

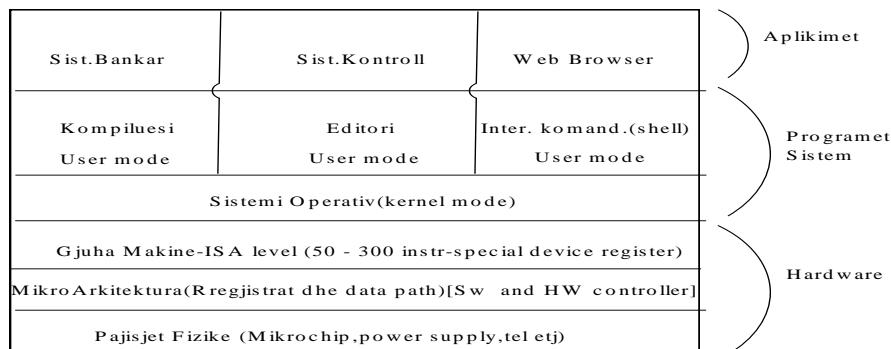
KAPITULLI I

HYRJE

Një sistem modern kompjuteri përbehet nga një ose me shume procesore, memorje qendrore, disqe, printerë, një tastjere, një monitor, nderfaqe network dhe pajisje të tjera hyrese/dalese (input/output). E thene ndryshe një sistem është shume i komplikuar. Për të shkruar programe qe ndjekin dhe përdorin të gjitha keto komponentë është vertet një pune shume e veshtire. Për ketë arsyе komjuterat jane të pajisur me një shtresa software-i qe quhet sistem operativ, detyra e të cilit është të menaxhoje të gjitha keto pajisje (device) dhe të siguroj programeve të përdoruesit një nderfaqe me të thjeshtë me hardware-in. Pra subjekti i ketij libri jane pikerisht keto sisteme.

Ne fig.1.1 tregohet menyra e vendosjes ne një sistem operativ. Ne fund kemi hardware-in, i cili ne shumicen e rasteve ka dy ose me shume nivele (shtresa). Ne nivelin me të ulet jane pajisjet fizike, të cilat përbehen nga qarqe të integruara chip-esh, tela (wires), tuba me reze katodike dhe pajisje të tjera të ngjashme me keto. Menyra e konstruktimit dhe e punimit të tyre është pune e inxhinierit elektrik.

Ne nivelin pasardhes kemi mikro arkitekturen, ne të cilën pajisjet fizike grupohen për të formuar njësi funksionale. Ky nivel përmban disa regjistra brenda ne CPU dhe një rrugë kalimi të dhenash (data path) qe përfshin një njësi arithmetike llojike (ALU-Arithmetic Logic Unit). Ne çdo cikel clock-u merren dy operanda nga regjistrat dhe me to kryhen veprime ne ALU. Rezultati ruhet ne një ose me shume regjistra. Ne disa makina, menyra e operimit të data path-it kontrollohet nga një software i quajtur mikrogram. Ne makina të tjera kontrollohet nga qarqe hardware-ike.



(Figure 1.1.Një system llogaritës)

Qellimi i data path-it është të ekzekutoje disa bashkesi instrukzionesh. Disa nga keto mund të behen ne një cikel të vetëm data path, të tjerat kerkojne me shume cikle data-path. Keto instruksione mund të përdorin regjistrat ose pajisje të tjera hardware-ike. Hardware-i dhe instruksionet të para se bashku nga një programues i gjuhes assembler formojne nivelin ISA (Instruction Set Architecture). Shpesh ky nivel quhet gjuhe makine.

Gjuha makine ka 50 deri ne 300 instruksione, shumica përdoren për levizur të dhena ne makine, për veprime arithmetike dhe për të krahasuar vlera. Ne ketë nivel pajisjet I/O kontrollohen duke ngarkuar vlera ne regjistra special të pajisjeve (device register). Për shembull, një disk mund të komandohet të lexoje duke ngarkuar ne regjistrat e tij vlerat e adreses se diskut, adreses se memories qendrore, numrin e byte-ve dhe veprimin (lexim ose shkrim). Praktikisht nevojiten me shume parametra dhe statusi që kthehet nga drive-ri pas një operacioni të caktuar është vertet shume kompleks. Vec kesaj, ne shume pajisje I/O kohezimi (timing) luan një rol të rendsishem ne programim.

Për të mos patur kete kompleksitet sigurohet një sistem operativ. Ky përbehet nga një shtrese software që pjeserisht fsheh hardware-n dhe i jep programuesit pér të punuar me një bashkesi instruksionesh me të përshtatshem.

Ne krye të sistemit operativ kemi software-t e sistemeve. Ketu kemi interpretuesin e komandave (shell), sistemet dritare (window systems), kompiluesit, editoret dhe programe të varura nga aplikacione të tillë. Është e rendesishme të kuptojme se keto programe nuk jane pjese e sistemit operativ, edhe pse ato sigurohen nga prodhuesit e kompjuterave. Kjo është një ceshtje kritike dhe delikate. Zakonisht sistemi operativ është ajo pjese e software-it qe vepron ne kernel mode ose supervisor mode. Ky mbrohet nga nderhyrjet e përdoruesit nepërmjet hardware-it (duke injoruar pér momentin disa mikroproçesore qe nuk kane fare mbrojtje hardware-ike). Kompiluesit dhe editoret veprojne ne user mode. Ne qoftë se një përdorues nuk pelqen një kompilues, ai mund të shkruaje vetë një tjetër të tille, por ai nuk mund të shkruaje një manovrues clocku të interrupteve (clock interrupt handler), i cili është pjese e sistemit operativ dhe qe normalisht mbrohet nga hardware ne menyre qe përdoruesi të mos e modifikoje atë.

Disa here, ky dallim ne sistemet e futura (mund të mos kene kernel mode) apo ne sistemet e interpretuara (sistemet operuese të bazuar ne Java të cilët përdorin interpretimin pér të ndare komponentët dhe jo hardware-in) mund të zbehet. Për kompjuterat tradicional sistemi operativ është akoma ai qe vepron ne kernel mode.

Ne shume sisteme ka programe të cilët veprojne ne user mode, por qe ndihmojne sistemin operativ ose përfomojne funksione të privilegjuara. Për shembull, shpesh kemi një program qe i lejon përdoruesit të ndryshoje fjalekalimin. Ky program nuk vepron ne kernel mode dhe nuk është pjese e sistemit operativ, por kryen një funksion të tille qe ne një fare menyre duhet jetë i mbrojtur.

Ne disa sisteme, kjo ide merret ne një forme shume ekstreme dhe pjeset e atij qe tradicionalisht quhet sistem operativ veprojne ne një hapesire pér përdoruesin (user space). Ne sisteme të tillë është shume e veshtire të vendosesh një kufi të quartë. Çdo gjë qe vepron ne kernel mode është quartësish pjese e sistemit operativ, por disa programe qe veprojne jashtë kesaj jane pjese të diskutueshme të tij, ose të paktën jane të lidhur ngushtë me të.

Me ne fund, sipër programeve sistem kemi programet aplikative. Keto programe jane marre ose jane shkruar nga përdoruesi pér të bere të mundur zgjidhjen e problemeve

të ndryshme, si operim me fjalet, llogaritjet inxhinierike apo grumbullimin e informacionit ne një database.

1.1. CFARE ËSHTË NJË SISTEM OPERATIV?

Shumica e përdoruesve të kompjuterit kane pasur eksperience me ndonjë sistem operativ, por është shume e veshtire të fiksosh se cfare bën saktësish një sistem operativ. Pjese e problemit është edhe fakti qe sistemi operativ kryen dy funksione qe rrenjësish nuk kane lidhje me njëri tjetrit, zgjerimin e makines (extending the machine) dhe menaxhimin e burimeve (managing the recourses). Përgjithesisht duhet të keni degjuar vetëm për njëren nga keto dy funksione. Le ti shohim të dy funksionet me ne detaje.

1.1.1. SISTEMI OPERATIV SI NJË EXTENDED MACHINE

Sic kemi përmendur, arkitektura (bashkesia e instruksioneve, organizimi i memorjes, I/O dhe struktura e bus-eve) e shume kompjuterave ne nivelin e gjuhes makine është e thjeshtë dhe e njojur nga programi, vecanerisht për pajisjet I/O. Për ta konkretizuar me shume ketë le të shohim se si behet një floppy disc I/O duke përdorur chips kontrollues të përshtatshem NEC PD765, të cilët përdoren ne shumicen e kompjuterave personal Intel-based.(Përgjatë librit ne do të përdorim termat “floppy disk” dhe “diskretë” ne vend të njëri-tjetrit). PD765 ka 16 komanda, secila e specifikuar me një ngarkese mes 1 dhe 9 byte-sh ne një device regjistër. Keto komanda jane për të lexuar dhe shkruajtur të dhena, për të levizur krahun e diskut (disc arm) dhe për të formatuar track-et, si dhe për të filluar, ndjere, rifilluar, rikalibruar kontrolluesin dhe drive-rat.

Komandat me themelore jane lexim dhe shkrim, secila prej tyre kerkon 13 parametra, të paketuara ne 9 bite. Keto parametra specifikojne terma të tille si adresa e bllokut të diskut për tu lexuar, numri i sektoreve për track, menyra e regjistrimit ne mediumin fizik dhe cfare të beje me shenjën e adreses se të dhenave të fshira (deleted-data-address-mark). Mos u merzisni ne se nuke kuptoni ketë ngatërrese fjalesh, kjo është pikërisht ceshtja paksa konfidenciale. Kur operacioni përbushet, chip-i kontrollues rikthen 23 statuse dhe fusha gabimi të paketuara ne 7 byte. Sikur kjo të mos mjaftonte programuesi i floppy diskut duhet gjithashtu të jetë vazhdimesh i vemendshem ne se motori është i ndezur apo i fikur. Ne se motori është i fikur, ai duhet të ndizet (me një vonese të gjatë startup-i) para se të dhenat të mund të lexohen apo të shkruhen. Motori nuk duhet lene ndezur për një kohe të gjatë sepse floppy disk demtohet. Ne ketë menyre programuesit i duhet të merret me kombinimin me të mire të vonesave të gjata të startup-it kundrejt demtimit të floppy disk-ut (dhe të humbase të dhena ne to).

Duke mos u futur ne detaje, duhet të jetë e qartë se një pjese e mire e programueseve nuk pelqejne të futen thelle ne programimin e floppy disqeve (ose hard disks, të cilët janë po aq kompleks dhe shume të ndryshem). Ajo cfare do një programues është e thjeshtë, të merret me abstraktsion të nivelit të lartë. Ne rastin e disqeve, abstraksion tipik do të ishte përbajtja e një bashkesie file-sh me emra ne disk. Secila file hapet për lexim ose shkrim, pastaj lexohet ose shkruhet dhe ne fund mbyllitet.

Programi, i cili fsheh të vertetën e hardware-it nga programuesi dhe shfaq një paraqitje të kendshme dhe të thjesht të file-ve të emeruara, të cilat mund të lexohen apo të

shkruhen është sistemi operativ. Sistemi operativ ashtu sic fsheh hardware-in nga programuesi dhe jep një paraqitje të thjeshtë të orientuar nga file-t, gjithashtu zhduk shume problem si; interrupt-et, kohezusat, menaxhimin e memorjes dhe dukuri të tjera të nivelit të ulet. Ne raste të tilla, sistemi operativ ofron një abstraksion me të thjeshtë dhe me të lehtë për tu përdoruar se sa ai i ofruar nga hardware-i përkatës.

Ne ketë kendveshtrim, funksioni i një sistemi operativ është të prezantoje programuesin me ekivalenten e një **extended machine** ose një **makine virtuale** qe është me e lehtë për tu programuar se hardware-i përkatës. Menyra se si sistemi operativ arrin ketë qellim është një histori e gjatë, të cilën do ta shohim ne detaja ne vazhdimin e librit. Sa për ta përbledhur, sistemi operativ siguron një shumellojshmeri sherbimesh të cilët merren nga programet duke përdorur instruksione speciale të quajtura thirrje të sistemit (system calls). Me tej ne ketë kapitull do të shohim disa nga thirrjet e sistemeve me të zakonshme.

1.1.2. SISTEMI OPERATIV SI NJË RE COURSE MANAGER

Koncepti i sistemit operativ si një sigurim të një nderfaqeje të thjeshtë për përdoruesit e tij është një veshtrim top-down. Një alternative, veshtrimi bottom-up konsiston ne faktin qe sistemi operativ është për të menaxhuar të gjitha pjeset e një sistemi kompleks. Kompjuterat modem përbehen nga procesoret, memoriet, kohuesat, disqet, mous-et, nderfaqesit e network-ut, printerat dhe një shumlojshmeri e njësive të tjera. Ne pamjen alternative, puna e sistemit operativ është për të siguruar një lokalizim të kontrolluar dhe të rradhitur të procesoreve, memorieve dhe njësive I/O përgjatë njësive të programeve të ndryshem konkuries me ta.

Imagjinoni se cfare do të ndodhne ne se 3 programe qe veprojne (running) ne të njëjtin kompjuter të mund të printonim outputin e tyre ne menyre të njëpasnjëshme ne të njëjtin printer. Rreshtat e pare mund të ishin nga programi i pare, pasardhesit nga program i dytë, për të vazhduar me ato të programit të tretë e keshtu me rradhe. Rezultati do të ishte një kaos. Sistemi operativ mund të sjelle rregull ne të tilla raste duke cuar të gjithe outputin e destinuar për ne printer, ne buffer. Kur një program mbaron, sistemi operativ mund të kopjoje outputin e tij nga disku ku ai ishte ruajtur për ne printer, nderkohe qe ne të njëjtën kohe programi tjetër mund të vazhdoje të gjeneroje me shume output, duke mos marre ne konsiderate faktin qe output nuk po shkon realisht ne printer (akoma).

Kur një kompjuter (ose network) ka përdorues të shumfishtë, nevoja për të menaxhuar dhe mbrojtur memorien, njësite I/O dhe burimet e tjera është akoma me e madhe, përderisa përdoruesit mund të interferojne të njëri-tjetri. Për me tepër, përdoruesit shpesh kane nevoje të ndajne jo vetëm hardware-in por edhe informacionin (file-t, database, etj), gjithashtu. Ne menyre të përbledhur kjo pamje e sistemit operativ tregon qe detyra e tij kryesore është të ruaj gjurmet e atij qe po përdor secilin burim, të plotësoj kerkesat e burimeve dhe të nderhyje ne kerkesat konfliktuale nga programe dhe përdorues të ndryshem.

Menaxhimi i burimeve përfshin burimet e multipleksuara (sharing) ne dy menyra: ne kohe dhe ne hapesire. Kur një burim është i multipleksuar ne kohe programe ose përdorues të ndryshem zene rradhe ne përdorimin e tij. Fillimisht një prej tyre e përdor burimin, pastaj një tjetër, e keshtu me rradhe. Për shembull, vetëm me një CPU dhe

programe qe duan të veprojne, sistemi operativ fillimisht lokalizon CPU ne një program, dhe pasi ai të ketë vepruar për një kohe të gjatë, një tjetër merr përdorimin e CPU, pastaj një tjetër dhe me vone përseri i pari. Përcaktimi se si burimi është i multipleksuar ne kohe, kush e ka rradhen dhe sa do të zgjase është detyra e sistemit operativ. Një shembull tjetër i multipleksimit ne kohe është ndarja e printerit. Kur pune shumefiske të printerit rradhiten për një printer të vetëm, duhet marre një vendim se cfare do të printohet pastaj.

Lloji tjetër i multipleksimit është multipleksimi ne hapesire, ne vend qe konsumatoret të zene rradhe, secili merr një pjese të burimit. Për shembull, memorja kryesore normalisht ndahet ndermjet disa programeve vepruese ne menyre qe secili të jetë resident ne të njëjtën kohe (për shembull, ne menyre qe të zene rradhe për përdorimin e CPU). Duke supozuar qe ka memorje të mjafueshme për të mbajtur programe të shumefishta është me eficiente të mbash disa programe ne memorie njehersh se sa një prej tyre, vecanerisht ne se ai ka nevoje vetëm për një sasi të vogel të totalit. Sigurisht qe kjo i rrit principet e ndershmerise, mbrojtjes e keshtu me rradhe, gje e cila është ne dore të sistemit operativ për ti zgjidhur. Një burim tjetër i multipleksuar ne hapesire është hard disku. Ne shume sisteme një disk i vetëm mund të mbaje file nga shume përdorues ne të njëjtën kohe. Lokalizimi i hapesires se diskut dhe ruajtja e gjurmave të atij qe po përdor secilin disk është një detyre tipike e menaxhimit të burimeve nga sistemi operativ.

1.2. HISTORIA E SISTEMEVE OPERATIVE

Sistemet operative kane qene ne zhvillim përgjatë ketyre viteve. Ne seksionet e meposhtme do të shohim shkurtimisht disa nga me kryesoret. Meqje sistemet operative historikisht kane qene të lidhur ngushtë me arkitekturen e kompjuterit ne të cilin veprojne, ne do të shohim ne gjeneratat e sukseshme të kompjuterave, ne menyre qe të shohim se si kane qene sistemet operative. Menyra e të parit të gjeneratave të sistemit operativ ne gjeneratat e kompjuterit është e patakt, por gjithsesi na krijon disa struktura ku përndryshe nuk do të kishim asnjë.

Kompjuteri i pare dixhital është dizenuar nga matematicieni anglez Charles Babbage (1792 – 1871). Edhe pse Babbage harxhoi shumicen e jetës se tij duke u munduar të ndertoje motorrin analistik (“analytical engine”) ai nuk mundi kurre të punoje ne të pasi ky ishte shume i varfer mekanikisht dhe teknologjia e ditëve të tij nuk mund të prodhonte gomat, mekanizmat dhe rrotat e nivelit qe ai kishte nevoje. E panevojshme të thuhet, motorri analistik nuk k ishte një sistem operativ.

Si një ane interesante historike, Babbage kuptoi qe do të mund ti duhej një software për motorrin e tij analistik, keshtu ai pajtoi ne pune një grua të re qe quhej Ada Lovelace, e cila ishte e bija e poetit të madh anglez Lord Byron, si programuesen e pare ne botë. Gjuha e programint Ada, quhet keshtu ne respekt të emrit të saj.

1.2.1. GJENERATA E PARE. VACCUM TUBES DHE PLUGBOARDS

Pas përpjekjeve pa sukses të Babbage, është bere një progres shume i vogel për konstruktimin e kompjuterave dixhital deri pas luftës se dytë botërore. Rreth viteve 1940, Howard Aiken ne Harvard, John von Neumann ne institutin e studimeve të avancuar ne Princeton, J. Prespër Eckert dhe William Mauchley ne universitetin e Pennsylvannia dhe Konrad Zuse ne gjermani, ndryshe nga të tjeret të gjithe patën sukses ne ndertimin e

makinave llogaritëse. Të parat përdoren përforcues mekanike por ishin shume të ngadalta, me cikle kohe të matur ne sekonda. Përforcuesit me vone u zevendesuan me vaccum tubes. Keto makina ishin të medha, duke mbushur dhoma të tëra me dhjetra mijera vaccum tubes, por ato ishin akoma miliona here me të ngadaltë se kompjuterat personal me të lire ne ditët e sotme.

Ne ato ditë, një grup i vetëm njerëzish dizenjuan, ndertuan, programuan, operuan dhe ruajtën secilen makine. Çdo programim behej ne gjuhe makine, shpesh duke lidhur plugboard-e për të kontrolluar funksionet kryesore të makines. Gjuhet e programimit ishin të panjohura (edhe gjuha assembly ishte e tille). Sistemet operative ishin të padegjuar. Operimi i zakonshem nga një përdorues ishte të hyntë për block të kohes, pastaj shkon poshtë ne dhomen e makines, vendos plugboard-in tij ne kompjuter dhe kalon disa ore duke shpresuar qe asnje nga 20000 apo dicka e tille vaccum tubes të mos digjen gjatë veprimit. Virtualisht të gjithe problemet ishin të drejtuara nga llogaritje numerike si studimi i tabelave të sinusit, kosinusit dhe logaritmeve.

Ne fillimet e para të 1950, rutina kishte zbuluar dicka me prezantimin epunched cards. Tani ishte e mundur të shkruaje programe ne card dhe të lexoje ato ne vend të përdorimit të plugboard-eve, përndryshe procedura ishte e njëjtë.

1.2.2. GJENERATA E DYTË (1955-65). TRANZISTORET DHE BATCH SYSTEMS

Prezantimi i tranzistoreve ne mes të viteve 1950 ndryshoi pamjen rrenjësisht. Kompjuterat u bëne të besueshem aq sa mund të prodhoheshin dhe tu shiteshin klientëve duke pritur qe të punonin aq gjatë sa të kryenin një pune ne menyre të suksesshme. Fillimisht k ishte një ndarje shume të qartë midis dizenjuesve, ndertuesve, operuesve, programueseve dhe personelitt mirembajtës.

Keto makina, tani të quajtura **mainframe**, ishin të mbyllura ne dhoma kompjuterash me ajer të kondicionuar, me staf operatoresh profesional qe i drejtonin. Vetëm korporata të medha si agjensite qeveritare ose universitetet mund të përballonin cmimin multimilion dollares. Për të bere një pune (job), (për shembull, një program ose një bashkesi programesh), një programuesi i duhet fillimisht ta shkruaje programin ne një letër (ne FORTRAN ose ne assemblers), pastaj ta shenoje (punch) atë ne card. Pastaj ai duhet të sjelle card deck-un ne dhomen e inputeve dhe t'ia dorezoje një operatori dhe të shkoje të pije një kafe derisa outputi të jetë gati.

Kur kompjuteri mbaron cfaredo lloj pune për të cilin ishte aktualisht duke punuar, një operator duhet të shkoje të printeri dhe të nxjerre outputin dhe ta coj atë ne dhomen e outputeve, keshtu qe programuesi mund ti mbledh me vone. Pastaj ai duhet të marre një nga deck cards qe i jane cuar nga dhoma e inputeve dhe të lexoje ne të. Ne qoftë se duhet kompliluesi FORTRAN, operatorit i duhet ta marre atë ne një kabinet filesh dhe të lexoje ne të. Shumica e kohes harxhohet nga operatoret duke levizur ne dhomat e makines.

Duke patur parasysh cmimin e lartë të pajisjes, nuk është për tu cuditur qe shume shpejt njerezit kerkuan menyra për të ulur (zevendesuar) kohen e humbur. Zakonisht zgjidhja me e pranueshme ishte Batch systems. Ideja e kesaj ishte të mblidheshin shume pune ne dhomen e inputeve dhe pastaj të lexoheshin ne një tape magnetik duke përdorur kompjuter relativisht të vogel, të pakushtueshem, si IBM 1401, i cili ishte shume

i mire ne leximin e cards, ne të kopjuarin e tape-ve dhe ne printimine outputeve, por jo aq i mire ne llogaritjet numerike. Të tjera makina me të kushtueshme, si IBM 7094, përdoreshin për llogaritjet reale. Kjo situatë tregohet ne figuren 1.2.

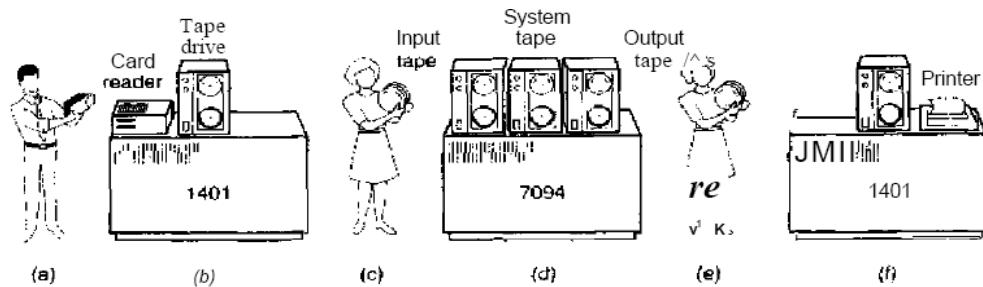


fig 1.2 (një batch sistem i hershem)

Pas një ore apo dicka e tille grumbullimi punesh, tape-i kthehet ne fillim dhe cohet ne dhomen e makines, ku vendoset ne një tape drive. Me pas operatori merre një program special (stërgjyshi i sistemit të sotëm operativ), i cili lexonte nga tape punen e pare dhe e kryente atë. Outputi ne vend qe të printohej shkruhej ne një tape të dytë. Pas çdo pune të mbaruar, sistemi operativ lexonte automatikisht punen tjetër nga tape dhe fillonte ta kryente atë. Kur të gjithe punet e grumbulluara mbaroheshin, operatori hiqte tape-t input dhe output, ne tape-in input vendoste punet e reja dhe tape-in output e conte ne një 1401 për printim **offline** (jo i lidhur me kompjuterin qendror).

Struktura e një pune tipike input tregohet ne figuren 1.3. Filloi me SJOB card, qe specifikontë maksimumin e kohes se veprimit (max run time) ne minuta, numrin qe duhej ngarkuar dhe emrin e programuesit. Pastaj vjen \$FORTRAN card, qe i tregon sistemit operativ të marre kompiluesin FORTRAN nga tape-i. Me pas vijonte programi qe duhej të kompilohej dhe pastaj një SLOAD card, qe drejtonte sistemin operativ të merrte programin objekt të sapo kompiluar.(Programet e kompiluara shpesh shkruheshin ne tape të gervishtura dhe duhet të merreshin saktë). Me pas vjen \$RUN card, qe i tregon sistemit operativ të veproje (run) ne program të ndjekur nga të dhenat. Ne fund, \$END card shenjon fundin e punes. Keto cards primitive kontrolli ishin paraprijesit e gjuheve moderne të kontrollit të puneve dhe interpretuesve të komandave.

Shume kompjutera të gjenerates se dytë përdoreshim shume për llogaritje inxhinierike dhe shkencore, si zgjidhja e ekuacioneve diferenciale me pjese qe shpesh gjenden ne fizike dhe inxhinieri. Ato programoheshin shume ne FORTRAN dhe ne gjuhen assembly. Të tille sisteme operative ishin FMS (The Fortran Monitor System), Sistemi operativ IMB për 7094.

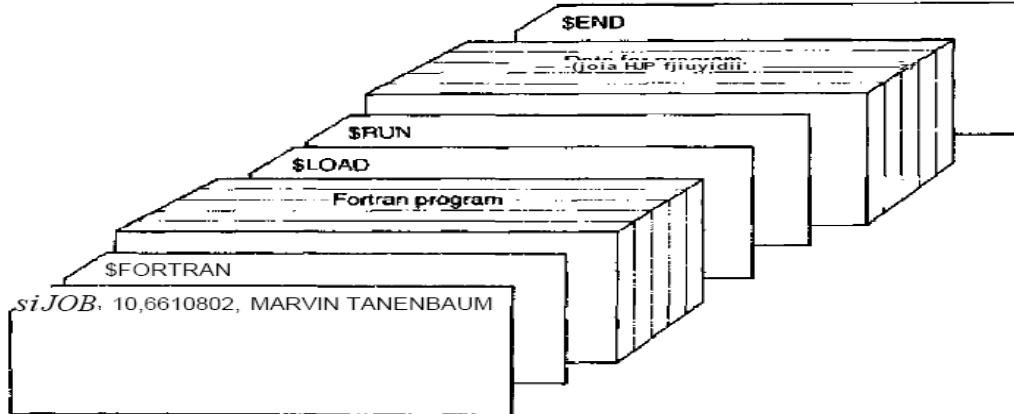


fig.1.3. Struktura e një pune tipike FMS

1.2.3. GJENERATA E TRETË (1965-1980). ICs DHE MULTIPROGRAMIMI

Ne fillim të viteve 1960, shumica e kompjuterave prodhoreshin ne dy linja prodhimi të vecuara dhe të papajtueshme. Ne njëren ane kishim kompjutera shkencore të një shkalle të lartë dhe word-oriented, sic është 7094, të cilët përdoreshin për llogaritje numerike ne shkence dhe inxhinieri. Ne anen tjetër, kishim kompjuterat komercial, character-oriented, sic është 1401, të cilët ishin gjeresisht të përdorur për klasifikimin e tape-ve (shiritit) dhe për printimin nga bankat dhe kompanitë e sigurimit.

Zhvillimi dhe mirembajtja e dy linjave plotësisht të ndryshme të prodhimi t ishte shume e kushtueshme për prodhuesit. Për me tepër, shume klientë të rinj të kompjuterave fillimisht donin një makine të vogel por me vone kerkonin një makine me të madhe ne të cilen të mund të vepronin të gjithe programet e meparshem por me shpejt.

IBM tentoi ti zgjidhte të dy problemet ne një të vetëm duke prezantuar keshtu System/360. 360 ishte një seri makinash të pajtueshme ne software duke filluar nga ato ne madhesi 1401 deri ne ato me të fuqishem se 7094. Makinat ndryshonin ne cmim dhe ne performance (memorje maksimale, shpejtësi procesori, numri i pajisjeve I/O të lejuara dhe keshtu). Teorikisht, përderisa shumica e makinave kane të njëjtën arkitekturë dhe bashkesi instrukSIONESH, programet e shkruara për një makine duhet të veprojnë edhe ne makinat e tjera. Vec kesaj, 360 është dizenuar ne menyre qe të merret me të dy llogaritjet shkencore dhe komerciale. Një familje e vetme makinash mund të kenaqë nevojat e të gjithe klientëve. Ne vitet ne vazhdim, duke përdorur teknologji me moderne IBM pati sukses të pajtueshme me atë të linjës 360, të njojur si seritë 370, 4300, 3080 dhe 3090.

360 ishte linja e pare kompjuterike e madhe qe përdori qarqe të integruara (JCs), keshtu qe krijon një avantazh ne cmim dhe ne performance kundrejt makinave të gjenerates se dytë, të cilët ishin të ndertuar nga transistore individuale. Kjo ishte një sukses i menjehershëm dhe shume shpejt ndertues të tjere të fuqishem adoptuan idene e krijimit të një familjeje kompjuterash të tille. Pasardhesit e ketyre kompjuterave janë akoma ne ditët e sotme ne përdorim nepër qendra të kompjuterit. Ne ditët e sotme janë

shume të përdorur për menaxhimin e database-ve të medha (për shembull, sistemet e rezervimit të linjave ajoore) ose përdoren si servera faqet World Wide Web qe duhet të veprojne me mijera kerkesa për sekonde.

Fuqia me e madhe e idese “një familje” papritur u be dobesia me e madhe e tij. Qelimi ishte qe të gjithe software-t, duke përfshire ketu edhe sistemin operativ, OS/360 duhet të punonin ne të gjitha modelet. Duhet të vepronate ne sisteme të vogla, qe shpesh zevendesonin 1401 për kopjimin e card-ave ne tape dhe ne sisteme të medha, qe shpesh zevendesonte 7094 për parashikimin e motit dhe llogaritje të tjera të medha. Duhet të ishte mire ne sistemet me pak pjese periferike dhe ne sistemet me shume pjese periferike. Duhej të punonte ne mjedise komerciale dhe shkencore. Mbi të gjitha, duhet të ishte eficiente për gjithe keto përdorime të ndryshme.

Nuk k ishte asnjë lloj menyre qe IBM (apo ndonjë tjetër) të mund të shkruanin një software për rrefimin e të gjithe ketyre nevojave konfliktuale. Rezultati ishte një sistem operativ jashtëzakonisht shume kompleks dhe gjigand, me shume mundesi 2 ose 3 here magnitude (madhesi) me të madhe se FMS. Përbehej nga miliona rreshta të shkruar ne gjuhe assembly nga mijera programues dhe përbante mijera viruse nga mijera lloje të ndryshme virusesh, të cilët kishin nevoje për një rrjedhe të vazhdueshme të shkarkimeve të reja ne një përpjekje për ti korriguar ato. Çdo shkarkim i ri korrigjonte disa virusë dhe sillte të tjere të rindërtuar, keshtu qe numri me shume mundesi qendronte po i njëjtë.

Një nga dizenjuesit e OS/360, Fred Brooks, me pas shkroi një liber të mprehtë (Brooks, 1996) qe përshkruante eksperiencat e tij me OS/360. Meqe do të ishte e pamundur të përmblidhnim librin ketu, të mjaftohemi duke thene qe kapaku tregon një tufe bishash kryelarta prehistorike në një gropë të erret.

Vec madhesise dhe problemeve gjigantë të tij, OS/360 dhe sisteme operative të prodhuar nga prodhues kompjuterash të tjere të ngjashem me keto aktualisht, përbushen nevojat e arsyeshme të shumices se bleresve. Ato gjithashtu popullarizuan mungesa teknike primare ne sistemet operative të gjenerates se dytë. Me shumë mundesi, me e rendesishme nga keto ishte **multiprogramimi**. Ne 7094, kur puna nderpritej për të pritur për kompletimin e operimit të një tape apo pajisjeve I/O, CPU rri koton deri kur I/O të mbaroje. Me llogaritjet e medha shkencore të lidhura me CPU, I/O është e rralle, keshtu qe kjo marreveshje e humbur nuk është domethene se. Me procesimin e të dhene komerciale, koha e pritjes se I/O shpesh mund të jetë 80% ose 90% e kohes totale, keshtu qe duhet të behej dicka qe të shmangtë të ndenjurit kot të CPU.

Zgjidhje e gjetur ishte ndarja e memories ne disa pjese me një pune të ndryshme për secilen pjese, sic tregohet ne figuren 1.4. Nderkohe qe një pune po pret qe të mbaroje I/O, një tjetër pune mund të përdore CPU. Ne qoftë se mund të mbaheshin pune mjaftueshem ne memorien kryesore ne të njëjtën kohe, CPU mund të qendronte e zene pothuajse 100% të kohes. Për të ruajtur ne memorie shume pune menjehere, kerkon hardware special për të mbrojtur çdo pune nga nderhyrjet dhe demtimet e të tjereve, por 360 dhe sisteme të tjere të gjenerates se tretë ishin pajisur me ketë hardware.

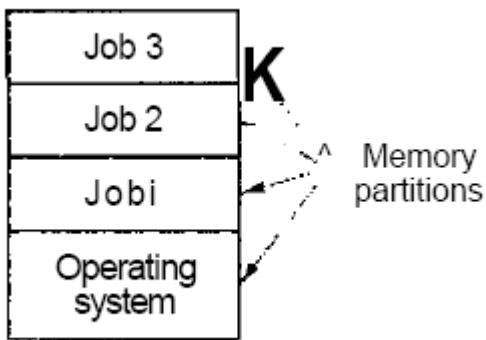


Fig.1.4. një sistem multiprogramming me tre pune ne memorie.

Një vecori tjetër e rendesishme e sistemeve operative të gjenerates se tretë ishte aftësia e leximit të puneve nga karta ne disk sa po ato të arrinin ne dhomen e kompjuterave. Sapo të mbaronte se vepruari një pune, sistemi operativ mund të marre një pune të re nga disku ne pjesen qe tanë është bosh dhe kjo pune mund të filloje veprimin. Kjo teknike njihet si **spooling** (nga operimi i menjehershëm periferik ne linjë) dhe ishte e përdorur për output, gjithashtu. Me spooling, 1401 nuk nevojiteshin me dhe mbajtja e tape-ve pothuajse u zhduk.

Megjithatë sistemet operative të gjenerates se tretë ishin të përshtatshem për llogaritje të medha shkencore dhe veprime me të dhenat komerciale masive, thellisht ato akoma ishin sisteme batch. Shume përdorues kerkonin akoma ditët e gjenerates se pare kur ato e kishin të gjithe makinen për vetë për disa ore dhe mund të rregullonin programet e tyre shume shpejt. Me sistemet e gjenerates se tretë, koha mes paraqitjes se punes dhe marrjes me pas të outputit ishte disa ore, keshtu vendosja keq e një presjeje të vetme mund të shkaktoje një komplikim të tille sa mund të deshtoje dhe programuesi të humbase një gjysem ditë.

Kjo deshire për kohe të shpejt responce (përgjigjeje) hapi rrugen për **timesharing** (ndarje kohe), variant i multiprogramimit, ku çdo përdorues ka një terminal online. Ne një sistem timesharing, ne qoftë se jane 20 përdorues logged in dhe 17 prej tyre po mendojne, flasin apo po pijne kafe, CPU mund të alokohet vetëm për tre punet qe duan sherbim. Për derisa njerezit për rregullimin e programeve zakonisht përdorin komanda të shkurtra (për shembull, kompilimi i një procedure pese faqeshe) me shume se komandat e gjata (për shembull, renditura e një file me miliona rekorde), kompjuteri mund të siguroje sherbim të shpejtë dhe interaktiv (bashkepunues) një numri përdoruesish dhe mbase mund të punoje ne pune grupi (batch job) ne background kur CPU po rri kot. Sistemi i pare serioz timesharing, **CTSS (Compatible Time Sharing System)**, u zhvillua ne **M.I.T.** ne një 7094 të modifikuar. Megjithatë timesharing nuk u be vertet e popullarizuar derisa nevojat për mbrojtje hardware u përhapen shume gjatë gjenerates se tretë.

Pas suksesit të sistemit CTSS, MIT, Bell Labs dhe elektrikut të përgjithshem, pastaj një prodhues i madh kompjuterash vendosi të nise me zhvillimin e “computer utility” (leverdi e kompjuterit), një makine qe do të përbollonte qindra përdorues timesharing të menjehershëm. Modeli i tyre ishte sistemi elektrik i shpërndare, kur nevojitet fuqi elektrike, vendoset një prize ne mur dhe ne të do të gjendet aq fuqi sa të nevojitet. Dizenjuesit e ketij sistemi, të njojur si **MULTICS (Multiplexed Information**

and Computing Service), parashikuan një makine super të madhe qe siguronte fuqi llogaritëse për çdo kend ne zonen Boston. Ideja qe makinat shume me të fuqishme se mainframe GE-645 do të shiteshin për mijera dollare, 30 vjet me vone ishte një trillim shkencor.

MULTICS ishte një sukses i vogel. Kjo ishte e dizenjuar për të suportuar qindra përdorues ne një makine pak me shume të fuqishme se një Intel 386-based PC, ndone se grumbullontë shume me shume kapacitet I/O. Kjo nuk është aq e cmendur sa mund të duket, përderisa njerezit e asaj kohe dinin të shkruanin programe të vogla dhe eficiente, aftësi qe me pas është humbur. Kishte shume arsyе qe MULTICS nuk pushtuan botën, arsyе jo pak e vogel ishte se ishte shkruar ne PL/1, dhe kompiluesi PL/1 ishte shume vite vone dhe kur ne fund arrinte mezi punonte. Për me tepër, MULTICS ishte jashtëzakonisht ambicioz për kohen e tij, me shume si motorri analistik i Charles Babbage ne shekullin e nentëmbedhjetë.

Për të bere të shkurtër një histori të gjatë, MULTICS prezantoi shume ide baze ne literaturen kompjuterike, por kthimi i saj ne një produkt serioz dhe ne një sukses komercial ishte shume me e veshtire nga c'mund ta priste çdonjëri. Laboratori Bell braktisi projektin dhe elektriku i përgjithshem la tëresisht biznesin e kompjuterave. Megjithatë, M.I.T. kembënguli dhe përfundimisht mori MULTICS për të punuar. Ne fund u shit si një produkt komercial nga kompania qe bleu biznesin e kompjuterave të GEs dhe e instaloi ne rreth 80 kompani të medha dhe universitete ne të gjithe botën. Nderkohe qe numrat e tyre ishin të vegjel, përdoruesat MULTICS ishin shume besnik. Për shembull, General Motors, Ford dhe agjensia nationale e sigurimit të shteteve të bashkuara, ne fund të viteve 1990 vetëm fiknin sistemet MULTICS, 30 vjet pasi MULTICS ishte shkarkuar (released)

Për momentin, koncepti i një utiliteti kompjuteri ka deshtuar por ai shume mire mund të kthehet ne formen e servera interneti masivisht të centralizuar me të cilët janë lidhur makinat user dumb (memec, të heshtur), me shumicen e punes qe ndodh ne serverat e medhenj. Motivimi ketu është qe shumica e njerezve nuk duan të administrojnë një sistem kompjuterik nazeli dhe me një kompleksitet rrítës dhe do të preferonin qe atë pune të behej nga një skuader profesionistësh për kompanine qe drejton serverin. E-commerce akoma po zhvillohet ne ketë drejtim, me kompani të ndryshme qe drejtojne emaile ne servera mikroproçesore me të cilat lidhen makinat e thjeshta të klientëve.

Përvec mungeses se suksesit komercial, MULTICS pati një influence të madhe ne sistemet operativ pasues. Ajo është përshkruar ne (Corbato, 1972; Corbato and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; dhe Saltzer, 1974). Ajo akoma ka një faqe Web aktive www.mulrticaris.org ne të cilën ka shume informacion për sistemin, dizajnesit e tij dhe për përdoruesit e tij.

Një zhvillim tjetër i madhe gjatë gjenerates se tretë ishte rritje e jashtëzakonshme e minikompjuterave, duke filluar me DEC PDP-1 ne 1961. PDP-1 kishte 4K fjale 18-biteshe, por për 5120000 për makine (me pak se pese përqind e cmimit të një 7094) shiteshin si hotcakes (shiteshin shume). Për lloje të vecanta punesh jonumerike, ky ishte pothuaj aq i shpejt sa 7094 dhe i dha fillimin një industrie të plotë të re. Ajo shume shpejt u ndoq nga një seri PDP-sh të tjera (ndryshe nga familja IBJvTs) duke arritur kulmin ne PDP-11.

Një nga shkencetaret e kompjuterave ne laboratorin Bell qe kishte punuar ne projektin MULTICS. Ken Thompson, me pas gjeti një minikompjuter PDP-7 qe nuk po

përdorej nga asnjë dhe kishte si qellim të shkruantë një version të MULTICS stripped-down dhe me një përdorues. Kjo pune me vone u zhvillua ne sistemin operativ UNIX, i cili u be i populluar ne botën akademike, me agjensite qeveritare dhe me shume kompani.

Historia e UNIX tregohet tjetërkuad (për shembull, Salus, 1994). Ne kapitullin 10 do të japid pjesë nga kjo histori. Tani mjaftohuni duke ju thene se ngaqë kodi burim ishte gjeresisht i përdorshem, organizata të ndryshme zhvilluan versionet e tyre të pakonkurueshem qe të conin ne një kaos. Dy nga versionet e medha të zhvilluara janë System V nga AT&T dhe BSD (Berkeley Software Distribution) nga universiteti i Californise ne Berkeley. Keto gjithashtu kishin variante të vegjel të tyre. Për të bere të mundur shkrimin e një programi qe mund të veproje ne çdo sistem UNIX, IEEE zhvilloi një standart për UNIX, të quajtur POSIX, qe suportohet nga shumica e versioneve UNIX. POSIX përcakton një nderhyrje minimale të thirrjeve sistem qe duhet të përkrahet nga sistemet UNIX konformant. Ne fakt, edhe disa sisteme të tjere operativ tashme suportojne (përkrahin) nderhyrjen POSIX.

Si me tutje, është mire të përmendet qe ne 1987, autor i leshoi një klonim të vogel të UNIX, të quajtur MINIX, qe përfshintë suportimin e POSIX. Është i përdorshem një liber i cili përshkruan opërimet e brendeshme të tij dhe liston kodet burim ne një appendix (Tanenbaum and Woodhull, 1997), MINIX gjendet free (duke përfshire kodet burim) ne internet ne URL www.cs.vu.nl/~ast/minix.html

Deshira për prodhimin e një versioni free të MINIX beri qe studenti Finlandez, Linus Torvalds, të shkruantë LINUX. Ky sistem u zhvillua ne MINIX dhe fillimisht suportontë disa dukuri të ndryshme të MINIX (për shembull, sistemi i filave MINIX). Edhe pse pas shume shtrirjesh ne shume menyrat ai akoma ruan një pjesë të madhe të struktura themelore si MINIX dhe UNIX. Shumica e atyre cfare do të thuhet për UNIX, përgjigjet edhe për System V, BSD, MINIX, Linux dhe versioneve të tjera, gjithashtu edhe klonimeve të UNIX.

1.2.4. GJENERATA E KATËRT (1980-SOT) PERSONAL COMPUTERS

Me zhvillimin e qarqeve **LSI (Large Scale Intëgratin)**, chipe qe përmbajne mijera tranzistore ne një centimetër kator silikoni, linden **PC (personal computers)**. Ne terma arkitekturore, personal computers (fillimisht të quajtur microcomputers), nuk ishin shume të ndryshem nga minikompjuterat e klases **PDP-11**, por ne terma të cmimit ishin plotësisht të ndryshem. Ne një departament, ne një kompani apo universitet ku mundesia për të pasur një minikomputer është *h*, chipi mikroproçesor beri të mundur qe për *n* individ të kishin *h* prej tyre një personal computer të tyrin.

Ne vitin 1974, kur doli Intel me 8080, CPU i pare 8-bitesh për qellime të përgjithshme, duhej një sistem operativ për 8080 ne menyre qe të jetë i mundur testimi i tij. Intel ia kerkoi qe të shkruantë një të tille, njërit prej konsulentëve të tij, Gary Kildall. Fillimisht Kildall dhe një shok i tij ndertuan një kontrollues për floppy disqet e rinj, Shugart Associatës 8-inch dhe bashkuan floppy disk-un me 8080, duke krijuar keshtu mikrokompjuterin e pare me disk. Me pas Kildall shkroi për të një sistem operativ disk-based të quajtur **CP/M (Control Program for Microcomputers)**. Intel duke mos menduar se keta mikrokompjutera kishin shume të ardhme, kur Kildall kerkoi për të

drejtat e CP/M, Intel pranoi kerkesen e tij. Me pas Kildall formoi një kompani të tijen për të ndihmuar për zhvillimin dhe shitjen e CP/M, Digital research.

Ne 1977, Digital Research rishkruan CP/M të tille qe ishte e përshtatshme të vepronte (run) ne shume minikompjutera qe përdornin 8080, Zilog Z80_h dhe të tjere chipe CPU. Shume programe aplikative u shkruan qe të vepronin ne CP/M duke bere keshtu të dominontë botën mikrollogaritëse për rreth 5 vjet.

Ne fillim të viteve 1980, IBM dizenjoi PC IMB dhe kerkoi për software qe të vepronte ne të. Njerez të IBM kontaktuan Bill Gatës për licensen e interpretuesit BASIC të tij. Gjithashtu i kerkuan atij ne se njihtë ndonjë sistem operativ qe të vepronte ne PC, Bill Gatës u sugjeroi atyre të kontaktonin Digital Research, kompanine dominuese të sistemeve operative të asaj kohe. Kildall, duke bere sigurisht qe të ishte vendimi me i gabuar ne biznes qe njihet ne histori, refuzoi të takohej me IBM duke i derguar një vartës madje. Për me keq, avokati i tij nuk pranoi as të firmostë marreveshjen e mbyllur të IBM, duke mbuluar PC akoma të paprezantuar. Si pasoje, IBM i kerkoi përseri Gatës ne se ai mund ti siguronte një sistem operativ.

Ne kohen qe IBM u kthye përseri tek Bill Gatës, ky i fundit mori vesh qe një ndertues lokal kompjuterash, Seattle Computer Products, kishte një sistem operativ të përshtatshem, **DOS (Disk Operating System)**. Ai iu drejtua atyre dhe u kerkoi ta blintë (gjoja 50000 \$), të cilën ata e pranuan lehtë. Pastaj Gatës i ofroi IBM paketën DOS/BASIC të cilën IBM e pranoi. IBM dontë disa modifikime të caktuara, keshtu Gatës punesoi personin qe shkroi DOS, Tim Palerson si një nenpunes të ri të kompanise të tij, Microsoft, qe ti bënte keto modifikime. Sistemi i korrigjuar mori emer tjetër, **MS-DOS (Microsoft Disk Operating System)** dhe shume shpejt dominoi tregun e PC IBM. Një faktor kryesor ishte vendimi i Gatës (ne prapaveshtrim, shume i zgjuar) për të shitur MS-DOS kompanive kompjuterike për tu lidhur me hardware-in e tyre, krahasuar kjo me përpjekjen e Kildall për të shitur CP/M përdoruesve të fundit (end users).

Ne vitin 1983 IBM PC/AT doli me CPU Intel 80286. MS-DOS ishte përhapur tashme dhe CP/M ishte ne ditët e fundit. Me vone MS-DOS u përdor gjeresisht ne 80386 dhe 80486. Megjithese versioni fillestar i MS-DOS ishte i thjeshtë, versionet pasues përfshire dukuri me të avancuara duke përfshire shume të marra nga UNIX. (Microsoft ishte i informuar për UNIX, madje gjatë viteve të para të kompanise shistë versione të tij për mikrokompjutera, të quajtura XENI).

CP/M, MS-DOS dhe sisteme të tjera operative për mikrokompjuterat e hershem ishin të gjithe të bazuar ne shtypjen e komandave nga tastjera. Kjo përfundimish ndryshoi ne saj të kerkimit të bere nga Doug Engelbart, ne institutin e kerkimeve ne Stanford ne vitet 1960. Engelbart krijoj **GUI (Graphical User Interface)**, qe plotësoi me windows, icons, menus dhe mouse. Keto ide ishin të adoptuara nga kerkues ne Xerox PARC dhe të përfshira ne makinat qe ndertonin.

Një ditë, Steve Jobs qe shpiku kompjuterin Apple ne garazhin e tij, vizitoi PARC, pa një GUI, dhe menjehere kuptoi vlerat e tij potënciale, dicka qe menaxhimi i famshem i Xerox se beri (Smith dhe Alexander, 1988). Pastaj Jobs bashkoi një GUI me një Apple. Ky projekt na coi tek Lisa, qe ishte shume e shtrenjtë dhe nga ana tregetare deshtoi. Përpjekja e dytë e Jobs, Macintosh i Apple-it, ishte një sukses i madh jo vetëm se ishte me i lirë se Lisa, por ishte gjithashtu një **user shoqeror (user friendly)**, qe do të thotë qe ishte jo vetëm për përdorues qe nuk dinin fare rreth kompjuterave, por edhe për ata qe nuk kishin ndermend të mesonin dicka.

Kur Microsoft vendosi të ndertontë një pasues të MS-DOS, u influencua shume nga suksesi i Macintosh-it. Ai prodhoi një sistem të GUI-based të quajtur Windows, qe fillimisht veproi ne deget e MS-DOS. Për rreth 10 vjet, nga 1985-1993, windows ishte një mjedis grafik ne krye të MS-DOS. Sidoqoftë, duke filluar qe ne 1995 doli një version i lire (freestanding) i Windows, Windows 95, qe përfshintë shume dukuri të sistemit operativ ne të, duke përdorur bazat e sistemit MS-DOS vetëm qe programet e vjetra MS-DOS të boot-oheshin dhe të vepronin (run). Ne 1998 doli një version pak i modifikuar i ketij sistemi, i quajtur Windows 98. Sidoqoftë, Windows 95 dhe Windows 98 se bashku, akoma përbajne një sasi të madhe të Intel 16-bit gjuhe assembly.

Një sistem tjetër operativ Microsoft është Windows NT (New technology), i cili është i pajtueshem me Windows 95 ne një nivel të caktuar. Dizenjuesi kryesor i Windows NT ishte David Cuttër, qe ishte gjithashtu një prej dizenjuesve të sistemit operativ VAX/VMS, keshtu qe disa ide të VMS jane të pranishme edhe ne NT. Microsoft priste qe versioni i pare i NT të shfarostë MS-DOS dhe të gjitha versionet e tjera të Windows meqe ai ishte një sistem superior me rendesi, por nuk e beri një gje të tille. Vetëm me Windows NT 4.0 me ne fund ia doli ne një menyre të mire, vecanerisht ne network-un e koorporatave. Versioni i 5 i Windows NT u quajt Windows 2000, i cili doli ne fillim të vitit 1999. U krijua me qellimin qe të ishte pasuesi i Windows 98 dhe Windows NT 4.0. Ky gjithashtu nuk ia doli mbane plotësisht keshtu qe Microsoft doli akoma me një version tjetër të Windows 98 të quajtur **Windows Me (Millenium edition)**.

Një tjetër pretëndues kryesor ne botën e personal computers ishte UNIX (dhe derivatët e tij të ndryshme). Unix është me i fuqishmi ne workstations dhe ne kompjuterat e tjere high-end si network servers. Ky është popullarizuar vecanerisht ne makinat e fuqizuara nga chip-e RISC me performance të lartë. Ne kompjuterat pentium-based, Linux po behet një alternative e populluar e Windows për studentat dhe për shume përdorues korporatash. (Përgjatë librit do të përdorim termin “Pentium” për t’iu referuar Pentium I, II, III dhe IV).

Megjithese shume përdorues UNIX, vecanerisht programues me eksperience, preferojne një nderfaqe command-based se një GUI, pothuajse të gjithe sistemet UNIX suporojne sistem me *n* window i quajtur sistem **X Windows**, i prodhuar nga MIT. Ky sistem trajton menaxhimin baze window, duke lejuar përdoruesit të krijojne, fshijne, levizin dhe të rivendose dritaret (windows) me një mouse. Shpesh një GUI i kompletuar, sic është **Motif**, është i përdorshem për veprimin e tij ne krye të sistemit X Windows duke i dhene UNIX një paraqitje pak a shume si Machintosh apo Microsoft Windows, për ata përdorues të tij qe duan një gje të tille.

Një zhvillim intëresant qe filloi ne mes të viteve 1980 ishte rritja e network-eve (rrjetëve) të personal computers ku vepronte sistemi operativ, **network operating systems** dhe ai **distributed systems (sistemet e shpërndara)** (Tanenbaum dhe Van Stéen, 2002). Ne një sistem operativ network, përdoruesit jane ne dijeni për ekzistencën e shume kompjuterave dhe mund të futen (log in) për të levizur dhe kopjuar file-t nga njëra makine të tjera. Ne një grup makinash vepron një sistem operativ lokal i ketij grupi dhe ka gjithashtu përdoruesin (përdoruesat) e vet lokal.

Sistemet operativ network thellesisht nuk jane të ndryshem nga sistemet operativ single - processor (proçessor të vetëm). Duket qartë qe ato kane nevoje për një kontrollues të nderfaqes se networkut dhe disa software të nivelit të ulet qe ta drejtojne atë, aq mire sa

programet të arrijne login dhe aksesimin e fileve, por keto gjera shtese nuk e ndryshojne strukturen baze të sistemit operativ.

Një sistem operativ i shpërndare, ne dallim është i tille qe shfaqet të përdoruesit e tij si një sistem tradicional uniprocessor, edhe pse ai aktualisht është i përbere nga shume procesore. Përdoruesit nuk kane pse të jene ne dijeni se ku veprojne (run) programet e tyre dhe se ku jane të vendosur file e tyre, të gjitha keto do të behen automatikisht dhe ne menyre eficiente nga sistemi operativ.

Sistemet operative e verteta të shpërndara kerkojne me shume se thjesht shtimin e një kodit të vogel ne sistemin operativ uniprocessor, sepse sistemet e shpërndara dhe të centralizuara ndyshojne ne rruge kritike. Sistemet e shpërndare, për shembull, shpesh lejojne aplikime qe të veprojne ne disa procesor ne të njëjtën kohe, keshtu duke kerkuar algoritma skedulimi procesoresh me shume kompleks, ne menyre qe të optimizojne masen e paralelizmit.

Vonesat e komunikimit ne network shpesh do të thone qe keto (dhe të tjere) algoritma duhet të veprojne me informacion të pakompletuar, të vjetëruar madje dhe jokorrekt. Kjo situatë thellesisht është e ndryshme nga një sistem single-processor ne të cilin sistemi operativ ka informacion të kompletuar për gjendjen e sistemin.

1.2.5 ONTOGENY RECAPITULATES PHYLOGENY

Pas botimit të librit të Charles Darwin “Origjina e Specieve”, zoologjisti gjerman Ernst Haeckel konstatoi se “Ontogeny rikapitolon Phylogeny”. Me ketë ai dontë të thoshtë se zhvillimi i një embrioni (ontogeny) përsërit (recapitulatës) evoluimin e species (phylogeny). Me fjele të tjera, pas fertilizimit, një veze njëriu shkon nepër etapa për tu bere peshk, një derr dhe keshtu me rradhe përparrë se të kthehet ne një bebe njëri. Biologë modern e konsideruan ketë si një simplifikim të madh, por ajo akoma ka një fare të vertetë ne të.

Dicka analoge me ketë ka ndodhur edhe ne industrine kompjuterike. Çdo specie e re (mainframe, minicomputer, personal computer, embedded computers, smart cards etj) duket sikur po shkon drejt zhvillimit qe kane bere paraardhesit e tyre. Mainframet e pare ishin programuar plotësisht ne gjuhe assembly. Edhe programet shume komplekse, sic Jane kompilatoret dhe sistemet operative, ishin shkruar ne assemblér. Ne atë kohe dolen ne skene minikompjuterat. FORTRAN, COBOL dhe gjuhe të tjera programimi të nivelit të lartë zakonisht përdoreshin nepër mainframe, por asnjë nga minikompjuterat e rinj nuk programoheshin ne assemblér (për shkak të memories). Fillimisht kur u shpiken mikrokompjuterat (personal computers të hershem), ato gjithashtu programoheshin ne assemblér edhe pse ne atë kohe minikompjuterat programoheshin ne gjuhe programimi të nivelit të lartë. Edhe kompjuterat palmtop gjithashtu filluan me kode assembly por shume shpejt levizen ne gjuhe programimi të nivelit të lartë (përgjithesht sepse punet zhvilluese beheshin ne makina të medha). E njëjta gje është e vertetë për smart cards.

Tani le të shohim sistemet operative. Mainframet e pare nuk kishin mbrojtje hardware-ike dhe nuk suportonin multiprogramimin, keshtu qe ne to vepronin sisteme operative të thjeshtë qe merrej vetëm me një program manualisht të ngarkuar ne një kohe të caktuar. Me vone me sigurimin e hardware-it, sistemet operative suportonin të merreshin me shume programe njehersh, dhe me vone kishin aftësi të plota timesharing.

Ne fillimet e shfaqjes se minikompjuterave, ato gjithashtu nuk kishin mbrojtje hardware-ike dhe ne to vepronente një program manualisht i ngarkuar ne një kohe të caktuar, edhe pse ne atë kohe multiprogramimi ishte i vendosur mire ne botën e mainframeve. Gradualisht ato siguruan mbrojtje hardware-ike dhe aftësi qe dy ose me shume programe të vepronin ne të njëjtën kohe.

Gjithashtu edhe mikrokompjuterat e pare ishin të aftë qe ne to të vepronente një program i vetëm ne një kohe të caktuar, por me vone edhe keto siguruan aftësi të multiprogramimit. Ne të njëjtën menyre ndodhi edhe me smart cards dhe kompjuterat palmtop.

Disqet fillimisht u shfaqen ne mainframe me medha, pastaj ne minikompjutera, mikrokompjutera dhe keshtu me rradhe. Edhe ne ditët e sotme smart cards nuk kane harddisk, por me ardhjen e flash ROM, ato se shpejti do të kene të njëjtën ekuivalence me të. Kur u shfaqen disqet ne fillim u zhvilluan shpejt sistemet e thjeshta file. Ne CDC 6600, mainframe me i fuqishem ne botë gjatë viteve 1960, userat e sistemeve file kane aftësi për krijimin e fileve dhe me pas e deklarojne atë të përhershme, qe do të thotë qe ai qendron ne disk edhe pas mbylljes se programit të krijuar. Për aksesimin e një file të tille me vone, një program duhet të lidhet me një komande speciale dhe të jape passwordin e tij (e plotësuar kur file u be i përheshem). Ne fakt, ishte një direktori e vetme e ndare nga të gjithe përdoruesit. Kjo ishte për të evitar konfliktet e emrave të fileve. Sistemet e fileve të minikompjuterave të hershem kishin një direktori të vetme të ndare nga të gjithe përdoruesit dhe keshtu edhe me sistemet file të hershem mikrokompjuter.

Memorja virtuale (aftësia për veprimin e me shume programeve se ne memorien fizike) pati një zhvillim të ngjashhem. Fillimisht u shfaq ne mainframe, minikompjutera, mikrokompjutera dhe gradualisht filloj ne sisteme të vogla e me të vogla. Networking kishte një histori të ngjashme.

Ne të gjitha rastet, zhvillimi i software u diktua nga teknologjia. Mikrokompjuterat e pare, për shembull kishin pak a shume 4KB memorie dhe s'kishin mbrojtje hardware-ike. Gjuhet e nivelit të lartë dhe multiprogramimi ishin shume, qe një sistem i vogel, i tille të merrej me to. Meqe mikrokompjuterat u shnderruan ne personal computers modern, ato siguruan hardware-in e nevojsphem dhe me vone software-in e nevojsphem të merret me dukuri me të avancuara. **Duket se ky zhvillim edhe përvitetë tjera qe vijne**. Gjithashtu fusha të tjera mund të kene të njëjtën rrjedhe rimisherimi, por ne industrije e kompjuterave duket se ndodh me shpejt.

1.3. LLOJET E SISTEMEVE OPERATIVE

E gjitha kjo histori dhe ky zhvillim na kane lene një shumellojshmeri të gjere sistemesh operative, prej të cilave jo të gjithe janë gjeresisht të njojur. Ne ketë seksion do të prekim shkurtimisht shtatë prej tyre. Me vone gjatë librit do të shohim disa prej ketyre llojeve të ndryshme të sistemeve operative.

1.3.1. SISTEMET OPERATIVE MAINFRAME

Ne skajin me të lartë është sistemi operativ për mainframe, ato kompjutera me madhesi sa një dhome qe akoma gjenden ne koorporatat kryesore të qendrave të të dhenave. Keto kompjutera e dallojne veten e tyre nga personal computers ne terma të kapacitetit të tyre I/O. Një mainframe me 1000 disqe dhe mijera gigabyte të dhena nuk

është i pazakonte. Një personal kompjuter me keto të dhena me të vertetë qe do të ishte i vjetër. Mainframe gjithashtu po bejne dicka ne përgjigje të serverave Web, servera për faqe tregetare elektronike të një shkalle të lartë dhe servera për transaksionet biznes-me – biznes.

Sistemet operative për mainframe janë veshtiresisht të orientuar drejt procesimit të shume puneve njehersh, shumica e të cilave duan madhesi të madhe I/O. Ato ne menyre tipike ofrojnë tre lloje të ndryshme sherbimesh: batch (grumbull, tufe), transaction processing (përpunimet transaksion) dhe timesharing. Një sistem batch është një sistem qe përpunon punet rutine. Përpunimi claims ne një kompani sigurimi behet ne menyre tipike nga sistemet batch. Sistemet e përpunimit transaksion merren me numra të medhenj të keresave të vogla, për shembull, përpunimi check (kontrollo) ne një banke apo rezervimet e linjave ajrore. Çdo njësi pune është e vogel, por sistemi duhet të merret me qindra ose mijera për sekonde. Sistemet timesharing lejojnë shume përdorues qe të kryejne pune ne një kompjuter njehersh, si ndertimin e një database të madhe. Keto funksione janë të lidhur ngushtë: Sistemet operative mainframe shpesh i bejne të gjitha keto. Një shembull i sistemit operativ mainframe është OS/390, pasardhes i OS/360.

1.3.2. SISTEMET OPERATIVE SERVER

Një nivel me poshtë janë sistemet operative server. Ato veprojnë ne servera, të cilët përdorin shume personal computers, workstations, ose mainframe. Ato i sherbejnë shume përdoruesve ne të njëjtën kohe mbi një rrjet dhe lejojnë përdoruesit të ndajne burimet e hardware-it dhe software-it. Serverat mund të sigurojnë sherbim print, sherbim file, apo sherbim Web. Krijuesit e internetit kane shume makina server qe të suportojnë klientët e tyre dhe Web site përdorin servera për mbledhjen e faqeve Web (Web pages) dhe të merren me keresat qe vijnë. Sistem operativ tipik server është UNIX dhe Windows 2000. Gjithashtu edhe Linux po fiton vend për servera.

1.3.3. SISTEMET OPERATIV MULTIPROCESSOR

Një menyre e zakonshme për të marre fuqi llogaritëse të lartë është të lidhim shume CPU ne një sistem të vetëm. Duke u varur saktësisht se me cfare janë lidhur dhe cfare ndajne, keto sisteme quhen kompjutera paralel, multikompjutera ose multiproçesor. Ata duan sisteme operative speciale, por shpesh keto janë variacione ne sistemet operative server, me dukuri speciale komunikimi dhe lidhjeje.

1.3.4. SISTEMET OPERATIVE PERSONAL COMPUTERS

Kategoria tjeter është sistemet operative personal computers. Puna e tyre është kriji i një nderfaqeje për një përdorues të vetëm. Ato janë gjeresisht të përdorur për përpunimin e fjales (word processing), spreadsheets dhe aksesim interneti. Shembull për ketë janë windows 98, Windows 2000, sistemi operativ Macintosh, dhe Linux. Sistemet operative Personal computers janë gjeresisht të njobur, keshtu qe duhet një prezantim shume i shkurtër i tyre. Ne fakt, shume njerez nuk janë ne dijeni të ekzistënces të sistemeve të tjere.

1.3.5. SISTEMET OPERATIV REAL-TIME (KOHE REALE)

Një lloj tjetër i sistemeve operativ është ai real-time. Keto sisteme janë karakterizuar duke pasur kohen si parametër kyc. Për shembull, ne sistemet e kontrollit të procesit industrial, kompjuterat real-time duhet të mbledhin të dhena rrethe procesit të prodhimit dhe ta përdore atë për të kontrolluar makinat ne fabrike. Shpesh takohen deadline (viza qe nuk duhet të kalohen). Për shembull, ne qoftëse një makine leviz poshtë një rresht assembly, veprime të caktuara ndodhin ne momente të caktuara, ne qoftë se një robot saldimi, saldon shume heret apo shume vone, makina do të shkatërritet. Ne qoftë se veprimi duhet absolutisht të ndodhe ne një kohe të caktuar, kemi një sistem real-time hard.

Një tjetër lloj sistemi real-time është sistemi real-time soft, ne të cilin mungesa e një deadline rastesor është e pranueshme. Audio dixhitale apo sistemet multimedia bien ne ketë kategori. Sisteme të njojur real-time janë: VxWorks dhe QNX.

1.3.6. SISTEMET OPERATIV EMBEDDED (TE NDERFUTURA)

Duke vazhduar poshtë ne sisteme të vogla e akoma me të vogla, vijme ne kompjuterat Palmtop dhe ne sistemet embedded. Një kompjuter Palmtop ose ndryshe një **PDA (Personal Digital Assistant)** është një kompjuter i vogel qe mund të qendroje ne një xhep kemishe dhe kryen një numer të vogel funksionesh, si një liber elektronik adresash. Sisteme embedded veprojne ne kompjutera ku pajisjet kontrolluese përgjithesisht nuk mendohen si kompjutera, si për shembull, TV sets, furre mikrovale, dhe telefonat mobile. Keto shpesh kane karakteristika të sistemeve real-time por gjithashtu kane madhesi, memorie dhe frenim fuqie (power restriction) qe i bejne ato speciale. Shembuj për sisteme operative të tille kemi PalmOS dhe WindowsCE (Consumer Electronics)

1.3.7. SISTEMET OPERATIVE SMART CARD

Sistemi me i vogel operativ vepron ne smart cards, të cilët janë pajisje me madhesine e një karte krediti qe përbajne një chip CPU. Ata kane një fuqi përpunimi shume të ashpër dhe shtengese memorieje. Disa prej tyre mund të kryejne vetëm një funksion, si pagesat elektronike, por të tjere mund të kryejne shume funksione ne të njëjtën smart card. Shpesh keto janë sisteme proprietar.

Disa smart cards janë të orientuar ne Java. Ajo cfare kjo do të thotë është qe ROM ne smart card ka një interpretues për makinen virtuale Java (Java Virtual Machine, JVM). Programe të vogla ne Java downloadohen ne card dhe me pas interpretohen nga JVM. Disa nga keto card mund të merren me shume programe ne Java ne të njëjtën kohe, duke cuar keshtu ne multiprogramim dhe ne nevojen për skedulim. Menaxhimi dhe mbrojtja e burimeve gjithashtu behet një problem kur dy ose me shume programe janë prezantë ne të njëjtën kohe. Me keto probleme duhet të merret sistemi operativ prezent ne card.

1.4. RISHIKIMI I HARDWARE KOMPJUTERIK

Një sistem operativ është i lidhur ngushtë me hardware-in e kompjuterit ku ai vepron. Ai tregon bashkesine e instruksioneve të kompjuterit dhe menaxhon burimet e tij. Qe të punoje, ai duhet të dije një pjese të madhe të hardware-it, të paktën si shfaqet hardware te programuesi.

Konceptualisht, një personal computer i thjeshtë mund të konspektohet ne një model të ngjashem me atë të Fig.1.5.CPU, memoria, pajisjet I/O janë të gjithe të lidhur nga një bus sistem dhe komunikojne me njëri-tjetrin me anen e ketij bus-i. Personal computers modern kane një strukture me komplekse, duke përfshire ketu edhe shume bus-e, të cilët do ti shohim me vone. Sa për tanë ky model është i mjaftueshem. Ne seksionet e ardhshme ne do të rishikojme shkurtimisht keta komponentë dhe do të ekzaminojme disa prej tyre, ata të cilët projektuesit e sistemeve operative janë të interesuar.

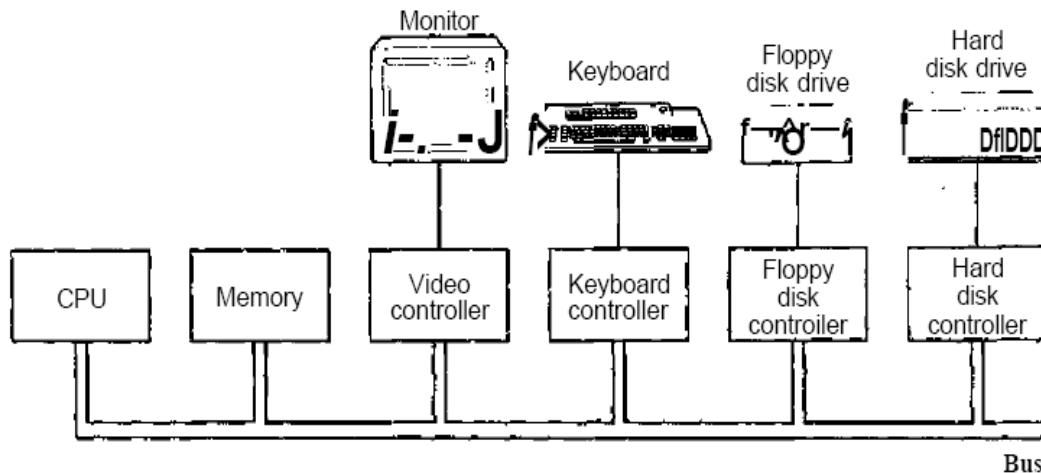


Figure 1-5. Some of the components of a simple personal computer.

1.4.1. PROÇESORET

CPU është “truri” i kompjuterit. Ai merre instruksione nga memoria dhe i ekzekuton ata. Cikli baze i çdo CPU është marrja e instruksionit të pare nga memoria, të dekodoje atë për të përcaktuar tipin dhe operandet e tij, të ekzekutoje atë dhe pastaj marrja, dekodimi, ekzekutimi i instruksioneve pasues. Ne ketë menyre kryhen programet.

Çdo CPU ka një bashkesi specifike instruksionesh qe ai mund të ekzekutoje. Keshtu një Pentium nuk mund të ekzekutoje programet SPARC dhe një SPARC nuk mund të ekzekutoje programet Pentium. Ngaqe aksesimi i memorjes, për të marre një instruksion apo një fjalë data zgjat me shume se ekzekutimi i një instruksioni, të gjithe CPU përbajne disa regjistra ne brendesi të tyre, për të mbajtur variablat kryesore dhe rezultatet paraprake. Keshtu bashkesia e instruksioneve përgjithesisht ka instruksione për

të marre një fjale nga memoria ne regjistër, dhe për të ruajtur një fjale nga regjistri ne memorie. Instruksione të tjere kombinojne dy operanda nga regjistrat, memoria apo të dy ne një rezultat, si mbledhja e dy fjalave dhe ruajtja e rezultatit ne një regjistër apo ne memorie.

Vec regjistrave të përgjithshem të përdorur për të mbajtur variablat dhe rezultatet paraprake, shumica e kompjuterave kane disa regjistra speciale qe jane të dallueshem për programuesin. Një nga keto është numeruesi i programeve (program counter), qe mban adresen e memories të instruksionit pasardhes qe do të merret. Pasi ai instruksion është marre, program counter shenjon pasardhesin e tij.

Një tjetër regjistër është stack pointer, qe shenjon ne krye të stack-ut aktual ne memorie. Stack përmban një frame për çdo procedure qe ka hyre por akoma nuk ka dale. Një frame stack i procedures mban keto parametra input, variabla lokale, variabla temporan qe nuk mbahen ne regjistër.

Akoma një tjetër regjistër është PSW (Program Status Word). Ky regjistër përmban bitin e kodit kusht, qe vendosen instruksione krahasimi, prioritet i CPU, mode (user apo kernel), dhe disa bite të tjere kontrolli. Programet user mund të lexojne të plotë PSW, por ne menyre tipike mund të shkruajne vetëm disa nga fushat e tij. PSW luan një rol të rendesishem në thirrjet sistem dhe I/O.

Sistemi operativ duhet të jetë ne dijeni të të gjithe regjistrave. Kur multipleksohet ne kohe CPU, sistemi opeartiv shpesh do të ndaloje programin vepruese për të restartuar një tjetër. Çdo here qe ndalon një program veprues, sistemi operativ duhet të save-je të gjithe regjistrat keshtu qe ata mund të ruhen kur programi të veproje përseri me vone.

Për të rritur performancen, projektuesit e CPU kane lene modelin e thjeshtë të marrjes, dekodimit dhe ekzekutimit të një instruksioni ne një kohe. Shume CPU moderne kane aftësi për ekzekutimin e me shume se një instruksioni ne të njëjtën kohe. Për shembull, një CPU mund të ketë të ndare njësite e marrjes, dekodimit dhe ekzekutimit keshtu qe nderkohe qe po ekzekutohet instruksioni n , mund të dekodoje instruksionin $n+1$ dhe të marre instruksionin $n+2$. Një organizim i tille quhet pipeline dhe ilustrohet ne fig.1.6(a) për një pipeline me tre stade. Pipeline me të gjatë janë të zakonshem, ne shumice e projektimeve pipeline, menjehere qe pipeline merret ne pipeline, ai duhet të ekzekutohet. Pipeline u shkakton shkruesve të kompliluesit dhe shkruesve të sistemit operative dhimbje koke sepse ato u tregojne atyre kompleksitetët e makines ne të cilin ndodhen.

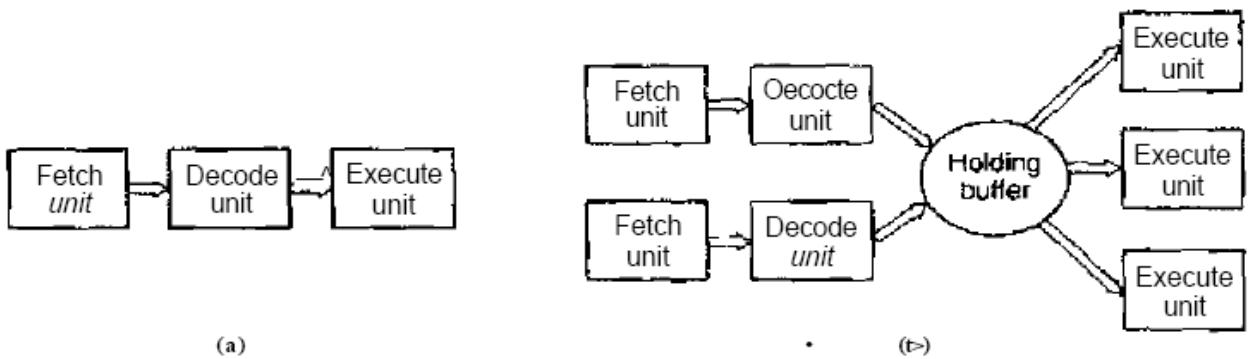


Figure 1-11. In) A three-stage pipeline, (b) A superscalar CPU.

Një projektim akoma me i avancaur se pipeline është një CPU superscalar, e treguar ne fig 1.6 (b). Ne ketë projektim, shume njësi ekzekutimi jane prezantë, për shembull, një për arithmetik integer, një për arithmetike me presje notuese dhe një për opéracionet Booleane. Dy ose me shume instruksione merren njehersh, dekodohen dhe hidhen ne një buffer mbajtës deri sa të ekzekutohen. Kur një execution unit është i lirë sheh ne bufferin mbajtës ne se ka ndonjë instruksion qe ai mund ta marre, dhe ne qoftë se ka e merr instruksionin nga bufferi dhe e ekzekuton atë. Një implikim i ketij projektimi është se instruksionet e programit shpesh nuk ekzekutohen sipas rrades. Për pjesen me të madhe, i takon hardware të siguroje qe rezultati i prodhuar është i njëjtë me atë qe do të kishte bere implementimi sekuencial, por, sic mund të shohim, një sasi e merzitshme e kompleksitetit është ngecur ne sitemin operativ.

Shumica e CPU, duke përgashtuar ato me të thjeshtat, të përdorura ne sistemet embedded kane dy mode, moden kernel dhe atë user, sic është përmendur dhe me pare. Zakonisht një bit ne PSW kontrollon moden. Gjatë veprimit të moden kernel, CPU mund të ekzekutoje çdo instruksion ne bashkesine e tij të instruksioneve dhe të përdore çdo njësi hardware-ik. Sistemi operativ vepron ne mode kernel, duke i dhene të drejtë për të gjithe hardware-in.

Ne ndryshim, programet user veprojne ne mode user, qe lejon vetëm një nenbashkesi të instruksioneve të ekzekutohen dhe një nenbashkesi të njësive të përdoren. Përgjithesisht, të gjithe instruksionet duke përfshire mbrojtjen I/O dhe të memories janë të palejuara ne mode user. Gjithashtu edhe vendosja e bitit të PSW ne kernel mode është e ndaluar.

Për të marre sherbime nga sistemi operativ, një program user duhet të beje një thirrje sistemi (system call), e cila si të themi ze ne gracke kernelin dhe kerkon sistemin operativ. **Intruksioni TRAP ndryshon nga moda user ne atë kernel dhe fillon sistemin operativ.** Kur puna përfundon, kontrolli kthehet të programi user të instruksioni qe vjen pas thirrjes sistem. Detajet e procesit të thirrjeve sistem do ti shpjegojmë me vone ne ketë kapitull. Si një shenim ne tipografi, ne do të përdorim fontin Helvetica me shkronja të vogla për të dalluar thirrjet sistem ne tekstin veprues, si kjo: **read**.

Është e rendesishme të themi se ka trape të tjera vec atyre të instruksioneve për ekzekutimin e një thirrjeje sistem. Shumica e trapeve të tjere shkaktohen nga hardware

për të lajmeruar situata përjashtuese si përpjekjen për të pjestuar me 0 apo një underflow të presjes notuese. Ne çdo rast sistemi operativ merr kontrollin dhe duhet të vendose cfare të beje. Disa here programi duhet të përfundoje me një error. Here të tjera error-i mund të injorohet (një numer i nenrrjedhur mund të vendoset 0). Ne fund, kur programi ka shpallur ne advancim se kerkon të merret me kondicione të llojeve të caktuara, kontrolli i kalon përseri programit duke e lene atë të merret me problemin.

1.4.2. MEMORIA

Komponenti i dytë kryesor ne çdo kompjuter është memoria. Idealisht, një memorie duhet të jetë shume e shpejtë (me i shpejtë se ekzekutimi i një instruksioni keshtu qe CPU nuk varet nga memoria), shume e madhe dhe shume e lire. Asnjë teknologji aktuale nuk i përbush të gjitha keto synime, keshtu qe është ndjekur një rruge tjetër. Sistemi i memories është i konstruktuar si një hierarki shtresash, sic tregohet ne fig 1.7.

Shtresa e sipërme përbehet nga regjistrat e brendshem ne CPU. Ato përbehen nga i njëjti material si ai i CPU dhe jane po ashtu po aq të shpejtë sa CPU. Si pasoje, nuk ka asnje vone se ne aksesimin e tyre. Kapaciteti i disponueshem i ruajtjes ne menyre tipike është 32x32-bit ne një

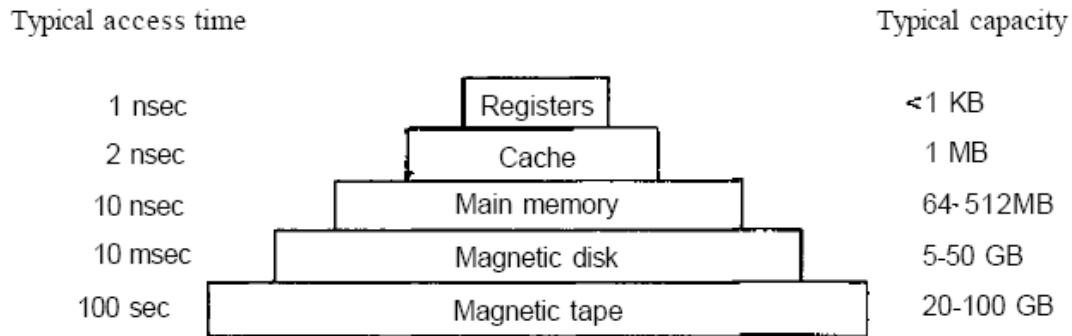


Figure 1-7, A typical memory hierarchy. These numbers are very rough approximations.

CPU 32-bit dhe 64x64-bit ne një CPU 64-bit. Me pak se 1KB ne të dy rastet. Programet duhet të menaxhojne regjistrat (të vendosin se cfare duhet të mbahet ne to), ne software.

Me pas vjen memoria Chace, e cila ne përgjithesi është e kontrolluar nga hardware. Memoria kryesore ndahet ne linja chace, ne menyre tipike 64 byte, me adresë 0 deri ne 63 ne linjën chace 0, adresat 64 deri ne 127 ne linjën chaces 1 dhe keshtu me rradhe. Linjat chace me të përdorura mbahen ne një chace me shpejtësi të lartë e vendosur brenda ose shume afer CPU. Kur programi duhet të lexoje një fjale memorieje, hardware chace kontrollon ne se reshti i nevojitur është ne chache ose jo. Ne qoftë se është, qe quhet **chace hit**, kerkesa plotësohet nga chace dhe ne memorie nuk dergohet asnje kerkese me ane të busit të memories kryesore. Chace hit normalisht kerkon dy cikle clocku. Chace misses (deshtim) duhet të shkoje ne memorie me një penalitet të konsiderueshem kohe. Memoria chace është e limituar ne madhesi qe përkon me koston e saj të lartë. Disa makina kane dy, madje edhe tre nivele chace, secila me e ngadaltë dhe me e madhe se ajo para saj.

Me pas vjen memoria kryesore. Kjo është ajo qe bën punet me të renda ne sistemin e memories. Memoria kryesore shpesh quhet **RAM (Random Access Memory)**, me të vjetër e quajne atë edhe **core memory**. Aktualisht, memoriet jane dhjetë ne qindra megabytes dhe duke u rritur ne menyre të shpejtë. Të gjitha kerkasat e CPU qe nuk mund të përmbushen ne chace shkojne ne memorien kryesore.

Me pas ne hierarki jane disqet magnetike (hard disks). Ruajtja ne disk është dy here me pak e kushtueshme se RAM për bit dhe shpesh dy here me e madhe gjithashtu. Problemi i vetëm është se koha e aksesimit të të dhenave ne të është rreth tre here me e ngadaltë. Kjo shpejtësi e ulet është nga fakti se një disk është një pajisje mekanike, sic tregohet ne fig 1.8.

Një disk përbehet nga një ose me shume pajisje metalike ne forme pjatë qe rrrotullohen ne 5400, 7200 ose 10800 rpm. Gjithashtu kane edhe boshtë të një krahu mekanik, mbi pjata nga qoshet, të ngjashme me krahus pickup ne një fonograf të vjetër 33 rpm. Informacioni shkruhet ne disk ne një seri qarqesh të koncentruar. Ne çdo pozicion krahu të dhene, secila nga kokat mund të lexoje një pjese unazore e quajtur **track**. Të gjithe tracks sebasaku për një pozicion krahu të dhene formojne një **cylinder**.

Çdo track është e ndare ne disa numra sektoresh, ne menyre tipike 512 byte për sektor. Ne disqet moderne, cilindrat e jashtëm përmbajne me shume sektore se ata të brendeshmit. Për të levizur krahus nga një cilinder ne cilindrin me pas kerkon rreth 1msec. Për ta levizur atë ne një cilinder të rastesishem kerken 5 deri ne 10 msec, duke u varur nga drive-i. Sapo krahu të jetë ne

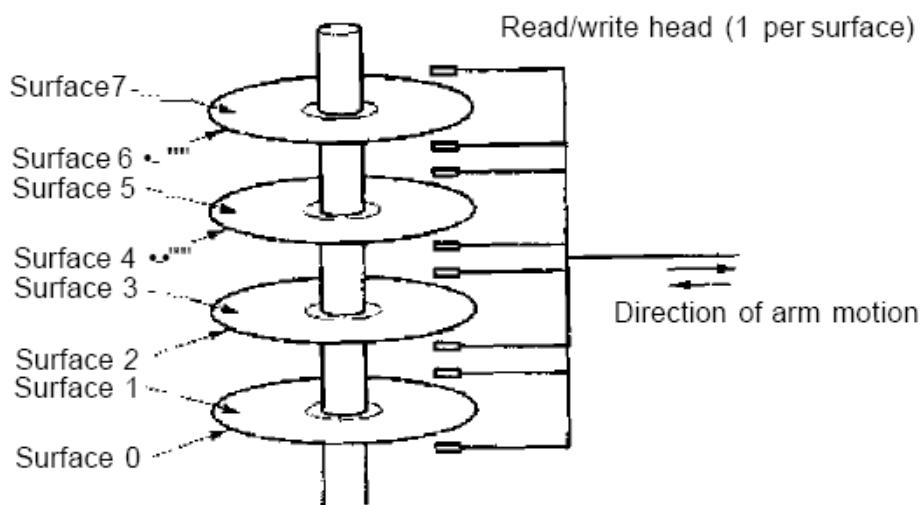


Figure 1-8. Structure of a disk drive.

krahun e duhur, drive-i duhet të prese për sektorin e nevojitur qe të rotullohet poshtë kokes, një vone se shtese prej 5 deri ne 10 msec qe varet nga rpm i drive-it. Sapo sektori të jetë poshtë kokes, leximi ose shkrimi behet ne ne rate prej 5MB/sec ne disqet e ngadalta deri ne 160MB/sec ne ata të shpejtët.

Shtresa e fundit ne hierarkine e memories është tape-i magnetik (shiriti magnetik). Ky mjet shpesh përdoret si një rezerve i ruajtjes ne disk dhe për mbajtjen e bashkesive të të dhenave shume të medha. Për aksesimin e një tape, fillimisht duhet të vendoset ne një

lexues tape, nga një person ose nga një robot (marrja e automatizuar e tape është e zakonshme ne instalimet me database të madhe). Pastaj tape mund të duhet të rrotullohet përpara për të arritur ne bllokun e kerkuar. Kjo mund të kerkonte disa minuta. Plus i madh i tape është jashtëzakonisht pak i kushtueshem për bit dhe i levizshem, qe është e rendesishme për tape rezerve qe duhet të ruhen jashtë ne menyre qe të rezistojne nga zjarret, përmbytjet, termetet, etj.

Hierarkia e memories qe kemi diskutuar është tipike, por disa instalime nuk i kane të gjitha shtresat ose kane të tjera pak të ndryshme (si një disk optik). Gjithesesi akoma ne të gjitha prej tyre, kur një ulet poshtë ne hierarki, koha e aksesimit të rastesishem rritet dramatikisht, kapaciteti rritet ne të njëjtën menyre dramatike dhe kosto për bit ulet pamase. Si pasoje, hierarkitë e memories do të jene edhe ne vitet qe vijne.

Vec llojeve të memorieve të diskutuara me sipër, shume kompjutera kane një sasi të vogel të qendrueshme memorie me aksesim të rastesishem (random access memory). Ndryshe nga RAM, memoriet e qendrueshme nuk e humbasin përbajtjen e tyre kur nderpritet energjia. ROM (Read Only Memory) programohet ne fabrike dhe me pas nuk mund të ndryshohet. Është e shpejtë dhe jo e kushtueshme. Ne disa kompjutera, ngarkuesi bootstrap, qe përdoret për të filluar kompjuterin, ndodhet ne ROM. Gjithashtu, disa karta I/O me ROM-in merren me kontrollin e pajisjeve të nivelit të ulet.

EEPROM (Electrically Erasable ROM) dhe flash RAM jane gjithashtu të qendrueshem, por ne ndryshim nga ROM mund të fshihet dhe të rishkruhet. Sidoqoftë, për ti shkruar ato kerkon me shume se të shkruash RAM, keshtu ato përdoren ne të njëjtën menyre si ROM, vetëm me një dukuri shtese qe tanë bën të mundur të korrigjosh viruset ne programet qe mbajne duke i rishkruar ata ne fushe.

Akoma një tjetër lloj memorije është CMOS, qe është e paqendrueshme. Shume kompjutera përdorin memorien CMOS për të mbajtur kohen dhe datën aktuale. Memoria CMOS dhe qarku i ores qe inkrementon kohen ne të, jane të fuqizuar nga një bateri të vogel, keshtu qe koha update-et (freskohet) ne menyre të saktë, edhe kur kompjuteri është i pavene ne prize. Memorja CMOS gjithashtu mund të mbaje parametrat e konfigurimit, se nga cili disk të boot-oje. CMOS përdoret sepse harxhon pak **power** saqe bateria e vendosur qe ne fabrike mund të zgjase për disa vjet. Gjithsesi, kur fillon të deshtoje, kompjuteri mund të filloje të ketë semundjen Alzheimer, qe harron gjerat qe i ka njojur për vite me rradhe, si për shembull, nga cili disk të boot-oje.

Le të fokusohemi ne memorien kryesore për pak. Shpesh është e pelqyeshme të mbash shume programe ne memorie njehersh. Ne qoftë se një program bllokohet duke pritur qe një lexicm diskut të kompletohet, një program tjetër mund të përdore CPU, duke i dhene një shfrytëzim me të mire CPU. Gjithsesi, me dy ose me shume programe njehersh ne CPU, duhet të zgjidhen dy probleme:

1. Si të mbrojme programet nga njëri-tjetri dhe tjetrin nga të gjithe ata
2. Si të merret me zhvendosjen

Shume zgjidhje jane të mundeshme. Megjithatë të gjitha prej tyre përfshijnë pajisjen e CPU me hardware special.

Problemi i pare është i dukshem, por i dyti është pak me delikat. Kur një program kompilohet dhe linkohet (bashkohet), kompliluesi dhe linkuesi nuk e dine se ne c'vend të memories fizike do të ngarkohet kur programi është ekzekutuar. Për ketë arsyen, ato

zakonisht supozojne qe do të filloje ne adresen 0 dhe aty do të vendoset instruksioni i pare. Supozojme se instruksioni i pare merre një fjale nga memorja ne adresen 10000. Tani supozojme se programi i plotë dhe të dhenat jane ngarkuar ne adresen 60000. Kur ekzekutohet instruksioni i pare, do të deshtoje sepse do ti referohet fjalet ne 10000 ne vend të fjalet ne 60000. Për të zgjidhur këtë problem, duhet ose të zhvendosim programin ne kohen e ngarkimit, të gjejme të gjitha adresat dhe ti modifikojme ato, gje e cila mund të behet por është **e kushtueshme, ose të (skanim i keq)**

Zgjidhja me e thjeshtë tregohet ne fig 1.9(a). Ne ketë figure shohim një kompjuter të pajasur me dy regjistra special, regjistri baze dhe regjistri limit (Vini re qe ne ketë liber, numrat qe fillojne me Ox jane ne hexadecimal). Kur një program është run, regjistri baze vendoset të shenjoje fillimin e tekstit të programit, dhe regjistri limit tregon sa të medhenj jane kombinimet e teksit të programeve dhe të dhenave. Kur një instruksion duhet të merret, hardware kontrollon ne se program counter është me pak se regjistri limit, dhe ne qoftë se është, e shton atë ne regjistrin baze dhe shumen e dergon ne memorie. Ne menyre të ngjashme kur programi kerkon të marre një fjale të dhenash (për shembull, nga adresa 10000), hardware automatikisht shton përmbajtjen e regjistrit baze me atë të adreses dhe shumen e dergon ne memorie. Regjistri baze e bën të pamundur për një program ti referohet një pjese memorije poshtë tij. Për me tepër, regjistri limit e bën të pamundur ti referohet një pjese të memories sipër tij. Keshtu kjo skeme zgjidh qe të dy problemet, atë të mbrojtjes dhe atë të zhvendosjes ne saj të dy regjistrave të rinj dhe një rritje të vogel të ciklit të kohes (për të përfornuar kontrollin dhe shtimin limit (limit check and addition))

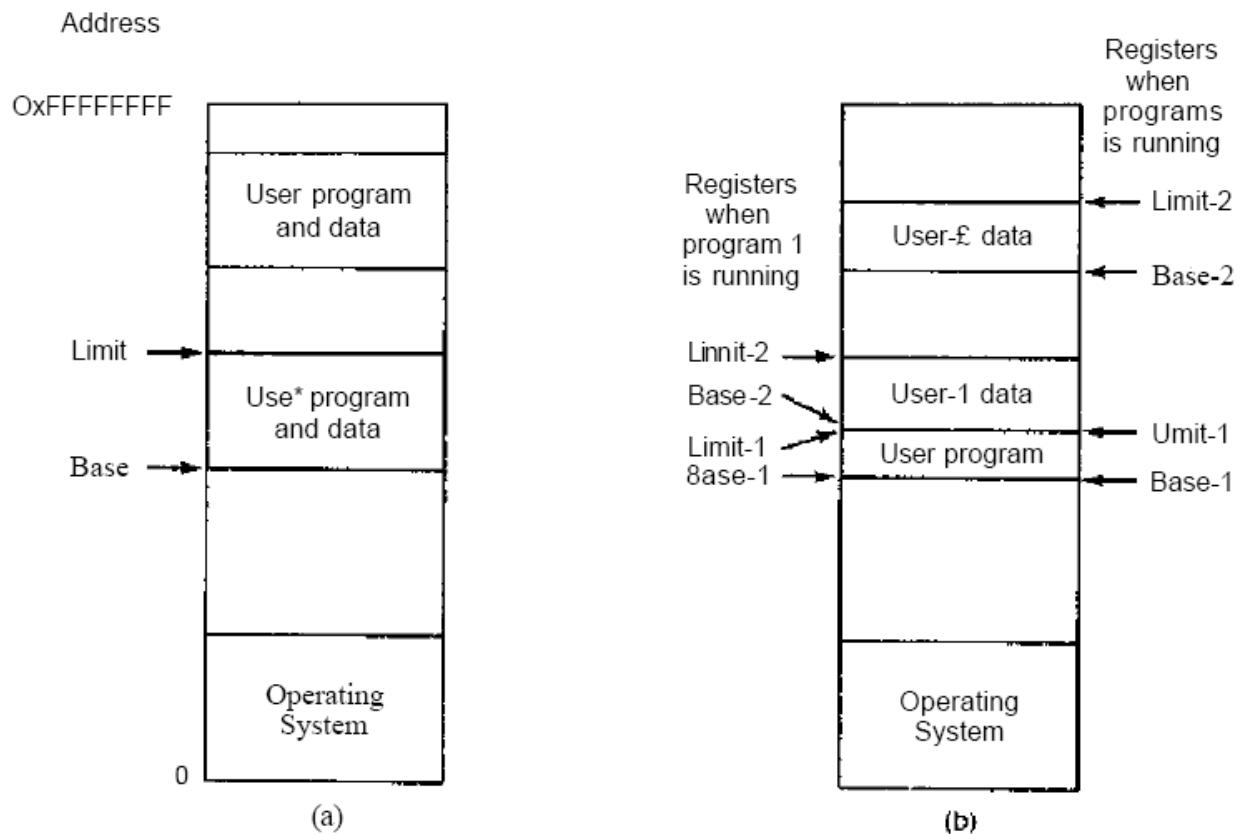


Figure 1-9. (a) Use of one base-limit pair. The program access memory between the base and the limit, (b) Use of two base-limit pairs. The program code is between Base-1 and Limit-1 whereas the data are between Base-2 and Limit-2.

Rezultati i kontrolluar dhe i hartuar është ne konvertimin e një adrese të gjeneruar nga programi, e quajtur **adrese virtuale**, ne një adrese të përdorur nga memoria, e quajtur **adrese fizike**. Ajo qe bën kontrollin dhe hartimin quhet **MMU (Memory Management Unit)**. Është e vendosur mes CPU-se dhe memories.

Një MMU me e sofistikuar tregohet ne fig 1.9(b). Kete ne kemi një MMU me dy ciftë registrash baze dhe limit, një për tekstin e programit dhe një për të dhenat. Program counter-i dhe të gjitha referencat e tjera të tekstit të programit përdorin ciftin e I-re dhe ato të të dhenave përdorin ciftin II-të. Si rrjedhim, tani është e mundur të kesh shume përdorues qe ndajne të njëjtin program me një kopje të vetme ne tij ne memorie, gje jo e mundur me skemen e pare. Kur programi 1 po vepron, katër regjistrat Jane të vendosur të shenuar me shigjeta ne të majtë të fig 1.9(b). Kur po vepron programi 2, ato Jane vendosur të shenuar me shigjeta ne të djathtë të figures. Ekzistojne MMU akoma me të sofistikuara. Disa prej tyre do ti shohim me vone ne ketë liber. Ajo qe duhet të veme re kete është qe menaxhimi i MMU-se duhet të jetë një funksion i sistemit operativ, përderisa përdoruesit nuk mund të jene të besuar se e bejne ne menyre korakte.

Dy aspekte të sistemit memorie kane efekt kryesor ne performance. Se pari, chace-të fshehin shpejtësine relativisht të ulet të memories. Kur një program ka qene ne

veprim për pak kohe, chace është plot me linjat chace të ketij programi, duke dhene një performance të mire. Sidoqoftë, kur sistemi operativ ndryshon nga njëri tek tjetri, chace mbetet plot me linjat chace të programit të pare. Ato qe duhen nga programi i pare duhet të merren një e nga një nga memoria fizike.

Se dyti, kur behet ndryshimi nga një program të tjetri, regjistrat MMU duhet të ndryshohen. Ne fig 1.9(b), duhet të risetohen vetëm 4 regjistra, qe nuk është shume problem, por ne MMU të verteta, duhet të ringarkohen shume regjistra, ne menyre dinamike apo ne menyre eksplikite, sipas nevojes. Secila rruge qe duhet ndermarre kerkon kohe. Morali i kesaj është se ndryshimi nga njëri program të tjetri, sic quhet **context switch**, është një biznes me të vertetë shume i kushtueshem.

1.4.3. PAJISJET I/O

Memorja nuk është i vetmi burim qe duhet të menaxhohet nga sistemi operativ. Pajisjet I/O gjithashtu bashkeveprojne ngushtë me sistemin operativ. Sic e pame ne fig 1.5, pajisjet I/O përgjithesisht përbehen nga dy pjesë: kontrolleri dhe pajisja vetë. Kontrolleri është një chip ose një bashkesi chipesh ne një plugboard qe fizikisht kontrollon pajisjen. Ai pranon komanda nga sistemi operativ, për shembull, të lexoje të dhena nga pajisja dhe ti marre ato.

Ne shume raste, kontrolli aktual i pajisjes është shume i komplikuar dhe i detajuar, keshtu qe është puna e kontrollerit ti paraqese një nderfaqe me të thjeshtë sistemit operativ. Për shembull, një kontroller diskur mund të pranoje një komande leximi për sektorin 11206 nga disk 2. Pastaj kontrolleri duhet të konvertoje ketë numer linear sektori ne një cilinder, sektor dhe koke (head). Ky konvertim mund të jetë i komplikuar nga fakti qe cilindrat e jashtëm kane me shume sektor se ato të brendeshmit dhe disa sektor të demtuar jane hartuar ne të tjere të rindj. Pastaj kontrolleri duhet të përcaktoje se kraku i diskut ne cilin cilinder është dhe ti jap atij një sekunce pulsesh për të levizur brenda apo jashtë numerit të kerkuar të cilindrave. Atij i duhet të prese deri kur sektori i duhur të rrullohet poshtë kokes dhe pastaj të filloje leximin dhe ruajtjen e biteve, ashtu sic ato vijnë nga drive, të heqë paraardhesit dhe të llogarise checksum-in. Se fundmi, ai duhet të grumbulloj bitet hyrese ne fjale dhe ti ruaj ne memorie. Për të bere gjithe ketë pune, kontrollerat shpesh përbajne kompjuterave embedded qe programohen për të bere punen e tyre.

Pjesa tjetër është pajisja aktuale vetë. Pajisjet kane nderfaqe të thjeshta, sepse nuk mund të bejne shume dhe ti bejne ato standarte. Kjo e fundit duhet sepse për shembull çdo kontroller disk IDE mund të merret me çdo disk IDE. IDE qe është Integrated Drive Electronics, është tipi standart i diskut ne Pentium dhe disa kompjuterave të tjere. Përderisa nderfaqja aktuale e pajisjes është e fshehur pas kontrollerit, gjithe ajo c'ka sheh sistemi operativ është nderfaqa të kontrolleri qe mund të jetë shume e ndyshme nga nderfaqja e pajisjes.

Meqe çdo lloj kontrolleri është i ndryshem, duhen software të ndryshme për të kontrolluar secilin nga ato. Software qe merret me kontrollerin, duke i dhene atij komanda dhe duke pranuar përgjigjet, quhet driveri i pajisjes (device driver). Ndertuesit e kontrollerave batch duhet të pajisin një driver për secilin sistem operativ qe suportojne. Keshtu për shembull, një skaner mund të shkoje me një driver për Windows 98, Windows 2000 dhe UNIX.

Driveri, qe të veproje ne moden kernel, duhet të vendoset ne sistemin operativ. Teorikisht, driverat mund të veprojne edhe jashtë modes kernel, por pak sisteme aktuale e suportojne ketë mundesi sepse ai kerkon aftësine për të lejuar një driver për hapesiren e përdoruesit, për të mund të aksesuar pajisjen ne një menyre të kontrollueshme, dukuri e suportuar rralle. Ka tre menyra qe driveri mund të vendoset ne kernel. Menyra e pare është të rilidhesh (relink) kernelin me një driver të ri dhe pastaj ta reboot-osh sistemin. Të shumtë jane sistemet UNIX qe punojne ne ketë menyre. Menyre e dytë është qe të bejme një hyrje (entry) ne një file të sistemit operativ duke i treguar qe i duhet një driver dhe pastaj reboot-ojme sistemin. Ne kohen e boot-imit, sistemi operativ shkon dhe gjen driverat qe i duheshim dhe i merr ato. Ne ketë menyre punon Windows. Menyra e tretë është qe sistemi operativ të jetë ne gjendje të pranoje drivera të rinj gjatë veprimit dhe ti instalojë ata pa patur nevojen qe ti boot-oje. Kjo menyre ka pasur përdorim të paktë por ne ditët e sotme po gjen me shume shtrirje për përdorim. Pajisje të tillë si USB apo IBEB 1394 (të diskutuara me pare) gjithmone duan drivera të ngarkuara ne menyre dinamike.

Çdo kontroller ka një numer të vogel regjistrash qe përdoren për të bere të mundur komunikimin me të. Për shembull, një kontroller disku minimal mund të ketë regjistra për të specifikuar adresen e diskut, adresen e memories, numrin e sektoreve dhe drejtimin (lexim apo shkrim). Për aktivizimin e kontrollerit, driveri merr një komande nga sistemi operativ dhe me pas e përkthen atë ne vlerat e duhura për të shkruar ne regjistrat e pajisjes.

Ne disa kompjutera, regjistrat e pajisjes jane hartuar ne hapesiren e adresave të sistemit operativ, keshtu ato mund të lexohen apo shkruhen si fjale të zakonshme të memories. Ne kompjutera të tille nuk nevojitet asnjë instruksion special I/O dhe programet user mund të mbahen larg nga hardware-i duke mos vendosur keto adresa memorjeje ne shtrirje të tyre (për shembull, duke përdorur regjistrat baze dhe regjistrat limit). Ne kompjutera të tjere, regjistrat e pajisjes jane ne një hapesire speciale të portave I/O, ku secili regjistër ka një adresë portë. Ne keto makina, instruksione speciale IN dhe OUT jane të disponueshem ne moden kernel për të lejuar driverat të lexojne apo të shruajne regjistrat. Skema e meparshme eliminon nevojen e instruksioneve speciale I/O por përdor një pjese të hapesires se adresave. Kjo e fundit nuk përdor fare hapesire të adresave por kerkon instruksione speciale. Qe të dy keto sisteme jane gjeresisht të përdorshem.

Inputi dhe outputi mund të behen ne tre menyra të ndryshme. Ne metoden me të thjeshtë, një program user leshon (kryen) një thirrje të sistemit, të cilën kerneli e shnderron me pas ne një thirrje procedure për driverin e duhur. Me pas driveri fillon I/O dhe qendron ne një right loop dhe ne menyre të vazhdueshme duke i bere sondazh pajisjes të shoh ne se është bere (zakonisht ka disa bit qe tregojne se pajisja është akoma e zene). Kur I/O ka përfunduar, driveri vendos të dhenat aty ku ato nevojiten (ne qoftë se ka të tille) dhe kthehet. Me pas sistemi operativ i kthen kontrollin thirresit. Kjo metode quhet **busy waiting**.

Metoda e dytë është qe driveri të startoje pajisjen dhe ti kerkoste atij ti jape një interrupt kur të mbaroje. Ne atë pike driveri kthehet. Pastaj sistemi operativ, ne qoftë se nevojitet, bllokon thirresin dhe sheh për ndonjë pune tjetër për të bere. Kur kontrolleri hap fundin e transferimit, për të sinjalizuar kompletimin gjeneron një **interrupt**.

Interruptet Jane shume të rendesishme ne sistemin operativ, keshtu qe le të ekzaminojme me afer ketë ide. Ne fig 1.10(a). shohim një proces tre-hapash për I/O. Ne

hapin e pare, driveri i tregon kontrollerit duke shkruar ne regjistrat e pajisjes se tij se cfare të beje. Me pas kontrolleri starton pajisjen. Kur kontrolleri ka mbaruar shkrimin ose leximin e numrave të byteve qe i është caktuar të transferoje, sinjalizon chip kontrollerin e interruptit duke përdorur linja busi të caktuara ne hapin e dytë. Ne qoftë se kontrolleri i interruptit është i përgatitur për të pranuar interruptin (i cili mund të mos jetë ne qoftë se është i zene me një tjetër me prioritet me të lartë), ai shpall një pin chipin CPU-se duke e informuar atë, ne hapin e katërt, kontrolleri interruptit vendos numrin e pajisjes ne bus ne menyre qe CPU mund ta lexoje atë dhe të dije se cila pajisje ka mbaruar (shume pajisje mund të jene ne veprim ne të njëjtën kohe).

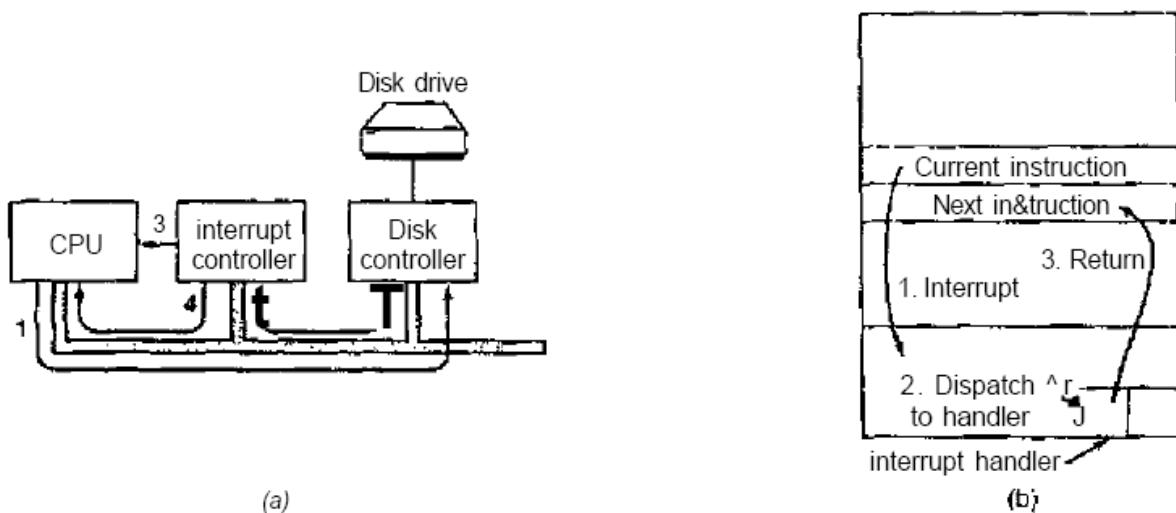


Figure 1-10. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves: taking the interrupt, running the interrupt handler, and returning to the user program.

Sapo CPU ka vendosur të marre interruptin, program counter-i dhe pastaj PSW, ne menyre tipike shtyhen ne stack-un aktual dhe CPU ndryshon ne kernel mode. Numri i pajisjes mund të përdoret si një index ne pjese të memories për të gjetur adresen e mbajtësit të interruptit për pajisjen. Kjo pjese e memories quhet interrupt vector. Sapo mbajtësi i interruptit (pjese e driverit për pajisjen qe sinjalizon interrupt) të ketë startuar, ai heq stacked program counter dhe PSW dhe i ruan ato, pastaj pyet pajisjen për të mesuar statusin e saj. Kur mbajtësi i interruptit ka përfunduar, ai kthehet ne programin user qe me pare ishte ne veprim, ne instruksionin e pare qe akoma nuk ishte ekzekutuar. Keto hapa tregohen ne fig.1.10(b).

Metoda e tretë për të bere punet I/O përdor një chip special DMA (Direct Memory Access) qe mund të kontrolloje rrjedhen e biteve mes memories dhe disa kontrollerave pënderhyrjen konstantë të CPU. CPU e nderton chipin e DMA, duke i treguar sa byte të transferoje, duke përfshire adresat e pajisjes dhe memories dhe drejtimin, dhe e le të vazhdoje. Kur behet chipi i DMA, ai shkakton një interrupt, i cili trajtohet sic thame me

sipër. Hardware-i DMA dhe I/O ne përgjithesi do të përshkruhet me shume ne detaje ne kapitullin 5.

Shpesh interruptet mund të ndodhin ne momente të papërshtatshme, për shembull, kur një tjetër mbajtës interrupti (interrupt handler) është ne veprim. Për ketë arsy, CPU ka një menyre qe të beje të paaftë interruptet dhe pastaj ti aftësoje përseni me vone. Nderkohe qe interruptet jane joaktive, çdo pajisje mund të gjeneroje interrupte, por CPU nuk nderpritet deri sa të aktivizoje përseni interruptet. Ne qoftë se, pajisje të shumta mbarojne qe interruptet jane të c'aktivizuar, kontrolleri i interruptit vendos se cilin të lere të parin, zakonisht bazuar ne prioriteten statik i shenuar të çdo pajisje. Pajisja me prioritet me të lartë fiton.

1.4.4. BUSET

Organizimi i figures 1.5. ishte i përdorur për vite me rradhe ne minikompjutera dhe gjithashtu ne personal computer IBM. Gjithesesi, me arritjen e procesoreve dhe memories me të shpejtë, aftësia e një busi të vetëm (dhe sigurisht busi IBM PC) për tu marre me të gjithe trafikun ishte sforcuar deri ne piken me të fundit. Dicak duhet të behej. Si rezultat i kesaj, u shtuan buse shtese, për të pajisje I/O me të shpejta dhe gjithashtu për të minimizuar trafikun CPU-memorie. Si pasoje e ketij evoluimi, një sistem i madh Pentium aktualisht Jooks, dicak si ne fig.1.11.

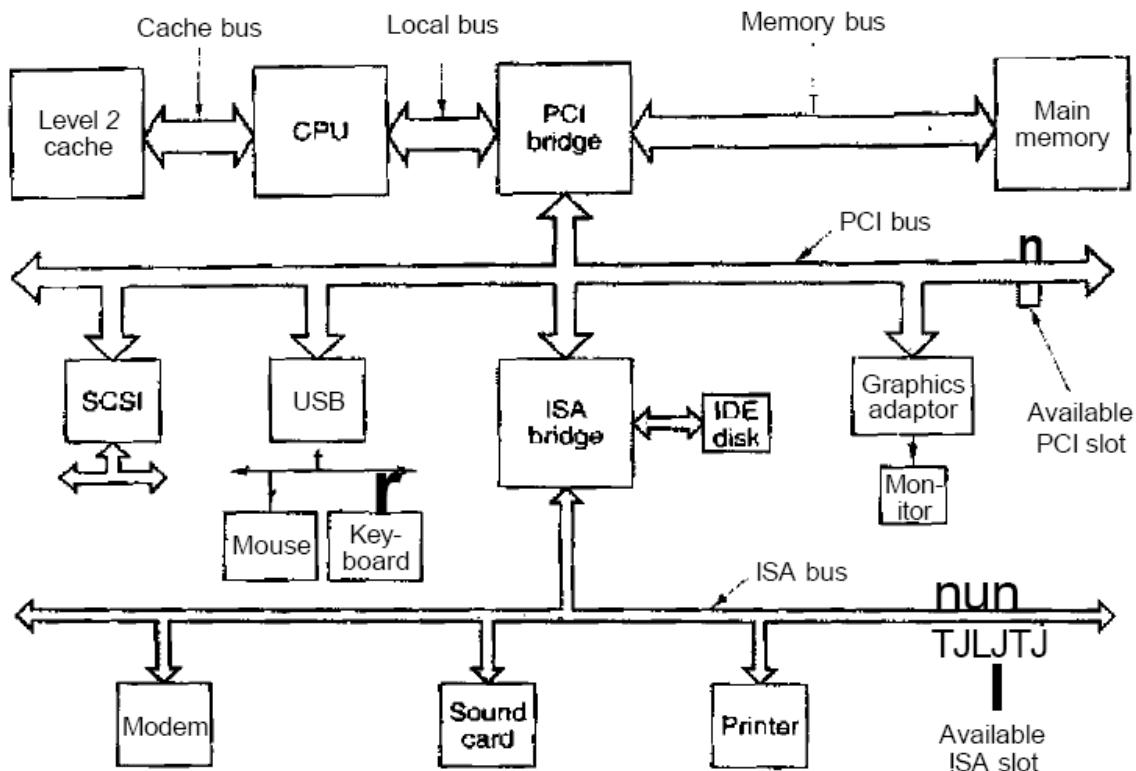


Figure 1-11. The structure of a largo Pemiurri system

Sistemi ka teë buse (chace, local, memory, PCI, SCSI, USB, IDE dhe ISA), ku secili ka rate (shkalle) transferimi dhe funksion të ndyshem. Sistemi operativ duhet të jetë ne dijeni të të gjithe ketyre për konfigurimin dhe menaxhimin. Dy buset kryesore janë origjinalët e IBM PC busi ISA (Industry Standard Architecture) dhe pasuesi i tij, PCI (Peripheral Component Interconnect). Busi ISA, qe fillimisht ishte busi IBM PC/AT, vepron ne 8.33MHz dhe mund të transferoje dy byte njehersh, ne një maksimum prej 16.67 Mb/sec. Ai është i përfshire për pajtueshmeri të prapambetur të kartave I/O të vjetra dhe të ngadalta. Busi PCI u krijuar nga Intel si një pasardhes i busit ISA. Ai mund të veproje ne 66MHz dhe të transferoje 8 byte ne të njëjtën kohe, për një data rate prej 528 MB/sec. Shumica e pajisjeve I/O me shpejtësi të lartë tanë përdorin busin PCI. Edhe dida kompjutëra jo-Intel gjithashtu përdorin busin PCI ne saj të numrit të madh të kartave I/O të përdorshme për të.

Ne ketë konfigurim, CPU bashke bisedon me uren e chipit PCI nepërmjet busit lokal (local bus), dhe ura e chipit PCI bashke bisedon me memorien nepërmjet një busi të dedikuar memories, shpesh qe vepron ne 100MHz. Sistemet pentium kane ne chip një chace nivel i I-re dhe jashtë chipit një chace shume me e madhe niveli i II-të, të lidhur me CPU me busin e chace-se.

Për me tepër, ky sistem përbën tre buse të specializuar: IDE, USB dhe SCSI. Busi IDE është për të lidhur pajisjet periferike si, disqet dhe CD-ROM me sistemin. Busi IDE është një dege e nderfaqes se kontrollerit të diskut ne PC/AT dhe tanë është standarte ne afersisht të të gjithe sistemet Pentium-based për hard diskun dhe shpesh për CD-ROM.

Busi USB (Universal Serial Bus) u krijuar për të lidhur të gjitha pajisjet e ngadalta I/O, si tastjeren dhe mouse-in, me kompjuterin. Ai përdor një lidhës 4 fijesh, dy nga të cilat sigurojnë fuqi elektrike për pajisjen USB. USB është një bus i centralizuar ne të cilin një pajisje rrenjë u bën sondazh pajisjeve I/O çdo 1 msec ne qoftë se ka trafik apo jo. Ai mund të manovroje një shume totale ngarkimesh prej 1.5 MB/sec. Të gjithe pajisjet USB ndajne një driver të vetëm pajisjesh USB, duke e bere të panevojshme instalimin e një driveri të ri për çdo pajisje të re USB. Për pasoje, pajisjet USB mund të shtohen ne kompjuter pa patur nevoje të reboot-ohen.

Busi SCSI (Small Computer System Interface) është një bus me performance të lartë i caktuar për disqe të shpejta, skanera dhe pajisje të tjera qe duan gjeresi brezi të konsiderueshme. Ai mund të veproje deri ne 160 MB/sec. Ai ka qene prezent ne sistemet Macintosh qe ne fillimet e krijimit të tyre dhe gjithashtu është i populuar ne UNIX, dhe ne disa sisteme Intel-based.

Akoma një bus tjetër (qe nuk është treguar ne fig 1.11.) është IEEE 1394. Ndonjehere ky quhet FireWire. Apple e përdor për implementimin e tij të 1394. Ashtu si USB, IEEE 1394 është një serial bitesh por është projektuar për transfertën e paketave me shpejtësi deri ne 50 MB/sec, duke e bere atë të përdorshem për lidhjen e cacorderave dixhitale dhe pajisje multimedia të ngjashme me një kompjuter. Ndryshe nga USB, IEEE 1394 nuk ka një kontroller qendror. SCSI dhe IEEE 1394 hasin konkurence nga versionet e shpejta qe jane zhvilluar të USB.

Për të punuar ne një sistem si ai i fig.1.11, sistemi operativ duhet të dije se cfare është aty dhe ta konfiguroje atë. Kjo nevoje coi Intelin dhe Microsoftin të projektojne një sistem për PC i quajtur **plug and play**, qe bazohet ne një koncept të ngjashem të implementuar fillimisht ne Apple Machintosh. Para plug and play, çdo kartë I/O kishte një nivel fiks të kerkesave për interrupt dhe adresa fiksë për registrat I/O. Për shembull,

tastiera ishte interrupt 1 dhe përdortë adresat 0x60 deri 0x64, kontrolluesi i floppy disk-ut ishte interrupt 6 dhe përdortë adresat 0xF0 deri ne 0xF7 dhe printeri ishte interrupt 7 dhe përdortë adresat 0x378 deri ne 0x37A, dhe keshtu me rradhe.

Deri ketu çdo gje ishte ne rregull. Shqetësimet erdhen kur përdoruesi bleu një kartë zeri dhe një kartë modem dhe duhet të përdoreshin të dyja, le të themi interrupt 4. Ato ishin konfliktuale mes njëra tjetres dhe dhe nuk mund të punonin se bashku. Zgjidhja ishte përfshirja e celesave **DIP** ne çdo kartë I/O, dhe instruktimi i përdoruesit qe ti vendose ato ne menyre qe të zgjedhin një nivel interrupt-i dhe adresa të pajisjeve I/O qe nuk jane konfliktuale me ndonjë tjetër ne sistemin e përdoruesit. Adoleshentët të cilët dedikuan jetën e tyre ne **ngatërresarat** e hardware-it të PC ndonjehere mund ta bënin ketë pa gabime. Fatkeqesisht, asnjë tjetër nuk mundi, gje e cila coi ne kaos.

Ajo cfare plug and play bën është të pasurit e një sistemi qe automatikisht të mbledhe informacion rreth pajisjeve I/O, kryesisht të përcaktoje nivelet e interrupt-eve dhe adresat I/O dhe me pas ti tregoje seciles kartë se cilët Jane numrat e saj. Shkurtimisht, kjo punon sic tregohet ne Pentium. Çdo Pentium ka një parentboard (formalisht i quajtur me pare motherboard përparrë se korrektesia politike të godistë industrine kompjuterike). Ne parentboard ndodhet një program i quajtur **BIOS (Basic Input Output System)**. BIOS-i ka software I/O të nivelit të ulet, përfshire këtu edhe procedurat e leximit të tastierës, shkrimit ne screen, berja e disk I/O. Ne ditët e sotme, ai mbahet ne një RAM të shpejtë, qe është i qendrueshem por mund të update-ojet nga sistemi operativ kur gjenden virusë ne BIOS.

Kur kompjuteri boot-ojet, fillon BIOS-i. Fillimisht sheh se sa RAM është instaluar dhe ne se tastiera apo pajisje të tjera kryesore Jane instaluar dhe veprojne ne menyre korakte. Ai fillon me skanimin e bus-eve ISA dhe PCI për të zbuluar të gjitha pajisjet e lidhura. Disa nga keto pajisje ne menyre tipike Jane **trashegime (legacy)** (të projektuar para krijim të plug and play) dhe kane nivele fikse të interrupt-eve dhe adresash I/O (me shume mundesi Jane të vendosur nga celesa ne kartën I/O, por jo të modifikueshme nga sistemi operativ). Keto pajisje Jane të regjistruara. Gjithashtu edhe pajisjet plug and play Jane të regjistruara. Ne qoftë se pajisjet qe Jane prezantë Jane të ndryshme nga ato pajisje të boot-imit të fundit të sistemit, ato konfigurohen.

BIOS-i me pas përcakton pajisjen e boot-imit duke provuar një listë të pajisjeve të ruajtur ne memorjen CMOS. Përdoruesi mund ta ndryshoje ketë listë duke hyre ne programin e konfigurimit BIOS direkt pas boot-imit. Ne menyre tipike, behet një përpjekje për të boot-uar nga floppy disk. Ne qoftë se kjo deshton provohet CD-ROM. Ne qoftë se as floppy dhe as CD-ROM nuk Jane prezant sistemi boot-ojet nga hard disk. Sektori i pare nga pajisja e boot-imit lexohen ne memorje dhe ekzekutohet. Ky sektor përmban një program qe normalisht ekzaminon tabelen e pjeseve ne fund të sektorit të bootimit për të përcaktuar se cila pjese është aktive. Me pas lexohet një ngarkues boot-imi sekondar nga ajo pjese. Ky ngarkues lexon ne sistemin operativ nga pjesa aktive dhe fillon.

Me pas sistemi operativ i kerkon BIOS-it të marre informacionin konfigurues. Për çdo pajisje, ai kontrollon ne se secila prej tyre ka driver-in e pajisjes. Ne qoftë se jo, i kerkon përdoruesit të vendose një floppy disk apo CD-ROM qe përmban driver-in (i siguruar nga prodhuesit e pajisjes). Sapo ai të ketë të gjithe driver-at e pajisjeve, sistemi operativ i ngarkon ato ne Kernel. Me pas ai inicializon tabelat e tij, krijon çdo lloj background-i qe u nevojitet proçeseve dhe starton një login program ose GUI ne çdo

terminal. Të paktën kjo është menyra qe supozohet të funksionoje. Ne jetën e përditshme plug and play është tepër e pabesueshme saqë njerezit e quajne plug and pray.

1.5. KONCEPTET E SISTEMIT OPERATIV

Të gjithe sistemet operative kane koncepte baze të caktuara si proceset, memorja dhe file-t të cilat janë kryesore ne kuptimin e tyre. Ne pjeset e me poshtme ne do të shohim disa prej ketyre koncepteve baze edhe pse shkurtimisht, si një prezantim. Do ti shohim me të detajuara secilen prej tyre me vone ne ketë liber. Për të ilustruar keto koncepte here pas here do të shohim shembuj të ndryshem, zakonisht të marre nga UNIX. Ne menyre tipike shembuj të ngashhem ekzistojne edhe ne sisteme të tjera gjithashtu.

1.5.1. PROÇESET

Një koncept kryesor ne të gjithe sistemet operative është **proçesi**. Ne menyre paresore një proçes është një program ne ekzekutim. Me çdo proçes shoqerohet hapesira e adresave të tij (address space), qe është një listë e vendndodhjeve të memorjes nga një minimum (zakonisht zero) ne një maksimum, të cilën proçesi mund ta lexoje dhe shkruaje. Hapesira e adresave përmban programin e ekzekutueshem, të dhenat e programit dhe stack-un e tij. Gjithashtu me çdo proçes shoqerohen disa bashkesi regjistrash, duke përfshire ketu program counter, stack pointer dhe regjistra të tjere hardware, dhe të gjithe informacionin tjetër të nevojitur për veprimin e programit.

Ne shume detaje do të vijme përsëri të proceset ne kapikullin II, por tani për tani menyra me e lehtë për të marre një ndjesi të mire intuitë për një proçes është të mendojme rrëth sistemeve timesharing. Ne menyre periodikë, sistemi operativ vendos të ndaloje se vepruari një parogram dhe të filloje një tjetër, sepse i pari ka patur me shume se koha e tij e ndarjes se CPU-se ne sekondin paraardhes.

Kur një proçes pezullohet përkohesisht si ky, duhet të restart-ojet me vone ekzaktësisht ne të njëjtin stil qe kishte para se të nderpritej. Kjo nenkupton qe i gjithe informacioni rrëth proçesit duhet ne menyre eksplicitë të ruhet diku gjatë pezullimit. Për shembull, proçesi mund të ketë disa file të hapura për lexim ne të njëjtën kohe. Për secilen nga keto shoqerohet një pointer qe jep pozicionin aktual (numrin e byte-ve apo record-eve qe duhet të lexohen me pas). Kur një proçes është përkohesisht i pezulluar, të gjithe keto pointer-a duhet të ruhen ne menyre qe një thirrje leximi qe ekzekutohet pas ristartimit të proçesit të lexojë të dhenat e duhura. Ne shume sisteme operative, çdo informacion rrëth secilit proçes, vec përbajtjeve të hapesires se adresave të veta, ruhet ne një tabele sistemi operativ e quajtur **tabela proçes (process table)**, e cila është një grup strukturash, një për çdo proçes aktualisht ekzistëntë.

Keshtu, një proçes i pezulluar vec të tjerash përbehet nga hapesira e adreses se tij, zakonisht e quajtur core image (ne nder të memorieve core magnetike të përdorura ne ditët e meparshme), **tabelen e tij të hyrjeve të proçesit (process table entry)**, qe vetë kjo përmban regjistrat e tij.

Thirrjet sistem kryesore të menaxhimit të proçesit **jane ato merren me krijimin** dhe mbarimin e proceseve. Konsideroni një shembull tipik. Një proçes i quajtur

interpretuesi i komandave (**command intérpretér**) ose **shell** lexon komandat nga një terminal. Përdoruesi apo ka shkruar një komande duke kerkuar qe një program të kompilohet. Shelli tani duhet të krijoje një proces të ri ne menyre qe kompiluesi të veproje. Kur procesi ka mbaruar kompilimin, ai ekzekuton një thirrje sistem qe të mbaroje edhe vetë ai.

Ne qoftë se një proces mund të krijoje një apo me shume procese të tjera (të referuara si **proçese femije**) dhe keto procese gjithashtu të mund të krijojnë procese femije, shume shpejt do të arrijme ne strukturen e pemes proces (process tree structure) të figures 1.12. Proçeset e lidhura, qe bashkepunojne për të bere disa pune, shpesh kane nevoje për të komunikuar me njëri-tjetrin dhe të sinkronizojnë aktivitetët e tyre. Ky komunikim quhet **komunikimi intérproçes** dhe do të jetë me i detajuar ne kapitullin e dytë.

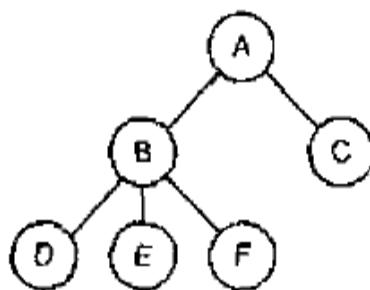


fig 1.12. Një peme proces, procesi A ke krijuar dy proçese femije, B dhe C. Proçesi B ka krijuar tre proçese femije, D, E, dhe F.

Të tjera thirrje sistemi të proçesit jane të përdorshme për të kerkuar me shume memorie (apo të le memorien e papërdorur), për të pritur për përfundimin e një procesi femije dhe për të vendosur programin e tij mbi një tjetër të ndryshem.

Here pas here, nevojitet të transportohet informacion ne një proces veprues i cili nuk është duke pritur për ketë informacion. Për shembull, një proces i cili po komunikon me një proces tjetër ne një kompjuter tjetër e bën ketë duke derguar mesazhe ne procesin tjetër nepërmjet një rrjeti kompjuterik. Për tu ruajtur nga mundesia qe një mesazh mund të jetë ose përgjigja e tij mund të jetë humbur, derguesi mund të kerkoj qe sistemi i tij operativ ta informoje atë pas një numri të caktuar sekondash, keshtu qe ai mund të ritransmetoje mesazhin përsëri ne qoftë se akoma nuk ka marre një pranim (acknowledge). Pas vendosjes se ketij kohuesi, programi mund të vazhoje të beje pune të tjera.

Pasi kalon numri i caktuar i sekondave, sistemi operativ i dergon proçesit një sinjal alarm. Ky sinjal shkakton të proçesi pezullimin, përkohesisht, e cfaredoqoftë të atij qe po bën, ruan regjistar e tij ne stack dhe fillon të ekzekutoje një procedure të vecantë të manovrimit të sinjalit, për shembull, të ritransmetoje një mesazh qe ka mundesi të jetë humbur. Pasi përfundon manovrimi i ketij sinjali, proçesi veprues kthehet ne gjendjen e tij qe ishte para sinjalit. Sinjalet janë software analoge të interrupteve hardware-ike dhe vec mbarimit të kohes mund të jene të gjeneruara nga një shumellojshmeri shkaqesh.

Shume trape të zbuluara nga hardware, si ekzekutimi i një instruksioni ilegal ose përdorimi i një adrese të gabuar, gjithashtu konvertohen ne sinjale për procesin fajtor.

Çdo personi të autorizuar për të përdorur një sistem i caktohet një **UID (User IDentification)** nga administratori i sistemit. Secili proces i filluar ka UID-in e personit, i cili e filloj ketë proces. Një proces femije ka po të njëjtin UID si procesi i tij prind. Përdoruesit mund të jene pjesetare të grupeve, tek secili nga të cilët ka një **GID (Group IDentification)**.

Një UID, (ne UNIX) i quajtur **superuser**, ka fuqi speciale dhe mund të shkel shume nga rregullat e mbrojtjes. Ne instalime të medha, vetëm administratori i sistemit e di fjalekalimin e nevojsphem për tu bere një superuser, por shume nga përdoruesit e zakonshem (vecanerisht studenta) kushtojne një përpjekje të konsiderueshme duke provuar për të gjetur ndonjë të metë të sistemit qe i lejon ata të superusera pa patur nevoje për fjalekalim.

Proçeset, Komunikimin Nderproçesorial dhe tëma të lidhura me keto do të studiohen ne kapitullin e dytë.

1.5.2. DEADLOCKS (SITUATA PA RRUGE DALJE)

Kur dy ose me shume sisteme bashkeveprojne, ata disa here mund të gjenden ne situata për të cilat nuk mund të ketë një rrugëdalje. Një situatë e tille quhet një deadlock.

Deadlocket mund të shpjegohen mire me një shembull të jetës reale, me të cilin çdokush është i familiarizuar; deadlock-u ne trafik. Shikoni figuren 1.13(a). Ketu 4 autobuse po afrohen ne një kryqezim. Pas ketyre katër autobuseve ka edhe autobuse të tjere por nuk jane të treguar ne figure. Me shume pak fat të keq, katër të paret mund të arrijne ne kryqezim njëkohesisht, duke na cuar ne situatën si ne figuren 1.13(b), ne të cilën nuk ka rrugëzgjidhje sepse asnjeri nga ato nuk mund të vazhdoje përrpara. Secili ka bllokuar një nga të tjeret. Gjithashtu nuk mund të shkojne as mbrapa ngaqe ka autobusa të tjere pas tyre. Nuk ka asnje rruge të lehtë daljeje.

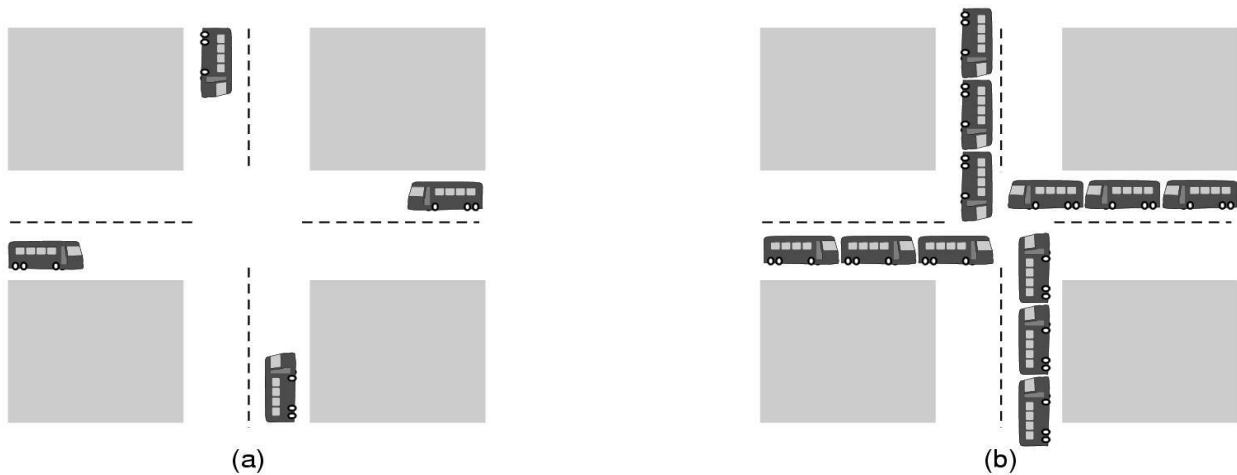


Fig.1.13.(a) Një rrezik deadlock (b)Një deadlock

Proçeset ne një kompjuter mund të ndodhen ne situata anologe ne të cilat nuk mund të bezne asgje ne progres. Për shembull, imaginoni një kompjuter me një tape drive dhe një CD-recorder.Tani imaginoni dy proçese ku qe të dy duan të bezne një CD-ROM nga të dhenat ne një tape. Proçesi 1 kerkon dhe i jepet ne përdorim tape drive. Ndersa proçesi 2 kerkon dhe i jepet ne përdorim CD-recorderi. Me pas proçesi 1 kerkon për CD-recorder dhe kjo kerkese pezullohet derisa të ta ktheje proçesi 2. Ne fund, proçesi 2 kerkon për tape drive dhe gjithashtu kjo kerkese pezullohet, sepse tashme ketë e ka proçesi 1. Ketu kemi një deadlock nga i cili nuk ka asnje dalje. Deadlocket dhe cfare mund të behet me to do ti shohim ne detaje ne kapitullin e tretë.

1.5.3. MENAXHIMI I MEMORIES

Çdo kompjuter ka një memorie kryesore të cilen e përdor për të mbajtur programet ekzekutuese. Ne një sistem operativ shume të thjeshtë, vetëm një program është ne memorie ne një kohe të caktuar.Për të ekzekutuar një program të dytë, programi i pare duhet të hiqet nga memoria dhe aty të vendoset i dyti.

Sisteme operative me të sofistikuara lejojne shume programe të jene ne memorie ne të njëjtën kohe.Qe ato të mos intérferojne me njëri-tjetrin (dhe me sistemin operativ), nevojitet disa lloj mekanizmi mbrojtës. Nderkohe qe ky mekanizem duhet të jetë ne hardware, ai kontrollohet nga sistemi operativ.

Kendveshtrimi i mesipërm ka lidhje me menaxhimin dhe mbrojtjen e memories kryesore. Një ceshtje ne lidhje me memorien e ndryshme por po aq e remdesishme është menaxhimi i hapesires se adresave të proçeseve. Normalisht, çdo proçes ka disa bashkesi adresash qe mund ti përdore, ne menyre tipike duke filluar nga 0 deri ne një maksimum. Ne rastin me të thjeshtë, vlera maksimale e hapesires se adresave të një proçesi është me e vogel se memoria kryesore. Ne ketë menyre, një proçes mund të mbushe hapesiren e adresave të tij dhe mund të mbahet i gjithi ne memorien kryesore.

Gjithesesi, ne shume kompjutera adresat jane 32 ose 64 biteshe, duke dhene një hapesire adresash përkësish prej 2^{32} apo 2^{64} bytes.Cfare do të ndodhte ne qoftë se një proçes ka me shume hapesire adresash se sa ka kompjuteri memorie kryesore dhe proçesi do ta përdor atë të gjithen? Ne kompjuterat e pare një proçes i tille ishte thjesht një pafat. Ne ditët e sotme egziston një teknike qe quhet memorie virtuale, ne të cilen sistemi operativ mban pjese të hapesires se adresave ne memorien kryesore dhe disa pjese ne disk **dhe i con keto pjese sa andej kendej** ndermjet tyre ashtu sic nevojiten. Ky funksion i rendesishem i sistemit operativ dhe funksione të tjera të lidhura me menaxhimin e memories do të shihen ne kapitullin e katërt.

1.5.4. INPUT/OUTPUT

Të gjithe kompjuterat kane pajisje fizike për marrjen e inputit dhe për nxjerrjen e outputit. Mbi të gjitha sa i mire do të ishte një kompjuter ne qoftë se përdoruesit nuk mund ti tregojne atij cfare të bez dhe të mos mund të marrin rezultatet pas përfundimit të punes se kerkuar. Ekzistojne shume lloje pajisjesh inputi dhe outputi, duke përfshire tastjeren, monitoret, printerat dhe keshtu me rradhe. Menaxhimi i ketyre pajisjeve i është lene sistemit operativ.

Për pasoje, çdo sistem operativ ka një nensistem I/O për menaxhimin e pajisjeve të tij I/O. Disa nga software-et I/O janë të pavarur nga pajisjet, qe do të thotë, veprojne me shume apo me të gjitha pajisjet me menyre shume të mire. Pjese të tjera të tij, si driverat e pajisjes, janë specifike për pajisje I/O të vecanta. Ne kapitullin e pestë do të shohim software-et I/O.

1.5.5. FILE-ET

Një koncept tjetër kryesor i suportuar virtualisht nga të gjithe sistemet operativ është sistemi file. Sic është thene me pare, një funksion kryesor i sistemit operativ është fshehja e karakteristikave të disqeve dhe pajisjeve të tjera I/O dhe ti prezantoje programuesit një model abstrakt të kendshem, të qartë të fileve të pavarura nga pajisja. Ne menyre të dukshme nevojiten thirrjet sistem për krijim e file-ve, levizjen e file-ve, leximin e file-ve dhe shkrimin e file-ve. Përpara se një fjale mund të lexohet, ajo duhet të vendoset ne disk dhe të hapet, dhe pasi si lexohet ajo duhet të mbyllët, keshtu thirrjet janë krijuar për të bere keto gjera.

Për sigurimin e një vendi për të mbajtur file-et, shumica e sistemeve operative kane konceptin e **direktive** si një menyre e grupimit të fileve sebashku. Një student, për shembull, mund të ketë një direktori për çdo kurs qe po ndjek (për programet e nevojitur për ato kurse), një tjetër direktori për **pjesen** e tij elektronike dhe akoma një tjetër direktori për faqen e tij World Wide Web. Thirrjet sistem janë të nevojshem për të krijuar dhe hequr direktoritë. Thirrjet gjithashtu janë krijuar për të vendosur një file ekzistues ne një direktori dhe për të hequr një file nga një direktori. Hyrjet ne direktori nuk të jene file apo mbase edhe direktori. Ky model gjithashtu ndikon ne rritjen e një hierarkie-sistemi file- sic është treguar ne figuren 1.14.

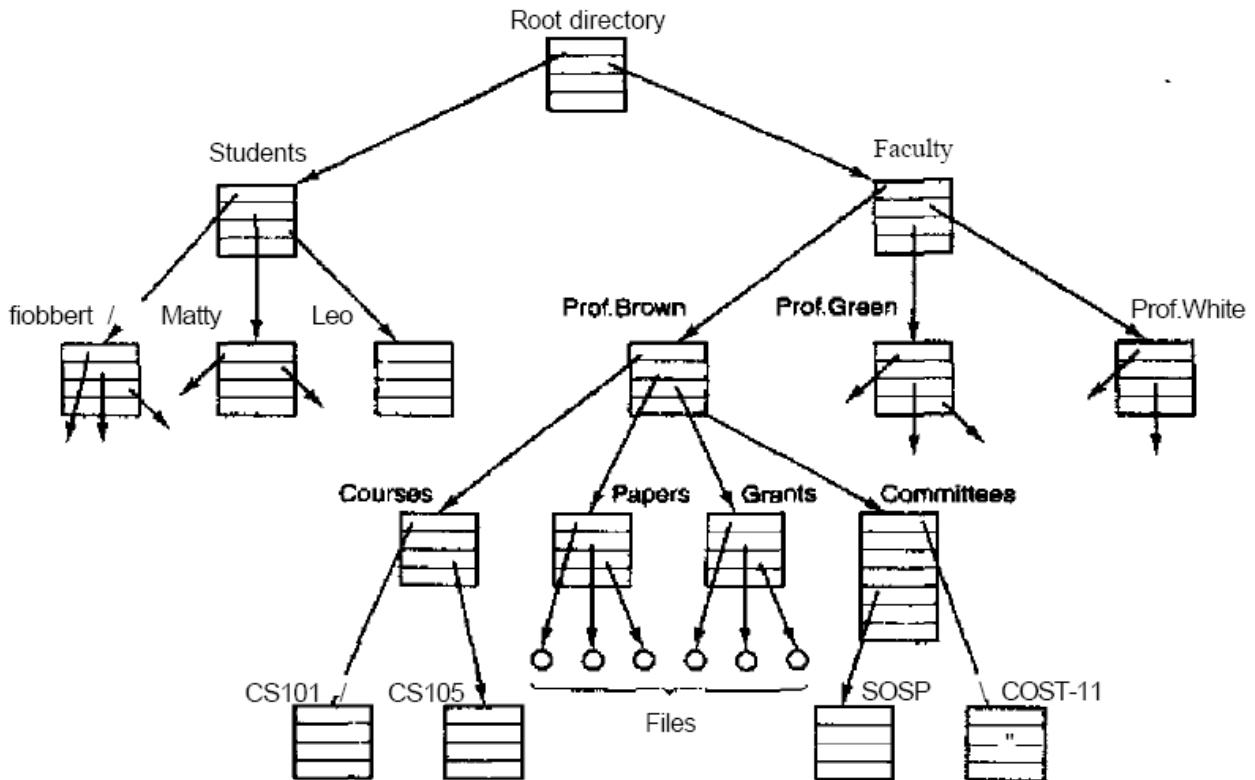


Fig. 1.14. Një sistem file për departamentin e një universiteti

Hierarkitë e proceseve dhe të file-eve se bashku jane të organizuar si peme, por ngashmeria ndalon ketu. Hierarkitë e proceseve zakonisht nuk jane shume të thella (me shume se tre nivele është e pazakonte) ndersa hierarkitë e file-eve jane ne përgjithesi katër, pese, apo edhe me shume nivele të thella. Hierarkitë e proceseve ne menyre tipike kane jetë të shkurtra, përgjithesisht të shumtën disa minuta, ndersa hierarkitë e direktive mund të ekzistojne përviteme rradhe. Pronesa dhe mbrojtja gjithashtu ndryshojne për proceset dhe file-t. Ne menyre tipike, vetëm një proces prind mund të kontrolloje apo edhe të aksesoje një proces femije, por pothuajse gjithmone ekzistojne mekanizma qe lejojne file-t dhe direktoritë të lexohen nga një grup me i gjere se vetëm të zotët e tyre.

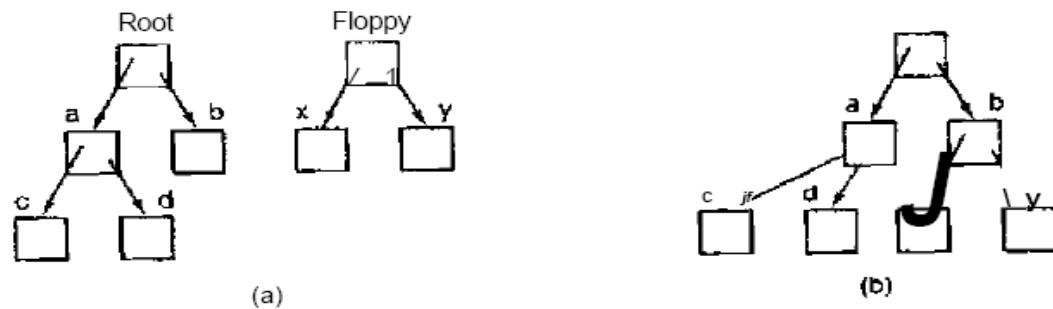
Çdo file brenda hierarkise se direktorive mund të specifikohet duke i dhene emrin e tij të rruges (path name) qe ne krye të hierarkise se direktorise, direktoria rrenjë. Path name të tille absolute përbehen nga lista e direktorive qe duhen kaluar nga direktoria rrenjë për të arritur ne file, me slashe qe ndajne komponentët. Ne Fig 1.14. rruga për file-in *CS101* është */Faculty/Prof.Brown/Courses/CS101*. Slashi i pare tregon qe rruga është absolute, qe është duke filluar nga direktoria rrenjë. Ne MS-DOS dhe Windows karakteri backslash (\) përdoret si ndares ne vend të karakterit slash (/), keshtu qe rruga e file-it e dhene me sipër do të shkruhej si; *\Faculty\Prof.Brown\Courses\CS101*. Përgjatë ketij libri ne përgjithesisht do të përdorim marreveshjen UNIX për pathet.

Ne çdo cast, secili proces ka një direktori punimi aktuale, ne të cilën kerkohen path names qe nuk fillojne me slash. Për shembull, ne fig.1.14, ne qoftë se

/Faculty/Prof.Brown ishin direktoritë e punimit, përdorimi i rruges me emer /Courses/CS101 do të na jeptë të njëjtin file si rruga absolute e dhene me sipër. Proçeset mund të ndryshojne direktorine e tyre të punimit nepërmjet një thirrjeje sistem që specifikon direktorine e re të punimit.

Përpara se një file të lexohet apo të shkruhet, ajo duhet të hapet, ne të cilën kohe kontrollohen autorizimet. Ne qoftë se aksesimi është i lejueshem, sistemi kthen një integer të vogel të quajtur pëershkruesi i file-it (file descriptor) që ta përdore ne opërimet pasuese. Ne qoftë se aksesimi është i ndaluar kthehet një kod error.

Një koncept tjetër i rendesisëhem ne UNIX është sistemi file mount. Pothuajse të gjithe personal computers kane një ose shumë drive floppy disk ne të cilat mund të vendosen dhe hiqen floppy disk. Për sigurim e një menyre me elegantë që të merret me media të levizshme (duke përfshire ketu CD-ROMs), UNIX lejon sistemin file ne një floppy disk të lidhet ne pemen kryesore. Merrni ne konsideratë situatën ne figuren 1.15.(a). Para thirrjes mount, sistemi file rrenjë, ne hard disk dhe një sistem i dytë file, ne një floppy disk, janë të ndara dhe të palidhura.



**Fig.1.15. (a).para mounting, file-ët ne drive 0 Jane të paaksesueshme
(b) pas mounting, ata Jane pjese e hierarkise se fileve**

Sidoqoftë, sistemi file ne floppy nuk mund të përdoret, sepse nuk ka asnjë menyre për të specifikuar path names ne të, UNIX nuk lejon path names të jene të parashtësuara (prefixed) nga një numer apo emer drive-i; ajo do të ishte saktësisht lloji i varesise se pajisjes qe sistemi operativ duhet të eliminoje. Ne vend të kesaj, thirrja sistem mount lejon sistemin file ne floppy të lidhet me sistemin file rrenjë kudo qe sistemi e do atë të jetë. Ne fig.1.15(b) sistemi file ne floppy është kaluar ne direktorine b, duke lejuar keshtu aksesimin /b/x dhe /b/y. Ne qoftë se direktoria do të kishte ndonjë file ata do të ishin të paaksesueshem underkohe qe floppy është kaluar, përderisa /b do të referoje direktorine rrenjë të floppy. (Të qenurit jo të aftë për të aksesuar keto file nuk është një problem aq serioz sa mund të duket ne fillim; sistemet file janë pothuajse gjithmone të kaluara ne direktori boshe). Ne qoftë se një sistem përmban shume hard disqe, ata mund të kalohen shume mire të gjithe ne një peme të vetme.

Një tjetër koncept i rendesisëhem ne UNIX është file-i special. File-t speciale janë krijuar ne menyre qe të befne pajisjet I/O të duken si file. Ne ketë menyre, ata mund të lexohen dhe të shkruhen duke përdorur të njëjtat thirrje sistem qe jane përdorur për leximin dhe shkrimin e file-ve. Ekzistojne dy lloje të file-ve speciale: file speciale block dhe file speciale character. Filet speciale block janë përdorur ne pajisjet model qe

përbehen nga një bashkesi e rastesishme blloqesh të adresueshem, si disjet. Me hapjen e një file special block dhe me leximin, say, block 4, një program ne menyre direktë mund të aksesoje bllokun e katërt ne device, pa vemandjen e struktura të sistemit file qe përbahet ne të. Ne menyre të ngjashme, filet special karakter jane të përdorur ne printerat model, modemat model dhe pajisje të tjera model qe pranojnë apo nxjerrin një rrjedhe karakteresh. Me marreveshje, filet speciale mbahen ne direktorine `/dev`. Për shembull, `/dev/lp` mund të jetë line printer.

Dukuria e fundit për të cilën do të diskutojmë ne ketë përbledhje është dicta qe lidhet me proceset dhe me filet gjithashtu: pipes. Një pipe është një renditje e pseudofile qe mund të përdoren për të lidhur dy procese sic tregohet ne figuren 1.16. Ne qoftë se proceset A dhe B duan të bisedojne duke përdorur një pipe, ata duhet ta vendosin atë ne avancim. Kur procesi A do të dergoje të dhena procesit B, ai shkruan ne pipe sikur ai të ishte një file output. Prosesi B mund të lexoje të dhenat duke lexuar nga pipe si të ishte një file input. Keshtu, komunikimi ndermjet proceseve ne UNIX ngjan shume me leximin dhe shkrimin e file të zakonshem. Menyra e vetme qe një proces mund të zbulojë se file-i output të cilit po i shkruan nuk është ne të vertetë një file, por një pipe është duke bere një thirrje speciale sistem. Sistemet file Jane shume të rendesishem. Do të kemi me shume për të thene rreth ketyre ne kapitullin 6 dhe gjithashtu ne kapitujt 10 dhe 11.

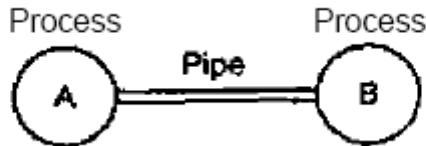


Fig.1.16.Dy procese të lidhur nga një pipe.

1.5.6. SIGURIA

Kompjuterat përbajne sasi të medha informacioni qe përoruesit shpesh duan ta mbajne konfidenciale. Ky informacion mund të përfshije postën elektronike, planet e biznesit, kthimet e taksave dhe shume me tepër. I lihet sistemit operativ të menaxhoje sigurimin e sistemit ne menyre qe filet, për shembull Jane të aksesueshme vetëm nga përdorues të autorizuar.

Si një shembull i thjeshtë, sa për të pasur një ide se si mund të funksionoje siguria, konsideroni UNIX. File-t ne UNIX Jane të mbrojtura duke u caktuar seciles një kod mbrojtës binar 9-bitesh. Kodi mbrojtës përbahet nga tre fusha 3-biteshe, një për pronarin, një për pjesetare të tjere të grüpuit të pronarit (përdoruesit Jane të ndare ne grupe nga administratori i sistemit) dhe një tjetër për çdokend tjetër. Çdo fushe ka një bit për akses leximi, një bit për akses shkrimi dhe një bit për akses ekzekutimi. Keto tre bite njihen si bitet rwx. Për shembull, kodi mbrojtës `rwxr-x--x` do të thotë qe pronari mund të lexoje, shkruaje apo të ekzekutoje file-in, pjesetare të tjere të grüpuit mund të lexoje apo të ekzekutojne (por jo të shkruajne), dhe çdokush tjetër mund të ekzekutoje (por jo të lexoje dhe të shkruaje) file-in. Për një direktori, x tregon autorizim të kerkimit. Një ‘dash’ do të thotë qe autorizimi korresponduesh mungon.

Vec mbrojtjes se file-s, ka edhe shume probleme të tjera sigurie. Mbrojtja e sistemit nga të nderhyrjet e padeshiruara, human dhe johuman (për shembull, viruset) është një prej tyre. Do të shohim disa prej tyre ne kapitullin e nentë.

1.5.7. SHELL-I

Sistemi operativ është kodi qe kryen thirrjet sistem. Editoret, komplilatoret, assembluesit, linkeruesit dhe interpretuesit e komandave kategorisht nuk jane pjese të sistemit operativ, edhe pse ato jane shume të rendesishem dhe të përdorshem. Për të mos ngatëruar gjerat, ne ketë seksion do të shohim shkurtimisht interpretuesin e komandave, të quajtur shell. Edhe pse nuk është pjese e sistemit operativ, ka përdorim të gjere ne shume dukuri të sistemit operativ dhe keshtu sherbën si një shembull i mire se si mund të përdoren thirrjet sistem. Gjithashtu është nderfaqja primare ndermjet përdoruesit dhe sistemit operativ, vec ne se përdoruesi është duke përdorur një nderfaqe grafike. Ekzistojne shume shell-e, duke përfshire sh, csh, ksh dhe bash. Të gjitha keto suportojne funksionalitetin e përshkruar me poshtë, qe burojne nga shelli origjinal (sh).

Kur një përdorues **logs in**, starton një shell. Shellli ka terminalin si input dhe output standart. Ai fillon me shtypjen e prompt, një karakter si ai i shenjës se dollarit, qe tregon përdoruesit qe shelli është duke pritur përmes tij. Kur procesi femije përfundon, shelli shtyp prompt përseri dhe mundohet të lexoje rreshtat pasardhes të inputit

Përdoruesi mund të specifikoje qe outputi standart të ridrejtohet tek një file, për shembull,

datë >file

Ne menyre të ngjashme, inputi standart mund të jetë i ridrejtuar, si ne

sort <file1 >file2

qe i kerkon programit renditës me inputin e marre nga file1 dhe outputin e derguar ne file2.

Outputi i një programi mund të përdoret si një input përmes një program tjetër duke i lidhur ata me një pipe. Keshtu

```
cat file1 file2 file3 | sort >/dev/lp
```

kerkon qe programi *cat* të lidhe tre file dhe të dergoje outputin ne *sort* për të rregulluar të gjithe rreshtat ne rend alfabetik. Outputi i *sort* është ridrejtuar ne file */dev/lp*, ne menyre tipike nga printeri.

Ne qoftë se një përdorues vendos një ampërsand pas komandes, shelli nuk pret për kompletimin e tij. Ne vend të kesaj ai menjehere i jep një prompt. Për pasoje,

```
cat file1 file2 file3 | sort >/dev/lp &
```

fillon renditja si një pune background, duke lejuar përdoruesin të vazhdoje normalisht të punoje nderkohe qe renditja vazhdon. Shelli ka një numer dukurish të tjera interesante, për të cilat nuk kemi kohe për ti diskutuar ketu. Shumica e librave ne UNIX diskutojne për shell (për shembull, Kernighan and Pike, 1984; Kochan and Wood 1990; Medinets, 1999; Newham and Rosenblatt, 1998: dhe Robbins, 1999).

1.5.8. RICIKLIMI I KONCEPTEVE

Shkencat kompjuterike, si shume fusha të tjera jane të udhehequra shume nga teknologjia. Arsyja qe romanet antike kishin mungesa makinash nuk është se atyre u pelqentë të ecnin ne kembe kaq shume. Arsyja ishte se ato nuk dinin se si ti ndertonin makinat. Personal computers ekzistojne jo sepse shume njerez kishin deshira të mbajtura përbrenda gjatë për të pasur një kompjuter të tyrin, por sepse tani është e mundur të prodhohen shume lire. Shpesh ne harrojme se sa shume ndikon teknologjia ne veshtrimin tone të sistemeve dhe është me vlore të reflektojme rreth kesaj, here pas here.

Ne vecanti, shpesh ndodh qe një ndryshim ne teknologji bie një ide të vjetëruar dhe zhduket shpejt. Gjithesesi, një tjetër ndryshim ne teknologji mund ta riktheje atë ide përseni. Kjo është e vertetë vecanerisht kur ndyshimi ka të beje me performancen relative të pjeseve të ndryshme të sistemit. Për shembull, kur CPU-të u bëne me të shpejta se memoriet, chace-të u bëne me të rendesishme për pëershpejtimin e memories të "ngadaltë". Ne qoftë se ndonjë ditë memoriet nepërmjet një teknologjie do të beheshin me të shpejta se CPU-të, chace-të do të zhdukeshin. Dhe ne qoftë se një teknologji CPU i bën përseni me të shpejta se memoriet, chace-të do të rishfaqeshin. Ne biologji, zhdukja është e përhershme, por ne shkencat kompjuterike, disa here është vetëm për disa vjet.

Si një pasoje e kesaj përkohshmerie, ne ketë liber here pas here do të shohim konceptet "e vjetëruara", sic jane, idetë qe nuk jane të pajtueshme me teknologjine aktuale. Sidoqoftë, ndryshimet ne teknologji mund të risillnin disa nga të keshtu quajturit koncepte të vjetëruara. Për ketë arsy, është e rendesishme të kuftojme pse një koncept është i vjetëruar dhe cfare ndryshimesh ne mjedis mund ta risillnin përseni.

Për të qartësuar me shume ketë pike, le të konsiderojme disa shembuj. Kompjuterat e hershem kishin bashkesi instruksionesh hardwired. Instruksionet ekzekutoheshin direkt nga hardware-i dhe nuk mund të ndryshoheshin. Me pas vjen mikrogramimi, ne të cilin një interpretues qe ndodhet ne të i beri instruksionet ne software. Ekzekutimi hardwired u be i vjetëruar. Me pas u krijuan kompjuterat RISC dhe keshtu mikroprogrami (ekzekutimi i intérpretuar) u be i vjetëruar sepse ekzekutimi direkt ishte me i shpejtë. Tani jemi duke pare rishfaqjen e intérpretimit ne formen apletëve Java qe dergohen nepërmjet internetit dhe të interpretuara ne ardhje. Shpejtësia e ekzekutimit

nuk është gjithemone vendimtare sepse vonesat e rrjetit jane shume të medha sa tentojnë të dominojne. Por edhe kjo, ndonjë ditë mund të ndyshoje.

Sisteme të hershme operative i shpërndanin file-t ne disk thjesht duke i vendosur ato ne sektore të afert, njëri pas tjetrit. Megjithese kjo skeme është e lehtë për tu implementuar, nuk është fleksibel sepse me rritjen e një file nuk ka vend mjaftueshem për ta mbajtur atë. Keshtu koncepti i fileve të shpërndare afer u be një koncept i vjetruar. Akoma kemi përreth CD-ROM. Ketu problemi i rritjes se file-s nuk ekziston. Papritura, gjithe thjeshtësia e shpërndarjes se filet afer u pa si një ide e madhe dhe tani sistemet file CD-ROM bazohen ne të.

Si ide përfundimtare, konsideroni lidhjen dinamike. Sistemi MULTICS ishte projektuar për të vepruar ditë e natë pa ndaluar asnjeherë. Për të sistemuar viruset ne software ishte e nevojesheme të kishte një menyre për të zevendesuar procedurat library, nderkohe qe ato ishin duke u përdorur. Koncepti i lidhjes dinamike u krijuua me ketë qellim. Pas vdekjes se MULTICS, koncepti u harrua për pak. Sidoqoftë, ai u rizbulua kur sistemet moderne operative kishin nevoje për një menyre për të lejuar shume programe për të ndare të njëjtat procedura library pa pasur kopjet e tyre private (sepse libraritë grafike ishin rritur shume). Shumica e sistemeve tanë suportojne edhe njehere disa forma të lidhjes dinamike. Lista vazhdon, por keto shembuj mund të jepin shume mire idene; një ide qe sot është e vjetruar, ne ser mund të jetë “ylli i mbremjes”.

Teknologjia nuk është i vetmi faktor qe udheheq sistemet dhe software. Gjithashtu ekonomia luan një rol të rendesishem. Ne vitet 1960 dhe 1970, me shume terminale ishin terminale mekanike printimi ose CTR të orientuara nga 25 x 80 nga karakteret, se terminale grafike bitmap. Kjo zgjedhje nuk ishte ceshtje e teknologjise. Terminalet grafike bit-map ishin ne përdorim përpara 1960, por është thjesht se ata kushtojne dhjetra mijera dollare secila. Vetëm kur Cmimi u ul ne menyre të hatashme njerezit (vec ushtarakeve) mund të mendonin për një terminal për një përdorues individual.

1.6. THIRRJET SISTEM

Nderfaqa ndermjet sistemit operativ dhe programeve user përcaktohet nga një bashkesi thirrjesh sistem qe sigurohen nga sistemi operativ. Për të kuptuar se cfare bën sistemi operativ ne të vertetë, duhet ta ekzaminojme ketë nderfaqe me afer. Thirrjet sistem të përdorshme ne sistem ndyshojne nga një sistem operativ ne një tjetër sistem operativ (edhe pse konceptet ne to tentojnë të jene të ngjashem)

Keshtu jemi të detyruar të bejme një zgjedhje mes (1) përgjithesimeve të paqarta (“sisteme operative kane thirrjet sistem për të lexuar file”) dhe (2) një sistemi specifik (“UNIX ka një thirrje sistem me tre parametra: Një për të specifikuar file-in, një për të treguar se ku duhet të vendosen të dhenat dhe një për të treguar sa byte të lexohen”)

Ne kemi zgjedhur rrugen e dytë. Kjo menyre ka me shume pune, por na jep me shume depërtim ne atë se cfare bën sistemi operativ ne të vertetë. Megjithese ky diskutim i referohet ne menyre të vecantë POSIX (International Standard 9945-1), por tanë edhe UNIX, System V, BSD, Linux, MINIX,etj, shume sisteme moderne operative të tjere kane thirrje sistem për përformimin e të njëjtëve funksione, edhe pse detajet ndryshojne. Përderisa mekaniket ne nxjerrjen e thirrjeve sistem Jane të varur shume nga

makina dhe shpesh duhet të shprehen ne kode assembly, sigurohet një procedure library për të bere të mundur berjen e thirrjeve sistem nga programe ne C dhe shpesh nga programe ne gjuhe të tjera gjithashtu.

Është e dobishme të mbajme ne mendje keto të me poshtmet. Çdo kompjuter me një CPU të vetme mund të ekzekutoje vetëm një instrukcion ne një kohe. Ne qoftë se një proces po ekzekuton një program user ne mode user dhe ka nevoje për një sherbit të sistemit, si leximi i të dhenave nga file, atij i duhet të ekzekutoje një instrukcion trap ose thirrje sistem për të transferuar kontrollin sistemit operativ. Me pas sistemi operativ gjen se cfare kerkon procesi thirres duke shqyrtau parametrat. Me pas ai kryen thirrjen sistem dhe i kthen kontrollin instrukzionit pas atij qe shkaktoi thirrjen sistem. Ne një kuptim, të berit e një thirrjeje sistem është si të berit e një lloji të vecantë thirrjeje procedure, vetëm thirrjet sistem hyjne ne kernel dhe thirrjet procedure jo.

Për të qartësuar mekanizmin e thirrjes sistem, le ti hedhim një sy të shpejtë thirrjes lexim të sistemit. Sic përmendem me sipër, ai ka tre parametra; i pari për të specifikuar file-in, i dyti për të shenuar bufferin dhe i treti për të dhene numrin e byteve qe duhen lexuar. Pothuajse si të gjithe thirrjet sistem, kerkohet nga programe ne C duke thirrur një procedure library me të njëjtin emer si ai i thirrjes sistem: *read*. Një thirrje nga një program ne gjuhen C mund të ngjaje me dicka të tille:

```
count = read(fd, buffer, nbytes);
```

Thirrja sistem (dhe procedura library) kthejne numrin e byteve aktualisht të lexuare ne *count*. Kjo vlore normalisht është sa *nbytes*, por mund të jetë me i vogel, ne qoftë se, për shembull, fundi i file-s është takuar befasisht gjatë leximit.

Ne qoftë se thirrja sistem nuk mund të kryhet, ose për shkak të një parametri të gabuar apo një errori të diskut, *count* vendoset -1, dhe numri error vendoset ne një variabel global, *errno*. programet duhet gjithmone të kontrollojnë rezultatet e thirrjes sistem për të pare ne se ka patur ndonjë error.

Thirrjet sistem kryhen ne një seri hapash. Për të qartësuar ketë koncept, le të ekzaminojme thirrjen lexim të pare me sipër. Gjatë përgatitjes për thirrjen e procedures library *read*, qe aktualisht bën thirrjen lexim të sistemit, programi thirres fillimisht fut parametrit ne stack, sic tregohet ne hapat 1-3 ne figuren 1.17. Kompilatoret C dhe C++ fusin parametrit ne stack me renditje mbrapër shembullt (reverse) për arsyen historike (duke iu dashur të bejne përametrin e pare ne *print*, stringa format, shfaqet ne krye të stackout). Parametrit e pare dhe të tretë thirren me vlera, por parametri i dytë me reference, qe do të thotë, qe kalohet adresa e bufferit (e përcaktuar me &), jo përbajtja e bufferit. Me pas vjen thirrja aktuale ne proceduren library (hapi 4). Ky instrukcion është instrukzioni i thirrjes normale procedure i përdorur për të kryer thirrjen e të gjithe procedurave.

Procedura library, me shume mundesi e shkruajtën ne gjuhen assembly, ne menyre tipike vendos numrin e thirrjes sistem ne një vend ku sistemi operativ po e pret atë, si ne një regjistër (hapi 5). Me pas ekzekuton një instrukcion TRAP për të ndryshuar nga mode user ne atë kernel dhe fillon ekzekutimin ne një adresë të caktuar ne kernel (hapi 6). Kodi kernel qe fillon kontrollon numrin e thirrjes sistem dhe me pas e dergon ne manovruesin e thirrjeve korrigjim të sistemit, zakonisht nepërmjet një tabele pointerash drejt manovruesit të thirrjeve sistem të indeksuara ne numrin e thirrjes sistem (hapi 7). Ne

ketë pike manovruesi i thirrjeve sistem fillon se vepruari (hapi 8). Sapo manovruesi i thirrjes sistem përfundon punen e tij, kontrolli mund të kthehet ne hapesiren e përdoruesit të procedures library tek instruksioni pasardhes i instruksionit TRAP (hapi 9). Kjo procedure me pas kthehet ne programin user ne menyre të zakonshme sic kthehen thirrjet e procedures (hapi 10).

Për të mbaruar punen, programi user duhet të pastroje stack-un, sic bën pas çdo thirrjeje procedure (hapi 11). Duke supozuar se stack është ne rrenie, si ndodh shpesh, kodi i kompluar inkrementon pointerin e stack-ut mjaftueshem për të hequr parametrat e futur para thirrjes *read*. Programi tani është i lire të beje cfare të doje.

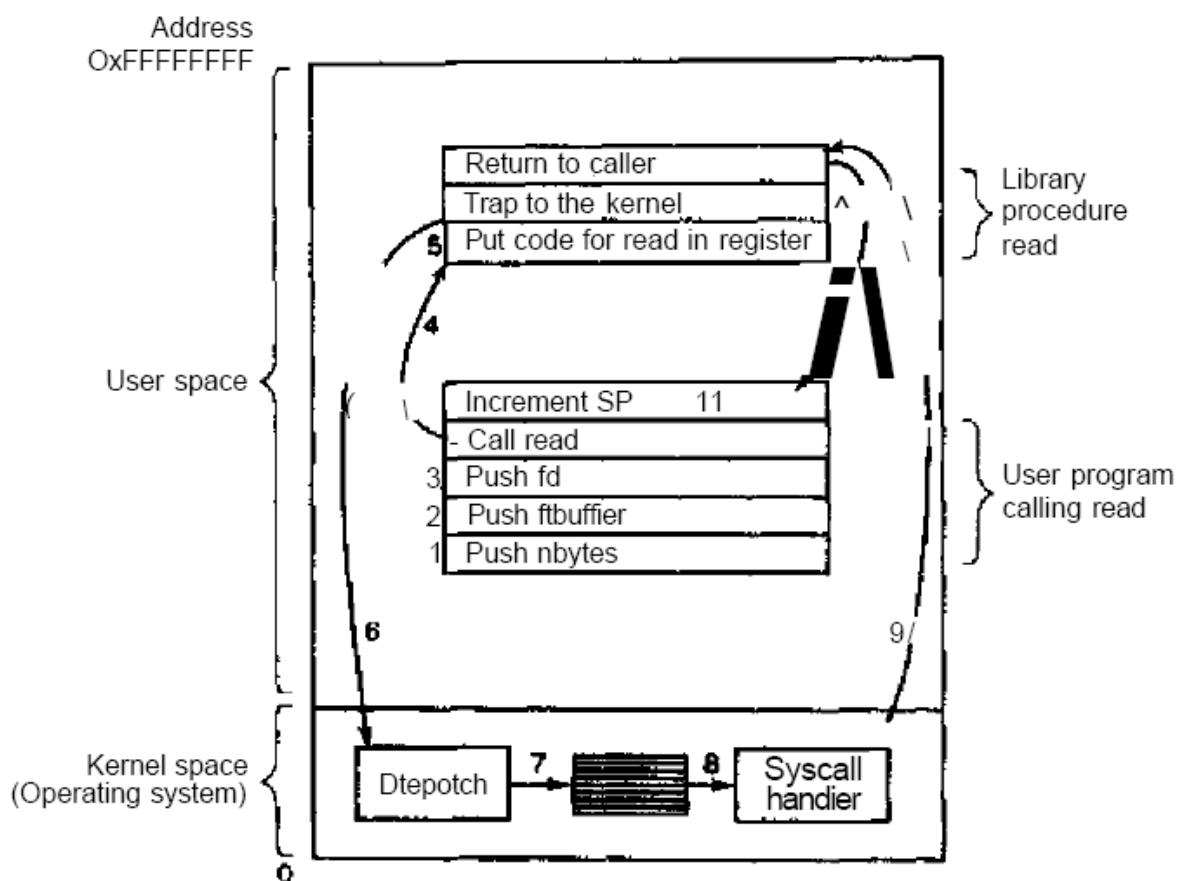


Fig.1.17. 11 hapat për të bere një thirrje sistem *read* (*id, buffer, nbytes*)

Ne hapin 9 me sipër thame “mund të kthehet ne hapesire e përdoruesit të procedures library...” për arsyet mire. Thirrja sistem mund të bllokoje thirresin, duke e ndaluar atë nga numerimi. Për shembull, ne qoftë se po mundohet të lexoje nga tastjera dhe asgje nuk është shtypur akoma, thirresi duhet të bllokohet. Ne ketë rast, sistemi operativ do të shohe ne se mund të ekzekutohet ndonjë proces tjetër. Me vone, kur inputi i deshiruar është

është i gatshem, ky proces do të marre vemandjen e sistemit dhe do të ndodhin hapat 9-11.

Ne paragrafet e meposhtëm, do të shqyrtojme disa nga thirrjet sistem POSIX me përdorim me të gjere ose ne menyre me specifike, procedurat library qe bejne keto thirrje sistem. POSIX ka rreth 100 thirrje procedura. Disa nga me të rendesishmet jane të listuara ne figuren 1.18, të përpunuara me marreveshje ne katër kategori. Ne tekst do të shqyrtojme shkurtimiqt çdo thirrje për të pare se cfare bën ajo. Ne një shtirje të madhe, sherbimet e ofruara nga keto thirrje përcakojne shumicen e atyre cfare duhet të beje sistemi operativ, përderisa menaxhimi i burimeve ne personal computers është minimal (të paktën i krahasuar me makina të medha me shume përdorues). Sherbimet përfshijne gjera si krijimin dhe përfundimin e proceseve, krijimin, fshirjen, leximin dhe shkrimin e fileve, menaxhimin e direktorive, kryerjen e inputit dhe outputit.

Process management

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information

Directory and file system management

Call	Description
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, name2, pointing to name1
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Fig 1.18. Disa nga thirrjet sistem kryesore të POSIX. Kodi i kthimit sështë -1 ne qoftë se ndodh ndonjë gabim. Kodet e kthimin janë si ne vijim: pid është një proces id, fd është një deskriptor file, n është një numer bytesh, position është një offset ne file, dhe seconds është një kohe e kaluar. Parametrat shpjegohen ne tekst.

Është e rendesishme të theksojme se paraqitje e thirrjeve procedura POSIX ne thirrjet sistem nuk është një ne një. Standartet POSIX specifikojnë një numer procedurash që një sistem i përshtatshem duhet të siguroje, por nuk specifikon ne se ato janë thirrje sistemi, thirrje library apo dicka tjeter. Ne qoftë se një procedure mund të kryhet pa kerkuar një thirrje sistem (pa trapping ne kernel), ajo pëgjithesisht do të jetë bere ne hapesireni user për arsyë të performancës. Sidoqoftë, shumica e procedurave POSIX kerkojnë thirrjet sistem, zakonisht me një grafike (mapping) procedure direktë ne një thirrje sistem. Ne pak raste, vecanerisht ku disa procedura të nevojshme janë vetëm variante të një tjetri, një thirrje sistem manovron me shume se një thirrje library.

1.6.1. THIRRJET SISTEM PËR MENAXHIM E PROCESIT

Grupi i pare i thirrjeve ne fiuren 1.18. merret me menaxhimin e proceseve. Fork është është ne menyre tipike shume mire për të filluar diskutimin. Fork është menyra e vetme për krijim e një procesi të ri ne UNIX. Ai krijon një kopje identike me procesin original, duke përfshire të gjithe deskriptoret e fileve, regjistrat-çdo gje. Pas forkut, procesi original dhe kopja e tij (prindi dhe femija) vazhdojnë ne rruge të ndara. Të gjithe variablat kane vlera identike ne kohen e forkut, por përderisa të dhenat e prindit janë kopjuar për të krijuar femijen, ndryshimet pasuese ne njërin nuk kane efekt ne tjetrin. (Teksti program, që është i pandryshueshem ndahet mes prindit dhe femijes). Thirrja fork kthen një vlere, e cila është 0 tek femija dhe e barabartë me identifikuesin e procesit femije ose **PID** tek prindi. Duke përdorur PID-in e kthyer, dy proceset mund të shohin se kush është procesi prind dhe kush është procesi femije.

Ne shumicen e rasteve, pas një fork-u, femija do të ketë nevoje për të ekzekutuar një kod të ndyshem nga prindi. Konsideroni rastin e shell-it. Ai lexon një komande nga terminali, krijon një proces femije, pret që femija të ekzekutoje komanden dhe me pas lexon komanden pasardhese kur përfundon femija. Për të pritur që të mbaroje femija, prindi ekzekuton një thirrje sistem waitpid, e cila thjesht pret derisa të përfundoje femija (çdo njëri femije ne se ekzistojne me shume se një). Waitpid mund të prese për një femije specifik ose për çdo njërin femije të vjetër duke vendosur parametrin e pare ne -1. Kur një waitpid përfundon, adresa e shenjuar nga parametri i dytë, *statloc*, do të evndoset ne

statusin exit të femijes (përfundim normal apo jonormal dhe vlera exit). Jane krijuar gjithashtu opsione të ndryshme, të specifikuar nga parametri i tretë.

Tani konsideroni se si është përdorur fork ne shell. Kur shtypet një komande, shell-i formon një proces të ri. Ky proces femije duhet të ekzekutoje komanden e përdoruesit. Ai e bën ketë duke përdorur thirrjen sistem execve, i cili shkakton qe imazhi i tij thelbesor i plotë të zevendesohet nga file i emeruar ne parametrin e pare. (Aktualisht, thirrja sistem vetë është exec, por disa proceduar library të ndryshme e therrasin atë me parametra të ndryshem dhe pak a shume me emra të ndryshem. Keto do ti trajtojme si thirrje sistem ketu). Ne figuren 1.19. tregohet një sheet shume i thjeshtësuar i cili ilustron përdorimin e fork-ut, waitpid, dhe gjithashtu execve.

Ne rastet me të përgjithshme, execve ka tre parametra; emrin e file-s qe do të ekzekutohet, një pointer ne grupin e argumentit dhe një pointer ne grupin e ambjentit. Keto do të përshkruhen shkurtimisht. Rutina të ndryshme library, duke përfshire *exec1*, *execv*, *execle*, dhe *execve* janë të krijuare për të lejuar parametrat të jene harruar apo specifikuar ne menyra të ndryshme. Përgjatë ketij libri ne do të përdorim emrin exec për të paraqitur thirrjen sistem të kerkuar nga të gjithe keto.

Le të konsiderojme rastin e komandes si

```
cp file1 file2
```

e përdorur për të kopjuar *file 1* ne *file 2*. Pas si shelli ka bere fork (degezim), procesi femije vendoset dhe ekzekuton file-n *cp* dhe i kalon asaj emrat e burimeve dhe file-t target.

```
#define TRUE 1

while (TRUE) {                                /* repeat forever */
    type_prompt();                            /* display prompt on the screen */
    read_command(command, parameters);        /* read input from terminal */

    if (fork() == 0) {                         /* fork off child process */
        /* Parent code. */
        waitpid(-t, &status, 0);              /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);       /* execute command */
    }
}
```

Fig 1.19.Një shell stripped-down.Përgjatë librit, TRUE pranohet të merret si 1.

Programi kryesor i *cp* (dhe programi kryesor i shumices se programeve të tjere ne C) përbajne deklarimin

```
main(argc, argv, envp)
```

ku *argc* është një numerimi i numrave të fjaleve ne rreshtin e komandes, duke përfshire edhe emrin e programit. Për shembull sipër *argc* është 3.

Parametri i dytë, *argv*, është një pointer ne një grup. Elementi *i* i ketij grupi është një pointer i stringes se i-të ne rreshtin komande. Ne shembullin tone, *argv[0]* do të shenontë ne stringen “*cp*” dhe *argv[1]* do të shenontë ne stringen “*file1*” dhe *argv[2]* do të shenontë ne stringen “*file2*”.

Parametri i tretë i *main*, *envp*, është një pointer i mjedisit, një grup stringash qe përbajne funksione të formes *name=value* të përdorur për të kaluar informacion ne një program si lloji i terminalit dhe emri i direktorise home. Ne figuren 1.19., nuk kalohet asnjë mjeshtë tek femija, keshtu parametri i tretë i *execve* është zero.

Ne qoftë se *exec* duket i komplikuar, mos u deshpëroni; ai është (nga ana kuqtimore) me kompleksi i të gjithe thirrjeve sistem POSIX. Të gjithe të tjerat jane me të thjeshta. Si shembull të një me thjeshtë, konsideroni *exit*, të cilën proçeset do ta përdorin kur kane mbaruar ekzekutimin. Ka një parametër, statusin *exit* (0 deri ne 255), i cili i kthehet prindit nepërmjet *stattoc* ne thirrjen sistem *waitpid*.

Proçeset ne UNIX kane memorien e tyre të ndare ne tre segmentë: **segmenti tekst** (kodi programit), **segmenti të dhene** (variablat) dhe **segmenti stack**. Segmenti i të dhene rritet ne drejtimin upward (lart) dhe stack rritet ne drejtimin downward (poshtë) sic tregohet ne figuren 1.20. Ndermjet tyre ndodhet një hapesire për hapesiren e papërdorur të adresave. Stack-u rritet automatikisht ne hapesire, sic mund të nevojitet, por zgjerimi i segmentit të të dheneve behet ne menyre eksplikite duke përdorur një thirrje sistem *brk*, qe specifikon adresen e re ku duhet të përfundoje segmenti i të dheneve. Kjo thirrje, sidoqoftë, nuk përcaktohet nga standarti POSIX meqene se programuesit jane të inkurajuar të përdorin proceduren library *malloc* për ruajtjet e alokuara dinamikisht, dhe implementimi ne të i *malloc* nuk ishte menduar të ishte subjekt i përshtatshem për standardizimin përderisa shume pak përdorues e përdorin atë ne menyre direktë.

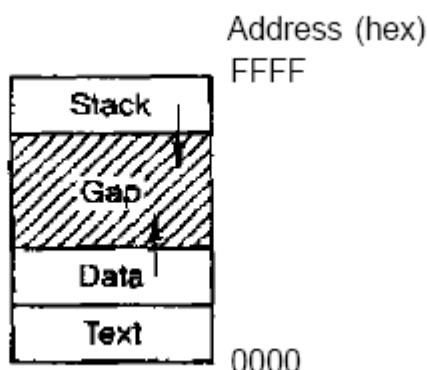


Fig.1.20. Proçeset kane tre segmentë: tekst, të dhena, she stack

1.6.2 THIRRJET SISTEM PËR MENAXHIMIN E FILE-VE

Shume thirrje sistem lidhen me sistemin file. Ne ketë pjese ne do të shohim thirrjet qe operojne ne file individuale; ne atë pasardhesin ne do të shqyrtojme ato qe përfshijne direktoritë ose sistemin file si një të plotë.

Për të lexuar apo për të shkruar një file, file fillimisht duhet të hapet duke përdorur open. Kjo thirrje specifikon emrin e file-s qe duhet të hapet, si një path name absolut ose relative me direktorine e punimit dhe një kod *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, qe do të thotë hape për të lexuar, shkruar apo të dyja se bashku. Për krijimin e një file të re përdoret *O_CREAT*. Descriptori file i kthyer me pas mund të përdoret për lexim dhe shkrim. Për ndryshe, file mund të mbyllët me close, qe bën descriptorin file të aftë për ripërdorim ne një hapje pasuese.

Thirrjet me shume të përdorura jane padyshim lexim dhe shkrim. Leximin e pame me pare. Shkrimi ka të njëjtat parametra.

Megjithese shumica e programeve lexojnë dhe shkruajnë file ne menyre të vazhdueshme, për disa aplikime programet duhet të jene të aftë për të aksesuar cilendo pjese të rastesishme të file-s. Me çdo file shoqerohet një pointer qe tregon pozicionin aktual ne file. Kur lexohet (shkruhet) vazhdimesht, normalisht ai shenon byte-in pasardhes qe do të lexohet (shkruhet). Thirja Iseek ndryshon vleren e pointerit të pozicionit, ne menyre qe thirrjet pasuese lexim apo shkrim mund të fillojne kudo ne file.

Iseek ka tre parametra: i pari është descriptori file për file-in, i dyti është pozicioni file-s dhe i treti tregon ne se pozicioni file-s është afer me fillimin e file-s, pozicionin aktual, ose fundin e file-s. Vlera e kthyer nga Iseek është pozicioni absolut ne file pas ndryshimit të pointerit.

Për secilen file, UNIX mban gjurmë të modes file (file normale, file speciale, direktori, dhe keshtu me rradhe), madhesine, kohen e modifikimit të fundit dhe informacion tjetër. Programet mund të kerkojne të shohin ketë informacion nepërmjet thirrjes sistem stat. Parametri i pare specifikon file-in qe do të inspektohet; i dyti është një pointer ne një strukture ku duhet të vendoset informacioni.

1.6.3 THIRRJET SISTEM PËR MENAXHIMIN E DIREKTORIVE

Ne ketë pjese do të shohim disa thirrje sistem qe kane kane të bejne me direktoritë ose sistemet file si një të plotë, me shume se thjesht një file specifik si ne paragrafin paraardhes. Dy thirrjet e para, kmdir dhe rmdir, përkatësisht krijojnë dhe heqin direktori boshe. Thirja tjetër është link. Qellimi i kesaj është të lejoje të njëjtën file të shfaqet me dy ose me shume emra, shpesh ne direktori të ndryshme. Një përdorim tipik është të lejuarit e disa pjesetareve të të njëjtit grup programimi të ndajne të njëjtën file, ku secili prej tyre të ketë file-in të shfaqur ne direktoritë e tyre, mundesisht me emra të ndryshem. Të ndarit e një file nuk është e njëjta me dhenien e një kopjeje private çdo pjestari të

grupit, sepse të paturit e një file të ndare do të thotë qe ndryshimet të bera nga cilido pjesetare i grupit jane menjehere të dukshme tek pjesetaret e tjere-ka vetëm një file. Kur behen kopje të një file, ndryshimet pasuese të bera ne një prej kopjeve nuk ndikon ne të tjerat.

Për të pare se si punon link, konsideroni situatën e figures 1.21(a). Këtu ka dy përdorues *ast* dhe *jim*, secili ka direktoritë e tija me disa file. Ne qoftë se *ast* tani ekzekuton një program qe përmban thirrjen sistem

```
link("/usr/jim/memo", "/usr/ast/notë");
```

file-i *memo* ne direktorine e *jim* tani është futur ne direktorine e *ast* me emrin *notë*. Pas kesaj */usr/jim/memo* dhe */usr/ast/notë* i referohen të njëjtës file. Ne se direktoritë e përdoruesve mbahen ne */usr*, */user*, */home*, apo diku tjetër është një vendim i thjeshtë i marre nga administratori i sistemit lokal.



Fig.1.21.(a) Dy direktori para linking */usr/jim/memo* ne direktorite e *ast*.(b)Të njëjtat direktori pas linking

Të kuptuarit se si funksionon link me shume mundesi do të na qartësoje se cfare bën ai. Çdo file ne UNIX ka një numer unik, qe e përcakton atë, numrin e tij *i*. Ky numri *i* është një index ne tabelen e i-nodes, një pér file, duke treguar se kush e ka file-in ku ndodhen bloqet disk të tij, dhe keshtu me rradhe. Një direktori është thjesht një file qe përmban një bashkesi (i-number, emrin ASCII) ciftësh. Ne versionet e para të UNIX, secila hyrje e direktorisë ishte 16 byte - 2 byte pér i-number dhe 14 byte pér emrin. Tani nevojitet një strukture me e komplikuar pér suportimin e emrave të gjatë të file-ve, por ne koncept një direktori është akoma një bashkesi (i-number, emrin ASCII) ciftësh. Ne figuren 1.21. *mail* ka i-number 16 dhe keshtu me rradhe. Ajo cfare bën link është thjesht krijimi i një hyrje direktorie të re me një emer (mundesisht të ri), duke përdorur i-number të files ekzistuese. Ne fig 1.21.(b), dy hyrje kane të njëjtin i-number (70) dhe keshtu i referohen të njëjtës file. Ne qoftë se me vone çdo njëra prej tyre fshihet, duke përdorur thirrjen sistem unlink, tjetra mbetet. Ne qoftë se fshihen të dyja, UNIX shikon qe nuk ekziston asnjë hyrje ne file (një fushe ne i-node mban gjurmë të numrit të hyrjeve direktori duke shenuar file-in), keshtu qe file-i fshihet nga disku.

Sic e kemi përmendur me pare, thirrja sistem mount lejon dy thirrje sistem të bashkohen ne një. Një situatë e zakonshme është të kemi file-n sistem rrenjë qe përfshin versionet binare (të ekzekutueshme) të komandave të zakonshme dhe të fileve të tjera të

përdorura gjeresisht, ne një hard disk. Me pas përdoruesi mund të vendose një floppy disk me file qe duhet të lexohen ne drive-in e floppy disk-ut.

Me ekzekutimin e thirrjes sistem mount, sistemi file i floppy disk mund të lidhet me sistemin file rrenjë, sic tregohet ne figuren 1.22. Një paraqitje tipike ne C për të përfurmuar mount është

```
mount("/dev/fd0", "/mnt", 0);
```

ku parametri i pare është emri i një file-i special block për drive 0, parametri i dytë është një vend ne peme ku duhet të kalohet (mounted) dhe parametri i tretë tregon ne se sistemi file duhet të caktohet lsim-shkrim apo vetëm lexim.



Fig.1.22.(a)sistemi file para mount.(b)Sistemi file pas mount.

Pas thirrjes mount, një file ne drive 0 mund të aksesohet thjesht duke përdorur rrugen e saj nga direktoria rrenjë ose nga direktoria e punimit, duke mos i përkitur drivet ne të cilin ndhodhet. Ne fakt, drive-t e dytë, tretë, dhe të katërt mund të kalohen kudo ne peme. Thirrja mount bën të mundur integrimin e medias të levizshme ne një hierarki file të integruar të vetme, duke mos qene nevoja e shqetësimit se ne cilen pajisje ndodhet file. Megjithese ky shembull përfshin floppy disks, hard disks apo pjese të hard disks (shpesh të quajtura **particione** ose **pajisje minore**) gjithashtu mund të kalohet ne ketë menyre. Kur një sistem file nuk nevojitet me, ajo mund të behet unmountëd nepërmjet thirrjes sistem unmount.

1.6.4 THIRRJET SISTEM TË PËRZIERA.

Gjithashtu ekziston një shumllojshmeri thirrjes së tjera sistem. Ketu do të shohim vetëm katër prej tyre. Thirrja chdir ndryshon direktorine aktuale të punimit. Pas thirrjes

```
chdir("/usr/ast/test");
```

një open ne file-in *xyz* do të hape */ttsr/ast/test/xyz*. Koncepti i një direktorit punimi eliminon nevojen e shtypjes gjatë gjithe kohes të path names absolute (të gjatë).

Ne UNIX çdo file ka një mode të përdorur për mbrojtje. Mode-a përfshin bitet read-write-execute për pronarin, grupin dhe të tjeret. Thirrja sistem chmod bën të mundur ndryshimin e mode-s se file-it. Për shembull, për të bere një file vetëm të lexueshme nga çdo njëri vec pronarit, do të ekzekutohej

```
chmod("file", 0644);
```

Thirrja sistem kill është menyre qe përdoruesit dhe proçeset user dergojne sinjale, ne qoftë se një proçes është preqatitur të kape një sinjal të vecantë, me pas kur mberrin, ekzekutohet një monovrues sinjali. Ne qoftë se proçesi nuk është i përgatitur për manovrimin e sinjalit, me pas ardhja e tij vret proçesin.

POSIX cakton disa procedura qe të merren me kohimin. Për shembull, koha thjesht kthen kohen aktuale ne sekonda, me 0 qe i korrespondon 1 Jan, ne mesnatë të 1970 (sapdita të ketë filluar dhe jo mbaruar). Ne kompjuterat ne fjale 32-bit, vlera maksimale qe time mund të ktheje është $2^{32}-1$ sekonda (duke supozuar qe përdoret një intërger pa shenjë). Kjo vlera korrespondon me pak me shume se 136 viti. Keshtu ne vitin 2106, sistemet UNIX 32-bit do të tërbohen, duke imituar problemin e famshem Y2K. Ne qoftë se ju aktualisht keni një sistem UNIX 32-bit, ju keshillohet ta ndryshoni atë për një 64-bit para vtitit 2106.

1.6.5. WINDOWS WIN32 API

Me pare jemi fokusuar kryesisht ne UNIX. Tani është koha të shohim shkurtimisht tek Windows. Windows dhe UNIX ndryshojne ne menyre thelbesore ne modelet përkatëse të programimit. Një program UNIX përbehet nga kodi qe bën një gje apo një tjetër, duke bere thirrje sistem për të pasur përformimin e sherbimeve të caktuara. Ne ndryshim nga kjo, një program Windows është normalisht i drejtuar nga ngjarja. Programi kryesor pret qe të ndodhe një ngjarje, me pas therret një procedure qe të monovroje atë. Ngjarje tipike jane goditja e butonave, levizja e mousit, shtypja e një butoni të mousit apo futja e një floppy disk-u. Me pas thirren manovruesit për të proçesuar ngjarjen, për freskimin e screen-it dhe freskimin e gjendjes se programit të brendshem. E gjithe kjo të con ne dicka me stil të ndryshem programimi nga ai UNIX, por meqene se fokusimi i ketij libri është ne funksionet dhe strukturat e sistemit operativ, keto modele të ndryshme programimi nuk do të na duhen shume.

Sigurisht, Windows gjithashtu ka thirrje sistemi. Me UNIX, pothuajse ka një marredhenie 1-me-1 ndermjet thirrjeve sistem (për shembull, lexim) dhe procedurave library (Për shembull, lexim) të përdorura për të kerkuar thirrjet sistem. Me fjale të tjera, për secilen thirrje sistem, ka afersisht një procedure library qe thirret për ta kerkuar atë, sic tregohet ne figuren 1.17. Për me tepër, POSIX ka vetëm rreth 100 thirrje procedura.

Me Windows, situata është ne menyre thelbesore e ndryshme. Si fillim, thirrjet library dhe thirrjet aktuale sistem Jane të ciftëzuara fort. Microsoft ka caktuar një bashkesi procedurash, të quajtura **Win32 API (Application Program Interface)** qe priten të përdoren nga programuesit për të marre sherbimet e sistemit operativ. Kjo nderfaqe është (pjesarisht) e suportuar ne të gjithe versionet e Windows qe ne Windows 95. Me ciftimin e nderfaqes nga thirrjet aktuale sistem, Microsoft ruan aftësine e ndryshimit ne kohe të thirrjeve aktuale të sistemit duke mos shfuqizuar programet ekzistuese. Ajo cfare aktualisht përbën Win32 është pak ambicioze meqene se Windows 2000 ka shume thirrje të reja qe me pare nuk ishin të disponueshme. Ne ketë pjese, Win32 nenkupton nderfaqen e suportuar nga të gjithe versionet e Windows.

Numri i thirrjeve Win32 API është jashtëzakonisht shume i madh, duke numeruar me mijera. Për me tepër, nderkohe qe shume prej tyre bejne kerkesen për thirrje sistem, një numer i dukshem cohet plotësisht ne hapesiren user. Si pasoje, me Windows është e mundur të shihet se cfare është një thirrje sistem (e përformuar nga kernel) dhe cfare është thjesht një thirrje library e hapesires user. Ne fakt, ajo cfare është thirrje sistem ne një version të Windows mund të behet ne hapesiren user ne një version tjetër dhe anasjelltas. Kur të diskutojme thirrjet sistem të Windows-it ne ketë liber, do të përdorim procedurat Win32 (aty ku është i duhur) meqene se Microsoft na garanton se keto do të jene të qendrueshme për shume kohe. Por është e rendesishme qe jo të gjitha prej tyre jane thirrje të verteta sistem (qe është, trap ne kernel)

Një tjetër komplikim është se ne UNIX, GUI (për shembull, X Windows dhe Motif) vepron plotësisht ne hapesiren user, keshtu të vetmet thirrje sistem të nevojitura për shkrimin ne një screen Jane write dhe shume pak shume pak të tjera të parendesishme. Sigurisht, ka një numer të madhe thirrjesh për X Windows dhe GUI, por ne një fare kuptimi, keto nuk Jane thirrje sistem.

Ne ndryshim, Win32 API ka një numer të madh thirrjesh për menaxhimin e dritareve, figurave gjeometrike, tekstit, font-it, scrollbars, dialog boxes, menutë dhe dukuri të tjera të GUI. Ne sipërfaqen qe nensistemi grafik vepron ne kernel (e vertetë për disa versione të Windows por jo për të gjitha), Jane thirrjet sistem, për ndryshe ata Jane thjesht thirrje library. A duhet ti trajtojme keto thirrje ne ketë liber apo jo? Meqene se ato nuk Jane ne të vertetë të lidhura me funksionet e një sistemi operativ, kemi vendosur të mos i trajtojme edhe pse ato mund të krijohen nga kerneli. Lexuesit e interesuar për Win32 API mund të konsultohen nga shume libra për ketë teme si për shembull, (Hart, 1997; Rector dhe Newcomer, 1997; dhe Simon, 1997).

Prezantimi i thirrjeve Win32 API ketu është jashtë temes, por ne jemi kufizuar ne keto thirrje qe afersisht korrespondojne me funksionalitetin e thirrjeve UNIX të listuara ne figuren 1.18. Keto Jane të listuara ne figuren 1.23.

Le të shohim shkurtimisht listën e figure 1.23. Create proces krijon një proces të ri. Kjo bën kombinimin e fork dhe execve ne sistemin UNIX. Ajo ka shume parametra qe specifikojne karakteristikat a proceseve të rinj qe Jane krijuar. Windows nuk ka një hierarki procesesh sic ka UNIX, keshtu qe nuk ekziston koncepti procesit femije dhe atij prind. Pas krijimit të një procesi krijuesi dhe i krijuari Jane të njëjtë. WaitForSingleObject përdoret për të pritur një ngjarje. Mund të pritet për shume ngjarje të mundeshme. Ne qoftë se parametri specifikon një proces, me pas thirresi pret qe procesi i specifikuar të dale, gje e cila behet duke përdorur ExitProcess.

Gjashtë thirrjet e tjera operojne ne file dhe Jane funksionalisht të ngjashem me homologet e tyre ne UNIX, edhe pse ndyshojne ne parametra dhe ne detaje. File-t mund të hapen, mbyllen, lexohen, shkruhen shume mire si ne UNIX. Thirrjet SetFilePointer dhe GetFileAttributës vendosin pozicionin e file-s dhe marrin disa veti të file-s.

Windows ka direktori dhe keto direktori Jane të krijuara nga CreateDirectory dhe RemoveDirectory, respektivisht. Kjo është gjithashtu një ide e direktorisë aktuale, e vendosur nga SetCurrentDirectory. Koha aktuale sigurohet duke përdorur GetLocalTime.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
Iseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Nderfaqa Win32 nuk ka marredhenie me file-t, sistemet file mountëd, sigurimin ose sinjalet, keshtu qe thirrjet korresponduese me ata të UNIX nuk ekzistojne. Sigurisht, Win32 ka një numer të madh thirrjesh qe nuk ekzistojne ne UNIX, vecanerisht për menaxhimin e GUI. Windows 2000 ka një sistem të nderlikuar sigurimi dhe gjithashtu suporton linkimet me file.

Është e rendesishme qe të theksojme dicka të fundit rrëth Win32. Win32 nuk është një uniform e tmerrshme apo0 një nderfaqe e qendrueshme. Fajtori kryesor për ketë është nevoja e të qenurit të krahasueshem me nderfaqen e meparshme 16-bit e përdorur ne Windows 3.x.

1.7. STRUKTURA E SISTEMEVE OPERATIVE

Tani qe kemi pare se si **duket** sistemi operativ nga jashtë (nderfaqa e programuesit), është koha të shohim nga brenda. Ne paragrafet e meposhtme, do të shqyrtojme pese struktura të ndryshme qe jane provuar për të marre ndonjë ide për spektrin e mundesive. Keto ne asnjë menyre nuk jane shtërues, por jadin një idetë të disa projektëve të provuara ne praktike. Përse projektimet jane sisteme njëbllokeshe (monolithic), sisteme me shtresa (layered), makina virtiale, exokernels dhe sistemet klient-server.

1.7.1 SISTEMET MONOLITIKE

Ne organizatat me të zakonshme të hershme, kjo arritje shume mire do të nentitullohej “The big mess (rremuje e madhe)”. Struktura është qe nuk ka asnje strukture. Sistemi operativ shkruhet si një bashkesi procedurash, ku secila prej tyre mund të therrase ndonjë tjetër sa here të jetë e nevojshme. Kur përdoret kjo teknike, çdo procedure ne sistem ka një nderfaqe të mire përcaktuar ne baze të parametrave dhe rezultateve, dhe çdonjëra është e lire të therrase ndonjë tjetër, ne qoftë se kjo e fundit siguron disa llogaritje qe i duhen të parit.

Për të ndertuar programin objekt aktual të sistemit operativ kur përdoret kjo arritje, fillimisht kompilohen të gjithe procedurat individuale ose file-t qe përbajne procedurat, dhe me pas i lidh ata se bashku ne një file objekt të vetëm duke përdorur linker-in e files. Ne terma të fshehjes se informacionit, thellesisht nuk ka asnje- çdo procedure është e dukshme tek çdo procedure tjetër (e kundervene me strukturen qe përmban module apo paketime, ne të cilin shume informacione fshihen brenda moduleve dhe vetëm pikat hyrese zyrtarisht të përcaktuara mund të thirren nga jashtë modulit).

Sidoqoftë, edhe ne sistemet monolitike është e mundur të kemi të paktën një strukture të vogel. Sherbimet (thirrjet sistem) të siguruara nga sistemi operativ kerkohen nepërmjet vendosjes se parametrave ne një vend të mire përcaktuar (për shembull, ne stack) dhe me pas me ekzekutimin e një instruksioni trap. Ky instruksion ndryshon makinen nga mode user ne atë kernel dhe transferon kontrollin tek sistemi operativ, e treguar si hapi 6 ne figuren 1.17. Me pas sistemi operativ merr parametrat dhe përcakton se cila thirrje sistem do të kryhet. Pas kesaj, ajo indeksohet ne një tabele qe përmban ne një vend të caktuar (slot) k një pointer tek procedura qe kryen thirrjen sistem k (hapi 7 ne figuren 1.17).

Ky organizim na sugjeron një strukture baze për sistemin operativ:

1. Një program kryesor qe kerkon ne proceduren e sherbimit të kerkuar.
2. Një bashkesi procedurash sherbimi qe kryejne thirrjet sistem
3. Një bashkesi procedurash të dobishme qe ndihmojne procedurat sherbim.

Ne ketë model, për çdo thirrje sistem ka një procedure sherbimi qe kujdeset për të. Procedurat e dobishme bejne gjera të nevojshme nga procedura sherbimi të ndryshme, si marrja e të dhenave nga programet user. Kjo ndarje e procedurave ne tre shtresa tregohet ne figuren 1.24.

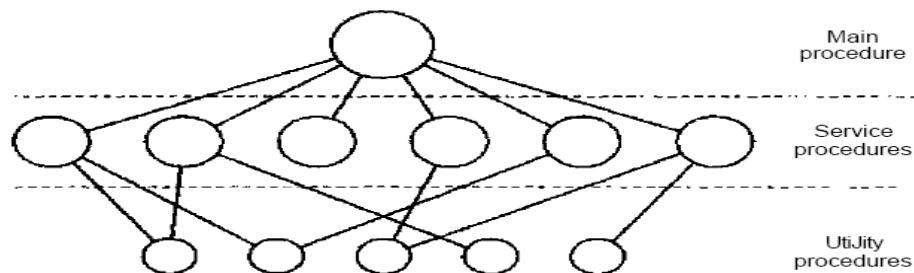


Fig.1.24.Një model i thjeshtë për sistemet monolitike.

1.7.2. SISTEMET ME SHTRESA (LAYERED)

Një përgjithesim i trajtimit të figures 1.24 është organizimi i sistemit operativ si një hierarki shtresash, ku çdonjëra është ndertuar mbi tjetren poshtë tij. Sistemi i pare i ndertuar ne ketë menyre ishte sistemi THE i ndertuar ne Technische Hogeschool Eindhoven ne Hollande nga E.W.Dijkstra (1968) dhe studentët e tij. Sistemi THE ishte një sistem i thjeshtë batch për një kompjuter hollandez, ElectroJogica X8, i cili kishte 32K fjale 27-bit.

Sistemi kishte 6 shtresa, sic tregohen ne figuren 1.25. Shtresa 0 merrej me alokimin e procesoreve, ndyshimin mes proceseve kur ndodhnin interrupte apo skadime të kohes. Sipër shtreses 0, sistemi përbehej nga procese vijues ku secili prej tyre mund të lidhtë programe duke mos u shqetësuar rreth faktit se shume procese po vepronin ne një procesor të vetëm. Me fjale të tjera, shtresa 0 siguroi multiprogramimin baze të CPU.

Shtresa	Funksionet
5	Operatori
4	Programet User
3	Menaxhimi input/output
2	Komunikimi Operator-proçes
1	Menaxhimi i memories dhe drum-it
0	Alokimi i procesorit dhe multiprogramimit

Figure 1-25, Structure of the THE operating Nyslem.

Shtresa 1 bënte menaxhimin e memories. Ajo alokontë hapesire për proceset ne memorien kryesore dhe ne 512K drum fjale të përdorur për mbajtjen e pjeseve të proceseve (faqeve) për të cilat nuk kishte vend ne memorien kryesore. Sipër shtëres 1, Proseset nuk kishin pse të shqetësoheshin ne se ato ishin ne memorie apo ne drum; software-i i shtreses 1 u kujdes për të bere të sigurt qe faqet të silleshin ne memorie kurdo qe ata mund të nevojiteshin.

Shtresa 2 bënte të mundur komunikimin ndermjet secilit proces dhe konsolit të operatorit. Mbi ketë shtrese secili proces ne menyre efektive kishte konsolin e tij të operatorit. Shtresa 3 kujdesej për menaxhimin e pajisjeve I/O dhe buffer-imin e rrjedhave të informacionit ne dhe jashtë tyre. Sipër shtreses 3 secili proces mund të merrej me pajisjet abstrakte I/O me karakteristika të kendshme, ne vend të pajisjeve reale me shume karakteristika. Shtresa 4 ishte aty ku gjendeshin programet user. Ato nuk kishin pse të shqetësoheshin rreth procesit, memories, konsolit ose menaxhimit I/O. Procesi sistem operator ishte i vendosur ne shtresen 5.

Një përgjithesim me i metejshem se ai i konceptit të shtresave ishte prezent ne sistemin MULTICS. Ne vend të shtresave, MULTICS ishte i përshkruar duke pasur një seri hallakash bashkeqendrore, ku ato me të brendeshmet ishin me të privilegjuara se ato të jashtmet (qe është efektivisht e njëjta gje). Kur një procedure ne një hallke (ring) të

jashtme dontë të therristë një procedure ne një hallke të brendeshme, ajo duhet të bënte ekulivalenten e një thirrjeje sistem, qe është një instrukcion TRAP, parametrat e të cilit ishin të kontrolluara me kujdes për vlefshmeri para se thirrja të lejohej të procedontë. Megjithese sistemi operativ i téri ishte pjese e hapesires se adresave të secilit proçes user ne MULTICS, hardware-i beri të mundur të përcaktontë procedurat individuale (aktualisht segmentët e memories) si të mbrojtura kundrejt leximit, shkrimit ose ekzekutimit.

Skema e shtresuar THE ishte me të vertetë vetëm një ndihme projektimi, sepse të gjithe pjeset e sistemit se fundmi ishin të lidhura se bashku ne një program objekt të vetëm, ne MULTICS, mekanizmi ring ishte me të vertetë shume prezentë ne kohen e ekzekutimit nga hardware-i. Avantazhi i mekanizmit ring është se ai shume thjesht mund të shtrihet ne strukturen e nensistemeve user. Për shembull, një profesor mund të shkruantë një program për të testuar dhe kategorizoje programet student dhe të ekzekutoje ketë program ne ring n , me programet student qe veprojne ne ring-un $n+1$, keshtu qe ata nuk mund të ndyshojne kategorine e tyre.

1.7.3. MAKINAT VIRTUALE

Nxjerrjet e para të OS/360 ishin pikesisht sistemet batch. Megjithatë, shume përdorues 360 donin të kishin timesharing (ndarjtë të kohes), keshtu IBM vendosi të shkruaje sisteme timesharing për të. Sistemi zyrtar IBM timesharing, TSS/360 u shpërndau vone dhe kur me ne fund arriti, ai ishte kaq i madh dhe kaq i ngadalta sa qe pak site ktheheshtin të ai. Ai u braktis përfundimisht pasi për zhvillimin e tij ishin harxhuar 50 milion \$ (Graham, 1970). Por një grup i qendres shkencore të IBM ne Cambridge, Massachusetts, prodhuan sistem rrenjësisht të ndyshem qe IBM përfundimisht e pranoi si produkt, dhe qe tani është gjeresisht i përdorur ne mainframet e tij të mbetur.

Ky sistem, fillimisht i quajtur CP/CMS dhe me vone i riquajtur VM/370 (Seawright and MacKinnon, 1979), ishte i bazuar ne një vrojtim të mprehtë; një sistem timesharing siguron (1) multiprogramimin dhe (2)një makine të zgjeruar me një nderfaqe me të përshtatshme se hardware-i minimal(bare). Thelbi i VM/370 është ndarja e plotë e ketyre dy funksioneve.

Zemra e sistemit, e njojur si **monitori i makines virtuale**, vepron ne hardware-in bare dhe bën multiprogramimin, duke i siguruar jo vetëm një por me shume makina virtuale shtreses se mepasme të sipërme, sic tregohet ne figuren 1.26. Sidoqoftë, ndryshe nga të gjithe sistemet e tjera operative, keto makina virtuale nuk jane makina të zgjeruara, me file-a dhe dukuri të tjera të kendshme. Ne të vertetë, ata jane kopje identike të hardware-it bare, duke përfshire mode-n user/kernel, I/O, interruptet dhe çdo gje tjetër qe kane makinat reale.

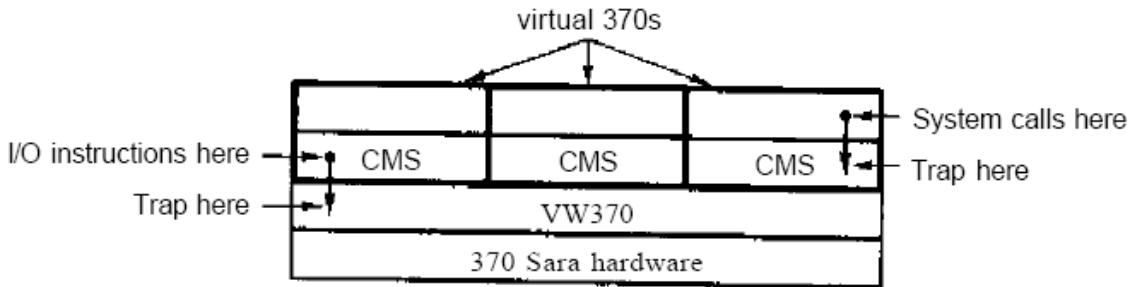


Fig.1.26. Struktura e VM/370 me CMS

Ngaqe çdo makine virtuale është identike me hardware-in e vertetë, çdo njëra mund të ekzekutoje cilido sistem operativ qe do të ekzekutohet direkt ne hardware-in bare. Makina virtuale të ndryshme mund të ekzekutojne sisteme operative të ndryshme, gje qe e bejne vazhdimisht. Disa ekzekutojne një nga paraardhesit e OS/360 përpunim veprimesh ose për batch, ndersa të tjere ekzekutojne sisteme interactive me një përdorues të vetëm të quajtura **CMS (Conversational Monitor System)** për përdorues interaktiv timesharing.

Kur një program CMS ekzekuton një thirrje sistem, thirrja behet trapped të sistemi operativ ne makine e tij virtuale, jo ne VM/370, pikerisht ashtu sic do të ishte ne se do të ekzekutohej ne një makine reale dhe jo ne një virtuale. Me pas CMS nxjerr instruksionet normale hardware-ike I/O për leximin e diskut të tij virtual apo cfaredo qe nevojitet për të kryer një thirrje. Keto instruksione I/O behen trapped nga VM/370, i cili me pas i intérpreton ato si pjese e simulimit të tij të hardware-it real. Duke ndare plotësisht funksionet e multiprogramimit dhe sigurimit të një makine me të zgjeruar, secila nga pjeset mund të jetë me e thjeshtë, me fleksibel dhe me e kollajtë për tu mbajtur.

Idea e makinave virtuale ne ditët e sotme është gjeresisht e përdorur ne një kontekst të ndryshem: ekzekutimi i programeve të vjetër MS-DOS (ose CPU të tjera Intel 32-bit). Gjatë projektimit të Pentium-it dhe software-it të tij, Intel dhe Microsoft kuptuan qe do të kishte një nevoje të madhe për ekzekutimin e software-ve të vjetër ne hardware-et e rinj. Për ketë arsy, Intel siguroi një mode viruale 8086 ne Pentium. Ne ketë mode, makina sillet si një 8086 (e cila nga kendveshtrimi i software-ve është identike me një 8088), duke përfshire adresimin 16-bit me një limit 1-MB.

Kjo mode përdoret nga Windows dhe sisteme të tjera operative për ekzekutimin e programeve MS-DOS. Keto programe fillohen ne mode virtuale 8086. Për sa kohe ata ekzekutojne instruksione normale, ata ekzekutojne ne hardware-in bare. Gjithsesi, kur një program bën përpjekje për të trap me sistemin operativ për të bere një thirrje sistem apo bën përpjekje për të bere ne menyre direktë I/O të mbrojtur, ndodh një trap ne monitorin e makines virtuale.

Ne ketë projektim jane të mundshem dy variante. Ne të parin, vetë MS-DOS është i ngarkuar ne hapesiren e adresave virtuale të 8086, keshtu qe monitori i makines virtuale thjesht reflekton mbrapër shembull trap-in tek MS-DOS, pikerisht sic do të ndodhë ne 8086 reale. Kur MS-DOS me vone provon të beje vetë I/O, ky opërim nderpritet dhe kryhet nga monitori i makines virtuale.

Ne variantin tjeter, monitori makines virtuale thjesht kap trap-in e pare dhe bën I/O vetë, meqene se ajo i di se cfare jane të gjitha thirrjet sistemit MS-DOS dhe keshtu di

se cfare është i supozuar të beje secili trap. Ky variant është me pak i pastër se ai i pari, përderisa rivalizon ne menyre korrekte vetëm MS-DOS dhe jo sistemet e tjera operative, sic bën i pari. Ne anen tjetër, është me i shpejtë, përderisa ruan problemet e fillimit të MS-DOS për të bere I/O. Një disavantazh i metejshem i MS-DOS-it aktualisht veprues ne mode viruale 8086 është se harxhon shume kohe me bitet enable/disable të interrupteve, të gjitha keto duhet të rivalizohen ne një kosto të konsiderueshme.

Është e rendesishme të theksojme se asnëra nga keto arritje jane me të vertetë të njëjtë me VM/370, meqene se makina qe do të rivalizohet nuk është një Pentium i plotë, por vetëm një 8086. Me sistemin VM/370 është e mundur të ekzekutohet vetë VM/370 ne makine virtuale. Me Pentium, nuk është e mundur të ekzekutohet Windows nw 8086 virtuale sepse asnë version i Windows nuk ekzekutohet ne një 8086; një 286 është minimumi edhe për versionet me të vjetra dhe rivalizuesi i 286 nuk është krijuar (duke lene menjane rivalizimin e Pentium). Gjithsesi, duke modifikuar pak binary-n Windows, ky rivalizim është i mundur madje edhe i përshtatshem ne produktët komerciale.

Tjetër përdorim i makinave virtuale, por ne një menyre të ndryshme është për ekzekutimin e programeve Java. Kur Sun Microsystems krijuan gjuhen e programimit Java, gjithashtu krijuan një makine virtuale (një arkitekturë kompjuteri) të quajtur **JVM (Java Virtual Machine)**. Kompilatori Java prodhon kod për JVM, i cili me pas ekzekutohet nga një interpretues JVM software-i. Avantazhi i kesaj arritjeje është se kodi JVM mund të transportohet nepërmjet internetit ne çdo kompjuter qe ka një interpretues JVM dhe të ekzekutohet atje. Ne qoftë se kompilatori ka prodhuar SPARC ose programe binare Pentium, për shembull, ato mund të mos jene transportuar dhe ekzekutohen kudo kaq lehtë. (Sigurisht Sun mund të kishte prodhuar një kompilator qe prodhonte binare SPARC dhe me pas të shpërndantë një interpretues SPARC por JVM është një arkitekturë me e thjeshtë për tu interpretuar). Një tjetër avantazh i përorimit të JVM është se ne qoftë se interpretuesi është implementuar sic duhet, qe nuk është plotësisht e parendesishme, Programet ardhese JVM kontrollohen për siguri dhe me pas ekzekutohen ne një mjedis të mbrojtur ne menyre qe ata të mos mund të vjedhin të dhena apo të bejne ndonjë dem tjetër.

1.7.4. EXOKERNELS

Me VM/370, çdo proces user merr një kopje identike të kompjuterit aktual. Me mode-n virtuale 8086 ne Pentium, çdo proces user merr një kopje identike të një kompjuteri tjetër. Duke vazhduar një hap me tej, kerkuesit a M.I.T. kane ndertuar një sistem qe i jep secilit përdorues një klonim të kompjuterit aktual, por me një nenbashkesi burimesh (Engler et al., 1995). Keshtu një makine virtuale mund të marre bloqe disku nga 0 -1023, tjetri me pas mund të marre blloqet nga 1024-2047 dhe keshtu me rradhe.

Ne shtresen e fundit, është një program qe vepron ne mode kernel dhe quhet exokernel. Puna e tij është alokimi i burimeve ne makinen virtuale dhe me pas kontrollon përpjekjet për përdorimin e tyre për tu siguruar qe asnë makine nuk po mundohet të përdor burimet e dikujt tjetër. Secila makine virtuale e nivelit user mund të ekzekutoje sistemin e tij operativ, si ne VM/370 dhe Pentium virtual 8086, përvèc qe secila është e kufizuar për të përdorur vetëm burimet qe ka kerkuar dhe qe me pas janë alokuar.

Avantazhi i skemes exokrenel është se ajo ruan një shtrese të mapping. Ne projektime të tjera, çdo makine virtuale mendon sikur ka disqet e tij, me blloqe ekzekutimi nga 0 deri ne një maksimum, keshtu qe monitori i makines virtuale duhet të mbaj tabelat për të rihartuar adresat e diskut (all të burimeve të tjere). Me exokernel ky rihartim është i panevojshem. **Exokerneli ka te beje me atë se cila gjurmë se cila makine virtuale i është caktuar cilit burim**. Kjo metode ka akoma avantazhit e ndarjes se multiprogramimit (ne exokernel) nga kodi i sistemit operativ user (ne hapesireni user), por me me pak mbingarkese, meqene se e gjithe ajo cfare duhet të beje exokerneli është të mbaje makinat virtuale larg të ngacmimive të të tjereve.

1.7.5. MODELI KLIENT-SERVER

VM/370 rritet ne thjeshtësi duke levizur një pjese të madhe të kodit tradicional të sistemit operativ (duke implementuar makinen e zgjeruar) ne një shtrese me të lartë, CMS. Sidoqoftë, vetë VM/370 akoma është një program kompleks sepse simulimi i një numri të 370 virtuale ne tëresi të tyre nuk edhtë aq e thjeshtë (vecanerisht ne qoftë se doni ta bëni atë ne menyre të arsyeshme eficiente).

Një tendencë ne sistemet moderne operative është ideja e të levizurit të kodit ne shtresa të metejshme të sipërme dhe të heqë sa me shume të jetë e mundur nga moda kernel, duke lene një **microkernel** minimal. Arritja e zakonshme është të implementuarit e shumices se sistemit operativ ne proceset user. Për të kerkuar një sherbim, si leximi i një blloku apo i një file-i, një proces user (tani i njohur si procesi **klient**) dergon kerkesen ne një proces **server**, i cili me pas bën punen dhe dergon mbrapa përgjigjen.

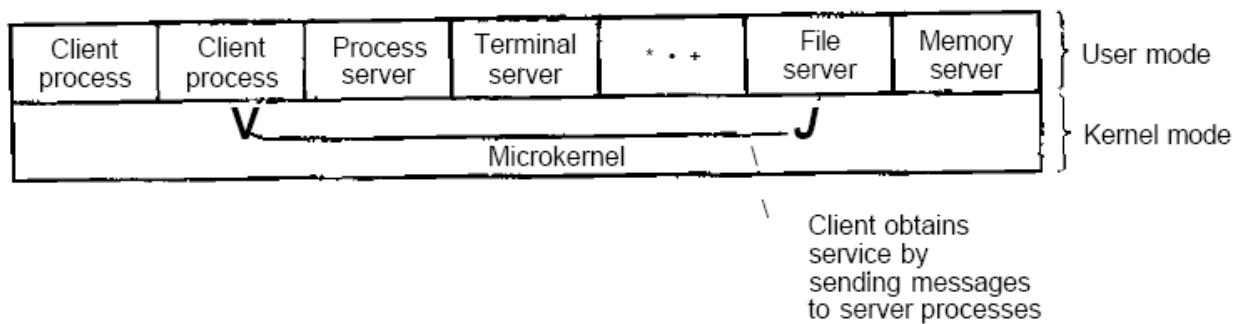


Fig.1.27. Modeli klient-server.

Ne ketë model të treguar ne figuren 1.27., e gjithe ajo cfare kerneli bën është organizimi i komunikimit mes klientit dhe serverit. Duke ndare sistemin operativ ne dy pjese, ku secila merret me një aspekt të sistemit si; sherbimi file, sherbimi proces, sherbimi terminal apo sherbimi memorie, secila pjese behet me e vogel dhe me e menaxheshme. Për me tepër, meqë të gjithe serverat veprojnë si procese të mode-s user, dhe jo ne kernel, ata nuk kane aksesim direkt ne memorie. Si pasoje, ne qoftë se gjendet një virus ne fileserver, sherbimi file mund të deshtoje, por kjo nuk do të beje të deshtoje e gjithe makina.

Një tjetër avantazh i modelit klient-server është përshtatshmeria e përdorimit ne sistemet e shpérndara (shikoni figuren 1.28.). Ne qoftë se një klient komunikon me një server duke derguar mesazhe, klienti nuk ka nevoje të dije ne se mesazhi merret lokalisht nga makina e tij, apo është derguar nepërmjet një rrjeti tek makina e serverit. Për sa kohe klienti është i interesuar, e njëjtë gje ndodh ne të dy rastet: dergohet një kerkese dhe vjen një përgjigje.

Figura e mesipërme e një kerneli qe merret me transportin e mesazheve nga klientët të serverat dhe anasjelltas nuk është plotësisht reale. Disa funksione të sistemit operativ (si ngarkimi i komandave ne regjistrat e pajisjeve fizike I/O) jane të veshtira, për të mos thene të pamundura, për tu bere nga programet ne hapesiren user. Ka dy menyra për tu marre me ketë problem. Një menyre është të kemi disa proçesse server kritike (për shembull, driverat e pajisjeve I/O) qe aktualisht të veprojne ne mode kernel, me aksesim të plotë ne hardware, por akoma të komunikojne me proçeset e tjera duke përdorur mekanizmin e mesazheve normale.

Menyra tjetër është të ndertojme një sasi minimale të mekanizmit ne kernel por vendimet e mencura ti leme ne dore të serverave ne hapesiren user (Levin et al., 1975). Për shembull, kerneli duhet të njohe qe një mesazh i derguar ne një adresë të caktuar nenkupton të marre përbajtjen e atij mesazhi dhe ta ngarkoje atë ne regjistrat e pajisjeve I/O, qe të filloje leximin e diskut. Ne ketë shembull, kerneli nuk duhet të prese sa bytet ne mesazh, për të pare ne se ishin gabim apo të pakumptimta, ai thjesht duhet të kopjoje ato ne regjistrat e pajisjeve ne disk, (me sa duket, disa skema për limitimin e mesazheve të tille duhet të përdoren vetëm ne proçese të autorizuar). Ndodh ne kontekste të ndryshme ne sistemin operativ here pas here.

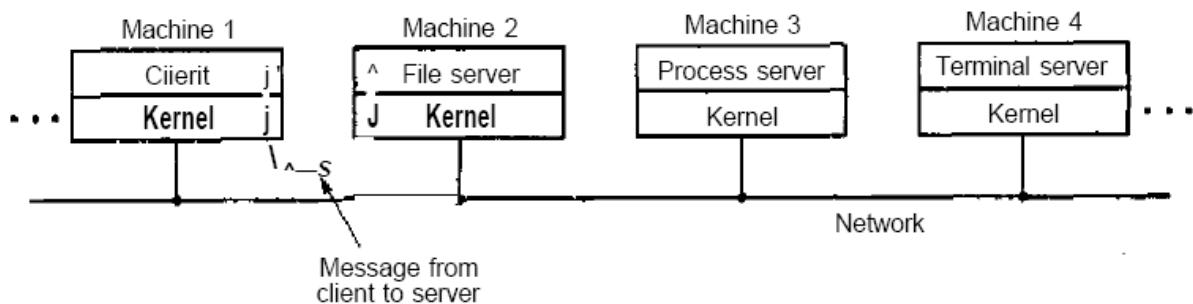


Fig.1.28. Modeli klient-server ne sistemet e shpérndare.

1.8. KERKIMI NE SISTEMET OPERATIVE

Shkenca kompjuterike është një fushe qe avancon shume shpejt dhe është e veshdre të parashikosh se ku do të shkoje. Kerkuesit ne universitetë dhe ne laboratoret industrial të kerkimit jane ne menyre të vazhdueshme duke menduar për ide të reja, disa prej të cilave nuk shkojne ne asnje vend por disa prej tyre behen themeloret e produktëve të se ardhmes dhe kane ndikim masiv ne industri dhe të përdoruesit. Të tregosh se cfare është secila duket me e lehtë se ne jetën reale. Ndodh se grurit nga mbeturinat është vertetë e veshdre sepse shpesh kerkon 20-30 vjet qe të ndikohet nga ideja.

Për shembull, kur presidenti Eisenhower ndertoi dep. e Defence's Advanced research Projects Agency (ARPA) ne 1958, ai po mundohej të mbantë ushtrine nga vrasja e flotës ushtarake detare dhe forcat ajrore nepërmjet buxhetit të kerkimeve ne Pentagon. Ai nuk po mundohej të krijonte internetin. Por një nga gjerat qe beri ARPA ishte financimi i disa kerkime universitare, gje qe shpejt coi ne rrjetin e pare eksperimental packet-switched, ARPANET. Vazhdoi deri ne 1969. Me lidhjen e rrjetëve të tjere të gjetur nga ARPA me ARPANet, lindi interneti. Atëherë interneti për 20 vjet përdorej nga kerkuesit ushtarake për dergimin e emailve të njëri-tjetri. Ne fillim të 1990, Tim Berners-Lee krijoi World Wide Web ne laboratorin e kerkimeve CERN ne Geneve dhe Marc Adreesen shkroi një browser grafik për të ne universitetin Illinois. Shume shpejt interneti ishte plot të rinj qe chatonin.

Kerkimet ne sistemet operative gjithashtu na cojne ne ndryshime dramatike ne sistemet praktike. Sic e kemi diskutuar me pare, sistemet e para komerciale kompjuterike ishin të gjithe sisteme batch, derisa M.I.T. krijoi timesharing interaktiv ne fillim të viteve 1960. Kompjuterat ishin të gjithe text-based derisa Doug Engelbart krijoi mouse-in dhe nderfaqen grafike user ne institutin kerkimore Stanford ne fund të viteve 1960. Kush e di se cfare do të vij me vone?

Ne ketë paragrafe do të shohim shkurtimisht ne disa kerkime ne sistemet operative qe jane bere gjatë 5 deri 10 vitet e fundit, thjesht të japim një atmosfere të atyre qe jane ne horizont. Ky prezantim nuk është i hollesishem dhe përgjithesisht është i bazuar ne publikimet ne revistat kerkimore dhe konferanca. Shumica e citimeve të publikimeve jane publikuar nga ACM apo nga Organizata e kompjuterave IEEE, apo USENIX dhe jane të diponueshem ne internet tek pjesetaret e ketyre organizatave dhe biliotekave dixhitale të tyre viziton:

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Virtualisht të gjithe kerkuesit e sistemeve operative kuptojne qe sistemet aktuale operative jane masive, infleksible, të pa pabesueshme, të pa sigurta dhe të ngarkuar me viruse, disa me shume e disa me pak (për të mbrojtur fajtorin emrat Jane të fshehtë). Për pasoje është bere një kerkim i gjere për menyren se si të ndertojmë sisteme fleksible. Shumica e kerkimeve kane lidhje me sistemet mikrokernal. Keto sisteme kane një kernel minimal, keshtu qe ka një shans qe ato mund të behen të besueshem dhe të regulluar. Ata gjithashtu jane fleksibel sepse shume sisteme operative reale veprojne shume si procese user mode, dhe keshtu mund të zevendesohen apo të përshtatën shume lehtë, mundesisht edhe gjatë ekzekutimit. Ne menyre tipike ajo cfare bën mikrokerneli është të merret me menaxhimin e burimeve dhe kalimin e mesazheve mes proceseve user.

Mikrokernalat e gjenerates se pare, si Amoeba (Tanenbaum et al., 1990), Chorus (Rozier et al., 1988), Mach (Accetta et al., 1986), dhe V (Cheriton, 1988), treguan se sisteme të tille mund të ndertoheshin dhe të përdoreshin për pune. Gjenerata e dytë po përpinqet të tregoje qe ato jo vetëm mund të punojne, por madje edhe me performance të lartë (Ford et al., 1996; Hartig et al., 1997; Liedtke 1995, 1996; Rawson 1997; dhe Zuberi et al., 1999). Bazuar ne mase të publikimeve duket se ky objektiv është arritur.

Ne ditët e sotme shumica e kerkimeve kernel është fokusuar ne ndertimin e sistemeve operative të zgjatshme (extensible). Keto jane ne menyre tipike sistemet mikrokernel me aftësi për të zgjeruar dhe përshtatur ne disa drejtime. Disa shembuj janë Fluke (Ford et al., 1997), Paramecium (Van Doom et al., 1995), SPIN (Bershad et al., 1995) dhe Vino (Seltzer et al., 1996). Disa kerkues gjithashtu po kerkojne se si mund të zgjerojnë sistemet ekzistuese (Ghormley et al 1998). Shume nga keto sisteme lejojnë sistemin operativ të shtojne kodin e tyre ne kernel, gje e cila na sjell problemin e dukshem se si të lejohen prapashtesat (extension) ne një menyre të sigurtë. Teknikat përfshijne intepretimin e extension-eve, kufizimin e tyre ne sandbox-et kodeve, përdorimin e gjuheve type-safe dhe shenimin e kodit (Grimm dhe Bershad, 1997: dhe Small and Seltzer, 1998). Drusich et al (1997) paraqet një opinion kunder, duke thene se do të duhej shume përpjektje ne mbrojtje për sistemet oser-extandable. Sipas tyre, kerkuesit duhet të kuptionin se cilët extensions ishin të dobishem dhe me pas ti bënин ato pjese normale të kernelit.

Edhe pse një arritje për të eliminuar sistemet operative plot me viruse dhe të pabesueshem është ti bejme ata me të vegjel, një tjetër me thelbesore është të fshijme të gjithe sistemin operativ. Kjo arritje po ndiqet nga gruپi Kaashoek ne VI.IT, ne kerkimet e tyre exokernel. Ketu idea është të kemi një shtrese të holle software qe vepron ne metalin bare, puna e vetme e të cilit është alokimi i sigurt i burimeve hardware tek userat. Për shembull, ai duhet të vendose kush do të përdore dhe cilen pjese të diskut, dhe ku do të vendosen paketat hyrese. Çdo gje tjetër i është lene ne dore proçeseve të nivelit user, duke bere të mundur ndertimin e sistemeve operative me qellime të përgjithshme dhe atyre hollesisht të specifikuara (Engler dhe Kaashoek, 1995; Engler et al, 1995: dhe Kaashoek et al, 1997).

1.9. PARASHTRIMI I PJESES TJETËR TË LIBRIT

Tani kemi përfunduar se prezantuari veshtrimin e përgjithshem të sistemeve operative. Është koha të futëmi me ne detaje. Kapitulli 2 trajton proçeset. Ai gjithashtu flet për karakteristikat e tyre dhe menyren e komunikimit të tyre me njëri-tjetrin. Gjithashtu ka një një numer shembujsh të detajuar se si funksionon komunikimi nderproçesor dhe si shmangim disa kurthe.

Kapitulli 3 trajton deadlocks. Ne ketë kapitull ne treguan shkrutimisht se cfare janë deadlocks, por ka me shume për të thene rrëth tyre. Gjithashtu diskutohen menyra për ndalimin e tyre.

Ne kapitullin 4 do të studiojme ne detaje menaxhimin e memories. Ketë do të shpjegojen tëmat e rendesishme të memories virtuale, me konceptet të lidhura ngushtë si paging dhe segmentimi.

Input/Output do të trajtohet ne kapitullin e pestë. Do të shihen konceptet e pajisjeve të pavarura dhe të varura. Disa pajisje të rendesishme, duke përfshire edhe disqet, tastjerat do të përdoren si shembuj.

Me pas ne kapitullin e gjashtë, vijme ne tëma të rendesishme të sistemeve file. Me një zgjerim të konsiderueshem, ajo cfare sheh përdoruesi është përgjithesisht sistemi file. Do të shikojme nderfaqen e sistemeve file dhe implementimin e sistemeve file.

Ne ketë pike do të kemi përfunduar studimin tone të principeve baze të sistemeve operative me një CPU të vetme. Gjithsesi, ka me shume për të thene, sidomos rrëth témave të avancuara. Ne kapitullin 7, do të shqyrtojme sistemet multimedia, të cilët kane një numer karakteristikash dhe vetish qe ndryshojne nga sistemet operative tradicionale. Vec të tjerave, skedulimi dhe sistemet file janë të ndikuara nga natyra e sistemeve multimedia. Një tjetër teme e avancuar janë sistemet me shume procesore, duke përfshire multiproçesoret, kompjuterat paralel dhe sistemet e shpërndare. Keto tema trajtohen ne kapitullin e 8.

Një teme shume e rendesishme është siguria e sistemeve operative, që trajtohet ne kapitullin 9. Nder tëmat e trajtuar në këtë kapitull janë threads (viruset dhe worm), mekanizmat mbrojtës dhe modelet e sigurimit.

Me pas kemi disa studime rasti të sistemeve operative reale. Keto jane UNIX (kapitulli 10) dhe Windows 2000 (kapitulli 11). Libri përfundon me disa mendime rreth sistemeve operative ne kapitullin 12.

1.10. NJËSITE METRIKE.

Për të shmangur çdo konfuzion, është e rendesistem të themi se ne ketë liber, si ne shkencen kompjuterike ne përgjithesi, njësите metrike janë përdorur ne vend të njësive tradicionale angleze. Prefikset themelore metrike janë të listuara ne figuren 1.29. Prefikset janë ne menyre tipike të shkurtuar me shkronjat e tyre të para, me njësите me të medha se 1 janë të kapitalizuara. Keshtu një database 1-TB ze 1012 byte për ruajtje dhe një 100 clock tick –e pces (ose 100 ps) çdo 10-10 sekonda. Përderisa milli dhe mikro fillojnë qe të dyja me shkronjë ‘m’, duhet të behej një ndryshin. Normalisht ‘m’ është për milli dhe ‘ μ ’ është për mikro.

Fig.1.29. Prefikset kryesore metrike

2

2. PROÇESET DHE THREAD-ET

Tashme ne jemi ne gjendje të fillojme një studim me ne detaje të dizenjimit dhe ndertimit të një sistemi operativ. Koncepti kryesor i çdo sistemi operativ është procesi: një abstrakim i një programi të ekzekutushem. Çdo gje tjetër varet nga ky koncept dhe është e rendesishme qe dizenjuesit e sistemit operativ (për shembull. studentët) të kene një njojje të kompletuar për atë cfare është procesi sa me shpejtë të jetë e mundur.

2.1 PROÇESET

Të gjithe kompjuterat modern Jane ne gjendje të bejne gjera të ndryshme ne të njëjtën kohe. Nderkohe qe një program user është duke u ekzekutuar, kompjuteri mund të lexoj nga një disk ose të nxjerri si output një tekst ne ekran ose ne printer. Ne një sistem multiprogramimi një proces mund kaloj nga një program ne një tjetër, duke ekzekutuar secilin për dhjetra ose qindra milisekonda. Nderkohe, duke folur rigorozisht, ne çdo moment të kohes, CPU ekzekuton vetëm një program, ne kohen prej 1 sekond, ai mund të punoj me programe të ndryshme, kjo i jep userave iluzionin e paralelizmit. Ndonjehere njerezit flasin për pseudoparalelizem ne ketë kontekst, për ta krahasuar me paralelizmin e vertetë të sistemeve multiproçessor (të cilët kane dy ose me shume CPU qe share-ojne të njëjtën memorie fizike). Duke patur parasysh shumëllojshmerine, aktivitet ne paralel Jane të veshtira për tu bere nga njerezit. Gjithsesi, dizenjuesit e sistemit operativ kane përfshire një model konceptual (proçeset sekuenciale) qe e bejne paralelizmin të lehtë për të kryer veprime me të. Ky model, përdorimet e tij dhe disa nga pasojat e tij përbejne subjektin e ketij kapitulli.

2.1.1 Modeli Proçes

Ne ketë model, të gjithe software-et e ekzekutushem ne kompjuter, ndonjehere duke përfshire dhe sistemin operativ, Jane ndertuar ne një numer sekuencial procesesh ose për shkurt procese. Një proces është thjesht një program i ekzekutueshem, duke përfshire vlerat e fundit të program counter, regjistrave dhe variablate. Ne menyre konceptuale çdo proces ka CPU e vet virtuale. Ne realitet, një CPU reale komuton procesin para dhe pas, por për të kuptuar sistemin, është me e thjeshtë për të menduar rreth një koleksioni

proçesesh qe ekzekutohen ne paralel, se sa të ndjekesh gjurmet se si CPU komuton nga programi ne program. Kjo menyre e shpejtë e komutimit para dhe pas është quajtur multiprogramim, sic pame ne kapitullin e pare .

Ne fig 2-1(a) shohim një kompjuter multiprogramimi me 4 programe ne memorie. Ne fig 2-1(b) shohim 4 proçese secili me program counter e vet llogjik dhe ato ekzekutohen ne menyre të pavarur nga njëri tjetëri. Sigurisht qe ka vetëm një program counter fizik, keshtu qe kur çdo proçes ekzekutohet, program counteri i tij llogjik ngarkohet ne program counter-in real. Kur ai ka përfunduar për kohen e mbetur, program counter fizik ruhet ne proçes kurse program counter llogjik ne memorie. Ne fig.2-1(c) do të shohim qe gjatë një intervali kohe, të gjithe proçeset kane bere një progres, po ne të vertetë ne një cast të kohes vetëm një proçes po ekzekutohet.

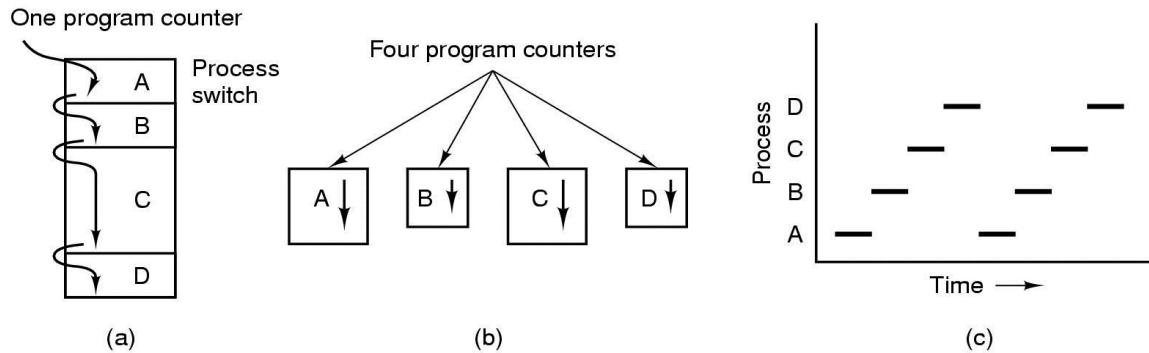


Fig.2-1 (a) Multiprogramimi për katër programe.(b)Modeli conceptual i katër proçeseve sekuenciale dhe të pavarur.

Me komutimin e CPU para dhe pas ndermjet proçeseve, shpejtësia me të cilen një proçes përformon llogaritjen e tij, nuk do të jetë uniforme dhe ne qoftë se i njëjti proçes ekzekutohet përsëri është e mundur të mos marrim ndonjë rezultat. Keshtu qe, proçeset mund të mos programohen duke ngritur supozime mbi sinkronizimin. Konsideroni, për shembull një paisje I/O qe nis një regjistrim shiriti për të rivendosur back up të fileve, ekzekutohet një (idle loop) 10000 here për të rritur shpejtësinë, e me pas zgjidhet një komande për të lexuar regjistrimin e pare. Ne qoftë se CPU vendos të komutoj ne një program tjetër gjatë idle loop, proçesi i regjistrimit mund të mos ekzekutohet derisa regjistrimi i pare të ketë kaluar tek koka lexuese. Kur një proçes ka kerkesa kritike ne kohe reale, si për shembull ngjarje të vecanta mund të ndodhin brenda një numri të specifikuar milisekondash, duhen marre masa të vecanta për tu siguruar qe keto të ndodhin. Normalisht shume proçese nuk jane prekur nga multiprogramimi i nenvizuar i CPU apo shpejtësia relative e proçeseve të ndryshme.

Ndryshimi ndermjet procesit dhe programit është delikat, por kritik. Ketu do të na ndihmontë një analogji. Le të marrim ne konsiderate një shkencetar kompjuterash i cili është duke pjekur një kek për ditlindjen e se bijes. Ai ka një recetë keku dhe një kuzhine të mbushur me të gjithe përberesit: mielli, vezet, sheqeri, vanilje e keshtu me rradhe. Ne ketë analogji, receta është programi (një algoritem i shprehur ne një kuptim të përshtatshem), shkencetari i kompjuterave është procesori dhe përberesit e kekut jane të dhenat ne hyrje. Një proces është një veprim qe konsiston ne leximin e recetës nga pjekesi, marrjen e përberesve dhe pjekja e kekut.

Tani imagjinoni qe djali i shkencetarit të kompjuterave vjen duke qare, duke thene se e ka pickuar një bletë. Shkencetari i kompjuterave fikson ku e nderpreu recetën (kjo është gjendja e ruajtjes se procesit të fundit), merr një liber të ndihmes se pare dhe i hedh një sy instrukSIONEVE ne të. Ketu ne shohim kalimin e procesorit nga një proces (pjekja) ne një proces të një niveli me të lartë prioriteti (administrimi i kujdesit mjeksor), ku secili ka program të ndryshem. Kur kujdesi ndaj të pickuara se bletës përfundoi, shkencetari i kompjuterave kthehet të keku aty ku e kishte lene .

Idea kryesore është se një proces është një veprim i llojeve të ndryshme. Ai ka një program, input, output dhe një gjendje. Një procesor i vetëm mund të ndahet ndermjet disa proceseve, me disa algoritma skedulimi të përdorur për të përcaktuar kur mbron puna e një procesi, dhe ti sherbehet një tjetri.

2.1.2 Krijimi i një procesi

Sistemet operative kane nevoje të sigurohen qe të gjithe proceset e nevojshme ekzistojnë. Ne sisteme shume të thjeshtë ose ne sistemet e dizenuara për një aplikim të vetëm (për shembull. kontrolli ne furren e një sobe), është e mundur qe të gjithe proceset e nevojshme të jene prezent kur sistemi aktivizohet. Ne menyre të përgjithshme sistemet kane nevoje për disa menyra për krijimin dhe përfundimin e proceseve sic Jane të nevojshme gjatë veprimeve. Le të shikojme disa prej ketyre ceshtjeve. Jane katër ngjarje të rendesishme qe shkaktojne krijimin e proceseve:

1. Inicializimi i sistemit
2. Eksekutimi i një thirrje sistem për krijimin e proceseve nga një proces ne ekzekutim
3. Kerkesa e një useri për krijimin e një procesi të ri.
4. Fillimi i një grapi pune.

Kur një sistem operativ boot-ohet, krijohen disa procese. Disa prej tyre Jane procese foreground, të cilët bashkeveprojne me user-in dhe kryejne detyra për to. Të tjeret Jane procese background, të cilët nuk Jane lidhur ne menyre të vecantë me user-in, por panvarësish kesaj ato kane disa funksione specifike. Për shembull, një proces background mund të dizenojhet për të pranuar email qe vijne, ai mund të jetë i paaktivizuar gjithe ditën dhe mund të aktivizohet kur vjen një email. Një proces tjetër

background mund të dizenjohet për të pranuar kerkesat qe vijne nga faqet Web, qe jane host ne atë makine, aktivizohet për ti sherbyer kerkeses qe ka ardhur.

Proçeset qe qendrojnë ne background për të trajtuar disa aktivitetë sic Jane email, faqe Web, lajme, printime e keshtu me rradhe, Jane quajtur **daemons**. Sistemet e medha kane shume të tilla. Ne UNIX, programi për shembull mund të përdoret për të listuar proçeset ne ekzekutim. Ne Windows 95/98/Me, duke shtypur një kohësish komandat ctrl-alt-del nga tastjera tregon se cfare po ekzekutohet ne atë moment. Ne Windows 2000 përdoret **task manager**.

Përvec proçeseve qe krijohen gjatë kohes se boot-imit, proçeset mund të krijohen dhe me vone. Zakonisht një sistem ne ekzekutim mund të përdori thirrjen sistem për të krijuar një ose me shume proçese të reja për të ndihmuar ne kryerjen e detyrave të tij. Krijimi i proçeseve është i nevojshem kur puna qe duhet të kryejne është e thjeshtë për tu formuluar ne terma me disa referime, por nga ana tjetër proçeset bashkëveprojne ne menyre të pavarur. Ne qoftë se një sasi e madhe të dhenash është térhequr nga një network për proçesin qe vijon, do të jetë me përshtatshme të kijohet një proçes për të térhequr të dhenat dhe për ti vendosur ne një buffer të share-uar derisa një proçes i dytë të zhvendos të dhenat dhe ato proçese. Ne një multiproçesor, duke lejuar qe çdo proçes të ekzekutohet ne CPU të vecanta do të bënte qe një pune të ectë shpejtë.

Ne sistemet itéraktive, user-at mund të startojne një program duke shtypur një komande ose (double) klik ne një ikone. Duke u nisur nga ky veprim starton një proçes i ri dhe ekzekutohet programi i selektuar. Ne komandat baze sistemet UNIX ekzekutojne X Windows, proçesi i ri vendoset ne dritaren ku u startua. Ne Microsoft Windows, kur një proçes startohet nuk ka një dritare, por ajo mund të krijoj një ose me shume. Ne të dy sistemet, user-at mund të kene shume dritare hapur ne të njëjtën kohe, secili duke ekzekutuar proçese të ndryshme. Duke përdorur mousin, user-i mund të selektoje një dritare dhe të nderhyj ne proçes, për shembull duke shtuar të dhena kur duhet.

Situata e fundit ne të cilën sistemet mund të krijohen aplikohet vetëm ne sistemet e grupuara (batch) të gjetura ne mainframe të medha. Ketu user-at mund ti nenshtrojne një grup pune sistemit. Kur një sistem vendos qe i ka burimet për të ekzekutuar një pune tjetër, ai krijon një proçes të ri dhe ekzekuton punen tjetër nga rradha e inputeve ne të.

Teknikisht, ne të gjithe keto raste, një proçes i ri është krijuar duke patur një proçes ekzistënt qe ekzekuton thirrjen sistem për krijimin e një proçesi. Ai proçes mund të jetë një proçes user i ekzekutus hem ose një proçes sistem i futur nga tastjera ose mouse, ose një batch manager proçess. Ajo cfare bejne keto proçese është se ekzekuton një thirrje sistem për të krijuar një proçes të ri. Kjo thirrje sistem i kerkon sistemit operativ të krijoj një proçes të ri, qe tregon ne menyre direktë ose jo qe cili program do të ekzekutohet ne të.

Ne UNIX ka vetëm një thirrje sistem për të krijuar një proçes të ri: **fork**. Kjo thirrje krijon një kopje ekzaktë të proçeseve thirres. Pas fork, të dy proçeset prind dhe femi, kane të njëjtën pamje ne memorie, të njëjtat stringje dhe të njëjtat open files. Ne përgjithësi proçesi femi, me pas ekzekuton execve ose një thirrje sistem të ngjashme qe ndryshon pamjen e memories, dhe ekzekuton një program të ri. Për shembull, kur një

user shtyp një komande, say, sort to the shell, shell ndalon procesin femi dhe kercen ne femijen e ekzekutushem. Arsyja për procesin me dy hapa është të lejoi femijet të manipulojne deskriptorin e fileve pas fork por para execve për të kryer riadresimin e input-eve standarte, output-eve standarte dhe erroreve standarte.

Ne Windows, ne kontrast, një funksion i vetëm Win32, CreateProcess, trajtohen të dyja krijimi i proceseve dhe ngarkimi i programit korrekt ne procesin e ri. Kjo thirrje ka 10 parametra, të cilat përfshijnë programin qe do të ekzekutohet, parametrat komand line të cilat ushqejnë atë program, atributetë ndryshme sigurie, bite qe kontrollojnë ne se open files janë trasheguar, informacionet me prioritet, specifikime për dritaren qe do të krijohet për procesin (ne qoftë se ka) dhe një pointer i një strukture ne të cilën informacioni mbi procesin e ri të krijuar i kthehet therritësit. Për të plotësuar CreateProcess, Win32 ka rreth 100 funksione të tjera për menaxhimin dhe sikronizimin e proceseve dhe tëmave qe kane lidhje.

Sin e UNIX ashtu dhe ne Windows, pasi procesi është krijuar, të dy prindi dhe femija kane hapsirat e tyre të adresave të ndara, edhe ne qoftë se procesi ndryshon një fjale ne hapsiren e adresave, ndryshimet nuk janë të dukshme nga proceset e tjera. Ne UNIX, fillimi i hapsires se adresave të femijes është një kopje e prindit, por aty janë përfshire dy hapsira adresash; memoria e shkrueshme nuk është e share-uar (ne disa implementime të UNIX share-ohet programi text ndermjet të dyve derisa ato të mos modifikohen). Është e mundur gjithashtu për krijimin e proceseve të reja të share-ohen disa nga krijuesit e tij, burime të tjera sic janë open files. Ne Windows, hapsira e adresave të femijeve dhe prinderve janë të ndryshme nga fillimi.

2.1.3 Mbarimi i Proçesit

Pasi një proces është krijuar, ai fillon të ekzekutohet dhe të kryej çdo detyre për të cilën është krijuar. Gjithsesi, asnjë gje nuk zgjat për gjithmone as proceset. Heret a vone proceset e reja do të përfundojnë, zakonisht i takon një nga kushtet e meposhtme:

1. Dalje normale (vullnetare)
2. Kur kemi error (vullnetare)
3. Gjatë një errori fatal (jo vullnetare)
4. Vrasja nga një proces tjetër ((jo vullnetare))

Shumica e proceseve përfundojnë pasi e kane kryer detyren e tyre. Kur një kompilator ka kompiluar programin qe i është dhene, kompilatori ekzekuton një thirrje sistem për ti treguar sistemit operativ qe ka përfunduar. Kjo thirrje është **exit** ne **UNIX** dhe **ExitProcess** ne **WINDOWS**. Programet screen-oriented gjithashtu mbështësin mylljet vullnetare. Fjalet e procesoreve, Internet browser dhe programe të ngjashme kane gjithmone ikona ose menu qe user-at mund të click-ojne për ti treguar procesit qe të zhvendosi çdo file të përkohshem qe ka hapur dhe me pas ta përfundoj.

Arsyeja e dytë e mbylljes është qe një proces zbulon një error fatal. Për shembull, ne qoftë se një user shtyp komanden:

cc foo.c

për të kompiluar programin foo.c dhe jo disa file qe ekzistojne, kompilatori vetëm sa ekziston. Ne përgjithesi proceset interaktive screen-oriented nuk ekzistojne kur jepen parametra të gabuar. Ne vend të tyre shfaqet një box dialogu dhe i kerkon përdoruesit të provoje dhe një here.

Arsyeja e tretë e mbylljes është një error i krijuar nga një proces, zakonisht kjo i takon një gabimi ne program. Shembulli përfshin ekzekutimin e një instruksioni ilegal, duke iu referuar një memorie qe nuk ekziston ose një pjestimi me zero. Ne disa sisteme (për shembull.UNIX), një proces mund ti tregoj sistemit operativ se ai deshiron ti trajtoje vet disa error-e, raste ne të cilat procesi është sinjalizuar (interrupt) ne vend qe të përfundoj kur ndodh gabimi.

Arsyeja e katërt qe një proces të përfundoj është qe një proces ekzekuton një thirrje system duke i thene sistemit operativ qe të vrasi ndonjë proces tjeter. Ne UNIX kjo thirrje është quajtur **kill**. Funksioni WIN32 korespondues është TerminatëProcess. Ne të dyja rastet vrasesi duhet të ketë autorizimin e nevojshem për të kryer vrasjen. Ne disa sisteme, kur një proces përfundon, si ne menyre vullnetare dhe e kunderta, të gjithe proceset qe krijoj vritën menjehere. As unix as windows nuk punojne ne ketë menyre.

2.1.4 Hierarkia e Proçeseve

Ne disa sisteme, kur një proces krijon një proces tjeter, proceset prind dhe femi vazhdojne të kene një fare lidhje ndermjet tyre. Proçesi femi mund të krijoj shume procese të tjera, duke krijuar keshtu një hirarki procesesh. Ndryshe nga bimet dhe kafshet qe përdorin riprodhimin sexual, një proces ka vetëm një prind (por një, dy ose me shume femi).

Ne UNIX, një proces dhe të gjithe femijet dhe pasardhesit e tij të mbledhur se bashku formojne një grup procesi. Kur një user dergon një sinjal nepërmjet tastjeres, sinjali i shpërndahet të gjithe anetareve të grupit të proceseve qe është lidhur me tastjeren (zakonisht të gjithe proceset qe janë krijuar ne dritaren e fundit). Ne menyre individuale çdo proces mund të kapi sinjalin, ta injoroj atë ose të marri veprimin standart, i cili është për tu vrare nga sinjali.

Një tjeter shembull ne të cilin hierarkia e proceseve luan rol është inicializimi i UNIX kur hapet. Një proces special, i quajtur init, është present ne pamjen boot. Kur fillon të ekzekutohet, lexohet një file qe tregon sa terminale ka. Me pas ai vecon nga një proces të ri për çdo terminal. Keto procese presin për dike të hyje ne sistem “log in”. Ne qoftë se hyrja pati sukses, proceset ne hyrje ekzekutojne një shell për të pranuar komandat. Keto komanda mund të vene ne pune shume procese, e keshtu me rradhe. Keshtu qe, të gjithe proceset ne të gjithe sistemin i përkasin një peme të vetme me init ne boot.

Ne kontrast, Windows nuk ka ndonjë koncept për hierarkine e proceseve. Të gjithe proceset jane të barabartë. Vendi i vetëm ne të cilin ka një dicka të ngashme me hierarkine e proceseve është kur krijohet një proçes, prindit i vendoset një shenjë speciale (e quajtur handle) që mund të përdori për të kontrolluar femijen. Gjithsesi, është e lire qe kjo shenjë të përdoret nga procese të tjera, keshtu neutralizohet hierarkia. Proceset ne UNIX nuk mund të përjashtojne femijet e tyre.

2.1.5 Gjendja e Proçeseve

Çdo proçes është një njësi e vecantë, qe ka program counter e tij dhe gjendjen e brendshme, proceset zakonisht kane nevoje të bashkohen me procese të tjera. Disa procese mund të gjenerojne disa output-e qe një proçes tjetër mund ti përdori për input. Ne komandat shell:

```
cat chapter1 chapter2 chapter3 | grep tree
```

proçesi i pare, ekzekuton **cat**, lidh tre filet. Proçesi i dytë, ekzekuton **grep**, selekton të gjithe reshtat qe përbajne fjalen “**tree**”. Ne varesi të shpejtësise relative të dy proceseve (e cila varet nga marëdhenia komplekse e programeve dhe sa kohe të CPU ka patur secilin prej tyre), mund të ndodhi qe **grep** të jetë gati për tu ekzekutuar, por nuk ka input qe presin për të. Ai qendron i bllokuar për sa kohe disa inputë të jene të disponueshme.

Kur një proçes bllokon, ai vepron keshtu sepse logjikisht nuk mund të vazhdoi, ne menyre tipike sepse ai është duke pritur për një input qe nuk është i disponueshem akoma. Gjithashtu është e mundur për proceset qe të jene ne menyre konceptuale gati dhe të jene ne gjendje të ekzekutohen dhe të ndalojne sepse sistemi operativ ka vendosur ti caktoj cpu një procesi tjetër për pak kohe. Keto dy kushte janë plotësisht të ndryshme. Ne rastin e pare përjashtimi është trasheguar ne problem (ju nuk mund të përdorni command line të userit pasi të jetë shtypur). Ne rastin e dytë është një teknike e sistemit (nuk ka CPU të mjaftueshme për ti caktuar çdo procesi një processor privat). Ne fig 2.2 ne shohim një diagrame gjendje qe tregon tre gjendje të proceseve:

1. Ekzekutimi (duke përdorur CPU ne atë moment)
2. Gati (ndalon përkohësisht për të lejuar një proces tjetër të ekzekutohet)
3. Bllokuar (e ndaluar të ekzekutohet derisa të ndodhin disa ngjarje të jashtme)

Llogjikisht, të dy ngjarjet e para jane të gjashme. Ne të dyja rastet CPU është e gatshme për tu ekzekutuar, vetëm e dyta, nuk ka përkohsisht një CPU të disponueshme për të. Gjendja e tretë është e ndryshme nga dy të parat, ne të procesi nuk mund të ekzekutohet edhe pse CPU mund të mos ketë asnjë gje tjetër për të bere.

Për keto tre gjendje jane të mundshme katër tranzacione, sic tregohet:

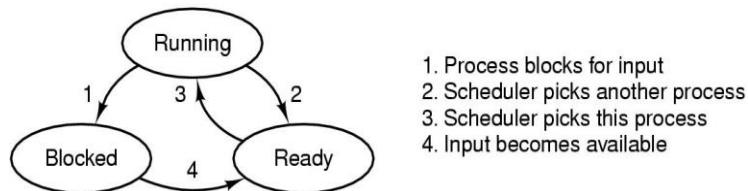


Fig2-2 Tranzicionet ndermjet gjendjeve të proçesit

1. Proçese të bllokuar për input
2. Skeduleri merre një proçes tjetër
3. Skeduleri merre ketë proçes
4. Inputet behen të disponueshme

Tranzicioni 1 ndodh kur një proçes zbulon qe nuk mund të vazhdoi. Ne disa sisteme proçesi mund të ekzekutohet nga një thirrje sistem, sic jane block dhe pause, për të hyre ne gjendjen e bllokuar. Ne sisteme të tjera, duke përfshire UNIX, kur një proçes lexon nga një pipe ose nga një file special (për shembull. një terminal) dhe nuk ka inputë të disponueshme, proçesi është i bllokuar automatikisht.

Tranzicionet 2 dhe 3 Jane shkaktuar nga proçesi i skedulimit, një pjese e sistemit operativ, pa proçesin madje duke ditur mbi të. Trazicioni i dytë ndodh kur skeduleri vendos të ekzekutoj proçese qe kerkojne një kohe të gjatë, dhe është koha për të lejuar një proçes tjetër të përdori CPU-ne. Tranzicioni i tretë ndodh kur proçeset e tjera kane share-uar ne menyre të drejtë dhe është koha qe proçesi i pare të përdori CPU për tu ekzekutuar dhe një here. Subjekti i skedulimit, është qe të vendosi se cili proçes do të ekzekutohet kur dhe për sa kohe, është i rendsishem; për ketë do të flasim me vone ne ketë kapitull. Shume algoritma jane ideuar për të balancuar duke konkuruar kerkesat me efiksiente për sistemet si një i tere dhe për proçeset ne vecanti. Ne do të studiojme disa prej tyre me vone ne ketë kapitull.

Tranzicioni 4 ndodh kur një ngjarje e jashtme për të cilën një proçes ka pritur të ndodhi. Ne qoftë se ne atë moment nuk është duke u ekzekutuar ndonjë proçes tjetër, tranzicioni 3 mund të aktivizohet dhe proçesi mund të filloj ekzekutimin. Ne të kundert do i duhej të

priste ne gjendjen gati për pak kohe derisa CPU të jetë e disponueshme dhe të vij rradha e tij.

Duke përdorur modelin proçes, është me e thjeshtë të kuptohet se cfare ndodh brenda një sistemi. Disa procese ekzekutojne programe qe marrin komanda nga një user. Proçeset të tjera janë pjesë e sistemit dhe handle task sic janë ekzekutimi i kerkesave për sherbimin e file-ve ose menaxhimi i detajeve të ekzekutimit të disqeve ose tape drive. Kur një disk dergon një interrupt, sistemi merr një vendim për të ndaluar ekzekutimin e proçesit qe është duke u ekzekutuar dhe ekzekuton proçesin disk, i cili ishte bllokuar duke pritur për interrupt. Keshtu qe ne vend qe të mendojme rrëth interrupt-ve, ne mund të mendojme rrëth proceseve user, proceseve terminal, proceseve disk e keshtu me rradhe, të cilët bllokohen kur janë duke pritur qe dicka të ndodhi. Kur një disk është lexuar ose karakteri i shtypur, proçesi qe është duke pritur për të është unblock dhe është ne gjendje të ekzekutohet përseri.

Nga kjo pikpamje i jepet një kuptim figures 2-3. Ketu niveli me i ulet i sistemit operativ është skeduleri, me një variacion ne procesesh ne krye të saj. Të gjithe trajtimet dhe detajet e nisjes dhe stopimit të proceseve janë të fshehura ne atë qe quhet skeduler, i cili nuk është ndonjë kod i madh. Pjesa tjetër e sistemit operativ është i strukturuar ne formen e proceseve. Shume pak sisteme reale janë të strukturuar ne ketë forme.

Proçeset				
0	1	...	n-2	n-1
Skeduler				

Fig2-3 Shtresa me e ulet e struktures proçes ne sistemin operativ trajtimi i interrupteve dhe skedulimit. Mbi shtresat jane proçeset sekuenciale.

2.1.6 Implementimi i Proçeseve

Për të implementuar modelin proçes, sistemi operativ përmban një tabele të quajtur **tabela proçes**, me një hyrje për proçes. (Disa autor i quajne keto hyrje **process control blocks**). Keto hyrje përbajne informacione rrëth gjendjes se proceseve, program counter i tij, stack pointer, vendodhjen ne memorie, gjendjen e open file-ve të tij, skedulimi i informacionit dhe çdo gje tjetër rrëth proceseve qe do të ruhen kur proçesi është switch nga gjendjet ekzekutim, gati ose bllokuar ne menyre të tille qe të ristarohet me vone sikur të mos ishte ndaluar asnjeherë.

Figura 2-4 tregon disa nga fushat me të rendesishme të një sistemi tipik.

Menaxhusi I proçeseve	Menaxhusi memories	I	Menaxhusi I file-ve	
Regjistrat			Direktoria root	
Program counter	Pointeri segment	ne	text	Direktoria e punes
Program status word	Pointeri segment	ne	data	File descriptors
Stack pointer	Pointeri segment	ne	stack	User ID
Gjendja proçesit	Pointeri segment	ne	stack	Grup ID
Prioriteti				
Parametrat e skedulimit				
ID proçesit				
Proçesi prind				
Grupi proçesit				
Sinjalet				
Koha kur starton një proçes				
Koha e përdorimit të CPU				
Koha e përdorimit të CPU nga femijet				
Koha e alarmit pasardhes				

Fig 2-4. Disa nga fushat e një tabelle tipike proçes entry

Fusha ne kolonen e pare lidhet me menaxhimin e proçeseve. Dy kolonat e tjera lidhen me menaxhimin e memories dhe menaxhimin e fileve. Duhet të vihet re me saktësi cila fushe

e tabeles se proçeseve ka varesine max nga sistemi, por kjo figure jep një ide të përgjithshme rrreth llojit të informacionit të nevojshem.

Tani le ti hedhim një sy tabeles se proçeseve, është e mundur të shpjegohet pak me shume rrreth iluzionit të shume proçeseve sekuenciale qe përbahen ne një makine me një CPU dhe shume paisje I/O. Lidhur me çdo klase të paisjeve I/O (floppy disk, hard disk, timers, terminale) është një vendodhje e quajtur interrupt vector. Ai përban adresen e procedures se sherbimit të inerrupteve. Supozojme qe proçesi user 3 po ekzekutohet kur ndodh një interrupt nga disku. Program counter i proçesit të 3 user, program status word dhe mundesisht një ose me shume regjistra jane futur ne stack nga një interrupt hardware. Me pas kompjuteri kercen ne adresen e specifikuar ne disk interrupt vector. Kjo është e gjitha qe behet ne hardware. Qe ketej e tutje procedurat e sherbimit të interrupt-it kryhen ne nivelin software.

Të gjithe interrup-et nisin duke ruajtur regjistrat, zakonisht ne tabelen e proçeseve për hyrje të proçesit të fundit. Me pas informacioni i futur ne stack nga interruptet është zhvendosur dhe stack pointeri është vendosur të pointoi ne një stack të përkohshem të përdorur nga menaxhuesi i proçeseve.

Veprime të tilla si ruajtja e regjistrave dhe vendosja e stack pointerit nuk mund të shprehen ne gjuhe të nivelit të lartë sic është C, keshtu ato ekzekutohen nga një gjuhe rutine e nivelit të ulet assembly, zakonisht përdoret e njëjtë gjuhe për të gjithe interruptet për sa kohe ruajtja e regjistrave është identike, nuk ka rendesi nga cfare Jane shkaktuar interruptet.

Kur kjo procedure përfundon, thirret një proçeure C qe kryen të gjithe punen e mbetur për një interrupt të specifikuar. (Ne supozojme qe sistemi operativ është i shkruar ne C, kjo është një zgjidhje e zakonshme për të gjithe sistemet operative reale). Pasi ka kryer detyren e tij, duke bere të mundur qe disa proçese të jene gati, skeduleri thirret për të përcaktuar kush është pasardhesi qe do të ekzekutohet. Pas saj, kontrolli i kalon kodit ne gjuhen assembler për të ngarkuar regjistrat dhe hartën e memories për proçesin aktual dhe të filloj ekzekutimin e tij. Menaxhuesit e interrupteve dhe skedulimi Jane pëmbledhur ne Fig.2-5. Vihet re mire qe detajet ndryshojne disi nga një sistem ne tjetrin.

1. Hardware stacks program counter, etj.
2. Ngarkimi hardware program counter të ri nga interrupt vector.
3. Procedura e gjuhes assembler ruan regjistrat.
4. Procedura e gjuhes assembler vendos stack të reja.
5. Procedura e interrupteve ne C vepron (lexon dhe fut ne buffer input-tët)
6. Skeduleri vendos cili është proçesi pasardhes për tu ekzekutuar.
7. Procedura C i rikthehet kodit assembler.
8. Procedura e gjuhes assembler fillon një proçes të ri.

Fig 2-5. Skeleti i asaj cfare bën niveli i ulet i sistemit operativ kur ndodh një interrupt

2.2 THREADE-t

Ne sistemet tradicionale operative, çdo proces ka një hapesire adresash dhe një thread kontrolli. Megjithatë ka shume situata ku është e nevojshe të ketë shume thread-e kontrolli ne të njëjtën hapesire adresash që ekzekutohen ne menyre gati paralele, si të ishin procese të ndare.

2.2.1 MODELI I THREAD-it

Modeli i procesit sic kemi diskutuar deri tani është i bazuar ne dy koncepte të pavarur: grupimi i burimeve dhe ekzekutimi.

Një menyre për ta pare një proces është ajo e një menyre grupimi burimesh të lidhur me njëri-tjetrin. Një proces ka një hapesire adresash që përmban një program tekst dhe të dhena, si dhe burime të tjera. Keto burime mund të përbajne file, procese femije, alarme, sinjale menaxhimi etj. Duke i vendosur bashke ne formen e një procesi, ato mund të menaxhohen me lehtë.

Koncepti tjetër i një procesi është ai i një thread-i ne ekzekutim. Thread-i ka një program counter qe shenjon ne instrukzionin pasardhes. Ai ka një regjistër i cili mban variablat të cilat po përdoren. Ka një stak i cili mban historine e ekzekutimit. Megjithese thread-i ekzekutohet ne një proces të caktuar, thread-i dhe procesi Jane koncepte të ndryshme dhe duhet të trajtohen vecmas njëri-tjetrit. Proseset përdoren për të grupuar burimet bashke; thread-et Jane entitetet qe përdoren për ekzekutim ne CPU.

Ajo cfare thread-et i shtojne modelit të procesit është fakti se lejojne shume ekzekutime ne të njëjtën hapesire të procesit. Duke pasur shume thread-e qe punojne ne paralel ne një proces është analoge si të kemi shume procese qe punojne ne paralel ne një kompjuter. Thread-et ndajne të njëjtën hapesire adresash, file dhe burime të tjera. Gjithashtu ato ndajne memorjen fizike, disqet, printerat etj. Meqe thread-et kane disa karakteristika si të proceseve ato shpesh quhet procese “**lightweight**”. Termi **multithreading** përdoret për të përshkruar shume thread-e ne një proces të vetëm.

Ne Fig. 2-6(a) shohim tre procese. Secili proces ka hapesiren e tij të adresave dhe një thread të vetëm kontrolli. Ne ndryshim ne Fig. 2-6(b) ne kemi një proces të vetëm me tre thread-e kontrolli. Megjithese ne të tre rastet kemi nga tre thread-e kontrolli, ne Fig 2-6(a) çdo njëri prej tyre vepron ne hapesira të ndryshme adresash, ndersa ne Fig 2-6(b) të tre ndajne të njëjtën hapesire adresash.

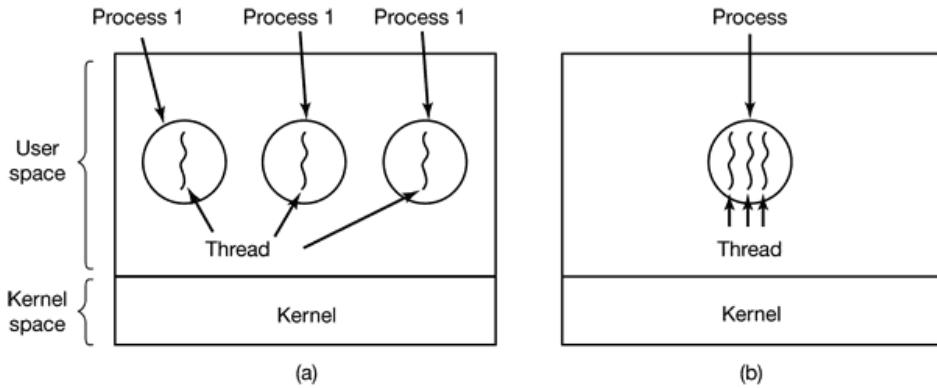


Figura 2-6 (a) Tre procese secili me një thread të vetëm. (b) Një proces me tre thread-e.

Kur një proces me shume thread-e ekzekutohet ne një sistem me një CPU, thread-et ekzekutohen sipas një rradhe të caktuar. Ne Fig. 2-1, ne pame se si funksionontë multiprogramimi i proceseve. Duke kaluar nga njëri proces ne tjetrin me shpejtësi të madhe sistemi krijonte idene e shume proceseve qe ekzekutohen ne paralel. Multithreading punon ne të njëjtën menyre.

Thread-et e ndryshem qe ndodhen ne një proces të vetëm nuk jane aq të pavarur sic mund të ishin dy procese të ndryshme, kjo nenkupton qe ato ndajne të njëjtat variabla globale. Meqe çdo thread mund të aksesoje çdo adresë memorje brenda hapesires se adresave të procesit, njëri thread mund të lexoje, shkruaje ose të fshije fare stakun e një thread-i tjeter. Nuk ka asnë lloj mbrojtje midis thread-eve sepse(1) **është e pamundur** dhe (2) **nuk është e nevojshme**. Ndryshe nga proceset ku secili proces mund të jetë prone e një përdoruesi të ndryshem dhe për ketë arsyet mund të kundershtojen njëri-tjetrin. Një proces i vetëm është gjithnjë prone e një përdoruesi të vetëm i cili mund të ketë krijuar disa thread-e ne të, të cilët me siguri jane krijuar për të bashkepunuar me njëri-tjetrin. Për vec se ndajne të njëjtën hapesire adresash thread-et ndajne dhe të njëjtën file, procese femije, alarme, sinjale etj. sic tregohet ne Fig. 2-7. Prandaj organizimi sipas Fig 2-6(a) do të përdorej kur të tre proceset nuk kane lidhje me njëri tjetrin, ndersa organizimi sipas Fig 2-6(b) do të ishte i përshtashem kur të tre thread-et merreshin me të njëjtën pune.

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figura 2-7. Kolona e pare tregon disa nga gjerat qe ndahen nga thread-et ne një proçes.

Kolona e dytë tregon disa nga gjerat qe jane private për çdo thread.

Si një proçes tradicional (proçeset me një thread), një thread mund të jetë ne një nga gjendjet: duke u ekzekutuar, i bllokuar, gati ose i përfunduar. Një thread ne ekzekutim po përdor CPU dhe është aktiv. Një thread i bllokuar pret për ndonjë ngjarje për ta zhbllokuar. Për shembull, kur një thread kryen një thirrje sistem për të lexuar nga tastiera, ai është i bllokuar derisa të kemi input nga tastiera. Një thread i bllokuar mund të prese për ndonjë ngjarje për tu zhbllokuar ose për ndonjë thread tjetër për ta zhbllokuar. Një thread i gatshem është përzgjedhur për tu ekzekutuar dhe pret derisa ti vij rradha. Tranzicionet midis gjendjeve të thread-eve jane të njëjta si tranzicionet midis proceseve.

Është për tu theksuar se çdo thread ka stakun sic tregohet ne Fig. 2-8. Staku i çdo thread-i përmban një frame për çdo procedure të thirrur por qe akoma nuk është kthyer prej saj. Ky frame përmban variablat lokale të procedures si dhe adresat e kthimit qe do përdoren kur të ketë mbaruar pune procedura. Për shembull, ne qoftë se procedura X therret proceduren Y dhe kjo e fundit therret proceduren Z, ndersa Z po ekzekutohet framet për X, Y dhe Z do të jene të gjitha ne stak. Çdo thread do të therras procedura të ndryshme dhe për rrjedhoje një histori ekzekutimi të ndryshme. Kjo është arsyja pse thread-i do stakun e tij.

Kur kemi rastin e multithreading, proçeset normalisht fillojnë me një thread të vetëm. Ky thread ka aftësine për të krijuar thread-e të tjere duke thirrur një procedure sic është, *thread_create*. Një parametër i *thread_create* zakonisht specifikon emrin e procedures për thread-in e ri qe do të ekzekutohet. Nuk është e nevojshme për të specifikuar asgje rrëth hapesires se adresave të thread-it të ri meqë ai ne menyre automatike ekzekutohet ne hapesiren e thread-it qe e krijoi. Ndonjehere thread-et kane një lidhje hierarkike, **lidhja prind-femije**, por shpesh kjo lloj lidhje nuk ekziston, të gjithe thread-et Jane të barabartë. Me ose pa një lidhje hierarkike thread-i krijues kthehet ne një thread identifikues i cili i ve emer thread-it të ri.

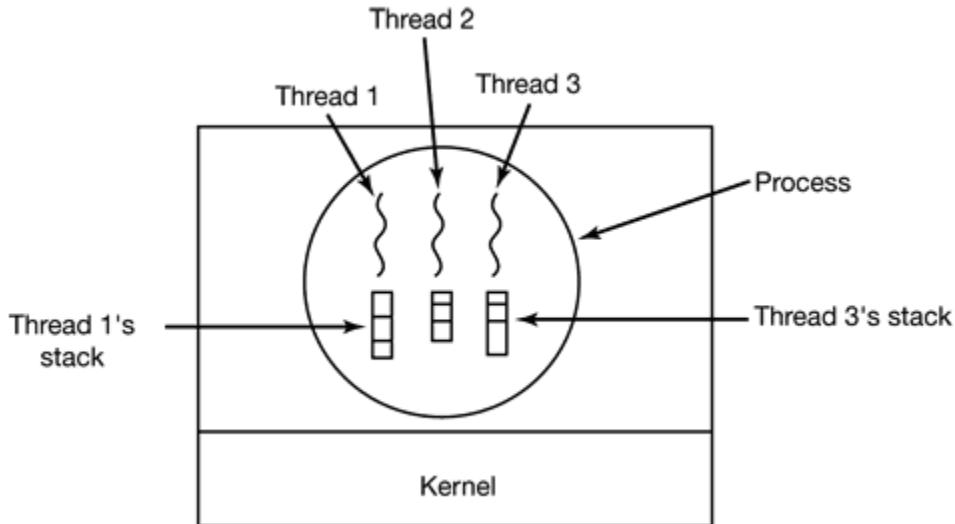


Figura 2-8 Çdo thread ka stakun e tij.

Kur një thread ka përfunduar punen e tij, ai mund të therrase një procedure, *thread_exit*. Ai me pas zhduket dhe nuk mund të përzgjidhet me. Ne disa sisteme thread-esh, një thread mund të prese për një thread tjetër pér të përfunduar, duke thirrur një procedure per shembull, *thread_wait*. Kjo procedure bllokon threadin qe e ka thirrur derisa një thread tjetër ka përfunduar. Ne ketë kendveshtrim krijimi dhe përfundimi i thread-eve është i ngjashem me krijimin dhe përfundimin e proceseve.

Një tjetër thirrje thread është edhe *thread_yield*, e cila e lejon një thread qe të lere CPU-ne ne menyre vullnetare pér të lejuar një thread tjetër pér tu ekzekutuar. Kjo lloj thirrje është e rendesishem pasi nuk kemi **timesharing** të detyruar i cili gjeneron interrupte qe krijojnë vonesa. Prandaj është e rendesishem pér thread-et qe ta lene ne menyre vullnetare CPU-ne pér ti dhene shansin thread-eve të tjere pér tu ekzekutuar. Disa thirrje të tjera lejojne një thread pér të pritur pér një thread tjetër pér të përfunduar punen, lejojne një thread pér të thene qe ai ka mbaruar një pune etj.

Nga sa pame duket se thread-et jane shume të dobishem, por ato shfaqin shume probleme ne modelin e programimit. Konsideroni efektët e thirrjes sistem ne UNIX **fork**. Ne se procesi prind ka shume thread-e a duhet ti ketë dhe femija? Megjithatë, ne qoftë se procesi femije merr po aq thread-e sa prindi i tij, cfare ndodh ne se një thread tek prindi ishte i bllokuar ne një thirrje leximi, pér shembull nga tastiera? A Jane tani dy thread-e të bllokuar pér leximin nga tastiera? Kur një rresht shkruhet a marrin të dy thread-et një kopje të saj? Apo merr vetëm prindi? Apo vetëm femija?

Një tjetër kategori problemesh lidhet me faktin se thread-et ndajne midis tyre shume struktura të dhenash. Cfare ndodh ne se një thread mbyll një file ndersa tjetri është duke e lexuar? Supozojme se një thread ve re se ka shume pak memorje dhe fillon të alokoje me shume memorje. Por ndodh qe të kemi një nderrim midis thread-eve dhe për pasoje, edhe thread i ri ve re se ka pak memorje dhe fillon të alokoje dhe ai me shume memorje. Memorja do të alokohet dy here. Keto probleme dhe shume të tjere duhen zgjidhur duke zgjedhur menyren e duhur të programimit të thread-eve dhe të dizjenjimit të tyre, ne menyre qe programet qe përdorin multithreading të punojne ne menyre korrekte.

2.2.2 PËRDORIMI I THREAD-eve

Pasi kemi shpjeguar se cfare jane thread-et, është koha për të shpjeguar pse duhen përdorur.

Arsyeja kryesore pse përdorim thread-et është se ne shume aplikacione, aktivitetët brenda tyre ekzekutohen ne cast. Disa nga keto aktivitetë mund të bllokohen here pas here. Duke e ndare një aplikacion të tille ne një bashkesi thread-esh qe ekzekutohen ne një menyre paralele, modeli i programimit behet me i thjeshtë.

Cfare thame me sipër e kemi pare edhe me pare. Është e njëjta arsyе pse kemi procese. Ne vend qe të mendojme për interrupte, tajmera etj ne mund të mendojme për procese qe punojne ne paralel. Vetëm se tani me thread-et kemi dicka të re: aftësine e një entiteti për të ndare një hapesire të caktuar adresash me entitetë të tjere, si dhe të gjithe të dhenat ne ketë hapsire adresash. Kjo aftësi është e rendesishme për disa lloje të caktuara aplikacionesh, kjo është arsyе se duke pasur shume procese (me hapesire adresash të ndare) nuk do të funksionoje.

Një argument i dytë pse kemi thread-e është se meqë nuk ka asnë lloj burimi të lidhur me to, Jane me të lehtë për tu krijuar dhe me pas shkatërruar se proceset. Ne shume sisteme, krijimi i një thread-i është 100 here me i shpejtë se krijimi i një procesi.

Argumenti i tretë pse kemi thread-e ka të beje me përfomancen. Ato nuk e rrisin përfomancen ne rast se ndodhen afer CPU (sepse rrëth CPU veprohet me shpejtësine e saj). Prandaj përdorimi i thread-ve është i ndjeshem ne rritjen e performancës se aplikacioneve kur përdoren për shembull ne pajisjet I/O.

Se fundmi, thread-et Jane mjaft të vlefshem ne sistemet me shume procesore ku mund të flitet për një paralelizem real.

Është me e lehtë për të pare se pse thread-et Jane të vlefshem duke dhene disa shembuj. Si shembull të pare supozoni një editor teksti. Shume editor e paraqesin dokumentin e tyre ne të njëjtën menyre se si do të dukej kur ai të printohej. Ne vecanti të gjithe ndarjet e rrjeshtave dhe ndarjet e faqeve Jane ne pozicionin e tyre të duhur ne menyre qe

përdoruesi mund ti shohe dhe ta ndryshoje dokumentin ne se duhet. Supozojme se përdoruesi po shkruan një liber. Nga pikepmja e autorit është me e thjeshtë për ta pasur librin si një file të vetëm, ne menyre qe modifikimet të behen me lehtë. Ne një rast tjetër mundet qe çdo kapitull të jetë një file me vetë. Por mund të kemi edhe rastin qe çdo seksion mund të jetë një file me vetë, ne ketë rast ndryshimet do të ishin shume të veshtira pasi do të ishin shume file për tu ndryshuar.

Tani konsideroni rastin kur një përdorues fshin një fjali nga faqja 1 e një dokumenti me 800 faqe. Pasi ka pare ndryshimin e bere përdoruesi do të bej një tjetër ndryshim ne faqen 600 dhe shkruan një komand ku i kerkon editorit të tekstit të shkoje ne atë faqe. Editori i tekshit duhet të riformatoje të gjithe librin deri tek faqja 600, pasi ai nuk e di se cili është rreshti i pare tek faqja 600, pa i pare të gjithe faqet deri tek 600. Mund të ketë një vone se deri sa të shfaqet faqa 600, kjo mund të sjelli qe përdoruesi të jetë i pakenaqur me sherbin min e ofruar.

Ne ketë pike thread-et mund të ndihmojne. Supozojme se editori i tekstit është i konceptuar si një program me dy thread-e. Njëri ndervepron me përdoruesin, ndersa tjetri merret me riformatimin e dokumentit. Ne momentin qe një rresht është fshire ne faqen e pare, thread-i qe ndervepron me përdoruesit i thotë thread-it tjetër të riformatoje të gjithe librin. Me pak fat riformatimi mund të behet me pare se përdoruesi të kerkonte të shohe faqen 600, ne menyre qe ajo të shfaqet menjehere kur të kerkohet nga përdoruesi.

Meqe jemi pse mos të shtojme një thread të tretë? Shume editore teksti kane aftësinë e ruajtjes se informacionit ne disk çdo disa minuta ne menyre automatike ne menyre qe ta mbrojne përdoruesin qe ta ruaje punen e tij dhe të mos rrezikohet nga një problem i sistemit, mungese energjie elektrike etj. Thread-i i tretë mund të merret me backup-et e diskut pa nderhyre tek dy thread-et e tjere. Situata me tre thread-e jezet ne Fig. 2-9.

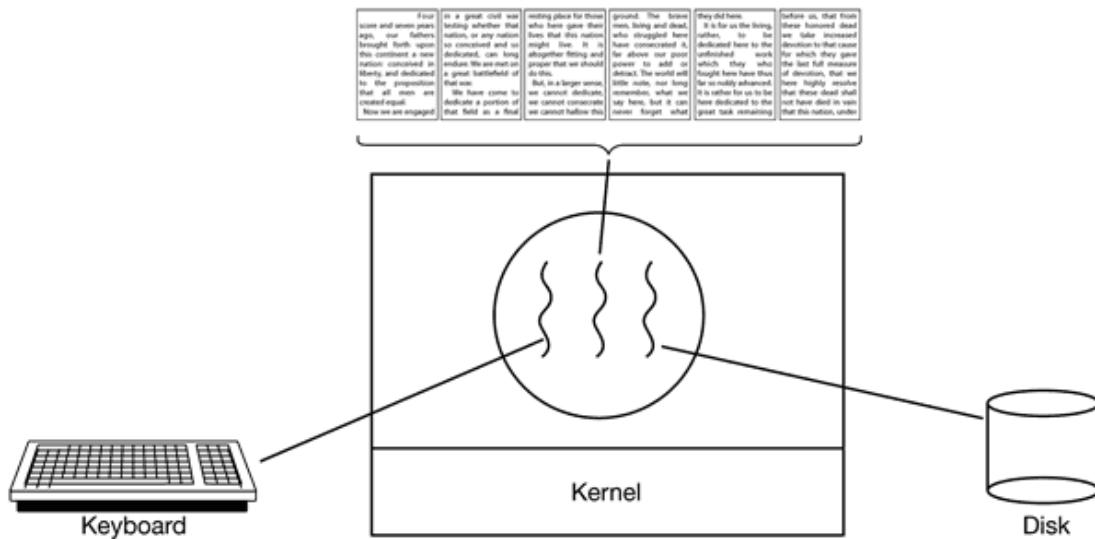


Figura 2-9 Një editor teksti me tre thread-e.

Ne se programi do të ishte me një thread të vetëm, atëherë sa here do të behej një backup ne disk, komandat nga tastiera apo mouse do të injoroheshin derisa të përfundonte backup-i. Me tre thread-e modeli i programit është me i thjeshtë. Thread-i i pare ndervepron me përdoruesin. I dyti riformaton dokumentin kur i kerkohet. Ndersa i treti ruan dokumentin ne menyre periodikë ne disk.

Duhet të jetë e qartë se ne qoftë se kemi tre procese të ndare nuk do të funksionontë, sepse duhet qe të tre thread-et të punojne ne të njëjtin dokument. Duke pasur tre thread-e ne vend të tre proceseve, ato ndajne të njëjtën memorje dhe keshtu kane akses ne dokumentin qe po editohet.

Le të shohim një shembull tjetër ku thread-et jane të dobishem: një server për site World Wide Web. Kerkesat për faqe shkojne ne server dhe faqet e kerkuara i dergohen klientit. Ne shume site disa faqe jane me të kerkuara se faqet e tjera. Web serverat e përdorin ketë fakt për të përmiresuar performancen duke mbajtur ato faqe qe aksesohen me shume ne memorjen kryesore duke mos pasur nevoje të kontrolloje diskun kur kerkohen keto faqe.

Një menyre për të organizuar një Web server jepet ne Fig. 2-10(a). Ketu njëri thread, dispatcher-i, lexon kerkesat qe vine nga rrjeti. Pasi ekzaminon kerkesen ai përzgjedh një thread “*punetor*” i cili është ne gjendje të bllokuar, e kalon ne gjendjen gati dhe i jep atij kerkesen.

Kur thread-i merr kerkesen ai sheh ne memorjen kryesore, ne të cilën të gjithe thread-et kane akses, ne se faqja e kerkuar ndodhet aty. Ne se jo ai starton një opération leximi për ta marre faqen nga disku dhe bllokohet derisa kerkesa e tij të plotësohet. Kur thread-i bllokohet gjatë kohes se leximit të diskut, një thread tjetër zgjidhet për tu ekzekutuar, me shume mundesi dispatcher-i.

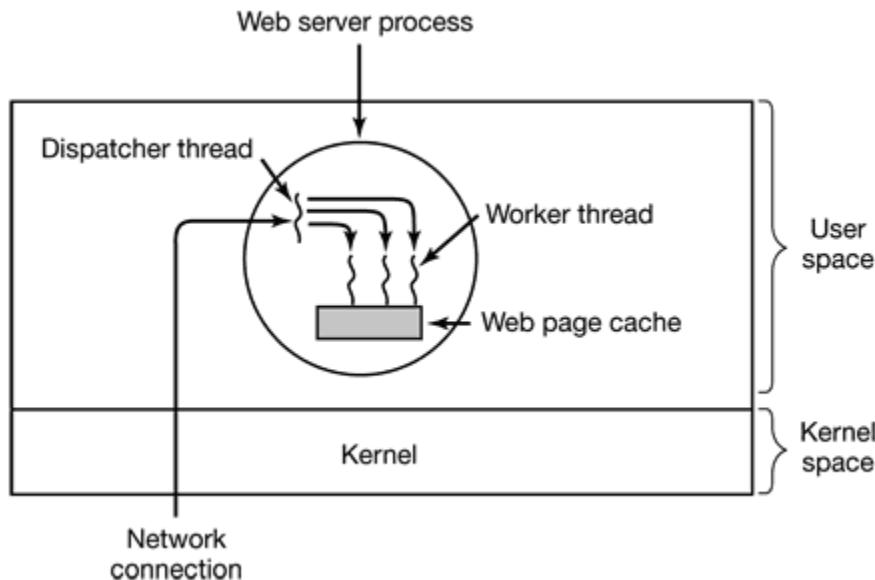


Figura 2-10. Një Web server multithread.

Ky model e lejon serverin të shkruhet si një grup thread-esh sekuencial. Programi i dispatcher-it konsiston ne një cikel të pafundem për të marre një kerkese dhe për t’ia

deleguar një thread-i tjetër. Kodi i ketyre thread-eve konsiston ne një cikel të pafundem pér të marre një kerkese nga dispatcher-i dhe pér të pare ne se faqja ndodhet ne memorjen kryesore. Ne se po, faqja i dergohet klientit, thread-i bllokohet dhe pret pér një kerkese tjetër. Ne se jo, thread-i e merr faqen nga disku, ia dergon klientit dhe me pas bllokohet ne pritje të një kerkese tjetër.

Ne vija të trasha kodi i përdorur jepet ne Fig.2-11. Ketu, si ne të gjithe librin, *TRUE* përfaqeson konstantën 1. Gjithashtu, *buf* dhe *page* Jane struktura të përshtatshme pér të mbajtur respektivisht një kerkese dhe një faqe Web-i.

```

while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf,
    &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf,
        &page);
    return page(&page);
}

```

(a) (b)

Figura 2-11. Një linjë kodi pér Fig. 2-10. (a) Thread-i dispatcher. (b) Thread-i “punetor”.

Mendoni se si mund të punonte Web serveri ne mungese të thread-eve. Një mundesi është qe ai të punonte si një thread i vetëm. Cikli kryesor i Web serverit merr një kerkese, e ekzaminon, e plotëson kerkesen por pa qene i aftë të marre një kerkese tjetër gjatë kesaj kohe. Kjo gje con ne një ulje të performancës.

Deri tani kemi pare një dy mundesi dizenjimi: një Web server multithread dhe një Web server i trajtuar si një thread i vetëm. Supozojme se kur një kerkese vjen një thread i vetëm e ekzaminon atë, ne se faqja e kerkuar ndodhet ne memorjen kryesore ai ia dergon klientit, ne se jo nis një opération leximi disku pa e bllokuar thread-in.

Serveri ruan gjendjen e kerkesen ne një tabele dhe me pas shkon dhe trajton ngjarjen tjetër. Ngjarja tjetër mund të jetë një kerkese pér një pune të re ose një përgjigje nga disku pér një opération të meparshem. Ne se është një pune e re, puna fillon. Ne se është një përgjigje nga disku, atëherë informacioni përkatës merret nga tabela dhe përgjigja vazhdon.

Ne ketë lloj menyre konceptimi, modeli sekuencial i proceseve qe kishim ne dy rastet e pare humbet. Gjendja e përpunimit të kerkesave ruhet ne menyre eksplicitë ne tabele sa here qe serveri kalon nga një kerkese ne tjetren. Me fjale të tjera ne po simulojme thread-et dhe staket e tyre. Një dizenjim i tille ne të cilën çdo veprim ka një gjendje e cila ruhet dhe përashton disa ngjarje qe mund të ndryshojne gjendjen quhet **finitë-state-machine**.

Tani duhet të jetë e qartë ajo se cfare mund të ofrojne thread-et. Ato bejne të mundur ruajtjen e idese se proceseve sekuencial qe rrisin dukshem pëformacen. Serverat me një thread të vetëm përdorin thirrjet bllokuese po lene pér të deshiruar ne performance.

Menyra e tretë arrin performance të lartë nepërmjet paralelizmit por përdor thirrje joblokuese dhe interrupte, por jane të veshtire për tu programuar. Keto modele përbidhen ne Fig. 2-12.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figura 2-12. Tre menyrat për të ndertuar një server.

Një shembull i tretë ne të cilën thread-et janë të dobishem janë aplikacionet qe do përpunojne sasi të madha të dhenash. Menyra me e thjeshtë është të lexosh ne një bllok të dhenash, ta përpunosh atë e me pas ta shkruash serisht. Problemi ketu qendron ne faktin se ne se vetëm thirrjet bllokuese janë të vlefshme, procesi bllokohet nderkohe qe të dhenat shkojnë e vine. Ne ketë menyre CPU mbetet pa pune nderkohe qe ka shume të dhena qe duhen përpunuuar. Kjo menyre nuk është e leverdiseshme prandaj duhet evituar ne se është e mundur.

Thread-et ofrojne një zgjidhje. Procesi strukturohet me një thread ne hyrje, një thread përpunues dhe një thread ne dalje. Thread-i i hyrjes lexon të dhenat nga një buffer i hyrjes. Thread-i përpunues merr të dhenat nga bufferi i hyrjes, i përpunon dhe vendos rezultatin ne një buffer dales. Bufferi i daljes i shkruan keto rezultate ne disk. Ne ketë menyre, hyrja, dalja, dhe procesimi behen ne të njëjtën kohe.

2.2.3 IMPLEMENTIMI I THREAD-eve NE USER SPACE

Ka dy menyra për të implementuar paketat e thread-eve: **ne user space dhe ne kernel**. Gjithashtu edhe një implementim hibrid është i mundshem. Tani do përshkruajme keto metoda, me avantazhet dhe disavantazhet e tyre.

Metoda e pare është vendosja e të gjithe paketës se thread-eve ne user space. Kerneli nuk ka asnjë informacion rreth tyre. Avantazhi i pare dhe me i dukshem është se ne nivelin user mund të implementohet paketa e thread-eve edhe ne ato lloj sistemesh operative qe nuk i suportojne thread-et.

Të gjitha keto implementime kane të njëjtën strukturre e cila ilustrohet ne Fig. 2-13(a). Thread-et ekzekutohen ne pjesen e sipërme të një **sistemi run-time**, i cili është një koleksion procedurash qe menaxhojnë thread-et. Ne kemi pare katër prej tyre: *thread_create, thread_exit, thread_wait, thread_yield*, por zakonisht ka me shume.

Kur thread-et menaxhohen ne user space, çdo procesi i duhet tabela e tij e thread-eve ne menyre qe kontrolloje keta të fundit. Kjo tabele është analoge me tabelen e proceseve qe

ndodhet ne kernel, ajo kontrollon program counter-in, stak pointerin, registrat, gjendjen, etj. Tabela e thread-eve menaxhohet nga sistemi run-time. Kur një thread kalon ne gjendjen gati ose të bllokuar, informacioni qe duhet për ta ristaruar ndodhet ne tabelen e thread-eve, ne të njëjtën menyre se si kerneli ruan informacionin për procesin ne tabelen e proceseve.

Ne se thread-i bën dicka qe mund ta bllokoje ai therret një procedure të sistemit run-time. Kjo procedure sheh ne se thread-i duhet të vendoset ne gjendjen e bllokuar. Ne se po, ajo ruan regjistrin e thread-it ne tabelen e thread-it, shikon ne tabele për një thread të gatshem për tu ekzekutuar dhe i ringarkon regjistrat e makines me vlerat e reja të thread-it. Derisa stak pointeri dhe program counteri të jene nderruar, thread-i i ri vjen ne jetë ne menyre automatike. Ne se makina ka një instruksion për të ruajtur të gjithe regjistrat dhe një tjetër për ti ngarkuar përseri të gjithe, i gjithe nderrimi i thread-eve mund të behet me pak instruksione. Berja e nderrimit të thread-eve ne ketë menyre është shume here me e shpejtë se sa do të ishte ne se threade-et do të ndodheshin ne kernel dhe është një argument shume i fortë ne favor të paketave të threade-eve ne user space.

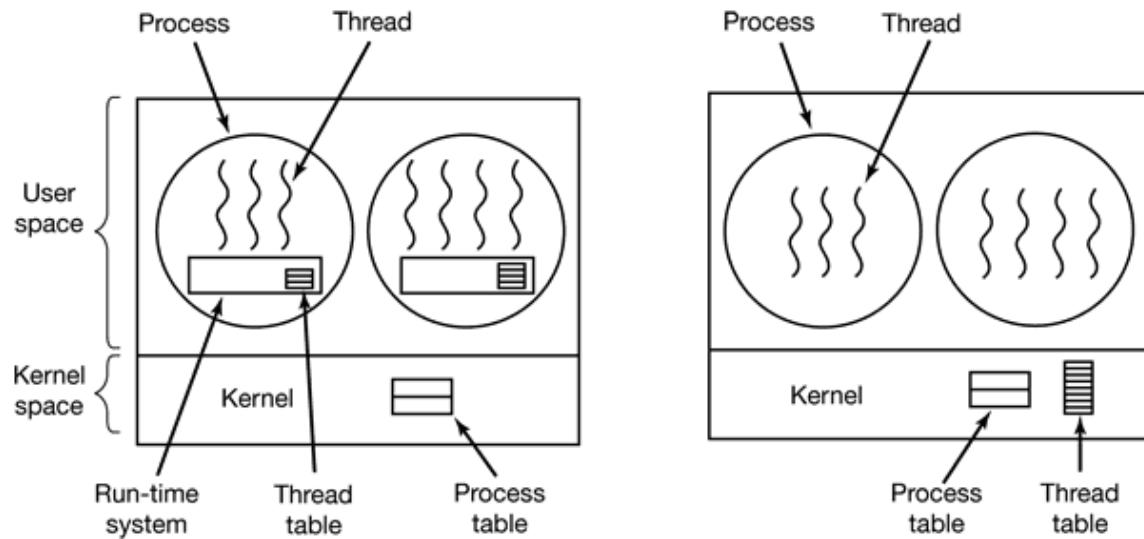


Figura 2-13. (a) Paketa e thread-ve ne user space.(b) Një paketë thread-esh e menaxhuar nga kerneli.

Megjithatë ka një ndryshim esencial me proceset. Kur një thread mbaron se ekzekutuari, per shembull, kur therret proceduren *thread_yield*, kodi i *thread_yield* ruan informacionin e thread-it ne tabelen e thread-eve. Me pas mund të therrase thread-in *scheduler* ne menyre qe të përzgjedhe një thread tjetër për tu ekzekutuar. Kur një thread mbaron se ekzekutuari, për shembull kur therret proceduren *thread_yield*, kodi i *thread_yield* ruan informacionin e thread-it ne tabelen e thread-eve.

Procedurat qe ruajne gjendjen e thread-it dhe schedulerin jane procedura lokale, prandaj thirrja e tyre është me eficiente se sa të besh një thirrje kernel. Pra nuk ekzekutohet asnjë instruksion “***trap***” . Kjo e bën përgjedhjen e thread-eve shume të shpejtë.

Thread-et ne nivelin user kane edhe disa avantazhe të tjere. Ato lejojne çdo të proces të ketë algoritmin e tij të schedulimit të përshtatur sipas nevojave të tyre. Për disa aplikacione fakti se nuk mund të shqetësohen ne se një thread ndalohet ne një moment të caktuar është një plus.

Duke lene menjane performancen e tyre të mire, thread-et ne nivelin e userit kane disa probleme të medha. Se pari, problem është implementimi i thirrjeve bllokuese të sistemit. Supozoni se një thread lexon nga tastiera me pare se një tast është shtypur. Është e palejueshme ne se e lejojme thread-in të beje një thirrje sistem, pasi do të kishim një bllokim të thread-eve të tjere.

Thirrjet sistem duhet të jene të gjitha jo-bllokuese, por kjo kerkon ndryshim ne sistemin operativ. Pastaj një nga argumentët qe përdoreshin thread-et ne user space ishte fakti se ato implementohen ne sistemet operative qe ekzistojne, dhe jo ne to qe duhen modifikuar. Plus ne rast se ndryshohet sistemi operativ duhen ndryshuar edhe shume programe të përdoruesve.

Një alternative tjetër është e mundshme. Ne se arrijme qe të përcaktojme me pare ne se një thirrje do bllokoje ose jo. Ne disa versione të UNIX, një thirrje sistem përgjedh daljet (*select*), kjo lejon thirresin të tregoje ne se një kerkese për lexim do bllokohet ose jo. Kur behet kjo thirrje, procedura *read* e librarise mund të zevendesohet me një të re qe ne fillim bën një thirrje *select*, dhe me pas lexon ne se është e sigurtë. Kjo gje kerkon rishkrimin e një pjese të librarise se thirrjeve sistem. Kodi i vendosur rrëth thirrjes sistem për të bere kontrollimin quhet **jacket** ose **wrapper**.

Pak a shume i ngjashem me problemin e thirrjeve bllokuese të sistemit është edhe ai i *page fault*. Pra kur faqja e kerkuar duhet të merret nga disku. Proçesi bllokohet ndersa faqja e kerkuar gjendet dhe lexohet. Ne se një thread shkakton një page fault, kerneli, edhe pse nuk di gje për ekzistencën e thread-it, bllokon të gjithe procesin derisa kerkesa për leximin nga disku të kompletuhet.

Një tjetër problem qe haset me thread-et qe implementohen ne user space: ne se një thread fillon të ekzekutohet,asnjë thread tjetër ne atë proces nuk do të ekzekutohet ne se thread-i i pare nuk e le ne menyre vullnetare CPU-ne. Brenda një proçesi të vetëm, nuk ka interrupte clock-u, duke bere të mundur përgjedhjen e proceseve me rradhe (round robin).

Një mundesi zgjidhje për thread-et qe ekzekutohen pafundesisht është duke bere sistemit run-time të kerkoje një sinjal clock-u (interrupt) çdo cast të caktuar kohe për të dhene kontrollin, por edhe kjo menyre është e veshtire për tu programuar. Interruptet periodikë ne një frekuencë të lartë nuk Jane gjithmone të mundeshme, edhe ne se do të ishin, overhead-i total do të ishte i konsiderueshem. Por edhe thread-i mund të doje një interrupt clock-u qe mund të intérferoje me përdorimin e sistemit run-time të clock-ut.

Argumenti kryesor kunder thread-eve ne user space është se programuesit zakonisht i duan thread-et pikerisht ne aplikacionet ku thread-et bllokohen shpesh, për shembull ne një Web server. Keto thread-e bejne thirrje sistem të vazhdueshme. Kur ndodh një instruksion “**trap**” ne kernel për tu marre me thirrjen sistem, kerneli merret me nderrimin e thread-eve, ne se thread-i me i vjetër është i bllokuar, ne se lejohet kerneli për të bere ketë gje eliminohet nevoja për të bere thirrje sistem *select*, të cilat kontrollojnë ne se thirrjet sistem për lexim jane të sigurta ose jo. Për aplikacione qe zhvillohen rrëth CPU-se dhe qe bllokohen shume rralle nuk është e nevojshme përdorimi i thread-eve.

2.2.4 IMPLEMENTIMI I THREAD-eVE NE KERNEL

Tani le të mendojme se për menaxhimin e thread-eve merret kerneli. Nuk duhet asnjë lloj sistemi run-time sic tregohet ne Fig. 2-13(b). Gjithashtu nuk ka asnjë lloj tabele thread-esh ne çdo proces. Ne vend të kesaj, kerneli ka një tabele thread-esh me ane të të cilit kontrollon të gjithe thread-et ne sistem. Kur një thread kerkon të krijoje një thread të ri ose të shkatërrroje një të vjetër, ai bën një thirrje ne kernel e cila ekzekuton kerkesen e thread-it duke ndryshuar tabelen e thread-eve.

Tabela e thread-eve qe ndodhet ne kernel mban registrat, gjendjen dhe informacione të tjera ne lidhje me thread-et. Ky infomacion është një shtese e informacionit qe kernelat tradicionale mbajne ne lidhje me proceset me një thread të vetëm.

Të gjitha thirrjet qe mund të bllokojne një thread jane implementuar si thirrje sistem. Kur një thread bllokohet, kerneli mund të ekzekutoje ose një thread tjetër nga i njëjti proces, ose një thread nga një proces tjetër. Me thread-et ne user space sistemi run-time ekzekutontë vetëm thread-e nga i njëjti proces derisa kerneli i merrte përdorimin e CPU-së.

Për shkak të kostos se lartë të krijimit dhe të shkatërrimit të thread-eve ne kernel, disa sisteme i “*riciklone*” thread-et e tyre. Kur një thread shkatërrohet, ai shenjohet si i paekzekutueshem, por struktura e tij e të dhenave qe ndodhet ne kernel nuk preket. Me vone, kur një thread i ri duhet të krijohet, një i vjetër riaktivizohet, duke mos harxhuar shume kohe. Riciklimi i thread-eve është gjithashtu i mundshem për thread-et ne user space, por meqë koha e cila fitohet është me e vogel, berja e riciklimit nuk është efektive.

Thread-et ne kernel nuk kerkojne thirrje sistem jo-bllokuese. Ne se një thread ne një proces shkakton një **page fault**, kerneli e ka të thjeshtë të shohe ne se procesi ka ndonjë thread tjetër të ekzekutueshem, ne se jo faqja merret nga disku. Disavantazhi kryesor është se kostoja e thirrjeve sistem është e konsiderueshme, overhead-i është i madh.

2.2.5 IMPLEMENTIMET HIBRIDE

Jane pare shume menyra për të kombinuar avantazhet e të dy menyrate për të implementuar thread-et. Një menyre është përdorimi i thread-eve ne kernel e me pas multipleksimi i thread-eve ne user space ne disa ose ne të gjithe thread-et kernel sic tregohet ne Fig.2-14.

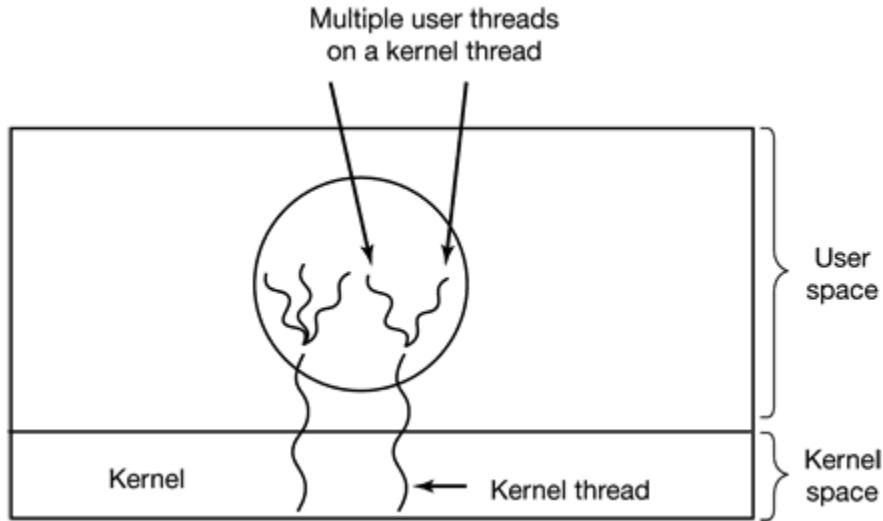


Figura 2-14 Multipleksimi i thread-eve të user space ne thread-et kernel.

Ne ketë dizenjim, kerneli është i vetëdijshem vetëm për thread-et kernel dhe i përzgjedh ato. Disa prej tyre mund të kene shume thread-e user të multipleksuara, keto thread-e (user) ekzekutohen sipas një rradhe të caktuar.

2.2.6 AKTIVIZIMET E SCHEDULERIT

Shume kerkues jane përpjekur për të kombinuar avantazhet e thread-eve ne nivelin user me avantazhet e thread-eve ne nivelin kernel. Me poshtë do të shohim një përpjekje për të bere një gje të tille dhe quhet *aktivizimet e schedulerit*.

Qellimet e aktivizimit të schedulerit Jane të imitojne funksionet e thread-eve kernel por me përformace me të mire dhe fleksibilitët me të lartë, zakonisht me paketat e thread-eve të implementuara ne user space. Ne vecanti, thread-et user nuk duhen të bejne thirrje blokuese speciale apo të shohe ne se është e sigurtë ne se bejme një thirrje sistem apo jo.

Eficencia arrihet ne se shmanget tranzicionet e panevojshme midis hapesires user dhe kernel. Sistemi run-time i user space mund t^e bllokoje thread-in sinkron dhe t^e përzgjedhe një t^e ri nga ana e tij.

Kur përdoren aktivizimet e schedulerit, kerneli i cakton një numer t^e caktuar procesoresh virtual p^r çdo proces dhe lejon sistemin run-time qe t^e alokoje thread-e p^r procesoret. Ky mekanizem mund t^e përdoret edhe ne sistemet me shume procesore ku procesoret virtuale mund t^e jene procesore reale. Numri i procesoreve virtuale qe i caktohet një procesi si fillim është një, por proceset mund t^e kerkojne p^r me shume ose ti lene procesoret ne se nuk i duhen me. Kerneli mund ti marre keto procesore virtuale qe jane alokuar njehere dhe ti caktoje ne procese t^e tjrrera qe kane nevoje p^r to.

Ideja kryesore qe e bën ketë skeme t^e funksionoje është se ne rastin kur kerneli kupton qe një thread është bllokuar, ai njofton sistemin run-time, duke kaluar si parametra ne stak numrin e thread-it ne fjale dhe një përshkrim p^r ngjarjen qe ka ndodhur. Njohja me atë qe ka ndodhur behet nga kerneli duke aktivizuar sistemin run-time ne një adresë t^e njohur

Ky mekanizem quhet **upcall**.

Njehere i aktivizuar, sistemi run-time mund ti rizgjedhe thread-et e tij, zakonisht duke vecuar thread-in e bllokuar dhe me pas duke marre një thread tjetër qe është ne gjendjen gati. Me vone, kur kerneli kupton se thread-i original mund t^e ekzekutohet përseri ai bën një tjetër **upcall** sistemit run-time p^r ta informuar p^r ketë ngjarje. Sistemi run-time, sipas vleresimit t^e tij, mund ose jo ta ristaroje menjehere thread-in ose e vendos ne listën e thread-eve t^e gatshem, p^r tu aktivizuar me vone.

Kur ndodh një interrupt hardware, kur një thread user është duke u ekzekutuar, CPU-ja e nderprere kalon ne kernel mode. Ne se interrupti shkaktohet nga një ngjarje qe nuk ka lidhje me procesin, kur menaxhuesi i interrupteve përfundon punen e tij, ai vendos thread-in e nderprere ne gjendjen qe ai ishte me pare. Por ne rast se interrupti qe ndodh ka lidhje me procesin qe është nderprere, thread-i i nderprere nuk ristartohet. Thread-i i nderprere pezullohet dhe sistemi run-time qe ishte startuar ne atë CPU vituale. Është me pas ne dore t^e sistemit run-time qe t^e përzgjedhe se cilin thread t^e përzgjedhe ne atë CPU: t^e nderprerin, një thread t^e ri t^e gatshem p^r ekzekutim apo ndonjë zgjedhje t^e tretë.

Një përashtim p^r aktivizimet e thread-eve është siguria rreth upcall-eve, një koncept qe prish strukturen qe trashegohet ne çdo sistem me shtresa. Normalisht, shtresa n ofron disa sherbime qe shtresa $n+1$ mund ti therrase, por shtresa n ndoshta nuk mund t^e therrase procedura nga shtresa $n+1$. Upcall nuk e ndjekin ketë princip t^e rendesishem.

2.2.7 THREAD-et POP-UP

Thread-et jane t^e vlefshem ne sistemet e shpërndare. Një shembull i rendesishem është se si kerkesat p^r sherbit menaxhohen. Menyra tradicionale është qe t^e bllokojme një

proçes ose një thread me një thirrje sistem **receive** qe pret për një mesazh. Kur mesazhi vjen ajo e pranon mesazhin dhe e përpunon kerkesen e tij.

Megjithatë, një tjetër menyre është e mundshme, ne të cilën ardhja e një mesazhi bën të mundur qe sistemi të krijoje një thread tjetër qe të menaxhoje mesazhin. Një thread i tillë quhet pop-up thread dhe ilustrohet ne Fig. 2-15. Një avantazh i madh i thread-eve pop-up është se meq Jane të ri, ato nuk kane histori, regjistra, stak, etj qe duhen të kthehen ne gjendje fillestare. Secili fillon si i ri dhe Jane identik midis tyre. Pra si përfundim përdorimi i tyre bën qe vonesa midis ardhjes se mesazhit dhe fillimit të përpunimit të tij mund të jetë shume e shkurtër.

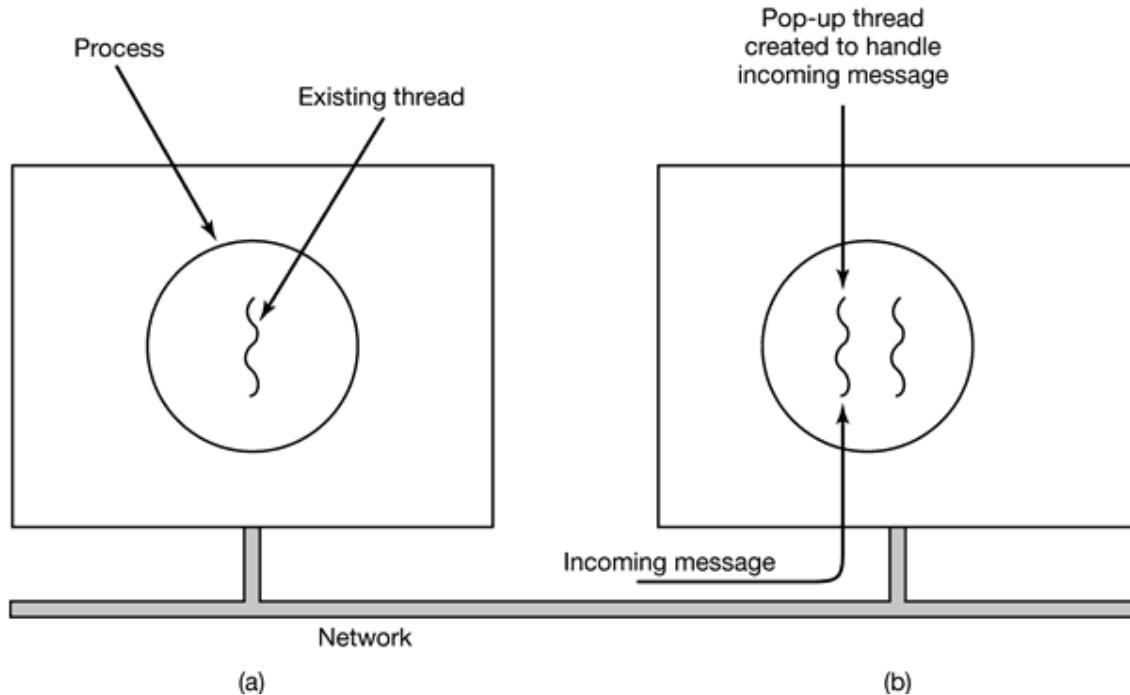


Figura 2-15. Krijimi i një thread të ri kur vjen një mesazh. (a) Para ardhjes se mesazhit. (b) Pas ardhjes se mesazhit.

Disa planifikime paraprake duhen bere kur përdoren thread-ot pop-up. Për shembull, ne cilin proçes ekzekutohet thread-i? Ne se sistemi e lejon qe thread-i të ekzekutohet ne kernel, thread-i mund të ekzekutohet aty. Duke patur thread-in pop-up qe ekzekutohet ne kernel është me i shpejtë dhe me i lehtë se ekzekutimi ne user space. Gjithashtu një thread pop-up ne kernel space mund të aksesoje me lehtësi të gjitha tabelat e kernelit dhe pajisjet I/O, qe mund të duhen për menaxhimin e interrupteve. Nga ana tjetër një *buggy thread kernel* mund të beje me shume deme se një *buggy user thread*. Për shembull ne se ai ekzekutohet për një kohe të gjatë nuk ka menyre për ta zevendesuar, dhe të dhenat qe vijnë mund të humbasin.

2.2.8 BERJA MULTITHREAD E KODEVE SINGLE-THREAD

Shume programe jane shkruar për procese me një thread. Konvertimi i tyre ne multithread është me i veshtire sesa duket. Me poshtë do të shohim disa nga veshtiresi që hasen.

Si fillim, kodi i një thread-i konsiston ne disa procedura, si tek proceset. Keto mund të kene variabla lokale, variabla globale dhe parametra procedurash. Variablat lokale dhe parametrat nuk shkaktojnë probleme, por variablat qe jane globale për një thread, por jo për të gjithe programin shkaktojnë probleme. Keto variabla jane globale ne sensin qe shume procedura brenda thread-it i përdorin, por thread-et e tjere logjikisht nuk duhet ti përdorin.

Për shembull, konsideroni variablin **errno** qe përdoret ne UNIX. Kur një proces ose një thread bën një thirrje sistem e cila deshton, kodi i gabimit vendoset ne **errno**. Ne Fig. 2-16, thread-i 1 ekzekuton thirrjen sistem **access** për të pare ne se ka të drejta për të aksesuar një file të caktuar. Sistemi operativ kthen përgjigjen ne variablin global **errno**. Pasi kontrolli është kthyer tek thread-i 1, por me pare ai ka një shans për të lexuar **errno**, scheduler-i vendos qe thread-i 1 ka pasur shume nga koha e CPU-se dhe vendos të kaloje tek thread-i 2. Thread-i 2 ekzekuton një thirrje **open** e cila deshton, e cila bën qe **errno** mbishkruehet dhe kodi i aksesimit të thread-it 1 të humbase përgjithnjë. Kur thread-i 1 të filloje me vone, ai do të lexoje vleren e gabuar dhe do të punoje jo të rregullt.

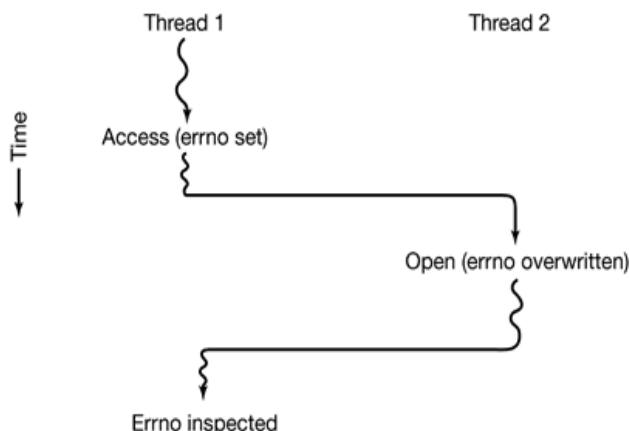


Figura 2-16. Konflikte midis thread-eve për përdorimin e një variabli global.

Shume zgjidhje për ketë problem jane të mundshme. Njëra është të ndalosh variablat global ne tëresi. Megjithese kjo do të ishte një zgjidhje, realisht nuk pranohet nga pjesa me e madhe e software-eve ekzistues. Një tjetër zgjidhje është ti vendosim çdo thread-i variablat e tyre global sic tregohet ne Fig.2-17. Ne ketë menyre çdo thread ka një kopje private të **errno** dhe variabla të tjere globale, keshtu qe konfliktet shmanget. Por ky vendim krijon një nivel të ri lirie, variablat qe jane të dukshme për të gjitha procedurat e

një thread-i, qe i shtohen variablave qe jane të dukshem vetëm për një procedure dhe variablat qe jane të dukshem kudo ne program.

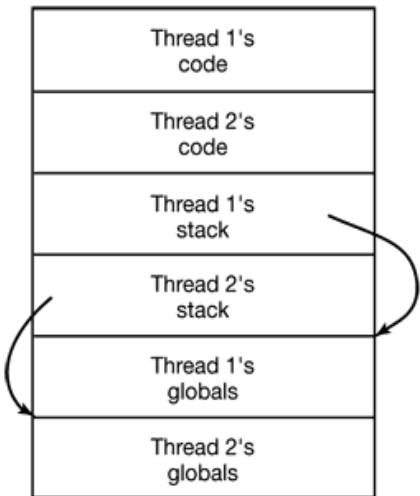


Figura 2-17. Thread-et mund të kene variabla globale private.

Aksesimi i variablave global private është pak i vesh tire, përderisa shume gjuhe programimi kane një menyre për të deklaruar variablat lokale dhe globale, por jo forma të ndermjetme. Është e mundur të alokosh një pjese të vogel të memorjes për globalet dhe t’ia kalosh çdo procedure ne thread, si ne parametër shtese.

Një tjetër zgjidhje konsiston ne faktin e krijimit të procedurave të reja ne librari, të cilat mund të krijojnë, vendosin dhe lexojnë variabla të ndermjetëm. Thirrja e pare mund të jetë si me poshtë:

```
create_global("bufptr");
```

Kjo alokon një pjese memorje për një pointer qe quhet **bufptr** ne një vend të vecantë të rezervuar për thread-in therritës. Nuk ka rendesi ku zgjidhet vendodhja, vetëm thread-i therritës ka akses tek variabla global. Ne se një thread tjetër krijon një variabel global me të njëjtin emer ai ruhet ne një vendodhje tjetër, ne menyre qe mos të krijoje konflikt me variablin ekzistues.

Dy thirrje nevojiten për të aksesuar variablat globale: Një për ti shkruar dhe tjetri për ti lexuar. Për ti shkruar përdoret dicta e tille:

```
set_global("bufptr", &buf);
```

Ajo ruan vleren e pointerit ne vendin e përcaktuar qe është krijuar me pare nga thirrja *create_global*. Për të lexuar një variabel global, thirrja është afersisht si me poshtë:

```
bufptr = read_global ("bufptr");
```

Kjo kthen adresen e ruajtur ne variablin global, por të dhenat e saj mund të aksesohen.

Problemi tjetër qe ekziston kur kthjeme një program single-thread ne multithread është se shume procedura librarish nuk të lejone të hysh. Kjo do të thotë qe nuk jane të dizenuara të kene një thirrje të dytë të bere ne çdo lloj procedure, ndersa e para nuk ka mbaruar. Për shembull, dergimi i një mesazhi ne rrjet mund të jetë i programuar të asembloje mesazhin ne një buffer fiks brenda librarise, me pas behet një instruksion “trap” ne kernel për ta derguar. Cfare ndodh ne se një thread ka asembluar mesazhin e tij ne buffer, me pas një interrupt clock-u detyron të behet një nderrim thread-esh, thread-i i ri menjehere mbishkruan bufferin me mesazhin e tij?

Ne menyre të ngjashme, procedura për alokimin e memorjes, sic është *malloc* ne UNIX, mbajne disa tabela rreth përdorimit të memorjes, per shembull, një list me vende të gatshme për tu përdorur ne memorje. Ndersa *malloc* është i zene duke azhornuar keto lista, ato mund të jene përkohesisht ne gjendje të parregullt, me pointera qe nuk pointojne ne ndonjë vend. Ne se ndodh një nderrim thread-esh, ndersa keto tabela jane ne gjendje të parregullt dhe një thirrje e re vjen nga një thread tjetër, një pointer i gabuar mund të përdoret, qe con ne një gabim të programit. Rregullimi i të gjithave ketyre gabimeve kerkon rishkrimin e të gjithe librarise.

Një zgjidhje e ndryshme është qe çdo procedure të vendosi një bit për të shenuar librarine qe ka ne përdorim. Çdo përpjekje nga ana e një thread-i tjetër për të përdorur librarine do të bllokohet. Megjithese kjo menyre mund të funksionoje ajo e eliminon parelizmin e mundshem.

Konsederoni një sinjal. Disa sinjale specifikojne logjikisht thread-et, disa të tjere jo. Për shembull ne se një thread bën një thirrje *alarm*, kjo ka kuptim për një sinjal të caktuar të shkoje të thread-i qe beri thirrjen. Megjithatë ne se thread-et jane implementura të gjithe ne user space, kerneli nuk di gje fare për ekzistencën e tyre dhe zor se mund ta dergoje sinjalin të thread-i i duhur. Një komplikacion tjetër ndodh ne se procesi mund të ketë një sinjal alarm duke pritur dhe shume thread-e qe bejne thirrje *alarm* ne menyre të pavarur.

Sinjale të tjere, si interruptet e tastieres, nuk Jane të specifikuar me thread-e. Kush duhet ti përgjigjet atyre? Një thread? Të gjithe thread-et? Një thread pop-up? Me tej, cfare ndodh ne se një thread ndryshon menaxhuesin e sinjalit pa i thene thread-eve të tjere? Cfare ndodh ne se një thread do të kape një sinjal të caktua, dhe një thread tjetër do ketë sinjal për të përfunduar procesin? Kjo situatë mund të ndodhe ne se një ose me shume thread-e ekzekutojne procedura standarte librarish dhe të tjeret jane *user-writtën*. Është e qartë qe keto deshira Jane të parealizueshme. Ne përgjithesi, sinjalet Jane të veshtire për tu menaxhuar ne një ambjent me një thread të vetëm. Por edhe përdorimi i tyre ne ambjentë me multithread nuk e lehtëson shume përdorimin e tyre.

Një problem i fundit i futur nga thread-et është menaxhimi i stakut. Ne shume sisteme, ne të cilët kemi overflow të stakut të procesit, kerneli thjesht i siguron atij procesi me shume stak ne menyre automatike. Kur një proces ka shume thread-e, ai gjithashtu ka edhe shume stake. Ne se kerneli nuk është ne dijeni të ketyre stakeve, ai nuk mund ti rrise ato ne menyre automatike ne rast se kemi një gabim ne stak (overflow).

Keto probleme nuk Jane të pakapërcyeshme, por tregonje se vetëm futja e thread-eve ne një sistem ekzistues pa një ridizenjim të sistemit nuk do të funksionoje. **Semantikat** e

thirrjeve sistem duhen të ripërcaktohen dhe libraritë duhet të rishkruhen. Keto ndryshime duhen të behen por gjithnjë duke ruajtur pajtueshmerine me programet ekzistuese.

2.3 Komunikimi i intërproceseve

Proçeset shpesh herë duhet të komunikojnë me proçese të tjera. Për shembull, në një pipeline shelli, outputi i një proçesi duhet ti kalohet proçesit të dytë dhe kështu deri në fund të linjës. Kështu është i nevojshëm një komunikim ndërmjet proçeseve, në një mënyrë të mirë-strukturuar, pa përdorur interruptet. Në pjesën e mëposhtme do të shohim disa nga problemet që lidhen me komunikimin e intérproceseve (IPC – Intérprocess Communication).

Shqyrtojmë shkurtimisht tre probleme këtu. I pari u përmend edhe më sipër: si një proçes mund ti kalojë informacion një tjetri. I dyti ka të bëjë me sigurimin që dy ose më shumë proçese të mos ndërhyjnë në rrugën e njëri-tjetrit kur merren me një aktivitet të rëndësishëm (supozojmë se dy proçese mundohen të kapin të dy 1 MB të fundit që ka mbetur në memorie). I treti ka të bëjë me rradhitjen e proçeseve kur ka varësi ndërmjet tyre: në qoftë se proçesi A prodhon të dhëna dhe proçesi B i printon ato, B duhet të presë derisa proçesi A të prodhojë të dhëna para se ti printojë ato. Do të shqyrtojmë secilin prej këtyre rasteve.

Është gjithashtu e rëndësishme të përmendim se dy nga këto probleme vlejnë në të njëjtën mënyrë edhe për threadet. I pari – kalimi i informacionit – është i lehtë për threadet duke qënë se ndajnë të njëjtën hapësirë adresash (threadet që ndodhen në hapësira adresash të ndryshme që kanë nevojë të komunikojnë komandohen nga proçeset komunikuese). Megjithatë, dy problemet e tjera – të rrinë larg nga rruga e njëri-tjetrit dhe renditja e përshtatshme – vlejnë edhe për threadet. Ekzistojnë të njëjtat probleme dhe vlejnë të njëjtat zgjidhje. Poshtë do të diskutojmë problemet në lidhje me proçeset, por mos harroni që të njëjtat probleme dhe zgjidhje vlejnë edhe për threadet.

2.3.1 Kushtet e përparësisë

Në disa sisteme operative, proçese që po punojnë në të njëjtën kohë mund të share-ojnë memorie të njëjtë, ku secili mund ta shkruajë ose lexojë. Memoria që share-ojnë mund të ndodhet në memorien kryesore (ndoshta në një strukturë të dhëna kernel) ose mund të jetë një file i share-uar: vendndodhja e memories së share-uar nuk ndryshon natyrën e komunikimit apo problemet që mund të lindin. Për të parë se si ndodh një komunikim interprocесеш në praktikë, marrim një shembull të thjeshtë por të shpeshtë: spooler i një printeri. Kur një proçes dëshiron të printojë një file, ai shkruan emrin e file-it në një **direktori spooler**. Një proçes tjetër, **printer daemon**, kontrollon përiodikisht për të parë a ka file të gatshme për t'u printuar dhe në qoftë se ka, ai i printon dhe heq emrat e tyre nga direktoria.

Imagjinojmë se direktoria spooler ka një numër të madh slotësh, me numrat 0, 1, 2, ..., dhe secili mund të mbajë një emër file. Gjithashtu imagjinojmë se janë dy variabla që share-ohen, *out*, e cila shenjon në filen pasardhëse që do të printohet, dhe *in*, që shenjon në slotin pasardhës të lirë në direktori. Këto dy variabla mund të ruhen në një file me gjatësi dy word, që mund të aksesohet nga të gjithë proceset. Në një çast të caktuar, slotët nga 0 në 3 janë bosh (filet janë printuar) dhe slotët nga 4 në 6 janë plot (me emrat e fileve në rradhë për tu printuar). Pak a shumë në të njëjtën kohë, proceset A dhe B vendosin që duan të rradhisin një file për printim. Kjo situatë është treguar në Fig. 2-18.

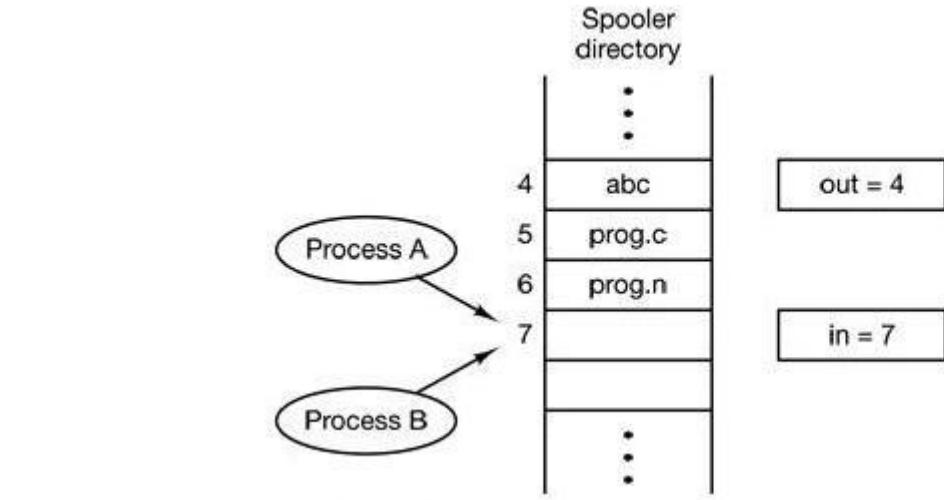


Figura 2-18. Dy procese që duan të aksesojnë memorie të share-uar në të njëjtën kohë.

Në një prioritet ku zbatohet ligji i Murphyt, mund të ndodhë si më poshtë. Procesi A lexon variablën *in* dhe ruan vlerën, 7, në një variabël lokale e quajtur *next_free_slot*. Menjehere ndodh një interrupt dhe CPU-ja vendos që procesi A ka vepruar mjaftueshmë, kështu kalon tek procesi B. Procesi B gjithashtu lexon variablën *in* dhe merr gjithashtu vlerën 7. Ai e ruan atë në variablën *e_tij* lokale *next_free_slot*. Në këtë moment të dy proceset mendojnë se slot i lirë pasardhës është 7.

Procesi B vazhdon rrjedhën. Ai ruan emrin e file-it të *tij* në slotin 7 dhe inkrementon variablin *in* në 8. Pastaj largohet dhe vazhdon gjëra të tjera.

Kështu, procesi A vazhdon ekzekutimin nga vendi ku e kishte lënë. Shikon në *next_free_slot*, gjen 7 aty dhe shkruan emrin e file-it në slotin 7, kështu fshin emrin që procesi B kishte vendosur. Më pas llogarit *next_free_slot + 1* që është 8, dhe bën variablin *in* 8. Direktoria spooler është e qëndrueshme, kështu printer daemon nuk do të dallojë asnje të gabuar, por procesi B nuk do të marrë asnjeherë outputin e dëshiruar. Përdoruesi B do të rrije në dhomën e printerit përvitetë tëra, duke shpresuar për një output që nuk vjen kurrë. Situata të këtilla, ku dy ose më shumë procese lexojnë ose shkruajnë disa të dhëna të share-uara dhe rezultati final varet nga kush ekzekutohet ekzaktësisht, kur quhen kushte përparësie (race conditions). Të bësh debug programeve që mbajnë kushte përparësie nuk është argëtim. Rezultatet e shumicës së testëve dalin mirë, por ndonjehere mund të ndodhë diçka e çuditshme dhe e pashpjegueshme.

2.3.2 Zonat kritike

Si i mënjanojmë kushtet e përparësisë? Celësi për të shhangur problemet në këtë rast dhe në shumë situata që kanë të bëjnë me share-imin e memories, share-imin e fileve dhe me share-imin e çdo gjëje tjeterë është të gjejmë një mënyrë që të lejojmë vetëm një proces, që të lexojë apo shkruajë të dhëna të share-uara në një çast të caktuar. E thënë me fjalë të tjera, na duhet një **përjashtim i ndersjelltë**, që do të thotë, na duhet një mënyrë për të siguruar që Në qoftë se një proces është duke përdorur një variabël apo file të share-uar, proçeset e tjera do të përjashtohen. Vështirësia më parë ndodhi sepse procesi B filloj të përdortë një nga variablat e share-uara para se procesi A të kishte përfunduar me të. Zgjedhja e veprimeve primitive të përshtatshme për të arritur përjashtimin e ndersjelltë është një problem i rëndësishëm design-i i të gjithë sistemeve operative dhe një tëmë që do të shqyrtojmë me vëmëndje në paragrafin pasardhës.

Problemi i mënjanimit të kushteve të përparësisë mund të formulohet edhe në mënyrë abstraktë. Një pjesë të kohës, procesi është i zënë duke bërë llogaritje të brendshme dhe gjëra të tjera që nuk çojnë në kushtet e përparësisë. Megjithatë, ndonjeherë një procesi i duhet të aksesojë memorie ose një file të share-uar, ose i duhet të bëjë gjëra të tjera që mund të çojnë në kushtet e përparësisë. Ajo pjesë e programit ku aksesohet memoria e share-uar quhet **zona kritike** ose **sekzioni kritik**. Në qoftë se ne mund të bëjmë të mundur që dy procese mos të jenë asnjeherë në zonat kritike në të njëjtën kohë, mund të mënjanojmë problemin e përparësisë.

Edhe pse kjo kërkesë mënjanon kushtet e përparësisë, nuk është mjaftueshëm për të pasur procese paralele që bashkëpunojnë korrektesisht duke përdorur të dhëna të share-uara. Ne duhet ti përbahemi katër kushteve për të pasur një zgjidhje të mirë:

1. dy procese nuk mund të jenë në të njëjtën kohë në zonat kritike respektive.
2. nuk mund të bëhen supozime mbi shpejtësите dhe numrin e CPU-ve.
3. një proces që ekzekutohet jashtë zonës së tij kritike nuk mund të bllokohet proçeset e tjera.
4. një proces nuk duhet të presë përgjithmonë për të hyrë në zonën e tij kritike.

Në mënyrë abstraktë, sjellja që duam tregohet ne Fig. 2-19. Procesi A hyn në zonën e tij kritike në kohën T_1 , në kohën T_2 procesi B tenton të hyjë në zonën e tij kritike, por dështon sepse një proces tjeterë është në zonën e tij kritike dhe ne lejojmë vetëm një në një çast të caktuar. Rrjedhimisht, procesi B pezullohet deri në kohën T_3 kur A largohet

nga zona kritike, kështu lejon B të hyjë menjeherë. Më pas B largohet (në T_4) dhe kthehemë sërish në situatën fillestare, ku asnjë proces nuk ndodhet në zonën kritike.

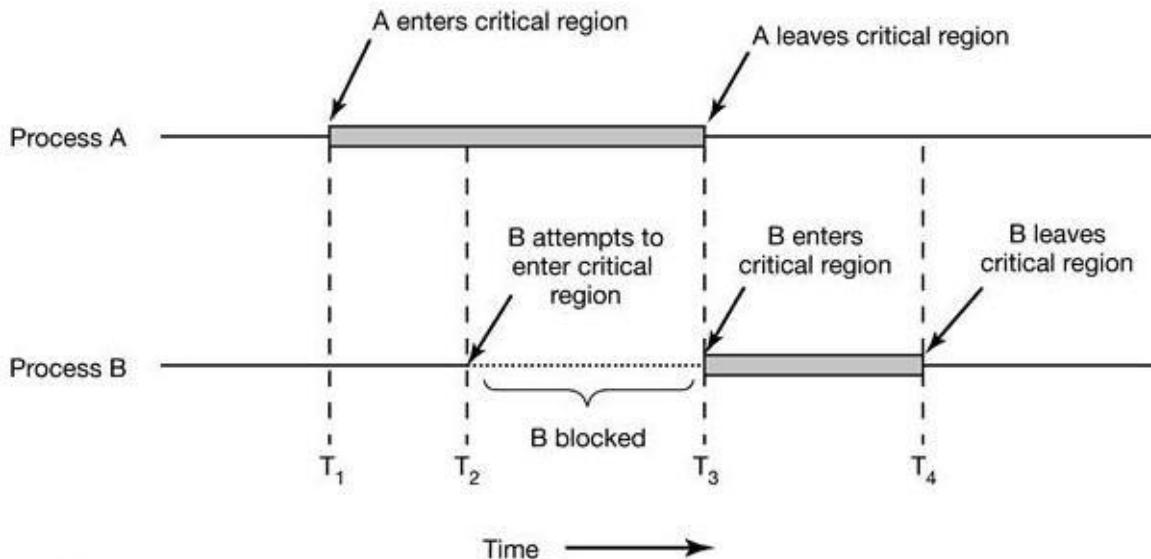


Figura 2-19. Përjashtimi i ndersjelltë me zonat kritike.

2.3.3 Përjashtim i ndersjelltë me Busy Waiting

Në këtë paragraf do të shqyrtojmë propozime të ndryshme për të arritur përjashtimin e ndersjelltë, në mënyrë që, ndërkohë që një proces është duke përpunuar memorien e sharuar në zonën e tij kritike, një tjeter mund të hyjë në zonën e tij kritike dhe kështu të hapë probleme.

Mbyllja e interrupteve

Zgjidhja më e thjeshtë është që çdo proces të mbylli interruptet menjeherë sa po të hyjë në zonën e tij kritike dhe lejimi i tyre menjeherë pasi të dalë. Me interruptet e mbyllura, nuk kemi interrupte të clockut. CPU-ja kalon nga një proces tek një tjeter si rezultat i interrupteve të clockut ose të tjerë, dhe me interruptet të fikura, CPU-ja nuk do të kalojë në një proces tjeter. Kështu, pasi një proces të ketë mbyllur interruptet, mund të shohë dhe ndryshojë memorien e sharuar pa patur frikë nga ndërhyrja e ndonjë procesi tjeter.

Kjo mënyrë nuk është e pëlqyer sepse nuk është e zgjuar ti japësh proceseve user mundësinë e fikjes së interrupteve. Supozojmë sikur njëri prej tyre e bën dhe nuk i ndez më? Kjo do të sillte fundin e sistemit. Për më tepër në qoftë se sistemi është multiproçesor, me dy ose më shumë CPU, mbyllja e interrupteve vlen vetëm për CPU-në

që ekzekutoi instruksionin e mbylljes. Të tjerët do vazhdojnë të funksionojnë dhe mund të aksesojnë memorien e share-uar.

Nga ana tjetër, shpesh herë është e dobishme që kerneli të mbyllë interruptet për pak instruksione ndërkohë që po ndryshohen variablat. Në qoftë se një interrupt ndodh kur lista e proceseve në gjendjen ready është e paqëndrueshme, mund të kemi kushtet e përparësisë. Përfundimi është: mbyllja e interrupteve muund të jetë një teknikë e dobishme brënda sistemit operativ, por nuk është i përshtatshëm si mekanizëm i përgjithshëm i përjashtimit të ndersjelltë për proceset user.

Variablat kyç (lock)

Si përpjekje të dytë le të përpinqemi të gjejmë një zgjidhje me sfotware. Konsiderojmë sikur kemi vetëm një variabël (kyç) të share-uar, që fillimisht është 0. Kur një proces kërkon të hyjë në zonën e tij kritike, teston variablën kyç. Në qoftë se kyçi është 0, procesi e kthen 1 dhe futet në zonën kritike. Në qoftë se kyçi është 1, procesi pret derisa të bëhet 0. Kështu, 0 do të thotë se nuk ka proces në zonën e tij kritike, dhe 1 do të thotë se ka një proces që ndodhet në zonën kritike.

Fatkeqësish, ideja përmban të njëjtën rrjedhë të gabuar që pamë në direktorinë spooler. Supozojmë se një proces lexon kryçin dhe shikon që është 0. përpara se ta kthejë në 1, skedulohet një proces tjetër, ekzekutohet dhe e kthen krycin në 1. Kur procesi i parë vazhdon ekzekutimin edhe ai do të kthejë krycin në 1, dhe dy procese do të ndodhen në zonat kritike përkatëse në të njëjtën kohë.

Tani mund të mendoni që mund të mënjanojmë këtë problem duke lexuar fillimisht variablën kyç, dhe më pas ta rikontrollojmë pak para se të shkruajmë në të, por kjo në të vërtëtë nuk ndihmon. Gara e përparësisë tani fillon në qoftë se procesi i dytë ndryshon variablën kyç menjeherë pasi procesi i parë e ka kontrolluar për herë të dytë.

Alternimi i Përpiktë

Një trajtim i tretë i problemit të përjashtimit të ndersjelltë tregohet në Fig. 2-20. Kjo pjesë programi, si pothuajse të gjitha në këtë libër, është shkruar në C. Është zgjedhur C sepse sistemet operative janë pothuajse gjithmonë të shkruara në C (ose nganjeherë në C++), por shumë rrallë në gjuhë si Java, Modula 3, apo Pascal. C-ja është e fuqishme, efikase dhe e parashikueshme, karakteristika kritike për shkruajtjen e sistemeve operative. Java, për shembull, nuk është e parashikueshme sepse mund të ngelet pa memorie në një moment kritik, dhe i duhet të thërrasë garbage collector në një moment të papërshtatshëm. Kjo nuk mund të ndodhë në C, sepse në C nuk ka garbage

collector. Një krahësim i përpiktë i C, C++, Java dhe katër gjuhëve të tjera është dhënë në (Prechelt, 2000).

Në Fig. 2-20, variabla integer *turn* (rradha), fillimi është 0, ruan kush e ka rradhën për t'u futur në zonën kritike dhe të shikojë apo ndryshojë memorien e sharuar. Fillimi është 0 kontrollon variablën *turn*, shikon që është 0 dhe hyn në zonën e tij kritike. Edhe procesi 1 shikon që është 0 dhe vendoset në një lak duke testuar vazhdimi është variablën *turn* për të parë kur do të bëhet 1. Testimi i vazhdueshëm i një variable derisa ajo të ketë një vlerë të caktuar quhet **busy waiting**. Zakonisht duhet evituar, meqënëse harxhon kohë të CPU-së. busy waiting përdoret vetëm në qoftë se është logjikisht e pritshme se do të marrë pak kohë. Një kyç që përdor busy waiting quhet **spin lock**.

```
while (TRUE) {
    while (turn != 0) /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1); /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Figura 2-20. Një propozim për zgjidhjen e problemit të zonave kritike. (a) Procesi 0. (b) Procesi 1. Në të dy rastet, vini re pikëpresjen në fund të ciklevë while.

Kur procesi 0 lë zonën kritike, vendos variablën *turn* në 1, që të lejojë procesin 1 të hyjë në zonën kritike. Supozojmë se procesi 1 mbaron zonën e tij kritike shpejt, kështu të dy proceset ndodhen në zonat e tyre jo-kritike, me variablën *turn* në 0. Tani procesi 0 ekzekuton gjithë ciklin e tij shpejt, del nga zona kritike dhe vendos *turn* në 1. Në këtë pikë *turn* është 1 dhe të dy proceset janë duke u ekzekutuar në zonat e tyre jo-kritike.

Papritur, procesi 0 mbaron zonën e tij jo-kritike dhe rikthehet në fillim të lakut. Fatkeqësisht, nuk i lejohet të futet në zonën e tij kritike tani, sepse *turn* është 1 dhe

proçesi 1 është i zënë me zonën e tij jo-kritike. Ai ngelet në lakun while derisa proçesi 1 kthen *turn* në 0. E thënë ndryshe, puna me turne nuk është ide e mirë kur njëri nga proçeset është shumë më i ngadaltë se tjetri.

Kjo situatë shkel kushtin 3 e vendosur më lart: proçesi 0 po bllokohet nga një proçes tjetër që nuk është në zonën e tij kritike. Po ti kthehem direktorisë spooling që diskutuan më lart, po të lidhim zonën kritike me leximin dhe shkrimin në direktorinë spooling, proçesit 0 nuk do ti lejohej të printontë një file sepse proçesi 1 është duke bërë diçka.

Në fakt, kjo zgjidhje kërkon që dy proçeset të altérnohen në hyrjen në zonat kritike, për shembull në file spooling. Asnjërit nuk do ti lejoheshin dy njëri pas tjetrit. Ndërkohë që ky algoritem mënjanon kushtet e përparësisë, nuk është një kandidat i vlefshëm sepse shkel kushtin 3.

Zgjidhja e Peterson

Duke kombinuar idenë e turneve me idenë e variablate kyç dhe variablate lajmëruese, një matematikan hollandez, T. Dekker, ishte i pari që gjeti një zgjidhje sfotware-ike për përjashtimin e ndërsjelltë që nuk kërkonte altérnim rigoroz. Për një diskutim mbi algoritmin e Dekker, shiko (Dijkstra, 1965).

Në 1981, G. L. Petëson zbuloi një mënyrë më të thjeshtë për të arritur përjashtimin e ndërsjelltë dhe në këtë mënyrë e bëri zgjidhjen e Dekker të pavlerë. Algoritmi i Petësonit është treguar në Fig. 2-21. Ky algoritem përmban dy procedura të shkruara në ANSI C, që do të thotë se për të gjithë funksionet e përcaktuara dhe përdorura duhet të caktohen prototipet e funksioneve. Megjithatë, për të kursyer hapësirë, nuk do të tregojmë prototipet në shembullin e mëposhtëm.

```
#define FALSE 0
#define TRUE 1
#define N    2    /* number of processes */

int turn;          /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process) /* process is 0 or 1 */
{
    int other;           /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;      /* set flag */
```

```

        while (turn == process && interested[other] == TRUE) /* null statement */;
    }

void leave_region (int process) /* process, who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

```

Figura 2-21. Zgjidhja e Petersonit për përjashtimin e ndersjelltë.

Para se të përdorë variablat e sharuara (para se të hyjë në zonën e tij kritike), çdo proces thirret *enter_region* me numrin e vet, 0 ose 1, si parametër. Kjo thirrje do ta bënte atë të priste, në qoftë se duhet, derisa të jetë pa rrezik të hyjë. Pasi të ketë mbaruar me variablat e sharuara, procesi thirret *leave_region* për të treguar që ka mbaruar dhe për të lejuar proceset e tjera të hyjnë, në qoftë se kjo kërkohet.

Le të shohim se si funksionon kjo zgjidhje. Fillimisht asnjë nga proceset nuk ndodhet në zonën kritike. Procesi 0 thirret *enter_region*. Tregon interesin duke setuar elementin e vektorit të tij dhe kthen *turn* në 0. Meqënë se procesi 1 nuk është i interesuar, kthehet menjeherë *enter_region*. Në qoftë se procesi 1 thirret tani *enter_region*, derisa sa *interested[0]* të bëhet *FALSE*, dhe kjo ndodh vetëm kur procesi 0 thirret *leave_region* për të dalë nga zona kritike.

Tani konsiderojmë rastin kur të dy proceset thërrasin pothuajse njëkohësisht *enter_region*. Të dy do të rregjistrojnë numrin e tyre tek *turn*. Vetëm ai numër që do të rregjistrohet i fundit vlen; i pari mbishkrumet dhe humbet. Supozojmë që të fundit e rregjistron procesi 1, kështu *turn* do të jetë 1. Kur të dy proceset të vijnë tek while, procesi 0 e ekzekuton zero herë dhe hyn në zonën kritike. Procesi 1 nuk futet në zonën kritike derisa procesi 0 të dalë nga zona e vet kritike.

Instruksioni TSL

Le të shikojmë tani një propozim që ka nevojë për pak ndihmë nga hardware. Shumë kompjutera, veçanërisht ata që projektohen për multiproçesorë, kanë një instruksion :

TSL RX, LOCK

(Test dhe Set Lock) që funksionon si më poshtë. Lexon përbajtjen e memories në fjalën *lock* (kyc) në rregjistrin RX dhe ruan një vlerë jo-zero në memorien me adresë *lock*. Veprimet e leximit dhe të shkrimit mbi fjalën janë të pandashme – asnje proces nuk mund të aksesojë fjalën derisa instruksioni të ketë përfunduar. CPU-ja që ekzekuton instruksionin TSL bllokon busin e memories për të mos lejuar CPU-të e tjera të aksesojnë memorien derisa të ketë përfunduar.

Për të përdorur instruksionin TSL, do të përdorim një variabël të share-uar, *lock*, për të koordinuar aksesimin e memories së share-uar. Kur *lock* është 0, çdo proces mund ta kthejë në 1 duke përdorur instruksionin TSL dhe më pas të lexojë dhe shkruajë në memorien e share-uar. Kur mbaron punë, procesi rikthen *lock* në 0 duke përdorur një instruksion të zakonshëm move.

Si mund të përdoret ky instruksion për të penguar që dy procese të futen në zonat kritike në të njëjtën kohë? Zgjidhja jepet në Fig. 2-22. Këtu tregohet një subrutinë me pesë instruksione në një gjuhë assembler fiktive (por tipike). Instruksioni i parë kopjon vlerën e vjetër të *lock* në një regjistër dhe kthen *lock* në 1. Vlera e vjetër krahasohet me 0. Në qoftë se është jo-zero, kyç është vënë, kështu programi kthehet në fillim dhe e teston sërisht. Në një moment ai do të bëhet 0 (kur procesi që për momentin ndodhet në zonën kritike e lë), dhe subrutina kthehet, me kyçin e vendosur. Të pastrosh kyçin është e thjeshtë. Programi ruan një 0 në *lock*. Nuk kërkohen intruksione të veçanta.

enter_region:

TSL REGISTËR, LOCK	kopjon lock në regjistër dhe vendos 1
CMP REGISTËR, #0	a ishte lock zero?
JNE enter_region	në qoftë se ishte jozero, lock ka qënë setuar, futu në lak
RET	return, futet në zonën kritike

Leave_region:

MOVE LOCK,#0	rregjistro 0 në lock
RET	return

Figura 2-22. Hyrja dhe dalja nga zona kritike duke përdorur instruksionin TSL.

Një zgjidhje e problemit të zonave kritike tashmë është e qartë. Përpara se të hyjë në zonën kritike, një proces thërret *enter_region* dhe kalon në busy waiting derisa të lirohet lock; pastaj merr lock-un dhe kthehet. Pas zonës kritike procesi thërret *leave_region*, që ruan një 0 në *lock*. Si gjithë zgjidhjet të bazuara në zonat kritike, proceset duhet të thërrasin *enter_region* dhe *leave_region* në kohën e duhur që metoda të funksionojë. Në qoftë se procesi bën hile, përjashtimi i ndërsjellë do të dështojë.

2.3.4 Sleep and Wakeup (gjumi dhe zgjimi)

Edhe zgjidhja e Petësonit, edhe ajo që përdor TSL janë të sakta, por të dy kanë nevojë për busy waiting. Në thelb, çfarë këto zgjidhje bëjnë është: kur një proces do të hyjë në zonën kritike, shikon a i lejohet. Në qoftë se jo, procesi thjesht pret derisa ti lejohet.

Ky trajtim harxon kohë të CPU-së dhe mund të ketë rezultate të papritshme. Konsiderojmë një kompjuter me dy procese, H me prioritet të lartë dhe L me prioritet të ulët. Rregullat e skedulimit janë të tilla që H mund të ekzekutohet kur të jetë në gjendjen gati. Në një çast të caktuar, me L në zonën e tij kritike, H bëhet gati për t'u ekzekutuar (për shembull, kompletohet një veprim I/O). H fillon busy waiting, meqënëse L nuk skedulohet kur H është duke u ekzekutuar, L nuk ka asnjeherë mundësinë të lerë zonën e tij kritike, kështu H mbetet në lak përgjithmonë. Kjo situatë nganjeherë referohet si **problemi i inversionit të prioritetit**.

Tani le të shikojmë disa primitiva të komunikimit të intérprocéseve që bllokohen në vënd që të humbin kohë të CPU-së kur nuk lejohen të futen në zonat kritike. Një nga më të thjeshtat është dyshja sleep dhe wake up. Sleep është një thirrje sistem që shkakton atë që bën thirrjen të bllokohet, do të thotë, të pezullohet derisa një proces tjetër ta zgjojë. Thirrja wakeup ka një parametër, procesin që duhet zgjuar. Ose të dy proceset mund të kenë secili nga një parametër, një adresë memorie që përdoret për të lidhur thirrjet sleep me ato wake up.

Problemi prodhues-konsumator

Si një shembull se si mund të përdoren këto primitiva, le të konsiderojmë problemin **prodhues-konsumator** (i njojur edhe si problemi **buffer-i kufizuar**). Dy procese ndajnë një buffer të përbashkët me madhësi të caktuar. Njëri prej tyre, prodhuesi, vendos informacion në buffer dhe tjetri, konsumatori e merr atë. (Është gjithashtu e mundur të përgjithësohet problemi duke patur m prodhues dhe n konsumatorë, por ne do të marrim parasysh vetëm rastin me një prodhues dhe një konsumator, sepse ky supozim thjeshtëson zgjidhjen).

Na lindin télashë kur prodhuesi do të vendosi një element tjetër në buffer, por është plot. Zgjidhja është që prodhuesi të vihet në gjumë, dhe të zgjohet kur konsumatori të ketë hequr një ose më shumë elementë. Në mënyrë të ngjashmme, në qoftë se konsumatori do të marrë një element nga bufferi dhe shikon që është bosh, vihet në gjumë derisa prodhuesi të vendosë diçka në buffer dhe ta zgjojë.

Ky trajtim duket i mjaftueshëm, por ai con në të njëjtat kushte përparësie që pamë më parë me direktorinë spooler. Na duhet një variabllë, *count*, për të ruajtur numrin e elementëve në buffer. Në qoftë se numri maksimal i variablave që mund të mbajë është *N*, kodi i prodhuesit do të testojë fillimisht në se *count* është *N*. Në qoftë se po, prodhuesi do të vihet në gjumë; në qoftë se jo, prodhuesi do të shtojë një element dhe do inkrementoje *count*.

Kodi i konsumatorit është i ngjashëm: fillimisht teston *count* për të parë në qoftë se është 0. Në qoftë se është, vihet në gjumë, në qoftë se është jo-zero, heq një element dhe dekrementon numërueshin. Secili prej proceseve shikon gjithashtu në se procesi tjetër duhet zgjuar, në se po, e zgjon. Kodi si për prodhuesin dhe konsumatorin tregohet në Fig. 2-23.

```
#define N 100      /* number of slots in the buffer */
int count = 0;    /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {          /* repeat forever */
        item = produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item);   /* put item in buffer */
        count = count + 1;    /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {          /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1;    /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);  /* print item */
    }
}
```

}

Figura 2-23. Problemi prodhues-konsumator me kushte përparësie.

Për të shprehur thirrjet sistem si sleep dhe wake up në C, do ti shprehim si thirrje për routina librarie. Ato nuk janë pjesë e librarive standarte të C por që do të jenë me shumë mundësi të disponueshme në çdo sistem që ka patur këto thirrje sistem. Procedurat *insert_item* dhe *remove_item*, që nuk tregohen, merren me rregullimin e futjes së elementëve në buffer dhe heqjen e tyre nga bufferi.

Tani kthehem i nuk kushtin e përparësisë. Mund të ndodhë sepse aksesimi i *count* është i pakufizuar. Mund të ndodhë situata e mëposhtme. Bufferi është bosh dhe konsumatori sapo ka lexuar *count* për të parë a është 0. Në atë çast, skeduleri vendos të ndalojë së ekzekutuari përkohësisht konsumatorin dhe të fillojë së ekzekutuari prodhuesin. Prodhuesi vendos një element në buffer, inkrementon *count* dhe vëren që tani është 1. Duke arsyetuar që *count* është 0, dhe kështu që konsumatori është në gjumë, prodhuesi thërret *wake up* për të zgjuar konsumatorin.

Fatkeqesisht, konsumatori nuk është ende illogjikisht në gjumë, kështu sinjali i zgjimit humbet. Kur ekzekutohet konsumatori, do të testojë vlerën e *count* që lexoi më parë, do të shohë që është 0 dhe do të vihet në gjumë. Në një çast prodhuesi do të mbushi bufferin dhe do të vihet në gjumë gjithashtu. Të dy do të mbeten në gjumë përgjithmonë.

Thelbi i problemit këtu është se thirrja *wake up* që i dërgohet një procesi që nuk është (ende) në gjumë, humbet. Në qoftë se nuk do të humbistë, gjithëcka do të punonte. Një rregullim i shpejtë do të ishte duke shtuar një **wakeup waiting bit** (biti i pritjes së zgjimit). Kur një *wake up* i dërgohet një procesi që është ende zgjuar, biti setohet. Më pas, kur procesi tenton të vihet në gjumë, në qoftë se *wake up* waiting bit është ndezur, do të fiket, por procesi do të mbetet zgjuar. *Wake up* waiting bit është një depozitë ndihmëse për sinjalat e zgjimit.

Ndërkohë që *wake up* waiting bit na shpëton problemin në këtë shembull të thjeshtë, është e thjeshtë të ndërtojmë problema me tre ose më shumë procese ku vetëm një bit i këtillë nuk është i mjaftueshëm. Mund të shtojmë edhe një bit tjetër ose ndoshta 8 apo 32, por si parim problemi mbetet.

2.3.5 Semaforët

Kjo ishte situata në 1965, kur E. W. Dijkstra (1965) sugjeroi të përdorej një variabël integer për të numëruar *wake up*-et e ruajtura për përdorim të mëvonshëm. Në propozimin e tij u fut një variabël e re e quajtur semafor. Një semafor mund të kishte një vlerë 0, që tregontë që nuk ishte ruajtur asnjë *wake up* ose një numër pozitiv, në qoftë se një ose më shumë *wake up*-e po prisnin.

Dijkstra propozoi dy veprime, down (poshtë) dhe up (lart) (përgjithësimet e sleep dhe wake up). Veprimi down kontrollon në se vlera është më e madhe se 0. Në qoftë se është kështu, dekrementon vlerën (përdor një wake up të ruajtur) dhe vazhdon. Në se vlera është 0, procesi vihet në gjumë pa përfunduar veprimin down për momentin. Kontrolli i vlerës, ndryshimi i saj dhe me raste vënie në gjumë, bëhet në një veprim të vetëm dhe të pandashëm (**atomic action**). Është e sigurt që, në qoftë se fillohet një veprim semafori, asnje proces tjetër nuk mund të aksesojë semaforin derisa veprimi të ketë përfunduar ose të jetë bllokuar. Kjo pandashmëri është thelbësore për zgjidhjen e problemeve të sinkronizimit dhe shmangien e kushteve të përparësisë.

Veprimi up inkrementon vlerën e adresës së semaforit. Në qoftë se një ose më shumë procese ishin në gjumë në atë semafor, të pamundur për të përfunduar një veprim të mëparshëm down, njëri prej tyre zgjidhet nga sistemi (per shembull,.., në mënyrë të rastesishmë) dhe lejohet të përfundojë veprimin e tij down. Kështu, pas një veprimi up në një semafor më procese në gjumë, semafori do të ngelet 0, por do të ketë një proces më pak në gjumë në të. Veprimi i inkrementimit të semaforit dhe zgjimit të një procesi është gjithashtu i pandashëm. Nuk ka procese që të bllokohen duke kryer një up, ashtu si nuk bllokoheshin duke kryer një wake up në modelin e mëparshëm.

Në paperin origjinal të Dijkstra, ai përdori emrat P dhe V në vend të down dhe up respektivisht, por meqenëse këto nuk kanë kuptim mnemonik për njerëzit që nuk flasin gjuhën hollandaze, do të përdorim termat down dhe up. Kjo u paraqit fillimisht në Algol 68.

Zgjidhja e problemit Prodhues-Konsumator duke përdorur Semaforët

Semaforët zgjidhin problemin e wake up-it të humbur, si tregohet në Fig. 2-24. Është thelbësore që të implementohen në mënyrë të pandashme. Mënyra normale është implementimi i up dhe down si thirrje sistem, me sistemin operativ që shkurtimisht mbyll interruptet ndërkohë që teston semaforin, e ndryshon atë, dhe në qoftë se është e nevojshme vë proceset në gjumë. Meqenëse këto veprime kanë nevojë vetëm për pak instruksione, mbyllja e interrupteve nuk dëmton. Në qoftë se përdoren shumë CPU, çdo semafor duhet të përdorë nga një varibël kyç, më instruksioni TSL që përdoret për të siguruar që vetëm një CPU ekzaminon semaforin. Sigurohuni të kuptoni se përdorimi i TSL për të parandaluar që disa CPU të aksesojnë semaforin në të njëjtën kohë është ndryshe nga përdormi i busy waiting nga prodhuesi dhe konsumatori që presin që tjetri të zbrazë ose mbushë bufferin. Veprimit të semaforit do ti duhen vetëm pak mikrosekonda, ndërkohë që prodhuesit dhe konsumatorit mund ti duhen relativisht shumë.

```
#define N 100           /* number of slots in the buffer */
```

```

typedef int semaphore; /* semaphores are a special kind of int */
semaphore mutex = 1; /* controls access to critical region */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0; /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) { /* TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        item = remove_item(); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}

```

Figura 2-24. Problemi prodhues-konsumator duke përdorur semaforët.

Kjo zgjidhje përdor tre semaforë: njëri i quajtur *full* (plot) për të ruajtur numrin e slotëve që janë plot, njëri i quajtur *empty* (bosh) për të ruajtur numrin e slotëve që janë bosh, dhe njëri i quajtur *mutex* për të siguruar që prodhuesi dhe konsumatori të mos aksesojnë bufferin në të njëjtën kohë. *Full* është fillimisht 0, *empty* është fillimisht i barabartë me numrin e slotëve në buffer, dhe *mutex* është fillimisht 1. Semaforët që inicializohen me 1 dhe që përdoren nga dy ose më shumë semaforë për të siguruar që vetëm njëri prej tyre mund të futet në zonën kritike në një çast të caktuar quhen **semaforë binarë**. Në qoftë se secili prej proceseve kryen një down menjehëre përparrë se të futet në

zonën e tij kritike dhe një up menjeherë pasi e lë atë, kemi patjetër një përjashtim të ndërsjelltë.

Tani që kemi një primitivë komunikimi intérprocésesh në dispozicion, le të rishikojmë sekuencën e interrupteve në Fig. 2-5. Në një sistem që përdor semaforët, mënyra natyrale për fshehjen e interrupteve është me anën e një semafori, fillimisht 0, i bashkëngjitur me të gjitha pajisjtët I/O. Menjeherë pas fillimit të një pajisje I/O, procesi menaxhues kryen një down në semaforin përkatës dhe bllokohet menjeherë. Kur vjen një interrupt, menaxhuesi i interrupteve kryen një up në semaforin përkatës, gjë që bën procesin të gatshëm të ekzekutohet sërisht. Në këtë model, hapi 5 në Fig. 2-5 konsiston në kryerjen e një veprimi up mbi semaforin e pajisjes, kështu që në hapin 6 skeduleri do të mund të ekzekutojë device managerin. Sigurisht, në qoftë se disa procese janë tashmë në gjendjen gati, skeduleri mund të zgjedhë të ekzekutojë një proces ndoshta më të rendësishëm më pas. Do të shohim disa algoritma që përdoren për skedulim më pas në këtë kapitull.

Në shembullin e Fig. 2-24, në të vërtëtë kemi përdorur semaforët në dy mënyra të ndryshme. Ky ndryshim është mjaftueshmërisht i rendësishëm për t'u bërë i quartë. Semafori *mutex* është përdorur për përjashtimin e ndërsjelltë. Është projektuar për të siguruar që në një kohë të caktuar vetëm një proces të lexojë ose të shkruajë bufferin e shoqëruar me variablat. Ky përjashtimi i ndërsjelltë duhet për të parandaluar një kaos. Do të studiojmë përjashtimin e ndersjelltë dhe se si ta arrijmë atë më shumë në paragrafin e ardhshëm.

Përdorimi tjetër i semaforëve është **sinkronizimi**. Semaforët *full* dhe *empty* nevojiten për të siguruar që disa sekuencia ngjarjesh të ndodhin apo të mos ndodhin. Në këtë rast, ata sigurojnë që prodhuesi të ndalojë së ekzekutuari kur bufferi është plot, dhe të ndalojë konsumatori kur është bosh. Ky përdorim është i ndryshëm nga përjashtimi i ndërsjelltë.

2.3.6 Mutex-et

Kur nuk na duhet aftësia e semaforëve për të numëruar, përdorim një version të semaforit të quajtur mutex. Mutex-et janë të mirë vetëm për manaxhimin e përjashtimit të njëanshëm në disa burime të sharuara ose pjesë kodit. Ata janë të lehtë dhe efikas në implementim, që i bën veçanërisht të përdorshëm në paketat e thread-eve që implementohen tërësisht në hapësirën user.

Mutex është një variabël që mund të ketë dy gjendje: e hapur dhe e kyçur. Rrjedhimisht, nevojitet vetëm 1 bit për ta përfaqësuar, por në praktikë përdoret shpesha herë një integer, ku 0 do të thotë e hapur dhe çdo vlerë tjetër e kyçur. Me mutex-in përdoren dy procedura. Kur një threadi (ose procesi) i duhet të aksesojë zonën e tij kritike, thërret një *mutex_lock*. Në qoftë se mutex është për momentin i hapur (që do të thotë që zona kritike është e lirë), thirrja kryhet dhe thread-i që kryen thirrjen është i lirë të hyjë në zonën kritike.

Nga ana tjetër, në qoftë se mutex është i kyçur, thread-i që ka kryer thirrjen bllokohet derisa thread-i në zonën kritike të ketë mbaruar dhe thérret *mutex_unlock*. Në qoftë se disa proçese janë bllokuar në mutex, zgjidhet rastesisht njëri prej tyre dhe lejohet të përdorë kyçjen.

Meqënë se mutex-et janë shumë të thjeshta, ato mund të implementohen shumë thjeshtë në hapësirën user në qoftë se mund të përdoren instruksione TSL. Kodi për *mutex_lock* dhe *mutex_unlock* që përdoren në një paketë thread-esh në nivelin user, tregohen në Fig. 2-25.

mutex_lock:

```
TSL REGISTËR,MUTEX | copy mutex to register and set mutex to 1
CMP REGISTËRS,#0 | was mutex zero?
JZE ok      | if it was zero, mutex was unlocked, so return
CALL thread_yield | mutex is busy; schedule another thread
JMP mutex_lock | try again later
ok: RET | return to caller; critical region entered
```

mutex_unlock:

```
MOVE MUTEX,#0 | store a 0 in mutex
RET | return to caller
```

Figura 2-25. Implementimi i *mutex_lock* dhe *mutex_unlock*

Kod i *mutex_lock* është i ngjashëm me kodin e *enter_region* që tregohet në Fig. 2-22 por me një ndryshim thelbësor. Kur *enter_region* nuk arrin të futet në zonën kritike vazhdon teston vazhdonisht (busy waiting). Përfundimisht, koha mbaron dhe skedulohet një proçes tjetër për t'u ekzekutuar. Në një çast proçesit që mban kyçin i vjen rradha për ekzekutim dhe e lëshon atë. Me thread-et situata është e ndryshme, sepse nuk ka clock që mbyll thread-et që kanë shumë kohë që ekzekutohen. Rrjedhimisht, një thread që tenton të marrë një kyç me busy waiting do të ngelet në lak përgjithmonë dhe nuk do të marrë kurrë kyçin, sepse nuk lejon asnjë thread të ekzekutohet dhe të lëshojë kyçin.

Këtu qëndron edhe ndryshimi ndërmjet *enter_region* dhe *mutex_lock*. Kur ky i fundit nuk arrin të kapë kyçin, thérret *thread_yield* për ti dhënë CPU-në një thread-i tjetër. Rrjedhimisht nuk ka busy waiting. Kur thread-i ekzekutohet herës tjetër, teston sërisht kyçin.

Meqënë se *thread_yield* është vetëm një thirrje drejt skedulerit të thread-eve në hapësirën user, është shumë i shpejtë. Si rrjedhim, as *mutex_lock* dhe as *mutex_unlock* nuk i duhen thirrje kernel. Duke i përdorur, thread-et në nivelin user mund të sinkronizohen plotësisht në hapësirën user duke përdorur procedura që kanë nevojë vetëm për një grusht instruksionesh.

Sistemi mutex që përshkruam më lart është një grup i thjeshtë thirrjesh. Për të gjithë sfotware-et ka gjithmonë një kërkesë për sa më shumë detyra dhe veçori, e njëjtë gjë vlen edhe për primitivat e sinkronizimit. Për shembull, një paketë thread-esh ofron një thirrje e quajtur *mutex_trylock*, që merr kyçin ose kthen një kod në rast se dështon, por nuk bllokohet asnjeherë. Kjo thirrje i jep thread-it mundësi të zgjedhë çfarë të bëjë si alternativë në vënd që të presë.

Deri tani na ka dalë një problem që e kemi mbuluar paksa por që në fund ja vlen ta shpjegojmë. Kur kemi një paketë thread-esh në hapësirën user, nuk kemi probleme me rastin kur shumë thread-e duan të aksesojnë të njëjtin mutex, meqënë se veprojnë në të njëjtën hapësirë adresash. Megjithatë, me shumicën e zgjidhjeve të mëparshme, si algoritmi i Petërsone apo semaforët, ka një supozim që nuk thuhet, që në rastin e shumë proceseve ata kanë të drejtë të aksesojnë të paktën disa memorie të share-uar, ndoshta edhe vetëm një fjalë, por diçka po. Në se proçeset kanë hapësira adresash pa lidhje me njëra-tjetrën, ashtu si kemi thënë vazhdimisht, si mund të share-ojnë variablën *turn* në algoritmin Petërsone apo në semaforë, apo në buffer çfarëdo?

Ka dy përgjigje. E para, disa nga strukturat e share-imit të të dhënave, si semaforët, mund të ruhen në kernel dhe të aksesohen vetëm me anë të thirrjeve sistem. Ky trajtim eliminon problemin. E dyta, shumica e sistemeve operative moderne (duke përfshirë Unix dhe Windows) ofrojnë një mënyrë që proçeset të share-ojnë një pjesë të hapësirës së tyre të adresave me proçese të tjera. Në këtë mënyrë, mund të share-ohen buffer apo dhe struktura të tjera të dhëna. Në rastin më të keq, që nuk është e mundur asgjë tjetër, mund të përdoret një file i share-uar.

Në se dy ose më shumë proçese share-ojnë shumicën apo të gjithë hapësirën e tyre të adresave, ndryshimi ndërmjet proçeseve dhe thread-eve mbulohet disi, por ekziston gjithësesi. Dy proçese që share-ojnë të njëjtën hapësirë adresash kanë gjithësesi open files, alarm timers dhe karakteristika të tjera të proçeseve të ndryshme, ndërkohë që proçeset brënda një proçesi i share-ojnë ato. Dhe është gjithmonë e vërtëtë që proçeset që ndajnë të njëjtën hapësirë adresash nuk kanë të njëtin efikasitet si thread-et e nivelit user meqënë se kernel është thellësish i përfshirë në manaxhimin e tyre.

2.3.7 Monitorët

Më anë të semaforëve komunikimi i intérproçeseve duket i thjeshtë, apo jo? Harrojeni. Shikoni me vëmendje rregullin e vendosjes së down-eve përpëra se të vendosni apo të tërhoiqni elementë nga buffer-i në Fig. 2-24. Supozojmë sikur dy down-et në kodin e prodhuesit të ishin ndryshuar në rradhë, kështu *mutex* do të dekrementohej përpëra *empty* dhe jo pas tij. Në se bufferi do të ishte plot, prodhuesi do të bllokohej, dhe *mutex* do të ishte 0. Rrjedhimisht, herën tjetër që konsumatori do të mundohej të aksesontë bufferin, do tëkryentenjë down mbi *mutex*, që tani është 0 dhe do të bllokohej gjithashtu. Të dy proçeset do të mbetësin të bllokuar përgjithmonë dhe nuk do të kryhej më punë. Kjo situatë e pafat quhet deadlock. Do ti studiojmë më mirë në Kap. 3.

Ky problem theksohet për të treguar sa i kujdeshëm duhet të jesh kur përdor semaforët. Një gabim i lehtë çon në një blokim të ashpër. Është si programimi në gjuhën assembler, vetëm se më keq sepse gabimet janë kushte përparësie (race conditions), deadlock, dhe forma të tjera sjelljeje të paparashikueshme.

Për ta bërë më të lehtë shkrimin e programeve të rregullta, Hoare (1974) dhe Brinch Hansen (1975) propozuan një primitivë me sinkronizim të nivlit të lartë të quajtur **monitor**. Propozimi i tyre ndryshonte shumë pak nga sa përshkruhet më poshtë. Një monitor është një grup procedurash, variablesh dhe struktura të dhënash që janë të gjitha të mbledhura bashkë në një tip të veçantë moduli apo paketë. Proseset mund të thërrasin procedurat që ndodhen në një monitor kur të duan, por nuk mund të aksesojnë strukturat e të dhënavë të brëndëshme të monitorit me anë të procedurave të deklaruara jashtë tij. Figura 2-26 ilustron një monitor të shkruar në një gjuhë imagjinare, Pidgin Pascal.

monitor example

integer *i*;

condition *c*;

procedure *producer*();

.

.

.

end;

procedure *consumer*();

.

.

end;

end monitor;

Figura 2-26. Një monitor.

Monitorët kanë një karakteristikë të rëndësishme që i bën ata të dobishëm për arritjen e përjashtimit të ndërsjelltë (mutual exclusion): në një cast të caktuar vetëm një proces mund të jetë aktiv në një monitor. Monitorët janë një konstrukt i gjuhës së programimit, kështu kompilatori e di që janë të veçantë dhe mund të merret me thirrjet drejt procedurave të monitorëve ndryshe nga thirrjet e tjera procedurë. Në përgjithësi, kur një proces thërret një procedurë nga monitori, instruksionet e para të procedurës do të kontrollojnë në se ka ndonjë proces tjetër aktiv në monitor. Ne së ka, procesi që ka kryer thirrjen do të pezullohet derisa procesi tjetër të lërë monitorin. Në se nuk ka proces tjetër duke përdorur monitorin, atëherë procesi thirrës mund të futet.

Implementimi i përjashtimit të ndërsjelltë tek monitorët është në dorë të kompilatorit, por një mënyrë e zakonshme është përdorimi i një mutex apo një semafori binar. Është më pak e mundur që diçka do të shkojë gabim, sepse është kompilatori dhe jo programuesi, që merret me përjashtimin e ndërsjelltë. Është e mjaftueshme të dimë që duke i kthyer të gjitha zonat kritike drejt procedurave të monitorëve, nuk do të ketë më dy procese që ekzekutojnë zonat e tyre kritike në të njëjtën kohë.

Edhe pse monitorët na paraqesin një mënyrë të thjeshtë për të arritur përjashtimin e ndërsjelltë, siç e pamë më lart, kjo nuk është e mjaftueshme. Na duhet edhe një mënyrë për të bllokuar proceset kur nuk mund të vazhdojnë më. Në problemin prodhues-konsumator, është e lehtë që të gjithë testët për buffer-full dhe buffer-empty të vendosen në procedurat monitor, por si do të bllokohet prodhuesi kur gjen buffer plot?

Zgjidhja qëndron në futjen e **variblave të gjendjes** dhe dy veprimeve në to, wait (prit) dhe signal (sinjal). Kur një procedurë monitor zbulon se nuk mund të vazhdojë më (per shembull,.., prodhuesi gjen buffer-in plot), ekzekuton një wait në ndonjë variabël gjendjeje, të themi, *full*. Ky veprim shkakton procesin që ka kryer thirrjen të bllokohet. Ai gjithashtu lejon një proces tjetër, që më parë nuk ishte lejuar, të futet tanë.

Ky proces tjetër, për shembull, konsumatori, mund të zgjоjë partnerin e tij duke ekzekutuar një sinjal mbi variablën e gjendjes mbi të cilën po pret partneri. Për të mënjanuar të paturin e dy proceseve aktive në monitor në të njëjtën kohë, na duhet një rregull që na tregon çfarë ndodh pas sinjalit. Hoare propozoi që procesi i sapozgjuar të lejohej të ekzekutohej, duke pezulluar tjetrin. Brinch Hansen propozoi të rregullojë problemin duke kërkuar që një proces që kryen një sinjal të dalë menjehershë nga monitori. Me fjalë të tjera, një rresht sinjal do të paraqitet si rreshti i fundit në një procedurë monitor. Ne do të përdorim propozimin e Brinch Hansen, sepse është më i thjeshtë në koncept dhe gjithashtu më i lehtë për t'u implementuar. Në se ekzekutohet një signal në një variabël gjendjeje në të cilën po presin disa procese, vetëm njëri prej tyre zgjohet, kush përcaktohet nga skeduleri i sistemit.

Ka dhe një zgjidhje të tretë, të cilën nuk e ka propozuar as Hoare as Brinch Hansen. Kjo është për të lënë sinjalizuesin të vazhdojë së ekzekutuar dhe të lejojë procesin në pritje të fillojë ekzekutimin vetëm pasi sinjalizuesi të ketë lënë monitorin.

Variablat e gjendjes nuk janë numërues. Ata nuk i ruajnë sinjalet për t'u përdorur më pas siç bëjnë semaforët. Kështu në se sinjalizohet një variabël gjendjeje në të cilën nuk ka proces duke pritur, sinjali humb përgjithmonë. Me fjalë të tjera, wait duhet të vijë përparrë signal. Ky rregull e bën implementimin shumë më të thjeshtë. Në praktikë nuk paraqitët ndonjë problem, meqënë se është e thjeshtë të ruhet gjendja e secilit proces me anë të variablate, në se duhet. Një proces që do të kryejë një veprim signal mund të shohë që ky veprim nuk është i domosdoshëm duke parë variablat.

Një skelet i problemit prodhues-konsumator me monitorë është dhënë në Fig. 2-27 në një gjuhë imagjinare, Pidgin Pascal. Avantazhi i përdorimit të gjuhës Pidgin Pascal në këtë rast është se ajo është e pastër dhe e thjeshtë dhe ndjek ekzaktësisht modelin Hoare/Brinch Hansen.

monitor *ProducerConsumer*

condition *full, empty;*

integer *count;*

procedure *insert(item: integer);*

begin

if *count = N* **then** **wait**(*full*);

insert_item(item);

count := count + 1;

if *count = 1* **then** **signal**(*empty*)

end;

function *remove: integer;*

begin

if *count = 0* **then** **wait**(*empty*);

remove = remove_item;

```
count := count - 1;  
if count = N - 1 then signal(full)  
end;
```

```
count := 0;  
end monitor;
```

```
procedure producer;  
begin  
    while true do  
        begin  
            item = produce_item;  
            ProducerConsumer.insert(item)  
        end  
    end;
```

```
procedure consumer;  
begin  
    while true do  
        begin  
            item = ProducerConsumer.remove;  
            consume_item(item)  
        end
```

end;

Figura 2-27. Një skicë e problemit prodhues-konsumator me monitorë. Vetëm një procedurë monitor është aktive në një çast të caktuar. Buffer-i ka N slotë.

Mund të mendohet se veprimet wait dhe signal duken të ngjashme me sleep dhe wake up, që më sipër pamë që kishin kushte përparësie fatale. Ato janë shumë të ngjashme, por me një ndryshim thelbësor: sleep dhe wake up dështuan sepse njëri proces po mundohej të futej në gjumë, tjetri po mundohej ta zgjontë. Me monitorët kjo nuk mund të ndodhë. Përjashtimi i ndërsjelltë automatik i procedurave monitor e garanton që, të themi, në se prodhuesi brënda një procedure monitor zbulon që bufferi është plot, do të mund të kompletojë veprimin wait pa u shqetësuar për mundësinë që skeduleri mund të kalojë tek konsumatori para se të kompletohet wait. Konsumatori nuk do të lejohet të futet në monitor para se të kompletohet wait dhe prodhuesi të jetë shënuar si jo në ekzekutim.

Edhe pse Pidgin Pascal është një gjuhë imaginare, ka disa gjuhë programimi që suportojnë monitorët, edhe pse jo gjithmonë në formën e projektuar nga Hoare dhe Brinch Hansen. Një nga këto gjuhë është edhe Java. Java është një gjuhë e orientuar nga objekti që suporton thread-et në nivelin user dhe gjithashtu lejon metodat (procedurat) të grupohen në klasa. Duke shtuar një fjalë kyçe në sinkronizim me deklarimin e metodës, Java garanton që në se një thread fillon ekzekutimin e një metode, asnjë thread nuk do të lejohet të fillojë ekzekutimin e metodave të tjera në atë klasë.

Një zgjidhje e problemit prodhues-konsumator duke përdorur monitoret në Java është dhënë në Fig. 2-28. Në zgjidhje ka katër klasa. Klasa e jashtme, *ProducerConsumer*, krijon dhe fillon dy threade, *p* dhe *c*. Klasa e dytë dhe e tretë, *producer* dhe *consumer*, përbajnë kodin për prodhuesin dhe konsumatorin respektivisht. Së fundi, klasa *our_monitor*, është monitori. Ajo përmban dy thread-e të sinkronizuara që përdoren për të futur elementë në bufferin e share-uar dhe për ti nxjerrë ato. Ndryshe nga shembulli i mëparshëm, këtu kemi treguar kodin e plotë për *insert* dhe *remove*.

Thread-et e prodhuesit dhe konsumatorit janë identike në funksion me homologët e tyre në shembullin e mëparshëm. Prodhuesi ka një lak të pafund që gjeneron të dhëna dhe i fut ato në një buffer të përbashkët. Konsumatori ka një lak të ngjashëm të pafund që merr të dhëna nga bufferi.

Pjesa interesante e programit është klasa *our_monitor*, e cila përmban buffer-in, variablat administrative, dhe dy metoda të sinkronizuara. Kur prodhuesi është aktiv brenda *insert*, e ka të sigurt që konsumatori nuk mund të jetë aktive brenda *remove*, duke e pasur pa rrezik ndryshimin e variablate dhe buffer-it pa frikën e kushteve të përparësisë. Variabla *count* ruan sa elementë ndodhen në buffer. Mund të marrë çdo vlerë nga 0 deri në $N - 1$, duke e përfshirë këtë të fundit. Variabla *lo* është indeksi i slotit të buffer-it nga do të ngarkohet elementi i ardhshëm. Në mënyrë të ngjashme, *hi* është indeksi i slotit të buffer-it ku do të vendoset elementi i ardhshëm. Lejohet që *lo* = *hi*, që do të thotë që në buffer mund të ketë 0 ose N elementë. Vlera e *count* tregon se cili rast qëndron.

Metodat e sinkronizuara në Java ndryshojnë nga monitorët klasik në një mënyrë thelbësore: Java nuk ka variabla gjendjesh. Por, ajo ofron dy procedura, wait dhe notify që janë ekuivalentët me sleep dhe wake up, por që kur përdoren brenda një metode të sinkronizuar, nuk janë shkak për kushte përparësie. Në tëori, metoda wait mund të ndërpritet, dhe këtë bën kodi që e rrethon atë. Java kërkon që të bëhet i qartë trajtimi i përjashtimeve. Për synimin tonë, imagjinojmë që *go_to_sleep* është mënyra për ta vënë në gjumë.

```

public class
ProducerConsumer {
    static final int N = 100;           // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor

    public static void main(String args[ ]) {
        p.start();      // start the producer thread
        c.start();      // start the consumer thread
    }

    static class producer extends Thread {
        public void run( ) { // run method contains the thread code
            int item;
            while(true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item (){ ... } // actually produce
    }

    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while(true) { // consumer loop
                item = mon.remove();
                consume_item (item);
            }
        }
        private void consume_item (int item) { ... } // actually consume
    }

    static class our_monitor {           // this is a monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // counters and indices
    }
}

```

```

public synchronized void insert (int val) {
    if(count == N) go_to_sleep(); //if the buffer is full, go to sleep
    buffer [hi] = val;          // insert an item into the buffer
    hi = (hi + 1) % N;         // slot to place next item in
    count = count + 1;          // one more item in the buffer now
    if(count == 1) notify();    // if consumer was sleeping, wake it up
}

public synchronized int remove( ) {
    int val;
    if(count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
    val = buffer [lo];          // fetch an item from the buffer
    lo = (lo + 1) % N;          // slot to fetch next item from
    count = count - 1;          // one few items in the buffer
    if(count == N - 1) notify(); // if producer was sleeping, wake it up
    return val;
}
private void go_to_sleep() { try{wait( );} catch{ InterruptedException exc) { };}
}
}

```

Figura 2-28. Një zgjidhje e problemit prodhues-konsumator në Java.

Duke e bërë përjashtimin e ndërsjelltë të zonave kritike automatik, monitorët e bëjnë programimin paralel me më pak mundësi gabimesh sesa me semaforët. Megjithatë edhe ata kanë një dizavantazh. Jo më kot dy shembujt tanë me monitorë ishin në Pidgin Pascal dhe Java dhe jo në C, ashtu sic janë edhe shembujt e tjerë në këtë libër. Ashtu si thamë më parë, monitorët janë një koncept gjuhe programimi. Kompilatori duhet ti njoħe ato dhe ti përshtasë për përjashtimin e ndërsjelltë. C, Pascal dhe shumica e gjuhëve të tjera nuk kanë monitorë, kështu është e paarsyeshme të presësh që kompilatorët e tyre të caktojnë rregulla për përjashtimin e ndërsjelltë. Në fakt, si do ta kuptonte kompilatori se cilat procedura ndodhen në monitor dhe cilat jo?

As këto gjuhë të tjera nuk kanë semaforë, por shtimi i tyre është i thjeshtë: na duhet të shtojmë në librari dy rutina të shkruara në gjuhën asembler për të përfaqësuar thirrjet sistem up dhe down. Kompilatoret nuk duhet as ta dijnë që ekzistojnë. Sigurisht që sistemet operative duhet të dinë që ka semaforë, megjithatë në qoftë se kemi një sistem operativ të bazuar në semaforë, mund të shkruajmë programe user në C ose C++ (apo edhe në asembler). Me monitorët, na duhet një gjuhë që ti ketë të përfshirë.

Një problem tjetër me monitoret dhe gjithashtu me semaforët, është që ata u projektuan për të zgjidhur problemin e përjashtimit të dyanshëm në një ose më shumë CPU, dhe të gjithë kanë akses në një memorie të përbashkët. Duke i vënë semaforët në memorien e share-uar dhe mbrojtja e tyre me anë të instruksioneve TSL, mund të

mënjanojmë kushtet e pëerparësisë. Kur shikojmë një sistem të shpërndarë që konsiston në shumë CPU, secili me memorien e tij personale, të lidhur nga një local area network, këto primitiva bëhen të papërdorueshme. Përfundimi është se semaforët janë të nivelist shumë të ulët dhe monitorët nuk janë të përdorshëm, përvèç se në disa gjuhë programimi. Dhe gjithashtu asnjëra nga primitivat e dhëna më sipër për shkëmbimin e informacionit ndërmjet makinave. Nevojitet diçka tjeter.

2.3.8 Shkëmbimi i Mesazheve

Ajo diçka tjeter është **shkëmbimi i mesazheve** (message passing). Metoda e komunikimit të intérprocéseve përdor dy primitiva, send (dërgo) dhe receive (merr), që si semaforët por jo si monitorët, janë thirrje sistem dhe jo konstruktë gjuhe. Si të tilla, ato mund të vendosen shumë thjeshtë në procedura librari, si për shembull;

```
send(destination, &message);
```

dhe

```
receive(source, &message);
```

Thirja e parë dërgon një mesazh në një vendndodhje të dhënë dhe e dyta merr një mesazh nga një burim i dhënë (ose prej një *çfarëdo*, në se për marrësin nuk ka rëndësi). Në se nuk ka asnje mesazh të mundshëm, marrësi bllokohet derisa të vijë një i tillë, ose mund të kthejë menjehere me një kod gabimi.

Problemet e projektimit të sistemeve për shkëmbimin e mesazheve

Sistemet e shkëmbimit të mesazheve kanë shumë probleme dhe çështje projektimi që nuk lindin me semaforët apo monitorët, në veçanti në se proceset komunikuese janë në makina të ndryshme të lidhura nga një network. Për shembull, mesazhet mund të humbën nga network-u. Për t'u mbrojtur nga humbja e mesazheve, dërguesi dhe marrësi mund të bien dakord që sapo të merret një mesazh, marrësi do të kthejë një mesazh **acknowledgement**. Në se dërguesi nuk ka marrë acknowledgement brënda një intervali kohor të caktuar, ai ritransmeton mesazhin.

Tani le të marrin në konsiderate çfarë ndodh në se mesazhi merret saktë, por humbet acknowledgement. Dërguesi do të ritransmetojë mesazhin, kështu marrësi do ta marrë atë dy herë. Është thelbësore që marrësi të dijë të dallojë një mesazh të ri nga një të vjetër. Zakonisht, ky problem zgjidhet duke vënë numra të njëpasnjëshëm në çdo mesazh.

Në se marrësi merr një mesazh që mban të njëjtin numër si mesazhi paraardhës, ai e di që mesazhi është duplikat dhe mund të injorohet. Komunikimi i suksesshëm është një pjesë e rëndësishme në studimin e rrjetave kompjuterike. Për më shumë informacion, shiko (Tanenbaum, 1996).

Sistemet e mesazheve duhet të merren edhe me pyetjen se si emërohen proçeset, në mënyrë që proçesi që specifikohet në një thirrje send apo receive të jetë i qartë. **Njohja** (autëntikimi) është gjithashtu një problem në sistemet e mesazheve: si mund të vërtëtojë klienti që po kommunikon me serverin e vertet dhe jo me një mashtrues?

Nga ana tjetër e spektrit, ka edhe probleme projektimi që janë të rëndësishme kur dërguesi dhe marrësi ndodhen në të njëjtën makinë. Njëra prej tyre është përformanca. Kopjimi i mesazheve nga një proçes tek një tjetër është gjithmonë më e ngadaltë sesa të bësh një veprim me semafor apo të aksesosh një monitor. Shumë punë është bërë për të bëre kalimin e mesazheve efikas. Cheriton (1984), për shembull, sugjeroi të limitohet size i mesazheve që të futet në regjistrat e makinës, dhe më pas të bëhet shkëmbimi i mesazheve duke përdorur regjistrat.

Problemi Prodhues-Konsumator me Shkëmbimin e Mesazheve

Tani le të shohim si mund të zgjidhet problemi prodhues-konsumator me shkëmbimin e mesazheve dhe jo memorie të share-uar. Një zgjidhje është dhënë në Fig. 2-29. Supozojmë që të gjithë mesazhet janë në të njëjtën madhësi, dhë që, mesazhet që janë dërguar, por nuk janë marrë akoma vendosen në buffer automatikisht nga sistemi operativ. Në këtë zgjidhje, përdoren N mesazhe, në analogji me N slotët e memories së share-uar në buffer. Konsumatori fillon duke i dërguar prodhuesit N mesazhe bosh. Sapo prodhuesi të ketë një element për ti dërguar konsumatorit, ai merr një mesazh bosh dhe kthen një të plotë. Në këtë mënyrë, numri total i mesazheve në sistem mbetet i pandryshueshëm në kohë, kështu që mund të ruhen në një sasi të caktuar memorie të ditur më parë.

Në se prodhuesi punon më shpejt se konsumatori, të gjithë mesazhet do të mbushen plot, duke pritur për konsumatorin: prodhuesi do të bllokohet, duke pritur për një mesazh bosh për t'u kthyer. Në se konsumatori punon më shpejt, atëherë ndodh e kundërta: të gjithë mesazhet do të janë boshe duke pritur që prodhuesi ti mbushë: konsumatori do të bllokohet, duke pritur për një mesazh plot.

```
#define N 100 /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m; /* message buffer */
```

```

while (TRUE) {
    item = produce_item( );      /* generate something to put in buffer */
    receive(consumer, &m);     /* wait for an empty to arrive */
    build_message (&m, item);   /* construct a message to send */
    send(consumer, &m);        /* send item to consumer */
}
}

void consumer(void) {
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);      /* get message containing item */
        item = extract_item(&m);   /* extract item from message */
        send(producer, &m);        /* send back empty reply */
        consume_item(item);       /* do something with the item */
    }
}

```

Figura 2-29. Problemi prodhues-konsumator me N mesazhe.

Ka shumë variante të mundshme për shkëmbimin e mesazheve. Si fillim, le të shohim si adresohen mesazhet. Një mënyrë është ti bashkangjisim çdo procesi një adresë të veçantë dhe mesazhet ti adresojmë drejt proceseve. Një mënyrë tjetër është të shpikim një strukturë të dhënash të re, të quajtur një **mailbox** (kuti postare). Një mailbox është një vend për të ruajtur një numër të caktuar mesazhesh, në përgjithësi numri caktohet kur krijohet mailbox-i. kur përdoren mailbox-et, parametrat e adresave, në thirrjet send dhe receive, janë mailbox-et dhe jo proceset. Kur një proces tenton ti dërgojë një mailbox-i që është plot, pezullohet derisa të largohet një mesazh nga mailbox-i dhe të hapë vend për një tjetër.

Për problemin prodhues-konsumator, edhe prodhuesi edhe konsumatori do të krijojnë një mailbox të madh sa për të mbajtur N mesazhe. Prodhuesi do të dërgojë mesazhe që pëmbajnë të dhëna në mailbox-in e konsumatorit, dhe konsumatori do të dërgojë mesazhe bosh në mailbox-in e prodhuesit. Kur përdoren mailbox-et, mekanizmi i buffer-it është i qartë: mailbox-i destinacion mban mesazhe që i janë dërguar procesit destinacion, por nuk janë pranuar akoma.

Ekstremiteti tjetër nga ai i të paturit mailbox-e është eliminimi i bufferit. Kur ndiqet ky trajtim, në se ekzekutohet send përpala receive, procesi dërgues bllokohet derisa të ndodhë receive, në këtë moment mesazhi mundet tanë të kalojë direkt nga dërguesi tek marrësi, pa përdorur buffer si ndërmjetës. Në mënyrë të ngjashme, në se veprimi receive kryhet i pari, marrësi bllokohet derisa të ndodhë një veprim send. Kjo

strategji shpesh herë njihet si një **rendezvous**. Është më i thjeshtë për t'u implementuar sesa një skemë mesazhesh me buffer, por është më pak fleksibël meqënë se dërguesi dhe marrësi janë të detyruar të punojnë me turne.

Shkëmbimi i mesazheve është shumë e përdorur në sisteme me programim paralel. Një sistem me programim paralel e mirënjojur është, për shembull, **MPI** (**Message-Passing Interface**). Është shumë i përdorur për llogaritje shkencore. Për më shumë informacion për këtë shikoni (Gropp et al., 1994; dhe Snir et al., 1996).

2.3.9 Barrierat

Mekanizmi i fundit i sinkronizimit është i përshtatshëm për situata me shumë proçese dhe jo për ato me dy proçese prodhues-konsumator. Disa aplikacione janë të ndara në fazë dhe kanë rregullin që asnjë proçes nuk mund të kalojë në fazën pasardhëse pa mbaruar të gjithë proçeset fazën paraardhëse. Kjo sjellje mund të arrihet duke vendosur një **barrierë** në fund të çdo faze. Kur një proçes mbërrin në barrierë, bllokohet derisa të gjithë proçeset të kenë arritur barrierën. Veprimi i një barriere është treguar në Fig. 2-30.

Në Fig. 2-30(a) shikojmë katër proçese që mbërrijnë në barrierë. Kjo do të thotë që po veprojnë dhe që nuk kanë arritur ende në fundin e fazës. Më pas, proçesi i parë përfundon gjithë punën e fazës së parë. Pastaj ekzekuton primitivën e barrierës, përgjithësisht duke thirrur një procedurë librarie. Proçesi pastaj pezullohet. Më pas, një proçes i dytë dhe më pas një i tretë përfundojnë fazën e parë dhe gjithashtu ekzekutojnë primitivat e barrierës. Kjo situatë ilustrohet në Fig. 2-30(b). më në fund, kur proçesi i fundit, C, mbërrin në barrierë, të gjithë proçeset lironen, si tregohet në Fig. 2-30(c).

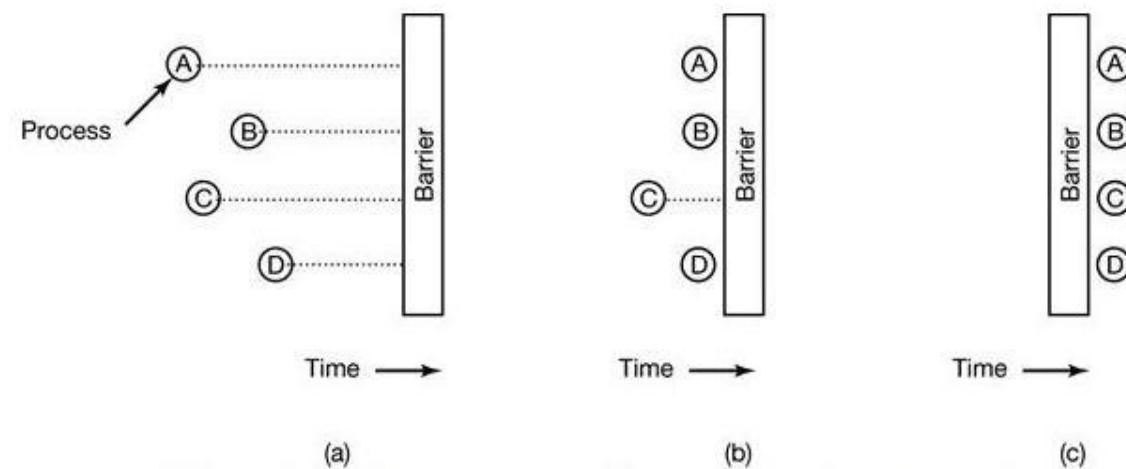


Figura 2-30. Përdorimi i një barriere. (a) Proçesi duke ju afruar barrierës. (b) Të gjithë proçeset veç njërit janë bllokuar. (c) Kur proçesi i fundit mbërrin në barrierë, të gjithë lejohen të kalojnë.

Si një shembull i një problemi që ka nevojë për barrierë, konsiderojmë një problem tipik në fizikë apo inxhinieri. Kemi një matricë tipike që përmban disa vlera fillestare. Vlera mund të përfaqësojnë temperaturat në pika të ndryshme të një fletë metalike. Ideja mund të jetë të llogaritet sa i duhet një flake të vendosur në njërin cep të japë efekt dhe të shpërhatet mbi gjithë fletën.

Duke filluar nga këto vlera, mbi matircë kryhet një transformim për të marrë versionin e dytë të matricës, për shembull, duke zbatuar ligjet e termodinamikës për të parë sa janë temperaturat pas ΔT . Më pas proçesi përsëritet dhe jep temperaturat në pika të caktuara, si një funksion i kohës ndërsa fleta nxehet. Algoritmi në këtë mënyrë prodhon një seri matricash me kalimin e kohës.

Tani imagjinojmë se matrica është shumë e madhe (të themi 1 milion me 1 milion), kështu na duhen proçeset paralele (mundësish në një multiproçesor) për të përshpejtuar llogaritjet. Proçese të ndryshme punojnë në pjesë të ndryshme të matricës, dhe kështu llogarisin elementet e matricës së re nga ajo e vjetra, duke u bazuar në ligjet e fizikës. Megjithatë, asnjë proçes nuk mund të fillojë përsëritjen $n + 1$ pa mbaruar përsëritja n , do të thotë, derisa të gjithë proçeset të kenë mbaruar punën e momentit. Mënyra për të arritur këtë qëllim është të programojmë çdo proçes të ekzekutojë një veprim barrierë pasi të ketë mbaruar pjesën e tij. Kur të gjithë të kenë mbaruar, matrica e re (input-i për përsëritjen e ardhshme) do të mbarojë dhe të gjithë proçeset do të lironen në të njëjtën kohë për të filluar ciklin e ardhshëm.

2.4. PROBLEMET KLASIKE IPC

Literatura e sistemeve operative është plot me probleme interesante të cilat jane diskutuar dhe analizuar duke përdorur një shumëllojshmeri metodash sinkronizuese. Ne paragrafet e me poshtme do të shqyrtojme tre probleme të mirenjohur.

2.4.1. PROBLEMI I DARKIMIT TË FILOZOFËVE

Ne 1965, Dijkstra parashtroi dhe zgjidhi një problem sinkronizimi qe e quajti dhe dining phylosophers problem (problemi i darkimit të filozofëve). Deri ne atë kohe, çdo njëri qe krijonte një primitive sinkronizimi u ndje i detyruar të demonstronte se sa elegantë ishte menyra e zgjidhjes se problemeve dining phylosophers. Problemi mund të konstatohet shume thjesht si me poshtë. Pese filozofe jane të ulur ne një tavoline të rrumbullaket. Çdo filozof ka një pjatë me spageti. Spageti është kaq rreshqitës sa një filozof ka nevoje për dy pirune për të. Ndermjet çdo cifti pjatash ka nga një pirun. Paraqitura e tavolines është e ilustruar ne figuren 2.31.

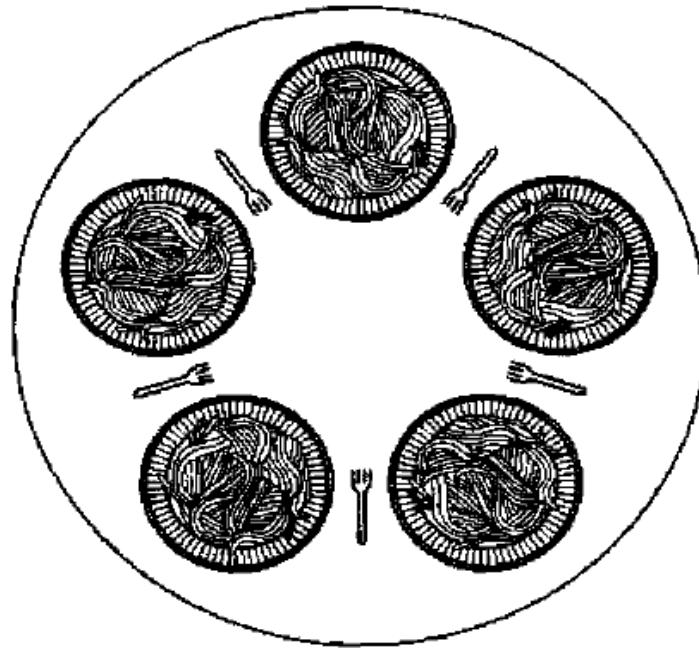


Fig.1.31. Dreka ne departamentin e filozofise.

Jeta e një filozofi konsiston ne perioda të alternuara ngrenieje dhe mendimi. (Kjo është dicka e abstraksionit, madje edhe për filozofet, por aktivitetet e tjera ketu janë të pavend.) Kur një filozof është i uritur, ai mundohet të marre pirunin e tij ne të majtë dhe ne të djathtë, një ne një kohe, ne një rend cfaredo. Ne qoftë se është i sukseshem ne sigurimin e dy piruneve ai ha për pak dhe me pas le pirunet, dhe fillon të mendoje. Pyetja kryesore është: A mund të shkruani ju një program për secilin filizof qe bën atë qe supozohet të beje dhe asnjeherë nuk mërzitet? (Është përmendur qe nevoja e dy pirunjëve është dicka artificiale; mbështetje duhet të ndryshojme ushqimin nga ai Italian ne atë Kinez, duke zevendesuar spageticin me orizin dhe pirunjët me shkopinjtë.)

Figura 2.32. tregon zgjidhje të dukshme. Procedura *take-fork* pret derisa piruni i specifikuar është i disponueshem dhe me pas e kap atë. Fatkeqesisht, kjo zgjidhje e dukshme është gabim. Supozoni qe pese filozofet marrin pirunjët ne të majtë të tyre, ne të njëjtën kohe. Asnjëri nuk është i aftë të marre pirunin ne të djathtë të tij. Ne qoftë se nuk mundet ai le pirunin ne të majtë, pret për pak kohe dhe pastaj përsërët të gjithe procesin. Ky propozim, gjithashtu, për një arsy apo për një tjetër deshton. Me shume pak fat të keq, filozofet mund të fillojne papritur algoritmin, marrin pirunjët ne të majtë të tyre, presin, marrin përsërët papritur pirunjët e tyre të majtë dhe keshtu me rradhe gjithmonë. Një situatë e tille ku të gjithe programet vazhdojnë të ekzekutojnë, por deshtojne ne berjen e çdo një progresi quhet starvation (vdekje nga uria). (Quhet starvation edhe ne se problemi nuk ndodh ne një restorant kinez apo italian).

```
#define N 5           /* number of philosophers */

void philosopher(int i) /* i: philosopher number, from 0 to 4 */
{

```

```

while (TRUE) {
    think();           /* philosopher is thinking */
    take_fork(i);     /* take left fork */
    take_fork((i+1) % N); /* take right fork; % is modulo operator */
    eat();            /* yum-yum, spaghetti */
    put_fork(i);      /* Put left fork back on the table */
    put_fork((i+1) % N); /* put right fork back on the table */
}
}

```

Fig.1.32. Problemi i darkimit të filozofeve pa zgjidhje.

Tani ju mund të mendoni “po ne qoftë se filozofi do të priste një kohe të rastesishme dhe jo të njëjtën kohe pas deshtimit të marrjes se pirunit ne të djathtë, shanci qe çdo gje do të vazhdoje ne lockstëp për edhe një ore është shume i vogel”. Ky vrojtim është i vertetë, dhe pothuajse ne të gjitha aplikimet të provuarit përseni me vone nuk është problem. Për shembull, ne sipërfaqen lokale të rrjetit Ethernet, ne qoftë se dy kompjutera dergojne paketa ne të njëjtën kohe, secili prej tyre pret një fare kohe të rastesishme dhe provon përseni; Ne praktike kjo zgjidhje funksionon mire. Sidoqoftë, ne disa aplikime, një do të preferontë një zgjidhje qe funksionon gjithmone dhe nuk mund të deshtoje ne saj të një serie numrash të rastesishem të papelqyeshem. Mendoni rreth kontrollit të sigurt ne një uzine nukleare.

Një përmiresim i figures 1.32. qe nuk ka asnë deadlock dhe asnë starvation është të mbrojme pese formulimet qe ndjekin thirrjen *think* nga një semafor binar. Para fillimit të marrjes se pirunjve, filozofi duhet të beje një zbritje ne *mutex*. Pas rivendosjes se pirunjve, ai duhet të bej një ngritje ne *mutex*. Nga kendveshtrimi teorik kjo zgjidhë është e përshtatshme. Por nga ana praktike ka një të metë ne performance: ne një cast kohe të caktuar vetëm një filozof mund të jetë duke ngrene. Me pese pirunj të disponueshem duhet të jemi të aftë të lejojme dy filozofë të hane ne të njëjtën kohe.

Zgjidhja e treguar ne figuren 2.33. është pa deadlock dhe lejon paralelizmin maksimal për një numer arbërtar filozofesh. Ai përdor një tabele, *state*, për të mbajtur gjurme ne se një filozof po ha, po mendon apo është i uritur (duke u munduar të marre pirunjtë). Një filozof mund të levize ne state-in ‘duke ngrene’ ne qoftë se asnë nga komshinjtë nuk po ha. Komshinjtë e filozofit të i-të përcaktohen nga makrot LEFT dhe RIGHT. Ne fjale të tjera ne qoftë se *i* është 2, LEFT është 1 dhe RIGHT është 3.

Programi përdor një grup semaforesh, një për çdo filozof, keshtu qe filozofet e uritur mund të bllokojnë ne qoftë se pirunjtë e nevojitur janë të zene. Vini re qe secili proces ekzekuton proceduren *philosopher* si kodin e tij kryesor, por procedurat e tjera, *take-forks*, *put-fork*, dhe *test* janë procedura të zakonshme, dhe nuk ndajne proceset.

```

#define N      5 /* number of philosophers */
#define LEFT   (i+N-1)%N /* number of i's left neighbor */
#define RIGHT  (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY   1 /* philosopher is trying to get forks */
#define EATING   2 /* philosopher is eating */

```

```

typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher (int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) { /* repeat forever */
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}

void take_forks(int i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

Fig.1.33. Një zgjidhje për problemin e darkimit të filozofeve

2.4.2. PROBLEMI I SHKRUESVE DHE I LEXUESVE

Problemi i darkimit të filozofeve është i dobishem për proceset e modelimit qe jane konkurese për akses eksklusive të një numri të limituar burimesh, si pajisjet I/O.

Një tjetër problem i famshem është problemi i shkruesve dhe i lexuesve (Courtois et al, 1971), modelet e të cilët aksesojnë ne një database. Imagjinoni, për shembull, një sistem rezervimi ajror, me shume procese konkuruuese që duan të lexojne apo të shkruajne. Është e pranueshme të kemi shume procese që lexojne database ne të njëjtën kohe, por ne qoftë se një proces ishte shkruajtur database-in, asnë proçes tjetër nuk mund të ketë akses te database, madje as edhe lexuesit. Pyetja është si i programoni lexuesit dhe shkruesit? Një zgjidhje tregohet ne figuren 2.34.

Ne ketë zgjidhje, lexuesi i pare që do të marre akses të database bën një zbritje ne semaforin *db*. Lexuesit pasues thjesht inkrementojne një numerues, *rc*. Sapo të ikin lexuesit, ato dekrementojne numeruesin dhe i fundit bën një ngritje të semaforit, duke lejuar një shkrues të blokuar, ne qoftë se ka një të tille, të futet.

Zgjidhja e prezantuar ketu ne menyre implicitë përmban një vendim delikat për të cilin ia vlen të komentojme dicka. Supozoni qe nderkohe qe një lexues po përdor database-in, vjen një lexues tjetër. Meqene se të paturit e dy lexuesve ne të njëjtën kohe nuk përbën ndonjë problem, lexuesi i dytë pranohet. Një lexues i tretë dhe të tjere pasues mund të pranohen ne qoftë se ato do të vijne.

Tani supozoni se vjen një shkrues. Shkruesi nuk mund të pranohet ne database përderisa shkruesit duhet të kene akses ekskluzive, keshtu shkruesi pezullohet. Me vone, shfaqen lexues të tjere. Për sa kohe qe të paktën një lexues është aktiv, lexues të tjere pasardhes pranohen. Si pasoje e kesaj strategjie, për sa kohe të ketë një furnizim të rregullt lexuesish, ata do të futen apo të mberrijne. Shkruesi do të mbahet i pezulluar derisa të mos ketë asnë lexues. Ne qoftë se vjen një lexues i ri, le të themi, çdo 2 sekonda dhe secili lexues kerkon 5 sekonda pér të bere punen e tij, shkruesi nuk do të futet asnjeherë.

Për të parandaluar ketë situatë, programi mund të shkruhet pak ndryshe: kur mberrin një lexues dhe një shkrues është duke pritur, lexuesi ne vend qe të pranohet menjehere pezullohet pas shkruesit. Ne ketë menyre shkruesit i duhet të prese vetëm pér lexuesit qe ishin aktiv kur ai mberrin, por nuk duhet të prese pér lexuesa qe vijnë pas tij. Disavantazhi i kesaj zgjidhjeje është se ajo con ne uljen e konkurencës dhe si pasoje ne një performance me të ulet. Courtois et al, prezantojnë zgjidhje qe u jep prioritet shkruesve.

```

typedef int semaphore;      /* use your imagination */
semaphore mutex = 1;        /* controls access to 'rc' */
semaphore db = 1;           /* controls access to the database */
int rc = 0;                 /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {          /* repeat forever */
        down(&mutex);       /* get exclusive access to 'rc' */
        rc = rc + 1;         /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader... */
        up(&mutex);          /* release exclusive access to 'rc' */
        read_data_base();    /* access the data */
        down(&mutex);       /* get exclusive access to 'rc' */
}

```

```

rc = rc - 1;          /* one reader fewer now */
if (rc == 0) up(&db); /* if this is the last reader... */
up(&mutex);          /* release exclusive access to 'rc' */
use_data_read();     /* noncritical region */
}
}

void writer(void)
{
    while (TRUE) {      /* repeat forever */
        think_up_data(); /* noncritical region */
        down(&db);       /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db);         /* release exclusive access */
    }
}

```

Fig.1.34. Një zgjidhje e problemit të lexuesve dhe shkruesve.

2.4.3. PROBLEMI I BERBERIT QE FLE

Një tjetër problem klasik IPC ndodh ne një dyqan berberi. Dyqani berberit ka një berber, një karrige berberi, dhe n karrige për tu ulur klientët qe presin, ne qoftë se ka. Ne qoftë se nuk ka asnë klient, berberi ulet ne karrigen e berberit dhe bie ne gjume, sic ilustrohet ne figuren 2.35. Kur vjen një klient, atij i duhet të zgjoje berberin qe po fle. Ne qoftë se gjatë kohes qe po pret floket e një klienti vijne klientë të tjere, ata ose ulen (ne qoftë se ka karrige bosh) ose ikin (ne qoftë se nuk ka karrige bosh). Problemi është të programojme berberin dhe klientët pa shkuar ne kushte race. Ky problem është i ngjashem me shume situata të gershetuara, si një zyre ne ndihme të shume personave me një sistem të kompjuterizuar pritjeje të thirrjes për mbajtjen e një numri të limituar thirrjesh hyrese.

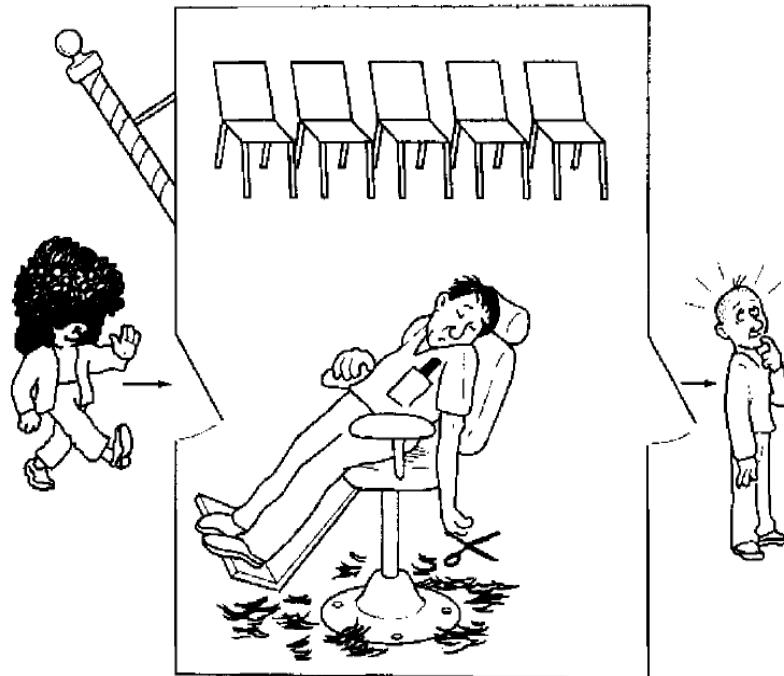


Fig.2.35.Berberi qe fle.

Zgjidhja jone përdor tre semafore: *customers*, i cili numeron klientët qe po presin (duke mos përfshire klientin ne karrigen berberit, qe nuk po pret), *barbers*, numrin e berbereve (0 ose 1) qe rrine kot, duke pritur për klientë dhe *mutex*, qe përdoret për përjashtimin e ndersjelltë (mutual exclusion). Gjithashtu duhet një variabel, *waiting*, e cila gjithashtu numeron klientët qe po presin. Është thelbesore një kopje e *customers*. Arsyeha e të paturit *waiting* është sepse nuk ka asnje menyre për të lexuar vleren aktuale të semaforit. Ne ketë zgjidhje, një klient qe hyn ne dyqan duhet të numeroje klientat qe po presin. Ne qoftë se ka me pak se numri i karrigeve, ai qendron; për ndryshe, iken.

Zgjidhja jone tregohet ne figuren 2.36. Kur berberi shfaqet ne mengjes për pune, ai ekzekuton proceduren *barber* sepse ajo fillimisht është 0. Me pas berberi shkon të fleje, sic tregohet ne fig.2.35. Ai fle derisa të vije klienti i pare.

```
#define CHAIRS 5           /* # chairs for waiting customers */
typedef int semaphore;    /* use your imagination */
semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;    /* # of barbers waiting for customers */
semaphore mutex = 1;      /* for mutual exclusion */
int waiting = 0;          /* customers are waiting (not being cut) */

void barber(void)
{
    whitë (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex);     /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
    }
}
```

```

        up(&barbers);      /* one barber is now ready to cut hair */
        up(&mutex);       /* release 'waiting' */
        cut_hair();        /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);       /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);    /* wake up barber if necessary */
        up(&mutex);        /* release access to 'waiting' */
        down(&barbers);    /* go to sleep if # of free barbers is 0 */
        get_haircut();     /* be seated and be serviced */
    } else {
        up(&mutex);       /* shop is full; do not wait */
    }
}

```

Fig.2.36. Një zgjidhje për problemin e berberit qe fle.

Kur vjen një klient, ai ekzekuton *customer*, duke filluar me sigurimin e *mutex* për të hyre ne një vend kritik. Ne qoftë se një tjetër klient futet pak me pas, i dyti nuk është i aftë të beje asgje derisa i pari të lere *mutex-in*. Me pas klienti kontrollon ne se numri i klientëve qe presin është me i vogel se i karrigeve. Ne qoftë se jo, ai le *mutex* dhe iken pa prere floket.

Ne qoftë se ka një karrige të disponueshme, klienti inkrementon variablin integer, *waiting*. Me pas ai bën një ngritje të semaforin *customers*, keshtu qe zgjon berberin. Ne ketë pike, klienti dhe berberi jane të dy të zgjuar. Kur klienti le *mutex*, berberi e rrëmben atë, bën administrimin dhe fillon prerjen e flokeve.

Kur mbaron prerja e flokeve, klienti del nga procedura dhe iken nga dyqani. Ndryshe nga shembujt e meparshem, nuk ka asnjë loop për klientin sepse secili pret vetëm një here floket. Berberi kthehet për të marre klientin tjetër. Ne qoftë se ka një të tille, behet një tjetër prerje flokesh. Ne qoftë se jo, berberi shkon të flej.

Është e rendesishme të themi se, edhe pse problemet e lexuesit dhe shkruesit dhe berberi qe fle nuk përfshijnë transferimin e të dhenave, ata akoma i përkasin IPC sepse ato përfshijnë sinkronizimin ndermjet shume proceseve.

2.5 SCHEDULIMI

Kur një kompjuter është i multiprogramuar, atëherë ai ka njëkohesisht shume procese qe konkurrojnë për CPU-ne. Kjo situatë ndodh gjithmone kur dy ose me shume procese janë ne të njëjtin moment ne gjendjen Gati. Neqoftëse vetëm një CPU është i gatshem atëherë duhet të behet një zgjidhje për procesin e ardhshem qe do të ekzekutohet. Pjesa e sistemit operativ qe e bën ketë zgjedhje quhet **schedulers** dhe algoritmi qe ai përdori quhet

algoritmi i schedulimit. Keto do jene pikat kryesore për të cilat do të flitet gjatë ketij kapitulli.

Shumica e problemeve qe përbetje schedulimin e procese Jane të njëjtë edhe ne schedulimin e threads edhe se ka ndryshime të vogla. Fillimisht do përqendrohem ne schedulimin e proceseve. Me vone do të shohim imtësisht schedulimin e thread-save.

2.5.1 Hyrje ne konceptin e Schedulimit

Ne kohet kur inputi behej me disqe magnetike, algoritmi I shcedulimit ishte shume i thjesht, ekzekuto punen e ardhshme ne disk. Me sistemin timesharing, algoritmi i schedulimit u be me kompleks sepse përgjithsisht kishte disa përdorues qe prisnin për sherbim. Disa mainframe ende kombinojne sherbimin batch dhe timesharing, duke kerkuar qe scheduleri të vendos në qoftë se një batch job ose një përdorues interactive ne një terminal të vazhdoj. (Një batch job mund të jetë një kerkese për të ekzekutuar me shume programe rradhazi, por ne ketë kapitull do e mendojme si një kerkeser për të ekzekutuar vetëm një program). Sepse CPU është një burim i vogel dhe një scheduler i mire mund të përmiresoje ndjeshem performancen dhe plotësimin e kerkesave të përdoruesit. Rrjedhimisht një pune e madhe është bere ne zhvillimin e algoritmeve të zgjuar dhe eficent.

Me ardhjen e kompjuterave personal, situatë ndryshoi ne 2 pika. E para, shumicen e kohes ka vetëm një process aktiv. Ne qoftë se një përdorues është duke shkruar një dokument ne një Word processor nuk mund të behet njëkohesisht me kompilimin e një programi ne background. Kur një përdorues shkruan një komand ne word processor, scheduleri nuk i duhet shume kohe për të kuptuar se kush process duhet ekzekutuar, sepse word procesori është kandidati i vetëm.

E dyta, kompjuterat jane bere aq të shpejt gjatë viteve saqe CPU nuk është me aq i vogel sa dikur. Shume programe për personal computers jane të kufizuar nga shpejtësia me të cilën përdoruesi mund të japi inputin (duke klikuar ose shtypur tastet), dhe jo nga shpejtësi e ekzekutimit të CPU-se. Ne kohet e shkuara një grup instruksionesh i duheshin një numer ciklesh CPU për ekzekutim, sot duhet e shumta disa sekonda. Edhe kur ekzekutohen njëkohesisht dy programe, si për shembull një word processor dhe një spreadsheet s'ka shume rendesi kush nis i pari sepse ka shume mundesi qe përdoruesi i pret të dy të përfundojne njëkohesisht. Si pasoje, skedulimi nuk ndikon shume ne kompjutarat e thjesht PC. Sigurisht, ka aplikacione qe praktikisht e "hane të gjalle" CPU-ne: të prodhosh një ore video me rezolucion të lartë nevojitet fuqi industriale përpunim imazhi ne secilen prej 108.000 frame-t ne NTSC(90.000 ne PAL), por keto aplikacione jane përjashtime nga rregulli.

Kur flasim për rrjeta serverash ose workstations, situata ndryshon. Ne ketë rast disa procese konkurrense për CPU, keshtu qe skedulimi ka rendesi. Për shembull kur CPU duhet të vendos ne se të ekzekutoj një update të ekranit pasi përdoruesi ka mbyllur një

dritare ose procesin qe dergon një email qe është ne radhe pritje, bën shume ndryshim ne përgjigjen e marre. Ne qoftë se ne do prisnim 2 sekonda deri sa të dergohej emaili dhe me pas të update-hej ekranin. Atëherë përdoruesi do të mendonte se sistemi është tepër i ngadalëtë ndersa ne qoftë se emaili do dergohej 2 sekonda me vone se atëherë përdoruesi as nuk do e kuptonte vonesen. Ne ketë rast skedulimi i procese ka shume rendesi.

Përvec se shqetësimit për të zgjedhur procesin e duhur për ekzekutim, por ai duhet të shqetësohet edhe për të bere sa me eficent përdorimin e CPU, sepse nderrimi i proceseve sjell shume vonesa. Fillimisht duhet të behet një kalim nga user mode ne kernel mode. Me pas gjendja e procesit qe po ekzekutohet duhet të ruhet, përfshire ketu edhe ruajtjen e regjistrave ne tabelen e regjistrave, keshtu mund të ringarkohen me vone. Ne shume sisteme harta e memorjes (për shembull, bitet e references se memorjes ne tabele) duhet të ruhen. Me pas një proces tjetër duhet të zgjidhet duket ekzekutuar algoritmin e schedulimit. Me pas MMU-ja duhet të ringarkohet me hartën e memorjes se procesit të ri. Ne fund procesi i ri duhet të filloj. Për me tepër, ndryshimi i proceseve e bën jo të vlefshme gjithe cache-ne, duke e detyruar atë të ringarkohet ne menyre dinamike nga memorja dy here (ne hyrjen ne kernel dhe ne daljen nga kerneli). Pra ndryshimi i shume proceseve ne sekonde mund te humbe një pjese të madhe të kohes se CPU, keshtu qe keshillohet kujdes.

Sjellja e proceseve

Pothuajse çdo process altëronon përpunimin kompjuterik me (disk) kerkesat I/O, sic tregohet ne fig. 2-37. Normalisht CPU-ja punon për një kohe të konsiderueshme para se të ndaloje, me pas një thirrje sistem behet për të lexuar një file ose për të shkruar një të tille. Kur përfundon thirrja sistem përfundon, CPU vazhdon përseri të përpunoje deri sa të ketë nevoje për të lexuar të dhena ose për të shkruar të dhena. Duhet kujtuar se edhe disa aktivitetë I/O quhen si përpunim. Për shembull, kur CPU kopjon bite ne një video RAM për të rinovuar pamjen ne ekran, kjo është përpunim dhe jo I/O, sepse CPU është ne përdorim. I/O ne ketë sens është kur një process futet ne gjendjen e bllokuar duke pritur një paisje të jashtme për të përfunduar punen e saj.

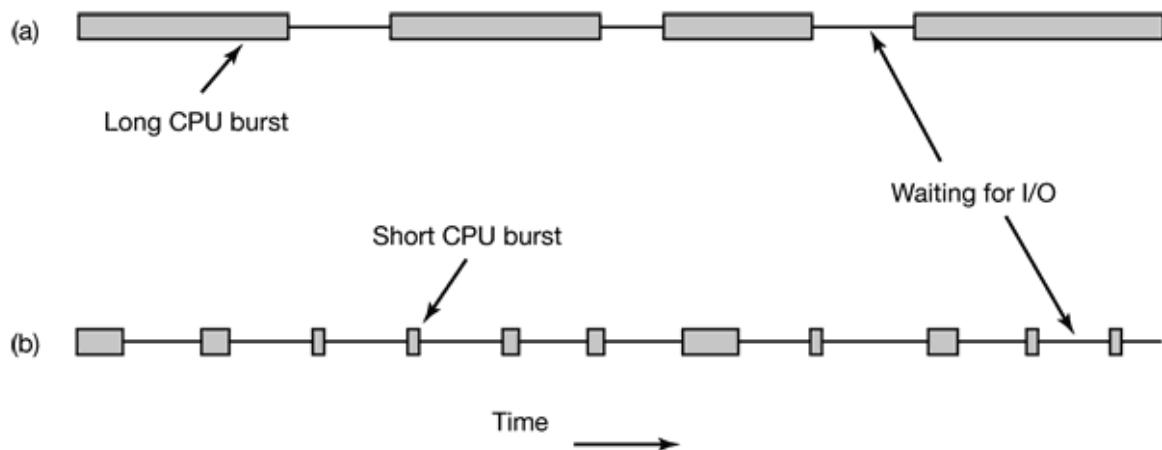


Figure 2-37. Impulset e përdorimit të CPU-të alternuara me momente pritje për I/O. (a) Kufijtë e një procesi ne CPU. (b) Kufijtë e një procesi I/O.

Fakti kryesor qe paraqitet ne Fig.2-37 është se disa procese, si ai i paraqitur ne Fig.2-37(a), shpenzojne shumicen e kohes duke përpunuar, ndersa disa të tjere si ne Fig.2-37(b) shpenzojne shumicen e kohes duke pritur I/O. Të paret quhen **compute-bound**; të dytët quhen **I/O-bound**. Proseset Compute-bound kane cikle ne gjata të CPU-se ndersa shume pak pritje të I/O, ndersa ato I/O bound kane pak cikle të CPU por shume pritje të I/O. Calesi është gjithmone ciklet e CPU-se dhe jo pritja e I/O.

Tani nuk kushton shume të kemi një CPU të shpejtë keshtu qe proceset tentojnë të jene I/O bound. Dhe me kalimin e kohes do të punohet shume me schedulimin e procese I/O bound sepse shpejtësia e CPU është shume e shpejtë se ajo e disqeve.

Kur të kryhet schedulimi

Një problem kryesor për sa i përket schedulimeve është kur të behet schedulimi. Del ne dukje se ka një numer të madh situatash ne të cilat duhet schedulimi. Fillimisht kur një process i ri krijohet, vendimi qe duhet të merret është ne se duhet të ekzekutohet procesi prind apo femije. Duke qene se të dy proceset jane ne gjendjen gati, duhet të behet një schedulim normal i cili mund të shkoj ne dy drejtime, domethene qe scheduleri mund të zgjedh midis procesit prind dhe femije.

Një vendim schedulimi duhet të behet kur një process ekziston. Ky process nuk mund të ekzekutohet (deri sa nuk ekziston me), pra disa procese të tjera nga lista e proceseve ne gjendjen gati. Ne qoftë se asnjë proces nuk është gati, do të ekzekutohet normalisht një proces i ardhur nga sistemi.

E treta, kur një proces bllokohet ne I/O, ne një semafor, ose për arsyet e tjera, duhet të zgjidhet një proces tjetër për tu ekzekutuar. Disa here arsyeta e bllokimit mund të luaj rol ne zgjidhjen e procesit të ri. Për shembull, ne qoftë se A është një proces i rendesishem dhe pret qe B të largohet nga zona kritike, ne qoftë se ne lejojme qe B të ekzekutohet me vone, do ti hapim rruge procesit A të vazhdoje fillimisht. Po problemi qendron ne faktin se scheduleri nuk e ka informacionin e saktë.

E katërta, kur ndodh një interrupt I/O, duhet të behet një vendim schedulimi. Ne qoftë se interrupti vjen nga një paisje I/O qe ka përfunduar punen, disa procese qe mund të ishin ne pritje të I/O tani mund të vazhdojnë ekzekutimin. Varet nga scheduleri ne qoftë se proceset ne gjendjen gati të ekzekutohen, ne se procesi qe po ekzekutohet ne momentin e interruptit duhet të vazhdoje ekzekutimin ose duhet të ekzekutohet një proces i ri.

Ne qoftë se një ore hardware prodhon interrupte periodikë me frekuencë 50 Hz, 60 Hz ose frekuencia të tjera, një vendim interrupti mund të behet çdo interrupt clocku ose çdo k-interrupt. Algoritmet e schedulimit mund të klasifikohen ne dy grupe sipas menyres si

kombinoohen me interruptet e clock-ut. Një algoritem schedulimi **nonpreemptive** zgjedh një proces për të ekzekutuar dhe e le të ekzekutohet deri ne bllokim (si ne I/O ashtu edhe për të pritur një proces tjetër) ose deri sa të lihet vullnetarisht CPU-ja. Edhe ne qoftë se ekzekutimi zgjat për ore të tëra, ai nuk do të ndalohet me force. Ne të vertetë asnje vendim schedulimi nuk behet gjatë interrupteve të clock-ut. Pas kryerjes se interruptit të clockut, procesi qe po ekzekutohej rikthehet ne ekzekutim.

Ndryshe nga rasti i pare, algoritmi i schedulimit **preemptive** zgjedh një proces dhe e lejon të ekzekutohet për një kohe të caktuar. Ne qoftë se ai është ende ne ekzekutim kur mbaron koha e caktuar, ai pushon se ekzekutuari dhe scheduleri zgjedh një proces tjetër për ekzekutim (ne qoftë se ka ndonjë të gatshem). Kryerja e schedulimit preemptive kerkon një clock interrupti ne fund të kohes se caktuar për ti dhene kontrollin e CPU-se përseni schedulerit.

Kategorit e algoritmeve të schedulimit.

Nuk është cudi qe ne sistuata të ndryshme nevojitet algoritme të ndryshme schedulimi. Kjo situatë lind për shkak se ekzistojne aplikime të ndryshme (dhe sisteme operative të ndryshme) kane qellime të ndryshme. Me fjale të tjera, cfare duhet të optimizoje scheduleri ndryshon nga sistemi. Ja vlen të dallojme tre mëjdice të ndryshme

1. Batch.
2. Interactive.
3. Real time.

Ne sistemet batch, nuk ka përdorues të padurueshem qe presin ne terminalin e tyre përgjigje. Rrjedhimisht algoritmet nonpreemptive ose preemptive, me periodë të gjatë për çdo proces Jane shpesh here të pranueshem. Kjo teknike redukton nderrimin e proceseve dhe përmireson performancen.

Ne një mëdis me përdorues interaktiv, algoritmet preemptive luajne rol kryesor qe një proces të përdori CPU-ne dhe ti mohoje sherbimin proceseve të tjera. Edhe se asnje proces nuk mund të ekzekutohet ne pafundesi, për shkak të një program bug, një proces mund ti bllokoje gjithe proceset pafundesisht. Preemption është i nevojshem për të shmangur ketë sjellje.

Ne sisteme me kushte real-time, preemption nuk është i mjaftueshem, disa here nuk nevojitet sepse proceset e dine qe mund të mos ekzekutohen për një kohe të gjatë dhe shpesh here e bejne punen e tyre dhe bllokohen. Ndryshimi midis sistemave interaktiv dhe sisteme real-time është se sistemet real-time ekzekutojnë programe qe mendohet të shkojne ne aplikacione të ardheshme. Ndersa sistemet interactive ekzekutojnë programe arbitrale qe nuk kooperojne.

Qellimet e algoritmeve të schedulimit

Ne menyre qe të ndertohet një algoritem schedulimi duhet të jetë e qartë idea se cfare bën një algoritem i mire. Disa qellime varen nga mjedisi (batch, interactive, real-time), por ka edhe qellime të njëjtë ne të gjitha sistemet. Disa qellime janë ne Fig. 2-38. Do ti diskutojme me poshtë.

Të gjithe sistemet

Drejtësia - Ndarja e CPU nepër procëse ne menyre të drejtë.

Policy enforcement - seeing that stated policy is carried out

Balance – Mbajtja e të gjithe pjesave të sistemit të zena.

Batch systems

Throughput – maksimizimi i puneve për ore.

Koha Turnaround – minimizimi i kohes midis submission dhe përfundimit.

Përdorimi CPU – Mbajtja e CPU të zene gjatë gjithe kohes.

Interactive systems

Koha e përgjigjes – përgjigja ndaj kerkesave me shpejtësi.

Proportionality – Plotësimi i kerkesave të përdoruesit.

Real-time systems

Realizimi brenda kohes se cakuar – Parandalimi i humbjes se të dhenave

Parashikueshmeria – Parandalimi i degradimit të cilesise ne sistemet multimediale.

Figure 2-38. Disa qellime të algoritmit të schedulimit ne rrethana të ndryshme.

Ne çdo rrethane, drejtësia është e rendesishme. Procëse konkuruese duhet të marrin sherbime konkuruese. Ti japesh një procesi me shume kohe CPU-je, se një procesi ekuivalent nuk është e drejtë. Sigrurisht, kategori të ndryshme procesesh duhet të trajtohen ndryshe. Mjafon të mendojme për qendren e kompjuterave ne një reaktor nuklear.

Të lidhur me drejtësinë janë edhe rregullat e sistemit. Ne qoftë se rregulli është qe procesi për kontrrollin e sigurise ekzekutohet kur të doje ai, edhe ne qoftë se payroll është 30 sekonda me vone se, scheduleri duhet të sigurohet qe ky rregull të zbatohet.

Qellim tjetër, të mbahen sa me shume të jetë e mundur të zena pjeset e sistemit. Ne qoftë se CPU-ja dhe paisjet I/O mund të mbahen ne pune gjithe kohes, kryen me shume pune

ne sekonde dhe ne qoftë se disa komponent Jane të gatshem, ne një system batch, për shembull, scheduleri ka kontroll mbi punet të cilat sillen ne memorje për tu ekzekutuar. Të kesh se bashku ne memorje disa procese CPU-bound dhe disa procese I/O-bound është ide me e mire, sesa të ngarkosh dhe të ekzekutosh gjithë proceset CPU-bound dhe me pas, pasi keto të përfundojnë, të ngarkohen dhe të ekzekutohen proceset I/O-bound. Ne qoftë se përdoret strategjia e dytë, kur proceset CPU-bound po ekzekutohen, ato do të jene ngjitur CPU-se dhe disku do të jetë bosh. Me pas kur vijne proceset I/O –bound, ato do të tentojnë drejt diskut dhe CPU-ja do të jetë bosh. Me mire është të mbajme të gjithe sistemin ne pune duke përdorur me kujdes një përzierje procesesh.

Manaxheret e qendrave të medha kompjuterike qe ekzekutojne shume procese batch shohim gjithmone tre karakteristika qe të vleresojne sa mire po punon sistemi i tyre: throughput, koha turnaround dhe përodrimi CPU. **Throughput**, është numri i puneve për ore qe sistemi kryen. Duke marre parasysh të gjitha, përfundimi i 50 puneve ne ore është me mire se 40 pune ne ore. **Koha Turnaround**, koha mesatare nga momenti qe fillon një pune batch deri ne momentin qe përfundon. Ajo mat mesataren sa duhet të pres përdoruesi për një përgjigje. Ketu rregulli është: sa me e vogel aq me mire.

Një algoritem qe maksimizon throughputin nuk siguron minimizimin e kohes turnaround. Për shembull, futja e një përzirjeje punesh të shkurtra dhe të gjata, një scheduleri qe gjithmone ekzekuton punet e shkurtra do të japi një thorughput maximal (domethene shume pune të shkurtra të ekzekutuar brenda një ore) por nga ana tjetër kemi një kohe shume të gjatë turnaround për punet e gjata. Ne qoftë se punet e shkurtra vijne vazhdimi, punet e gjata mund të mos ekzekutohen kurre duke e cuar kohen turnaround ne infinit dhe një throughput maksimal.

Përdorimi i CPU është edhe problem me sistemet batch sepse ne mainframe e medha-ja ku ekzekutohen sistemet batch, CPU është ende një shpenzim i madh. Manaxheret e ketyre qendrave kompjuterike gjithmone tentojnë të përdorin sa me shume CPU-ne. Megjithatë përdorimi i CPU-se nuk është një matës i mire. Cfare ka rendesi me shume, është punet qe kryhen nga sistemi ne një ore (throughput) dhe sa i duhet sistemit për të kthyer punen (koha turnaround). Të marresh përdorimin e CPU si tregues është si të vleresosh një makine me sa xhiro bën motorri ne një ore.

Për sistemet interactive, kryesisht sistemet time sharing dhe server, aplikohen qellime të tjera. Kryesori është minizimi i **kohes se përgjigjes**, qe është koha qe duhet nga dergimi i komandes deri ne marrjen e përgjigjes. Ne kompjuterat personal kur një proces background po ekzekutohet (për shembull, leximi ose ruajtja e një emaili nga network), një kerkese e përdoruesit për të nisur ose hapur një program duhet të ketë prioritet mbi punen qe është ne background. Ekzekutimi i gjithe kerkesave interactive ne fillim quhet një sherbim i mire.

Një tregues tjetër është **proportionality**. Përdoresit kane një ide (shpesh here të gabuar) për kohen sa duhet të zgjasë një veprim i caktuar. Kur një kerkese qe përdoruesit e

mendojne si komplekse zgjat shume, përdoruesit e pranojne ketë, por kur një kerkese e cila mendohet se thjesht kerkon shume kohe, atëhere përdoruesi zemerohet. Për shembull, ne qoftë se klikimi i një ikone e cila bën të mundur lidhjen me një internet provider me një modem analog do të duhet 45 sekonda të krijohet lidhja, përdoruesi është i detyruar ta pranoj ketë fakt. Por ne qoftë se do të duhej 45 sekonda për të shkeputur lidhjen atëhere kjo nuk do pranohej nga përdoruesi. Kjo sjellje vjen nga përceptimi i çdo përdoruesi se, krijimi i një lidhjeje do gjithmone me shume kohe se sa thjesht shkeputja e lidhjes. Ne disa raste si ky i apo scheduleri nuk ka site ndikoje ne kohen e përgjigjes, por ne raste të tjera, kryesisht kur vonesa është si pasoje e një renditjeje të keqe të proceseve.

Sistemet real-time kane karakteristika të ndryshme nga sistemet interactive keshtu qe ka edhe qellime të ndryshme schedulimi. Keto sisteme karakterizohen nga disa kohe zbatimi të caktuara qe duhet të respektohen. Për shembull, ne qoftë se kompjuteri po kontrollon një paisje qe prodhon të dhena me një fluks kostant, deshtimi ne ekzekutimin e procesit data-collection mund të coj ne humbje të të dheneve. Pra kerkesa kryesore ne sistemet real-time është respektimi i kohes se zbatimit.

Ne disa sisteme real-time, sidomos ato qe përfshijne multimedia, parashikuesheria është e rendesishme. Mos respektimi ne disa raste i kohes se zbatimi nuk është gjithmone fatal, por ne qoftë se procesi audio fillon të ekzekutohet me rrregullime, cilesia e zerit do të ulet. Edhe video është problem por veshi ka me shume ndjeshmeri ndaj gabimeve sesa videoja. Për të parandaluar problemin, procesi i schedulimit duhet të jetë shume i parashikueshem dhe i rregullt. Do të studiojme algoritmet batch dhe interactive ne ketë kapitull, por do të përqendrohem me shume ne schedulin real-time deri sa të shkojmë ne multimedia operating system.

2.5.2 Schedulimi ne sistemet Batch

Tani është moment për të kaluar nga schedulimi ne përgjithesi ne algoritmat specific të schedulimit. Ne ketë paragraph do të studiojme algoritmet e përdorura ne sistemet batch. Me pas do të studiojme sistemet interactive dhe real-time. Ja vlen të theksojme se disa algoritma janë të njëjtë si ne sistemet batch ashtu edhe ne ato interactive. Do ti studiojme keto me vone. Ketu do të studiojme algoritmat qe janë të përshtatshem vetëm për sistemet batch.

I pari qe vjen, i pari sherbehet (First-Come First-Served)

Ndoshta algoritmi me i thjeshtë i schedulimit është nonpreemptive **first-come first-served**. Me ketë algoritem, proceset aksesojne CPU sipas rrades qe kane kryer kerkesen. Pra normalisht ka vetëm një radhe të proceve ne gjendje gati. Kur vjen procesi i pare ai fillon menjehere dhe lejohet të ekzekutohet deri sa të doje. Çdo pune e re qe vjen vendoset ne fund të radhes. Kur procesi qe po ekzekutohet bllokohet atëhere procesi i pare qe është ne radhe do të ekzekutohet. Kur një process i bllokuar ndryshon gjendje dhe shkon ne gjendjen ready, si të ketë ardhur një pune e re, ky proces vendoset ne fund të radhes.

E mira e ketij algoritmi është se është i thjeshtë për tu kuptuar dhe i thjesht për tu programuar. Është gjithashtu i drejtë ne kuptimin sic është e drejtë qe biletat e para të një koncerti ose të një ndeshjeje ti marrin ne fillim ato qe jane ne radhe prej ores 2 të mengjesit. Me ketë algoritmet mjafton vetëm një listë e linkuar për të ruajtur proçeset qe jane ready. Zgjedhja e një proçesi mjafton vetëm të marrin proçesin e pare ne listën ready. Shtimi i proçesit ready ose zhbllokimi i një proçesi mjafton vetëm ta shtojme ne fund të radhes. Shume e thjeshtë!

Fatkeqsisht, first-come first-served ka gjithashtu një disavantazh të madh. Supozojme se ka një proçess compute-bound qe ekzekutohet për 1 sekonde dhe shume proçese I/O-bound qe përdorim pak kohe CPU-je, por qe secili prej tyre duhet të kryej 1000 lexime disku qe të përfundoje. Proçesi compute-bound ekekzekutohet për 1 sekonde, me pas lexon një block ne disk. Tani të gjithe proçeset I/O ekzekutohet dhe fillojne leximin e diskut. Kur proçesi copmutë-bound merr blockun nga disku, ai ekzekutohet edhe për 1 sekonde tjetër, duke u ndjekur nga gjithe proçeset I/O-bound ne vazhdim.

Rezultati final është se çdo proçess I/O-bound i duhen 1 sekonde për të lexuar një block të diskut, pra do i duhen 1000 sekonda për të mbaruar. Me një algoritem schedulimi qe zevendeson proçesin compute-bound çdo 10 sekonda do të bënte qe proçesi I/O-bound të përfundonte 10 sekonda ne vend të 1000 sekonda dhe duke mos ngadalesuar shume proçesin compute-bound.

Puna me e shkurtër e para (Shortest Job First)

Tani le të shohim një algoritem tjetër batch nonpreemptive qe pretendon se koha e ekzekutimit dihet paraprakisht. Ne një kompani sigurimi, për shembull, njerezit mund të parashikojne shume saktë sa kohe do të duhet për ekzekutuar një batch me 1000 thirrje, duke qene se pune të njëjta kryhen çdo ditë. Kur pune të të njëjtës rendesi presin ne radhe për ekzekutim, scheduleri zgjedh **punen me të shkurtër**. Shikojme ne Fig. 2-39. Shohim katër pune A,B,C dhe D me kohe ekzekutimi përkatëse 8, 4, 4 dhe 4 minuta. Ne qoftë se i ekzekutojme keto pune sipas kesaj rradhe, koha turnaround për A do të jetë 8 minuta, për B do jetë 12 minuta, për C do jetë 16 minuta dhe për D do të jetë 20 minuta me një mesatare prej 14 minutash.



Figure 2-39. Një shembull i schedulimit. Puna me e shkurtër e para. (a) Ekzekutimi i katër puneve sipas rendit original. (b) Ekzekutimi sipas rendit. Puna me e shkurtër e para.

Le të konsiderojme ekzekutimin e ketyre katër puneve duke përdorur, puna me e shkurtër e para, sic tregohet ne Fig. 2-39(b). Tani koha turnaround do të jene 4,8,12, 20 me një mesatare prej 11 minutash. Le të konsiderojme rastin e katër puneve cfardo me kohe

ekzekutimi a,b,c dhe d. Puna e pare mbaron ne kohen a, e dyta ne kohen a+b dhe keshtu me rradhe. Koha turnaround është $(4a+3b+2c+d)/4$. Është e qartë se a kontribon me shume ne mesatare se kohet e tjera, keshtu qe duhet të jetë puna me e shkurtër, me b ne vazhdim, me pas c dhe me pas d, si me e gjata e cila ndikon vetëm kohen e vet turnaround. E njëjtë llogjike aplikohet edhe ne raste kur kemi numer tjetër punesh.

Ja vlen të theksojme se algoritmi shortest job first është i vlefshem vetëm kur të gjitha punet jane të gatshme njëkohesisht. Si një kundershembull, le të konsiderojme 5 pune, nga A deri ne E, me kohe ekzekutimi përkatësish 2, 4, 1, 1 dhe 1. Koha e mberritjes se tyre është 0, 0, 3, 3 dhe 3. Fillimisht vetëm A dhe B mund të zgjidhen përderisa punet e tjera nuk kane ardhur ende. Duke zgjedhur punen e shkurtër të paren radha e ekzekutimit do të jetë A, B, C, D, E me një mesatare pritjeje 4.6. Megjithatë sipas radhes B, C, D, E, A mesatarja e pritjes është 4.4.

Koha e mbetur me shkurtër (Shortest Remaining Time Next)

Një version preemptive i shortest job first është **shortest remaining time next**. Me ketë algoritem, scheduleri zgjedh gjithmone procesin koha e mbetur e ekzekutimit të të cilit është me e shkurtër. Përseri edhe ne ketë rast koha e ekzekutimit duhet të jetë e ditur. Kur një punë e re vjen, koha e tij totale e ekzekutimit krahasohet me kohen e mbetur të ekzekutimit të procesit qe po ekzekutohet ne atë moment. Ne qoftë se puna qe erdhi do me pak kohe për ekzekutim se procesi qe po ekzekutohet, procesi qe po ekzekutohet do të pezullohet dhe puna e re qe erdhi do të ekzekutohet. Kjo skeme i ofron sherbim me të mire proceseve të shkurtra.

Schedulimi me tre nivele (Three-Level Scheduling)

Nga një pikpamje e caktuar, sistemet batch lejojne schedulimin me tre nivele të ndryshme sic është treguar ne Fig. 2-40. Pasi punet vijne ne system, ato fillimisht vendosen ne një radhe input qe ruhet ne disk. Admission scheduler vendos cilën pune të pranoj sistemi. Të tjeret mbahen ne radhen e inputit deri sa të zgjidhen. Një algoritem tipik për kontrollin e pranimit është qe të kerkohet një mix i puneve compute-bound dhe puneve I/O-bound. Alternativa tjetër, punet e shkurtra të zgjidhen menjehere, ndersa punet e gjata duhet të presin. Admission echeduler ka të drejtë të pranoj punë qe kane ardhur me vone dhe punë të tjera të presin.

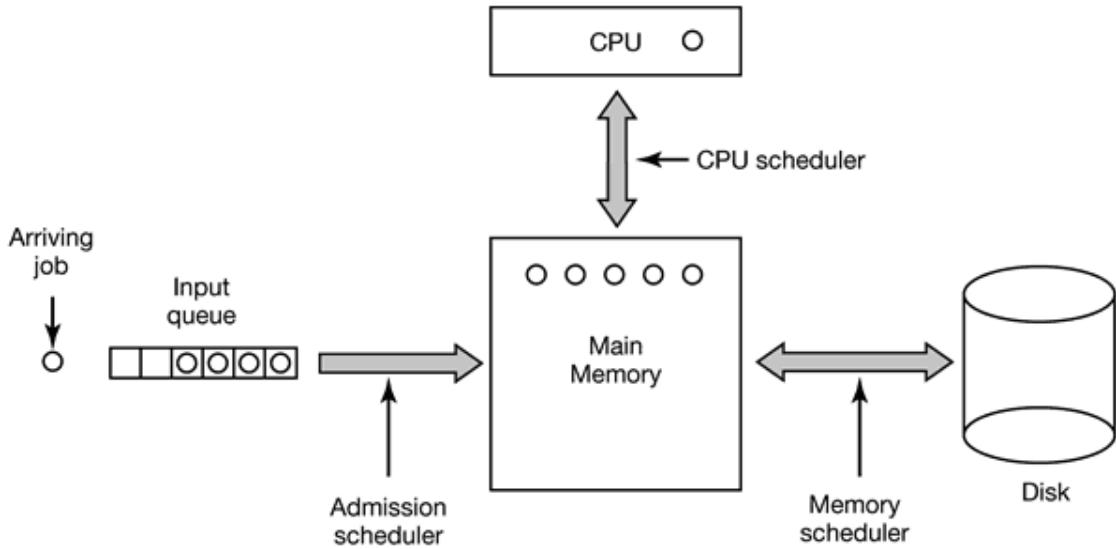


Figure 2-40. Schedulimi me 3 nivele

Ne momentin qe një pune pranohet ne sistem, një proces mund të krijohet për të dhe mund të pretendoje për CPU-ne. Megjithatë, mund të ndodhi qe numri i proceseve mund të jetë shume i madh sa nuk mund të ketë me vend bosh për to ne memorje. Ne ketë rast, disa nga proceset duhet të dergohen ne disk. Niveli tjetër i schedulimit është zgjedhja se cili process duhet të mbahet ne memorje dhe kush të mbahet ne disk. Do ta quajme ketë scheduler memory scheduler, sepse ai zgjedh cili process duhet të mbahet ne memorje dhe kush ne disk.

Ky vendim duhet të rishikohet shpesh për të bere të mundur qe proceset ne disk të marrin sherbimin qe kerkojne. Megjithese sjellja e një procesi nga disku nuk është i favorshem, ky rishikim nuk duhet të ndodhi me shume një here ne second ndoshta edhe me pak. Ne qoftë se përbajtja e memorjes kryesore rinovalohet shpesh, një pjese e madhe e bandwidthit të diskut do të shkoj dem, duke ngadalsuar I/O.

Për të përmiresuar performancen ne përgjithesi, scheduleri i memorjes duhet të vendlloj me kujdes sa procese do ne memorje, kjo quhet **grada e multiprogramimit** dhe llojin e proceseve. Ne qoftë se ai ka informacion se cili proces është computë-bound dhe cili është I/O-bound, ai mundohet të mbaj një përzierje të ketyre proceseve ne memorje. Si një parashikim i përafert, ne qoftë se një klase e caktuar procesesh ekzekutohen për 20% të kohes, mbajtja e 5 prej ketyre proceseve do të mbaj të zene gjatë gjithe kohes CPU-ne. Ne kapitullin e 4, do të shohim një model me të mire multiprogramimi.

Për të bere zgjedhjen e tij, scheduleri i memorjes ne menyre periodikë rishikon çdo proces ne disk për të vendlloj sur kush duhet të cohët ne memorje. Kriteret qe ndjek scheduleri për të bere zgjedhjen e tij janë keto:

1. Sa kohe ka kalur nga momenti qe procesi ka swapped in ose swapped out?
2. Sa kohe CPU-je i është dhene procesit kohet e fundit?

3. Sa i madh është procesi? (proçeset e vegjel nuk e marrin ketë rruge)
4. Sa i rendesishem është procesi?

Niveli i tretë i schedulimit është zgjedhja e njërit nga proçeset ne gjendjen ready qe ndodhen ne memorjen kryesore për tu ekzekutuar. Shpesh quhet scheduleri i CPU-se dhe është ai qe ne quajme thjesht scheduler. Keti mund të përdoret çdo algoritmet i vlefshem, si preemptive ashtu edhe nonpreemptive. Keti bejne pjese algoritmet e shpjeguara me lartë dhe ato qe do shpjegohen me poshtë.

2.5.3 Schedulimi ne sistemet interactive

Tani do të shohim disa algoritme qe përdoren ne sistemet interactive. Të gjithe keto mund të përdoren shume mire edhe ne schedulerin e CPU-se të sistemeve batch. Përderisa schedulimi me tre nivele nuk është i mundur ketu, schedulimi me dy nivele është i mundur dhe shpesh here i përdorshem (scheduleri i memorjes dhe scheduleri i CPU-se). Me poshtë do të përqendrohem me shume ne schedulerin e CPU-se.

Schedulimi Round-Robin

Le të shikojme disa algoritma specific schedulimi. Një nga me të vjetrit, me të thjeshit, me të drejtët dhe me të përdorshmit është algoritmi **round-robin**. Çdo procesi i caktohet një interval kohe, i quajtur **quantum**, gjatë të cilët ai lejohet të ekzekutohet. Ne qoftë se procesi është ende duke u ekzekutuar ne fund të quantumit, CPU-ja i jepet një procesi tjeter. Ne qoftë se procesi bllokohet ose përfundon para se të mbaroje quantum-i, nderrimi i CPU-se behet sigurisht kur bllokohet procesi. Round robin është i thjeshtë për tu implementuar. Gjithçka duhet të beje scheduleri është të mbaj një list të proceseve të ekzekutueshem, sic tregohet ne Fig.2-41 (a). Kur procesi e shfrytëzon quantumin e tij, ai vendoset ne fund të listës, sic tregohet ne Fig. 2-41(b)

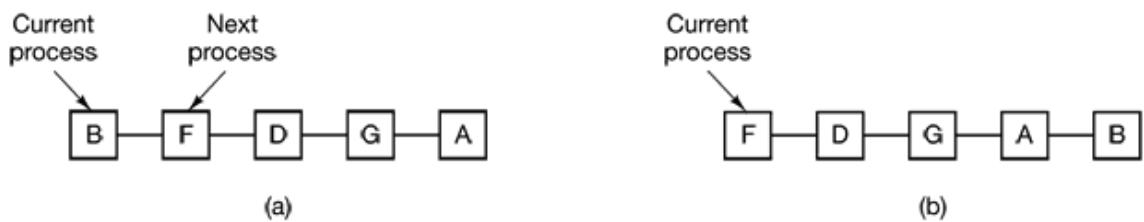


Figure 2-41. Schedulimi Round-robin. (a) Lista e proceseve të ekzekutueshem. (b) Lista e proceseve të ekzekutueshem pasi B e ka përfunduar quantumin e tij.

Problemi i vetëm me round-robin është gjatësia e quantumit. Kalimi nga një process ne tjetrin, kerkon një fare kohe gjatë të cilës behet administrimi, ruajtja e regjistrave ngarkues (loading) dhe hartës se memorjes, rinnovimi i tabelave të ndryshme dhe listave, pastrimi dhe ringarkimi i memorjes cache, etj. Supozojme se ky nderrim procesesh ose sic quhet shpesh context switch, zgjat 1 msec, duke përfshire edhe nderrimin e hartës se memorjes, pastrimin dhe ringarkimin e caches, etj. Gjithashtu supozojme se quantumi është caktuar 4msec. Me keto parametra pasi jane kryer 4 msec pune e vlefshme, CPU-

ja do shpenzoje 1 msec ne nderrimin e proceseve. Njëzet përqind e kohes se CPU do të shkoje dem me pune administrative. Duket qartë se kjo përqindje është e lartë.

Për të rritur eficencen e CPU-se, mund të cojme quantum le të themi 100 msec. Tani koha e shpenzuar është vetëm 1%. Por le të konsiderojme çdo të ndodh ne një sistem time sharing ne qoftë se dhjetë përdorues shtypin të njëjtin buton njëkohesisht. Dhjetë procese do të vendosen ne listën e proceseve të ekzekutueshem. Ne qoftë se CPU-ja është gati, procesi i pare do të ekzekutohet menjehere. I dyti mund të mos filloje pas 100 msec dhe keshtu me rradhe. I fundit do të jetë me fatkeqi dhe ndoshta mund të presi me shume se 1 sec deri sa të marri sherbimin e kerkuar, duke supozuar se gjithe të tjera e kane shfrytëzuar plotësisht quantumin e tyre. Shumica e përdoruesve do të marrin një përgjigje prej 1 sec të një komande si ngadalesim.

Një faktor tjetër është caktimi i quantumit me i gjatë sesa perioda kryesore e CPU-se, zevendesimi ndodh shume rralle. Ne të njëjtën kohe, shumica e proceseve do të kryejne një veprim bllokues para se quatumit të mbaroje, duke krijuar një nderrim procesi. Eleminimi i zevendesimit rrit performancen dhe nderrimi i proceseve do të ndodhi vetëm kur është llogjikisht e nevojshme, domethene kur një process bllokohet dhe nuk mund të vazhdoje.

Përfundimi mund të formulohet si me poshtë: Caktimi i quantumit të shkurtër shkakton një numer të madhe nderrime procesesh dhe ul eficencen e CPU-se, por caktimi i një quantumi të gjatë mund të uli përgjigjet ndaj kerkesave të gjata. Një quantum rrreth 20-50 msec është zgjedhja e përshtatshme.

Schedulimi me prioritet

Schedulimi Round-robin bën supozimin qe të gjithe proceset jane të njëjtë nga rendesia qe kane. Normalisht, njerez qe zotërojne ose punojne ne kompjutera multiuser kane ide të ndryshme mbi ketë teme.

Ne një univesitet radhitja mund të ishte dekani, profesoret, sekretaret, rojet dhe me ne fund studentët. Nevoja e faktoreve të jashtëm ne llogari na con tek **schedulimi me prioritet**. Idea baze është e tille: çdo procesi i caktohet një prioritet dhe procesi i ekzekutueshem me prioritetin me të lartë lejohet të ekzekutohet.

Edhe ne një PC me vetëm një përdorues, mund të ketë procese të shume fishta, disa me të rendesishme se të tjerat. Për shembull, një proces deamon qe dergon email dhe qendron ne background duhet ti caktohet një prioritet me të ulet, sesa një process qe shfaqet ne ekran një film ne real-time.

Për të menjanuar rastin kur proceset me prioritet të lartë të ekzekutohen deri ne një kohe të pacaktuar, scheduleri mund të uli prioritetin e procesit ne ekzekutim gjatë çdo tiku të ores (për shembull ne çdo interrupt të ores). Ne qoftë se ky veprim shkakton uljen e prioritetit nen nivelin e prioritetit të procesit të ardhshem, atëherë do të ndodhi një nderrim procesi. Alternativa tjetër, çdo procesi i caktohet një kohe maksimale ekzekutimi

quantum. Kur ky quantum përfundon, i jepet shansi proçesit të ardhshem me prioritet me të madh që të ekzekutohet.

Prioriteti mund të caktohet ne menyre statike ose dinamike. Ne një kompjuter ushtarak, proçeset fillojne si gjenerale me prioritet 100, si kolonele me 90, si majore me 80, kapitenet me 70, leitnantët me 60 dhe keshtu me radhe. Ndryshe, ne një qender kompjuterash komercial, punet me prioritet të lartë mund të kushtojne 100 \$ ne ore, prioritet të mesem 75 \$ ne ore dhe prioritet të ulet 50 \$ ne ore. Sistemi Unix ka një komande, *nice* qe bën të mundur uljen vullnetare të prioritetit të proçesit, ne menyre qe të hapi rruge proçeseve të tjere. Përdoret shume rralle.

Prioritetet mund të caktohen edhe ne menyre dinamike nga sistemet për të arritur një qellime të caktuar. Për shembull, disa proçese janë shume I/O bound dhe shpenzojne shume kohe për të pritur I/O që të përfundoje. Ne çdo moment kur ky proçes kerkon CPU-ne, ajo duhet ti jepet menjehere që të lejoje të niset kerkesa e ardhshme I/O, e cila mund të vazhdojne ne paralel punen me një proçes tjetër. Të lejosh të presi për një kohe të gjatë një process I/O bound, do të thotë ta kesh atë nepër kembe ne memorjen kryesore duke e zene atë. Një algoritem qe ofron sherbim të mire proçeseve I/O bound bazohet ne prioritet, ne $1/f$, ku f është fraksioni i quantumit të fundit qe proçesi përdor. Një proçes qe përdor 1msec nga 50 msec quantum do të ketë prioritet 50, ndersa një proçes qe ekzekutohet 25 msec para se të bllokohet do të ketë prioritet 2 dhe proçesi qe do përdori gjithe quantumin do të ketë prioritetin 1.

Shpesh here është e vlefshme të grupohen proçeset ne klasa prioriteti dhe me pas të përdoret schedulimi me prioritet midis klasave por round-robin brenda çdo klase. Figura 2-42 tregon një sistem me katër klasa prioriteti. Algoritmi i shcedulimit është si me poshtë: për aq kohe sa ka proçese të ekzekutueshme ne klasen 4, duhet të ekzekutohet secili prej tyre për një quantum, sipas metodes round-robin dhe kurre nuk duhet të shqetësohen nga proçese me klase prioriteti me të vogel, ne qoftë se klasa me prioritet 4 boshatiset, atëherë ekzekuto proçeset e klasses me prioritet 3 me round-robin. Ne qoftë se klasa me prioritet 4 dhe ajo me prioritë 3 janë të dyja bosh, atëherë ekzekutohen proçeset e klasses me prioritet 2 me round-robin dhe keshtu me rradhe. Ne qoftë se prioritetet nuk rregullohen here pas here, atëherë me porceset e klasave me prioritet të ulet do të ndodhë starvacioni.

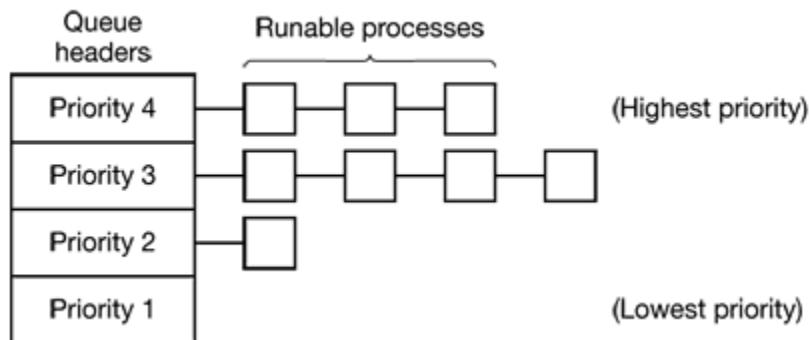


Figure 2-42. Një algoritem schedulimi me katër klasa prioriteti.

Rradhet e shumefishta

Një nga schedulerat e pare me prioritet ishte ne CTSS (Corbató et al., 1962). CTSS kishte një problem sepse nderrimi i proceseve ishte shume i ngadaltë për shkak se 7094 mund të mbaj vetëm një proces ne memorje. Çdo nderrim nenkupton dergimin e atij qe po ekzekutohej ne disk dhe marrjen e një të riu nga disku. Krijuesit e CTSS shpejt e kuptuan qe ishte me eficiente ti jepej një here ne një kohe të cakuar një quantum i madh proceseve CPU-bound sesa ti jepej shpesh here një quantum i vogel (për të reduktuar swaping). Nga ana tjetër ti jepej gjithe proceseve një quantum i madh do të conte ne një kohe përgjigjeje të gjatë. Zgjidhja e ketij problemi ishte krijimi i klasave me prioritet. Proseset ne klasen me të lartë do të ekzekutoheshin për një quantum të plotë. Proseset ne klasen me të lartë pasardhese do të ekzekutoheshin për 2 quanta. Proseset ne klasen pasardhese do të ekzekutoheshin për katër quanta. Sa here qe një proces çç gjithe quantat qe i ishin caktuar, ai ulet një klase me poshtë.

Si shembull, konsiderojme një process qe i nevojiten për të përfunduar 100 quanta. Fillimisht do ti jepet një quantum dhe me pas do largohet. Me pas do ti jepen 2 quanta para se të largohet. Ne vazhdim do ti jepet 4, 8, 16, 32, 64 quanta edhe se ai do të ketë përdorur vetëm 37 nga 64 quantat qe do i jepen ne fund. Do të nevojiten vetëm 7 largime (duke marre parasysh edhe ngarkimin fillestar) ne vend të 100 të tillave me një round-robin të thjeshtë. Ne vazhdim, procesi do të futet thelle e me thelle ne radhet e prioritetit, ai do të ekzekutohet rralle e me rralle, duke e ruajtur CPU-ne për proceset interactive të shkurtra.

Rregulli i meposhtëm u adoptua qe të parandalontë një proces qe dontë një kohe të gjatë për tu ekzekutuar kur filloj, por qe me vone u be interactive nga të ndeshkuarit përgjithmone. Gjithmone kur një carriage return është future ne një terminal, procesi qe i përket atij terminali cohët ne klasen me prioritet me të madh, duke supozuar se është duke u bere interactive.

Një ditë, një përdorues me një proces të rende CPU-bound zbuloi se vetëm duke ndenjur tek terminali dhe duke shtypur me crregullsi çdo disa sekonda, bënte cudi për kohen e përgjigjes. Ai i tregoi të gjithe shokeve. Morali është: Të besh gjera të sakta ne praktike është shume me veshtire se të jene të sakta ne téori.

Shume tipe të tjere algoritmash jane përdorur për ti caktuar proceseve klasen e prioritetit. Për shembull, sistemi XDS 940 (Lampson, 1968), i ndertuar ne Berkeley, kishte katër klasa prioriteti, të quajtura terminal, I/O, quantum i shkurtër, quantum i gjatë. Kur një proces qe është duke pritur për një input ne terminal zgjohet, ai menjehere cohët ne klasen me prioritet me të lartë (terminal). Kur një proces është duke pritur një block të diksut të behet gati, ai shkon ne klasen e dytë. Kur një proces është ende duke u ekzekutuar dhe quantum i tij mbaron, procesi vendoset fillimisht ne klasen e tretë. Megjithatë, ne qoftë se një proces e mbaron quantumin e tij shume here rresht pa u bllokuar për terminal ose I/O të tjera, atëherë ai vendoset ne fund të rradhes. Shume

sisteme të tjera dicka të ngjashme për të favorizuar përdoruesit dhe proçeset interactive ne lidhje me ato qe jane ne background

Proçesi me i shkurtër do jetë pasardhesi (shortest process next)

Me qe shortest job first gjithmone prodhon kohen mesatare me të ulet për sistemet batch, do të ishte mire sikur të përdorej edhe për sistemet interactive. Deri ne një fare pike mundet. Proçeset interactive përgjithesisht presin për një komande, ekzekutojne komanden, presin një komande, ekzekutojne komanden dhe keshtu me rradhe. Në qoftë se ne shohim ekzekutimin e çdo komande si pune e vecantë, munde të minimizojme përgjigjen e përgjithshme duke ekzekutuar me të shkurtren të paren. Problemi qendron ne gjetjen se cili proçes është me i shkurtri.

Një menyre është të bazohemi ne sjelljet e meparshme dhe të ekzekutohet proçesi me kohen e mbetur me të shkurtër. Le të supozojme se koha e mbetur për komande për disa terminale është T_0 . Tani supozojme se ekzekutimi i ardhshem do të jetë T_1 . Ne mund të llogarisim mbetjen duke marre Shumen me peshe të dy numrave qe, $aT_0 + (1 - a)T_1$. Duke zgjedhur vleren e a-se ne mund të vendosim qe ekzekutimet e vjetra të harrohen shpejt ose jo. Me një $a = 1/2$ ne marrim;

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Pas tre ekzekutimesh të reja, pesha e T_0 do të ketë rene ne $1/8$.

Teknika e llogaritjes se vleres pasardhese ne një seri duke përdorur mesataren me peshe të vlerave të matura me pare quhet **aging**. Është shume i shpeshtë rasti kur një parashikim duhet të behet ne baze të vlerave të meparshme. Aging është i lehtë të implementohet kur $a = 1/2$

Schedulimi i garantuar (Guaranteed Scheduling)

Një menyre tjetër e schedulimit është të behen prentime reale përdoruesit për performance dhe me pas ta mbash atë. Një prentim qe është realisht dhe i lehtë për tu mbajtur është ky: Në qoftë se ka n përdorues të loguar ne momentin qe po punojme, do të marrim rrëth $1/n$ e fuqise se CPU-se. E njëjtë gjë me një përdorues me shume proçese të gjithe të barabartë, secili do të ketë $1/n$ e cikleve të CPU-se.

Qe të mbahet ky prentim, sistemi duhet të mbaj shenim sa CPU ka pasur çdo proçes qe nga krijimi i tij. Ai me pas llogarit pjesen e CPU qe i përket secilit, koha qe kur është krijuar, pjesetuar me n. Duke qene se shuma e kohes se CPU-se qe çdo proçes ka është e ditur, është e drejtë llogaritja e raportit të kohes aktuale të konsumuar të CPU-se me kohen e paracaktuar të CPU-se. Një raport 0.5 nenkupton se proçesi ka marre vetëm gjysmen e asaj qe i takonte të merrte dhe një raport 2.0, nenkupton se një proçess ka marre dyfishin e kohes qe ishte caktuar. Algoritmi këtu është të ekzekutohet proçesi me reportin me të vogel deri sa reporti i tij të behet me i vogel se i konkurruesit me të afert.

Schedulimi llotari (Lottery Scheduling)

Berja e premtiveve dhe me pas lenia ne dore e tyre është një ide e mire, por e vesh tire për tu implementuar. Megjithatë një algoritem tjetër mund të përdoret për të dhene rezultate të ngjashme dhe të parashikueshme dhe me një implementim me të thjeshtë. Ky quhet **schedulimi llotari** (Waldspurger and Weihl, 1994).

Idea kryosore është ti japim proceseve bileta llotarie për burime të ndryshme të sistemit, si koha e CPU-se. Gjithmone kur duhet të behet një vendim schedulimi, një biletë llotarie zgjidhet rastesisht, dhe procesi qe ka ketë biletë merre burimin qe kerkon. Kur kjo aplikohet ne schdeulimin e CPU-se, sistemi mund të japi deri ne 50 here ne sekonde, dhe çdo fitues të marri 20 msec të kohes se CPU-se si cmim.

George Orwell thotë: "Të gjithe proceset jane të barabartë, por disa jane edhe me të barabartë". Proseset me të rendesishem mund të marrin bileta speciale për të rritur shanset e tyre përfitore. Në qoftë se ka 100 bileta dhe një proces ka 20 prej tyre, ai do të ketë 20% të shanseve përfituar llotarine, dhe do të ketë rrëth 20% të kohes se CPU-se. Ndersa ne schedulimin me prioritet nuk është e qartë se çdo të thotë të kemi një prioritet 40. Ketu rregulli është i qartë: zotërimi i një fraksioni f të biletave do të japi afersisht fraksionin f të burimeve ne fjale.

Schdeulimi llotari ka tipare interesante. Për shembull, në qoftë se një proces i ri del ne skene dhe ka siguruar disa bileta, ne rastin e pare kur do të hidhet llotaria ai do jetë ne gjendje të fitoje ne raport me numrin e biletave qe ka.

Proseset qe bashkepunojne mund të shkembejne bileta në qoftë se deshirojne. Për shembull, kur një proces klient dergon një mesazhe tek një proces server dhe me pas bllokohet, ai mund t'ja japi gjithe biletat e tij procesit server, duke i rritur shanset serverit qe të ekzekutohet. Kur serveri mbaron ekzekutimin, ai ja kthen biletat klientit qe ai të ketë mundesi të ekzekutohet. Ne fakt ne munges të serverave klientët nuk kane nevoje përf biletat.

Schedulimi me llotari mund të përdoren përf të zgjidhur probleme qe jane të veshitura përf tu menaxhuar me metoda të tjera. Një shembull është një video server ne të cilin shume procese ushqejne me video stream klientët e tyre, por me frame rate të ndryshme. Supozojme se proceset kerkojne frame nga 10, 20 dhe 25 frame/sec. Duke i caktuar ketyre proceseve 10, 20 dhe 25 bileta përkatësisht ato automatikisht do ta ndajne CPU-ne afersisht ne raportin 10:20:25.

Schedulimi me ndarje të drejtë (Fair-Share Scheduling)

Deri tani kemi supozuar se proceset schedulohen ne vetvetë, pa patur rendesi kush është pronari. Si rezultat, në qoftë se përdoruesi 1 nis 9 procese dje përdoruesi 2 nis 1 proces,

me round-robin ose me prioritete të njëjtë, përdoruesi 1 do marri 90% të CPU-se dhe përdoruesi 2 do të marri vetëm 10% të CPU-se.

Për të parandaluar ketë situatë, disa sisteme shikojne edhe se kush e zotërone procesin para se ta schedulojne. Ne ketë model, çdo përdoruesi i caktohet një fraksion të CPU-se dhe scheduleri zgjedh proceset duke ju bindur kesaj ndarjeje. Për shembull në qoftë se dy përdorues kane secili 50 % të CPU-se, ata do ta kene ketë duke mos u marre parasysh sa procese ato kane ne ekzistënce.

Si një shembull, konsiderojme një sistem me dy përdorues, ku secili prej tyre kane 50 % të CPU-se. Përdoruesi 1 ka katër procese A, B, C dhe D dhe përdoruesi dy ka vetëm një proces. Në qoftë se round-robin përdoret, atëherë një sekunca schedulim do të jetë kjo:

A E B E C E D E A E B E C E D E ...

Në qoftë se përdoruesit 1 do ti jepej dyfishi i kohes se CPU-se ne lidhje me përdoruesin 2 atëherë sekunca mund të jetë:

A B E C D E A B E C D E ...

2.5.4 Schdeulimi ne sistemet real-time

Një sistem real-time është një sistem ku koha luan rol kryesor. Nomalisht, një ose me shume paisje të jashtme gjenerojne stimuj dhe kompjuteri duhet të reagoj ne menyre të drejtë kundrejt secilit prej tyre me një pjese kohe të caktuar. Për shembull, kompjuteri ne një compact disk player merr bitet sic vine dhe duhet ti konvertoje ne music ne një interval shume të shkurtër. Në qoftë se llogaritja zgjat shume, muzika do të degjohet e vecantë. Sisteme të tjera real-time janë sistemet e monitorimit të pacientëve ne spitale, piloti automatic ne avion, kontrroli i robotit ne një fabrike. Ne të gjitha keto raste, të kesh përgjigjen e duhur por ta kesh me vone se është aq keq sa mos ta kesh fare.

Sistemet real-time ndahen ne **hard real time**, domethene qe ka limitë kohore qe duhet të zbatohen me saktësi. Grupi tjtër jane **soft real time**, domethene qe të mos zbatosh ndonjë limit është i papelqyeshem por i tolerueshem. Ne të dy rastet, arrija e sjelljes real-time arrihet duke duke ndare programin ne disa procese, ku sjellja e tyre është e parashikueshme dhe e njojur qe me pare. Përgjithesisht keto procese janë të shkurtër dhe ekzekutohen brenda një sekondi. Kur ndodh një ngjarje e jashtme është detyre e schedulerit për të schedular proçeset ne një menyre qe gjithe limitet kohore të respektohen.

Ngjarjet qe një system real-time duhet ti përgjigjet mund të kategorizohen si **periodic** dhe **aperiodic**. Një sistem mund ti duhet ti përgjigjet shume ngjarjeve periodikë, kjo varet nga sa kohe kerkon çdo ngjarje për përpunim, ndoshta mund të mos jetë e mundur përballoimi i gjithe ketyre proceseve. Për shembull, në qoftë se kemi m ngjarje periodikë dhe ngarja i ndodh me përiod P_i dhe i duhen C_i sekonda të kohes se CPU qe të menaxhoj çdo proces, keshtu pra ngarkesa mund të përballohet vetëm në qoftë se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Një sistem real-time kur përputhet me ketë kritër quhet **schedulable**.

Si një shembull, konsiderojme një soft real-time me tre ngjarje periodikë, me perioda 100, 200 dhe 500 msec, përkatësisht. Në qoftë se keto ngjarje kerkojne 50, 30 dhe 100 msec kohe CPU-je për ngjarje dhe sistemi është schedulable, sepse $0.5 + 0.15 + 0.2 < 1$. Në qoftë se një ngjarje e katërt me periodë 1 sec shtohet, sistemi do të ngelet schedulable deri ne momentin qe kjo ngjarje nuk do me shume se 150 msec të kohes se CPU-se.

Schedulimi real-time mund të jetë static ose dinamik. I pari i merr vendimet e tij para se sistemi të filloje pune. Tjetri i merr vendimet e tij gjatë kohes kur sistemi është ne ekzekutim. Schedulimi static punon vetëm kur ka informacion të saktë para ekzekutimit mbi punen qe do kryhet. Schedulimi dinamik nuk i ka keto kufizime.

2.5.5 Rregullat dhe mekanizmat

Deri tani, kemi supozuar se gjithe proceset ne system i përkasin përdoruesve të ndryshem dhe të gjithe konkurrojne për CPU-ne. Për sa kohe kjo është e vertetë, por ka raste kur një proces ka shume femije qe ekzekutohen nen kontrollin e tij. Për shembull, një sistem manaxhimi të dhenash mund të ketë shume femije. Çdo femije mund të punoje ne kerkesa të ndryshme ose secili mund të ketë funksion specific. Është totalisht e mundur qe procesi kryesor të ketë një ide të qartë se cili nga femijet e tij është procesi me i rendesishem. Por asnjë scheduler nuk e bën dot ketë.

Zgjidhja e ketij problem është ndarja e **mekanizmave të schedulimi** nga **rregullat e schedulimit**. Me pak fjale duhet të parametrizohet algoritmi i schedulimit, por parametrat mund të merren nga proceset user. Le të marrim edhe njehere si shembull rastin e data bases. Supozojme se kernel përdor një algoritem me prioritet dhe siguron një system call me ane të se ciles një proces mund të rregulloje prioritetin e femijes se tij. Ne ketë menyre prindi mund ta caktoje vetë si do të shkoje schdulimi i femijes se tij, edhe se nuk është ai qe e bën schedulimin.

2.5.6 Schedulimi I threadsave

Kur çdo proces kane thread-sa të shumefishta, kemi dy nivele të paralelizmit prezentë: procese dhe thread-sa.

Schedulimi ne keto lloj sistemesh ndryshon thellisht dhe është ne varesi thread-save të nivelit-user dhe thread-save të nivelit-kernel.

Le të konsiderojme thread-sat e nivelit user të parat. Duke qene se kernel nuk është ne dijeni të threadsave, ai punon si gjithmone, merr një proces le të themi A dhe i jep A-se quantumin e saj. Scheduleri i thread-save brenda A vendos qe thread të ekzekutoj, themi A1. Deri sa nuk ka interruptë clock-u ne multiprogramin e threadsave, ai mund të vazhdoje të ekzekutohet sa të doje. Në qoftë se ai mbaron quantumin e tij atëherë kernel do zgjedhi një proces tjetër.

Kur procesi A ekzekutohet prape, thread-si A1 do të vazhdoje ekzekutimin e tij. Do të konsumoje gjithe kohen e A deri sa të mbaroje. Sjellja e tij antisocial nuk do ndikoje proceset e tjera. Ato do të marrin atë qe mendon se është e drejtë schedulerit nuk ka rendesi se cka procesi A brenda.

Le të konsiderojme se thread-sat e A nuk kane shume pune ne çdo cikel të CPU-se, për shembull, 5 msec pune ne 50 msec quantum. Secili punon për pak dhe me pas ja jep CPU-ne schedulerit të thread-save. Kjo do të coj ne sekuencen *A1, A2, A3, A1, A2, A3, A1, A2, A3, A1*, para se kernel të ndryshojne ne B. Kjo situatë ilustrohet ne Fig. 2-43(a).

Algoritmi i schedulimit i përdorur nga sistemet run-time mund të jetë njëri nga keto të meposhtëmit. Me shume përdoren round-robin dhe prioriteti me schedulim. Problemi i vetëm është mungesa e interruptit të clock-ut.

Tani konsiderojme situatën me nivelin kernel të thread-save. Këtu kernel zgjedh një thread të caktuar për u ekzkeututar. Nuk merret parasysh kuj i përket thread-si por edhe mund të merret parasysh. Thread-sit i jepet një quantum dhe ai nderpritet kur ky quantum mbarohet me një quantum 50 msec dhe me thread-sa qe mbarojne për 5 msec, për 30 msec radha mund të jetë A1, B1, A2, B2, A3, B3. Kjo situatë është përshkruar ne figuren 2-43.

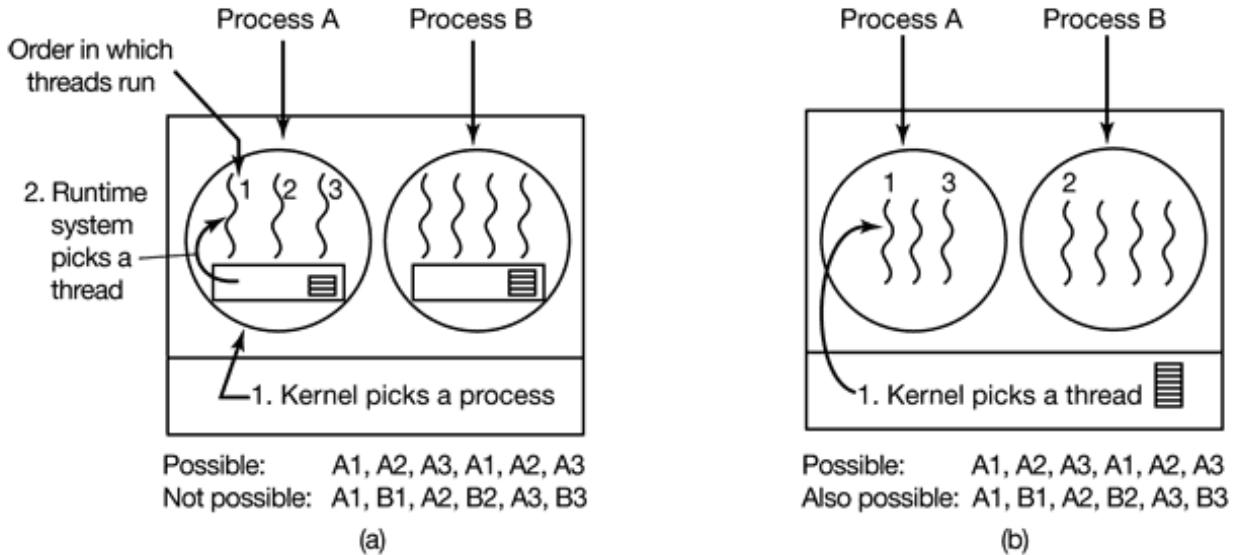


Figure 2-43. (a) schedulimi i mundshem i thread-save me 50-msec quantum dhe 5 msec për çdo ekzekutim thread-si.

Ndryshimi i thread-save të nivelit user dhe atyre të nivelit kernel është performance. Duke bere një ndryshim threadi ne user level thread-sat marrin dore të instrukSIONEVE makine. Me nivelin kernel duhet një nderrim i plotë, nderrim i hartës se memorjes, invalidimi i caches. Nga ana tjetër, me nivelin kernel threads, qenia e thread-save ne nivelin I/O nuk pushon gjithe proceset sic behet ne nivelin user.

Duke qene se kernel e di se kalimi nga një thread ne procesin A ne një thread ne procesin B është me kushtueshme se ekzekutimi i një thread-i tjetër ne A (për shkak se duhet të nderrohet memory map dhe cache), ai mund ta marri parasysh ketë fakt kur të marri vendimin e tij.

Një faktore tjetër i rendesishem është se thread-sat ne nivelin user mund të marrin një scheduler specific. Konsideroni për shembull, një Web server ne Fig. 2-10. Supozojme se një thread-s është bllokuar ne dispatcher thread dhe 2 threadsa punetore jane gati. Kush duhet të vazhdoje? Sistemi run-time duke ditur se cbën çdo thread mund të zgjedhi dispatcherin dhe keshtu mund të filloje një tjetër punetore punen. Kjo strategji rrit paralelizmin ku punetoret shpesh bllokohen ne disk I/O.

2.6 Kerkime mbi proceset dhe thread-sat

Ne kapitullin 1 pame disa nga kerkimet e bera ne strukturen e sistemeve operative. Tani do të shohim kerkimet mbi proceset. Disa subjektë Jane me të sqaruar se disa të tjere. Kerkimet bazohen ne subjeketë qe Jane të pazbuluara dhe jo ato qe kaneviteqe trajtohen.

Proçesi është një koncept shume i trajtuar. Gati çdo sistem ka një nocion të proçesit, një **mbajtës I për** grupimin e të dhenave qe kane të bejne me njëra-tjetren, si hapesira e adresave, thread-sat, file-t e hapur, ruajtja dhe lejet. Sisteme të ndryshme e bejne ndryshe grupimin, por ka vetëm ndryshime inxhinierike.

Thread-sat Jane ide me e re se proçeset, keshtu qe ka ende kerkime ne lidhje me to. Hauser et al. (1993), pa si programet reale përdorin thread-sat dhe nxorri 10 përfundime për përdorimin e thread-save. Schedulimi i threadsave multiproçessor dhe jo (Blumofe and Leiserson, 1994; Buchanan and Chien, 1997; Corbalán et al., 2000; Chandra et al., 2000; Duda and Cheriton, 1999; Ford and Susarla, 1996; and Petrou et al., 1999).

Gjithemone prane threadsave qendron sinkronizimi i threadsave dhe mutual exclusion. Me 1970 dhe 1980 nuk kishte shume pune mbi keto tëma, me shume ishin të fokusuar mbi performance. (e.g., Liedtke; 1993), mjetë për të dalluar gabimet e sinkronizimit (Savage et al, 1997) ose modifikimi i koncepteve të vjetra me menyra të reja. (Tai and Carver, 1996, Trono, 2000). Si përfundim POSIX dhe paketat e threadsave ende prodhohen dhe raportohen ne (Alfieri, 1994, and Miller, 1999).

2.7 Përbledhje

Për të fshehur efektët e interrupteve, sistemi operativ prodhon një model konceptual qe konsiston ne ekzekutimin e proceseve ne paralel. Proçeset mund të krijohen dhe përfundohen ne menyre dinamike.

Për disa aplikacione është mire të kemi thread-sa të shumefishtë kontrolli ne një proçes. Keto thread-sa schedulohen ne menyre të pavarur dhe secili ka stakun e vet, por çdo thread ne një proçes ndan një hapesire adresash të përbashketa. Thread-sat mund të implementohen ne hapesiren user ose kernel.

Proçeset mund të komunikojne me njëri-tjetrin duke përdorur komunikimin me primitive, si semaphore, monitore ose mesazhe. Keto duhen për të siguruar se asnë here dy proçese Jane ne zonen kritike ne të njëjtën kohe, kjo situatë do të conte ne kaos. Proçesi mund të jetë ne ekzekutim, i ekzekutueshem ose i blokuar dhe mund të ndryshoje gjendje kur ai ose një proçes tjetër ekzekutojne një nga primitivat e komunikimit. Komunikimi intérhead është i ngjashem.

Shume algoritme schedulimi njihen. Disa prej tyre Jane përdorur fillimisht ne sistemet batch, si për shembull shortest job first. Disa Jane të përdorshem ne të dy sistemet atë

batch dhe interactive. Ketu futen round robin, priority scheduling, multilevel queues, guaranteed scheduling, lottery scheduling dhe fair-share scheduling.

Disa sisteme bejne të quartë dallimin midis mekanizmave dhe rregullave të schedulimit, gje kjo qe bën të mundur qe përdoruesit të kene kontroll mbi algoritmat e schedulimit.

Kapitulli 3

DEADLOCK

Sistemet e kompjuterit jane plot me burime te cilet mund te perdoren vetem nga nje proces ne nje moment kohe. Shembuj te shumte jane printerat, tape driver dhe slots ne tabelat e brendeshme te sistemit.

Duke pasur 2-proçese ne te njejten kohe duke shkruar te printeri nuk ka kuptim. Duke pasur 2-proçese qe perdonin te njejtin file sistem, table slot do te mbaj te pa ndryshuar nje sistem te korruptuar. Per pasoj te gjitha sistemet operative kane aftesi (perkohesht) te hyrjes tek nje proces i madh ekskluziv tek disa burime te caktuara. Per shume aplikime nje proces do akses ekskluziv jo vetem tek nje burim por te disa, per shembull, supozojme 2 procese qe duan te shkruajn ne nje CD. Kerkesa e procesit A te perdoni skanerin dhe pastaj e liron ate. Procesi B eshte i programuar ndryshe, ai kerkon nje CD-recorder ne fillim dhe pastaj e liron gjithashtu ate. Tani A kerkon nje CD-recorder por kerkesa mohohet derisa B ta kete realizuar ate, per fat te keq ne vend qe B ta realizoj CD-recorder kerkon skanerin, ne kete cast te dy proceset do te bllokohen dhe do te rrijne keshtu per gjithmon, kjo situate perben nje deadlock.

Deadlock mund te ndodhi ne makina per shembull, shume zyra kane te perhapur nje rrjet local ne shume PC te lidhur ne te. Shume sherbime si skanerat, CD-recorder, printerat dhe tape drive jane te lidhur me rrjetin si burime te ndara te vlefshme te mbajne perdonusin ne ndonje makineri. Ne qofte se keto sherbime mund te realizohen se largu (nga perdonusat shtepiake) i njejtin lloj deadlock mund te ndodhi si ky qe shpjeguam. Situata me te nderlikuara mund te perfshijn 3, 4, 5 ose me shume sherbime dhe pordorues.

Deadlock mund te gjenden ne nje sere situatash sipas kerkesave te dedikuara I/O devices. Ne nje sistem data base, per shembull, nje program mund ti duhet te mbylli regjistrime te mundshme qe eshte perdonur per te lejuar kushte te rinje. Ne qofte se procesi A ka mbyllur regjistrimin R1 dhe procesi B mbyll regjistrimin R2, dhe atehere seicili proces perpiqet te bllokoj burimet e tjeterit, gjithashtu ne kete rast kemi deadlock. Keto burime mund te zene nje burim hardware ose software. Ne kete kapitull ne do te shikojme me nga afer deadlock, te shikojme si eshte ndare dhe te studiojme disa raste se si i ndalojme ose i lejojme ato. Megjithate ky material eshte rreth deadlock ne lidhje me sistemet operative, ato gjithashtu perdonen ne sistemet e data base dhe ne shume shkenca kompjuterike, keshtu qe ky lloj materiali eshte aplikuar ne shume lloje procesesh me multiproçese. Shume eshte shkruajtur rreth deadlock: 2-bibliografi me kete teme jene shfaqur ne (operating system review) dhe mund te perdonen per reference (Newton,1979:dhe Zobel,1983). Megjithese keto bibliografi jane te vjetra, pjesa me e madhe e punes per sa i perket deadlock eshte bere perpara 1980, keshtu qe keto jane akoma shume te perdonura.

3.1 BURIMET

Deadlock mund te zene vend kur proceset kane dhene pranin ne devicer, file e keshtu me radhe. Per ta bere diskutimin rreth deadlock sa me te perafert ne mund ti referohemi objekteve qe na japin burime. Nje burim mund te jete hardware (per shembull, tape driver) ose nje pjes informacioni (per shembull, nje regjister i blokuar ne nje data base). Nje PC normalisht mund te kete shume burime te ndyshme te cilat mund te perfitohen. Per shume burime instancat e identifikimit mund t'i lejojne atij si tape driver. Kur disa kopje te burimit jane te lejueshme shume prej tyre mund te perdoren per te kenaqur ndonje kerkesa te burimit. Ne menyre te permblehdur nje burim eshte nje dicka qe ne nje moment te kohes mund te perdoret nga nje proces.

3.1.1 BURIME TE PARANDALUSHEM OSE JO

Burimet mund t'i kemi ne 2 tipe: te parandalushme ose jo. Nje burim eshte i parandalushem ne qofte se mund te merret nga nje proces duke e fituar ate pa burime anesore. Memorja eshte nje shembull i burimeve te parandalueshem. Konsiderojme per shembull, nje sistem me 32 Mb te memorjes se perdorshme, nje printer dhe 2 procese me nga 32 Mb ku secili do te printoje dicka.

Procesi A kerkon dhe merr printerin pastaj nis te ndjeke rregullat e printimit. Perpara se ai te mbaroje me llogaritjet, ai e shpenzon kohen e tij te kuantizuar swapped out. Procesi B vepron dhe mundohet, por pa sukses per te aksesuar printerin. Fuqishem ne tani kemi nje situate deadlock sepse A ka printerin dhe B memorjen dhe asnjeri nuk mund te vazhdoj pa burimet e mbajtura nga tjetri; per fat eshte e mundur te marresh memorje nga B dhe t'ia japesh A. Tashme A mund te vazhdoje printimin dhe me pas ta kompletajo ate. Tani deadlock nuk shfaqet.

Nje burim i paparandalueshem ne contrast eshte nje qe nuk mund te merret nga pozicioni fillestar pa shkaktuar deshtimin e llogaritjeve: Ne qofte se procesi ka filluar te krijoje nje CD-ROM, te marresh papritur CD-Recorder dhe t'i japesh direkt atij nje proces tjeter mund te rezultoje me nje CD te parregullt. CD-Rekorder nuk parandalueshem ne nje moment albitrar. Ne per gjithesi deadlock perfshijne burime te paparandalueshme. Deadlock te fuqishem qe perfshijne burime te parandalueshme ne per gjithesi zgjidhen per te kaluar burimet nga nje proces ne nje tjeter, ky trajtimi qe ne po bejme do te fokusohet tek burimet e paparandalueshme. Seria e ngjarjeve qe kerkohen te ndiqen per te perdorur nje burim eshte dhene me pas ne nje forme abstrakte.

- 1.** Te kerkosh burimin.
- 2.** Te perdoresh burimin.
- 3.** Te rikthesh burimin.

Ne qofte se burimi nuk eshte i gatshem kur kerkohet, procesi i kerkimit eshte i detyruar te pres. Ne disa sisteme operative procesi blokohet automatikisht, kur nje kerkes burimi deshton dhe rivepron kur ai behet i vlefshem. Ne sisteme te tjera kerkesa deshton ne nje gabim kodi dhe kalon ne procesin e thirrjes, pret pak dhe fillon procesi.

Nje proces kerkesat e burimeve te te cilit sapo jane mohuar natyrshem zene vend ne nje nyje te fiksuar duke kerkuar burimin me pas bien ne qetesи pastaj e provojne perseri. Megjithese ku proces nuk eshte i blokuar, ne kuptimin e mirefillte, eshte njesoj sikur te ishte sepse nuk kryen asnje pune te dobishme. Ne trajtimin tone te radhes, ne do te trajtojme rastin kur nje procesi i mohohet kerkesa per burimin ai kalon ne gjendje sleep (nuk eshte aktiv).

Natyra ekzakte e kerkimit te burimit eshte komplet ne varesi te sistemit. Ne disa sisteme, kerkesa thirrje sistem, siguron qe te lejoje procesin per te pyetur per burimin ne menyre eksplikite. Ne te tjere, te vetmet burime qe sistemi operativ njeh, jane disa file speciale qe vetem nje proces mund ti hapi ne nje kohe te dhene. Keto hapen nga nje open call i zakonshem. Ne qofte se file vazhdon te jete ne perdonim thirresi erresohet derisa thirresi qe po e perdon e mbyll ate.

3.1.2 Marrja e burimeve

Per disa lloje burimesh sic jane registrat ne database e nje sistemi eshte detyre e procesit perdonues te menaxhoj perdonimin vetjak te burimit. Nje menyre e mundshme e lejimit te menaxhimit te burimit nga perdonuesi eshte shoqerimi i çdo burimi me nga nje semafor. Keta semafor jane te gjithe te inicializuar nga (mutex) dhe mund te perdoren mjaft mire te gjithe. Te tre keto shkalle qe permendem, jane te implementuara ne semafor sic jane treguar me poshte ne fig.3.1.

```
typedef int semaphore;      typedef int semaphore;  
semaphore resource-1;      semaphore resource -1;  
                           semaphore resource - 2;  
  
void process_A(void) {      void process_A(void) {  
    down(&resource - 1);      down (& resource-1);  
    use-resource -1();       down (& resource- 2),  
    up(&resource 1);          use both-resources();  
}  
up(&resource_1);           up(&resource-2);
```



Figure 3-1. Using a semaphore to protect the resources, (a) One resource, (b) Two resources.

Disa here proçeset duan dy ose me shume burime, ato mund te fitohen dale-ngadale sic tregohet ne (fig.3-h-p). Ne qofte se nevojiten me shume se dy burime ato fitohen njeri pas tjetrit. Kaq larg, kaq mire. Per aq kohe sa nje proçes eshte i perfshire çdo gje punon mire. Natyrisht qe vetem ne nje proçes nuk eshte e nevojshme te formosh perfitime burimesh derisa nuk ekziston nje tip gare per ta.

Tani le te marrim ne considerate nje situate me dy proçese A , B dhe dy burime. Dy skenare jane paraqitur ne fig. 3.2. Ne fig. 3.2 a te dy proçeset kerkojne burimet e te njejtin rregull. Ne fig. 3.2 b ata i kerkojne burimet ne menyra te ndryshme. Kjo diference mund te duket e paket , por nuk eshte e tille.

Ne fig. 3.2 a nje nga proçeset mund te fitoje burimin e pare me perpara se tjetri. Ai proçes mund te fitoje atehere ne menyre te suksesshme burimin e dyte: dhe te na beje neve te punojme. Ne qofte se proçesi tjeter tenton te fitoje burimin me perpara se ai te realizohet, proçesi tjeter thjesht do te bllokohet deri sa te behet i vlefshem.

Ne fig.3.2 b situate eshte e ndryshme. Mund te ndodhe qe nje proçes i fiton te dy burimet dhe efektivisht i bllokon te gjitha proçeset e tjera deri sa mbaroje.

Megjithate mund te ndodhe gjithashtu qe proçesi A te fitoje burimin e pare dhe B fiton burimin e dyte. Çdo njeri prej tyre tani bllokon tjetrin. Asnjë proçes tjeter nuk mund te ndodhe, kjo situate eshte nje deadlock.

Tani ne shohim se si shfaqet nje difference e vogel ne nje menyre kodimi, burimet e se ciles te fituara ne fillim kthehen per te bere diferençat ndermjet nje programi qe punon dhe nje programi qe deshton ne nje menyre te veshtire gjetjeje. Sepse deadlock mund te arrije ne menyre shume te lehte, shume prej kerkimeve kane shkuar ne ndarjen e tyre. Ky kapitull diskuton deadlock-et ne detaje dhe se cfare mund te behet me ta.

```

typedef int semaphore;

semaphore resource-1;      semaphore resource-1;
semaphore resource-2;      semaphore resource-2;

void process_A{void) {      void process-A(void) {
down(&resource -1);        down (& resource-1);

```

```

down(&resource 2);           down(&resource 2);
use-both-resources();        use-both-resources();
up(&resource-2);            up{ &resource-2};
up(&resource-1);            up(&resource-1);
}
void process-B(void) {
down(&resource-1);          down(&resource-2);
downj&resource -2);         do wn(& resource - 1);
use - both -resources{};    use- both - resources ( ) ;
up(&resource-2);            up(3cresource-1);
up(&resource-1);            up(&resource -2):
(a)                         (b)

```

Figure 3-2, (a) Deadlock-free code, (b) Code with a potential deadlock.

3.2 HYRJE TEK DEADLOCK

Deadlock-et mund te perkufizohen si me poshte:

Nje set proçesesh eshte deadlock (I ngecur) ne qofte se çdo proçes ne set eshte duke pritur nje ngjarje qe vetem nje proçes tjeter ne set mund ta shkaktoje: sepse nga te gjithe proçeset qe jane duke pritur asnje prej tyre nuk mund te shkaktoje ndarje, ngjarja qe mund t'i zgjoje te gjithe anetaret e tjere te setit dhe keshtu te gjitha proçest vazhdojne te presin per gjithmone.

Per kete model ne permblehdhim qe proçeset kane vetem nje thread te vetem dhe nuk kane interrupt-e te mundshme per te vene ne pune proçeset e bllokuara. Situata e jo-interrupt-it i duhet te kete (te beje prone te veten) nje proçes deadlock-u tjeter per te zgjuar dhe per te dhene nje alarm dhe me pas te shkaktoje ngjarjet qe te vene ne pune te gjitha proçeset e setit.

Ne rastet me te shumta ngjarjeje qe çdo proçes eshte duke pritue eshte realizimi i disa burimeve te njepasnjeshmë, pasuar nga nje anetar tejter i setit. Me fjale te tjera çdo anetare i proçesit te deadlock-ut eshte duke pritur per nje burim qe fitohet nga nje proçes deadlock. Asnje prej proçesave nuk ndodh, asnje prej tyre nuk realizon ndonje burim, dhe asnje prej tyre nuk mund te vihet ne pune.

Nr i proçesve dhe nr i llojit te burimeve te paraqitura dhe te kerkura jane te pa vlefshem. Keto rezultate jepen per ndonje lloj burimi duke perfshire hardware dhe software se bashku.

3.2.1 KUSHTET PER DEADLOCK

Coffman ka treguar qe 4 kushte nevojiten per nje deadlock.

1. Kushte perjashtuese, pershkruese çdo burim eshte per gjithesisht i shenuar kur nje proçes eshte i vlefshem.
2. Kushti i mbajtjes dhe i pritjes. Proçeset e rrjedhshem mbajne burimet e vjetra qe kerkojne burime te reja.
3. Kushti i paparashikueshem. Burimet e vjetra nuk mund te hiqen me force nga proçesi. Ato mund te shmangen dhe proçesi ti mbaje ato.
4. Kushti i pritjes cirkulare. Mund te ekzistoje nje rruge cirkulare e dy ose me shume proçesave ku secili prej tyre eshte duke pritur qe nje burim ta marre si nje anetar tjeter te zinxhirit.

Te gjitha keto 4 kushte mund te nevojiten qe te ndodhe nje deadlock. Ne qofte se ndonjeri prej tyre mungon asnje deadlock nuk ndodh. Eshte e nevojshme te thoni qe çdo njeri prej ketyre kushteve krijon nje tip kontrolli qe nje sistem mund ta kete ose jo .

A mund nje burim te regjistrohet ne me shume se nje proçes ne te njejten kohe? A mund nje proçes te mbaje nje burim dhe te kerkoste per nje tjeter? A mund nje burim te parashikohet ? A ekzistojne zinxhiret cirkulare ?

Me vone ne do te shohim se si deadlock-et mund te sulmohen per te bllokuar nje nga keto fusha.

3.2.2 MODELI DEADLOCK

Halt (1972) trgon se si keto 4 kushte mund te modelohen per te perdorur grafike direkt. Grafiku ka dy lloj shenjash. Proçest qe paraqiten me rrathe dhe burimet te paraqitur si katrore. Nje hark nga katrroi tek nje rrreth d.m.th. qe burimi eshte kerkuar, sigurohet dhe mbrohet nga proçesi. Ne fig 3.0 (a) burimi R eshte shenuar tek proçesi A. Nje pjese e proçesit tek nje burim d.m.th. qe proçesi eshte bllokuar perkohesisht duke pritur per burimin .

Ne fig. 3.3. b proçesi B eshte duke pritur per burimin S. Ne fig 3.3.c ne shohim nje deadlock. Proçesi C eshte duke pritur per burimin T i cili nderkohe kerkon te ndihmohet nga proçesi D. Proçesi D nuk eshte duke realizuar burimin T sepse eshte duke pritur per burimin U i cili mbrohet nga C. Te dy proçeset do te presin per gjithemon ciklin A me

grafik; kjo d.m.th. qe eshte nje deadlock qe perfshin proceset dhe burimet ne cikel (theksojme qe ka burime te ndryshme per çdo lloj). Ne kete shembull cikli eshte C-T-D-U-C .

Tani le te shohim nje shembull tjeter se sa grafike burimesh mund te perdoren. Imagjinoni qe ne kemi tre procese A, B dhe C dhe tre burime R,S,T.

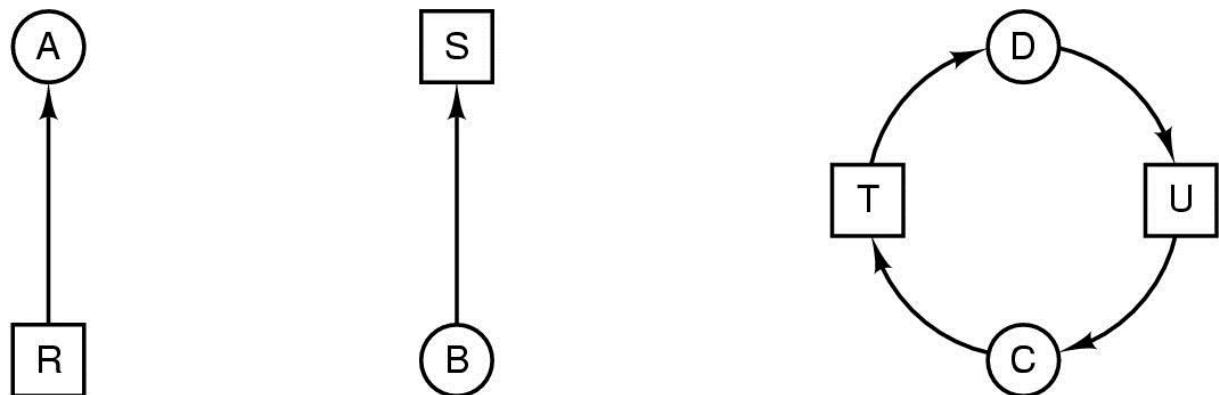


fig 3.4(a-e) (a) Resource allocation graph, (b) waiting queue, (c) dependency graph
a resource, (c) Deadlock,

Kerkesat dhe realizimet e tre proceseve jane te dhena ne fig 3.4(a-e). Sistemi operativ eshte prirur te kapi ndonje proces te pa bllokuar ne ndonje moment keshtu qe ai mund te vendosi te kapi A , ne momentin qe A mbaron pune .Me pas ai kap B qe ta kompletoje dhe ne fund kap C.

Ky rregull nuk lejon te ndodh ndonje deadlock (sepse nuk ka ndonje gare per burime), por gjithashtu nuk ka ndonje parallelizem te per gjithshem. Ne perfundim te burimeve te kerkuara dhe te realizuara procesi realizon dhe ben I/O. Kur proceset kafen njeri pas tjetrit, nuk ka ndonje mundesi qe ndersa nje proces tjeter mund te perdori CPU. Kjo kapje strikte e proceseve ne menyre sekuenciale mund te mos jete me e mira. Ne anen tjeter ne qofte se asnje nga proceset nuk ben asnje I/O ne per gjithesi ne qofte se bejne pune te tjera me te mira eshte me mire se te bejne nje (plackitje)ciklike.

Per disa rrethana te besh disa procese ne menyre te njepasnjeshe eshte me e mira. Tani le te supozojme se proceset i bejne te dy I/O dhe njehsimet, keshtu qe vjedhja

ciklike e shperndare eshte nje algoritem i programuar i arsyeshem. Kerkesat e burimve mund te ndodhin sipas rregullit tek fig **3.4.d.**

Ne qofte se keto 6 kerkesa mbahen me kete rregull 6 grafiket e burimeve rezultative jane te treguar ne fig **3.4(eKj)**. Pas kerkeses 4 qe eshte bere A bllokon pritjen per S sic pregohet ne fig **3.4.h**. Ne dy hapat e tjere B, C bllokohen gjithashtu , si perfundim forcon lidhjen ne cikel dhe deadlock nuk ndodh fig **1.4 (i,j)**

Megjithate sic kemi permendur SO nuk kerkon te kapi proceset me ndonje rregull special ne qofte se dhenia e nje kerkesa te vacant forcohet lidhet ngusht sipas nje deadlock SO thjesht mund ti pezulloj proceset pa dhene kerkesen (per shembull thjesht nuk programon procesin) derisa te sigurohet, te ruhet ne fig **3.4**. Ne qofte se SO di rreth varesis se deadlock ai mund te pezulloj B para se te japi S.

Duke vepruar vetem A dhe C ne mund te marrim kerkesat dhe r fig **3.4(k)** ne vend te **3.4(d)**. Keto sekuencia forcojne atehere grafikun e burimit te fig **3.4(i)-(q)**, e cila nuk lejon te ndodhin deadlock.

Pasi hapit (q), procesi B mund ti jepet S, sepse A ka mbaruar dhe C ka çdo gje qe i duhet.

Edhe ne se B mund te bllokohet kur kerkon 7, asnje deadlock nuk und te ndodhe, B thejsht do presi derisa te mbaroje C.

Me vone ne kete kapitull ne do te studiojme nje algoritem te detajuar per berjen e llogaritjeve qe nuk lejojn cuarjen e sistemit ne deadlock.

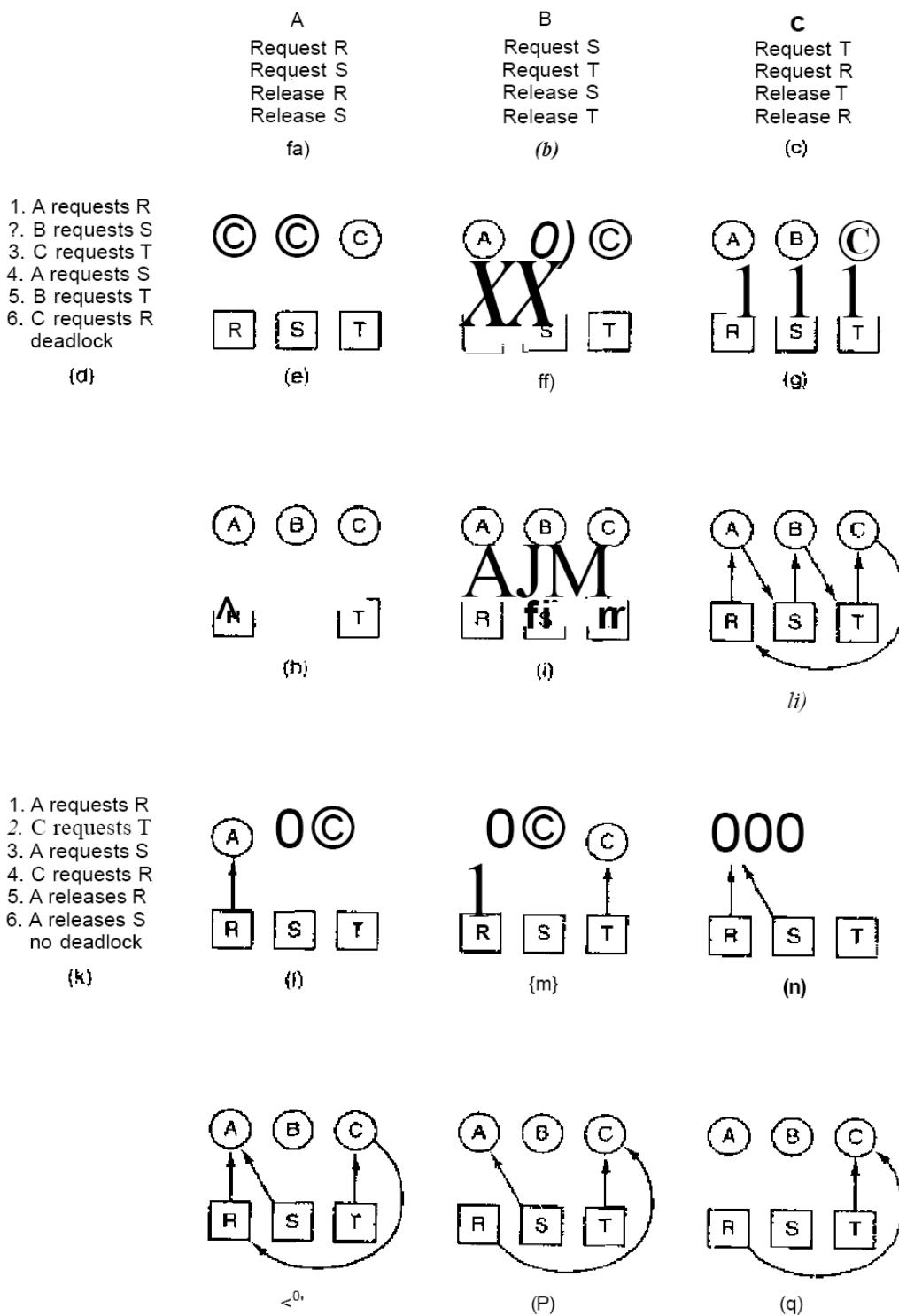


Figure 3-4. An example of how deadlock occurs and how it can be avoided.

Per momentin ajo qe duhet kuptuar eshte qe grafiket e burimeve jane nje table qe na lejon neve te shohim ne qofte se nje kerkesa dhe sekuenc realizimesh te dhena lejojne deadlock. Tani per tani ne marrim nje per nje kerkesat dhe realizimet dhe lejojme çdo hap te kerkoj grafikun per te pare nese permban ndonje cikel, gjithashtu nese kemi deadlock apo nuk kemi deadlock.

Pergjthesisht trajtimet jane mbi grafiket e burimet, jene bere per rastin e butimeve te vecanta per seicilin tip, grafiket e burimeve mund te gjeneralizohen per te shumefishuar burimet e te njejtit tip (Holt 1972).

Ne pergjithesi 4 strategji jane ndjekur per te shkurtuar deadlock.

1. Thjesht injoron problemet e gjithanshem. Mbase ngaqe ju do te injoroni ate ai do ju injoroje ju.
2. Dedekto, shtro, lejo te ndodhin deadlock dedektoje ate dhe vepro.
3. Shmangje dinamike nga llogaritjet e kujdeshme te burimeve.
4. Parandalim duke mohuar strukturalisht nje nga 4 kushtet e nevojshme per te shkaktuar nje deadlock,

Ne do te ekzaminojme seicilen prej ketyre metodave ne milimetra ne 4 sektionet pasardhes.

3.3 THE OSTRICH ALGORITEM

Afrimi me thjeshte eshte algoritmi skeletor: Fut kohen tende ne seksion dhe pretendo qe nuk ka asnje problem.

Njerez te ndryshem reagojne ndaj kesaj strategje ne menyre te ndryshme. Matematikisht nuk e konceptojne dot dhe thone qe deadlock duhet te parandalohet me çdo kusht. Inxhineret pyesin se sa here eshte kapur problemi, sa shpesh sistemi perplaset per arsyte te tjera dhe sa serioz eshte deadlock-u. Ne qofte se deadlock-u ndodh ne interval çdo 5 vejte, por sistemi perplaset me deshtimet e hardwar-eve, gabime te komplikuara dhe viruse te sistemit operativ ndodhin 1 here ne javë. Shume inxhiniere nuk do te preferonin te programonin nje penalitet te madh ne performance ose ne perfitim te eleminimit te deadlockut.

Per te bere kete contrast me specific shumica e sistemeve operative vuajne potencialisht nga deadlock-et qe nuk jane detektuar, duke u lene automatikisht te thyhen. Nr total i proceseve ne nje sistem percaktohet nga numri i hyrjeve nga nje proces table. Keto tabela proces Jane burime te perfunduar. Ne se nje dyjezim deshton sepse tabela eshte e plote nje perafrim i arsyeshem i programit duke bere dyjezimin, eshte nje kohe random dhe te provoje perseri.

Tani supozoni qe nje sistem UNIX ka 100 slote per proces, 10 programe fillojne te veprojne ku secili prej tyre duhet te krijoje 12 nenprocese. Me pas çdo njeri prej ketyre

proçeseve krijon 10 proceset origjinal dhe 90 procese te rind te cilet e ngarkojne tabelen. Çdo njeri prej 10 proceseve origjinal tani ulet ne nje nyje fundore, dyjezohet dhe deshton, kjo perben nje deadlock. Probabiliteti qe kjo te ndodhe eshte shume i vogel, por mund te ndodhe.

A do ta braktisim procesin dhe thirrjen per dyjezim per te eleminuar problemin?

Numri maksimal i fileve te hapura eshte i kufizuar ne menyre te njashme nga permusat e inodc table, nje problem i njashem ndodh dhe kur ajo mbushet plot. Hapesira e swap ne CD eshte nje tjeter burim i kufizuar, ne fakt shumica e çdo tabele ne sistemin operativ paraqet nje burim te kufizuar. A do ti braktisim ne te gjitha keto sepse mund te ndodhe qe nje koleksion i N proceseve mund te pretenduje Vn te totalit dhe çdo tjeter te pretenduje tjetren.

Shumica e sistemeve operative perfshire UNIX dhe WINDOWS thjeshte e injorojne problemin ne nje permbledhje, qe shume perdorues do t preferonin nje deadlock rastesor se sa nje rregull qe kufizon te gjithe perdoruesit ne nje proces, nje file i hapur dhe çdo gje tjeter. Ne qofte se deadlocket do te elemiñoheshin tek boshti kryesor nuk do te kete shume diskutime. Problemi eshte qe cmimi eshte shume i larte, aq shume sa vendosen dhe kufizime ne procese. Keto jane te ballafaqueshme ne nje situate te pakendshme midis nevojshmerise dhe korrekteses: nje diskutim tjeter i madh eshte se kush eshte me i rendesishem dhe per cfare.

Ne keto kushte zgjidhje te per gjitheshme jane veshtire per tu gjetur.

3.4 PERCAKTIMI I DEADLOCKEVE DHE RIFITIMI I GJENDJES

Nje teknike tjeter eshte detektimi dhe rivendosja. Gjate perdorimit te kesaj teknike sistemi nuk tenton te parandaloje ngjarjen deadlock, perkundrazi i lejon ato te ndodhin dhe tenton te identifikoje se kur mund te ndodhe dhe me pas mer masa per rivendojsen e gjendjes pas ndodhise. Ne kete seksion do te shikojme disa menyra per identifikimin e deadlock dhe disa menyra se si trajtohen rastet e rivendosjes se gjendjes.

3.4.1 DETEKTAMI I DEADLOCKEVE ME NJE BURIM PER SECILIN TIP

Le te fillojme me nje rast te thjeshte ku kemi vetem nje paisje per çdo tip. Si psh ne nje sistem te kemi vetem nje skaner nje CD-rom nje plotter nje tape drive por jo me shume se nje eksistencë per çdo klase paisjesh. Me fjale te tjera per momentin nuk do te trajtojme sisteme me paisje te dyjezuar per shembull, nje sistem me dy printerat. Keto lloje sistemesh do ti trajtojme me vone duke perdorur nje metode tjeter.

Per keto lloj sistemesh mund te ndertojme nje grafik burimesh (paisjesh) sipas ilustrimit ne figuren 3-3. Nese ky grafik permban nje apo me shume cikle athere mund te themi qe

ndodh nje ngjarje deadlock. Çdo proces qe eshte pjese e ketije cikli mund te quhet nje ngjarje deadlock. Nese nuk ekziston asnjë cikel atehere sistemi nuk ka ngjarje deadlock.

Si nje shembull me kompleks se ata qe kemi pare deri tani le te supozojme nje sistem me shtate procese, A deri ne G dhe gjashte paisje (burime) R deri ne W.

Rradha e paisjeve aktualisht ne perdomim dhe e atyre te kerkuara per perdomim nga proceset eshte si me poshte:

1. Procesi A kap R por deshiron S.
2. Procesi B nuk kap asgje por deshiron T.
3. Procesi C nuk kap asgje por deshiron S.
4. Procesi D kap U dhe deshiron S dhe T.
5. Procesi E kap V dhe deshiron V.
6. Procesi F kap W dhe deshiron S.
7. Procesi G kap U dhe deshiron U.

Pyetja ne kete rast eshte: “A mund te ndodhe ne kete sistem nje ngjarje deadlock? Nese po cilat jane proceset qe perfshihen ne kete ngjarje?”

Per tiu perjigjur kesaj pyetje nderotjme nje grafik burimesh si ne figuren 3-5(a). Ky grafik sic mund te shikohet dhe me pamje te pare permban nje cikel. Cikli paraqitet ne figuren 3-5(b). Nga ky cikel kuptojme se proceset D, E dhe G jane te perfshire brenda nje ngjarje deadlock. Proseset A, C dhe F nuk jane te perfshire ne nje ngjarje deadlock sepse S mund te vihet ne dispozicion per secilin nga keto procese, te cilat pasi perfundojne e lirojne burimin (paisjen) per perdomim; me pas dy proceset e tjere mund te perdomin burimin sipas rrades dhe ta lirojne per perdomim pas perfundimit.

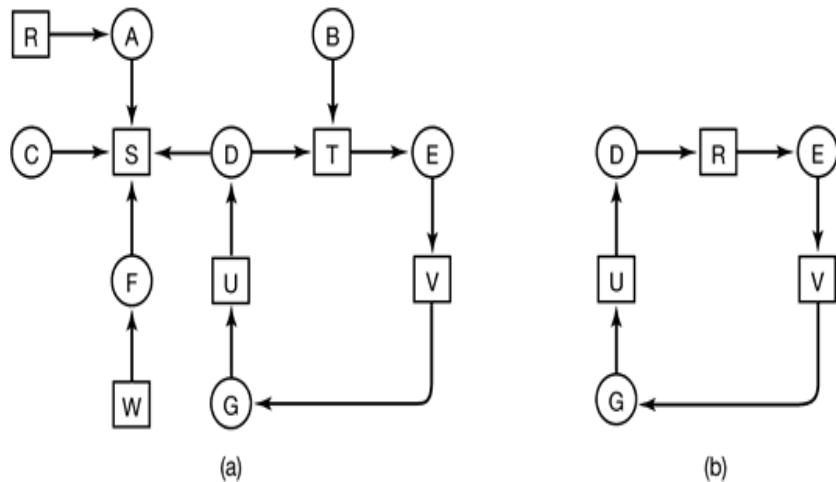


Figure 3-5. (a) A resource graph. (b) A cycle extracted from (a).

Megjidhese eshte relativisht e thjeshte per te identifikuar ngjarjet deadlock dhe proceset e perfshira ne te nga nje grafik i thjeshte, ne sistemet aktualisht ne perdomim nevojitet nje algoritem formal per identifikimin e ngjarjeve deadlock. Ekzistojne shume algoritme per identifikimin e cikleve ne grafik.

Me poshte do te japim nje shembull qe inspekton nje grafik dhe perfundon duke treguar nese ka apo jo nje ngjarje deadlock. Ky algoritem perdor nje strukture te dhenash te perbere nga nje liste nyjesh. Gjate ketij algoritmi harjet markohen per te dalluar nese ato jane inspektuar dhe per shmangjen e riverifikimeve te harqeve.

Algoritmi operon sipas hapave te specifikuar me poshte:

1. Per sejcilen nyje A^r ne grafik ndjekim 5 hapat duke mare nyjen V si nyje fillestare.
2. Boshatisim listen L dhe identifikojme harjet si te pa kontrolluara.
3. Shtojme nyjen aktuale dhe kontrollojme nese ne L shfaqet dy apo me shume here.

Nese po atehere grafiku permban nje cikel te listuar ne L dhe algoritmi perfundon duke identifikuar nje ngjarje deadlock.

4. Nga nyja e dhene kontollojme per harqe te pa kontrolluara. Ne se po atehere vazhdojme me hapin 5 ne te kundert kalojme direkte ne hapin 6.
5. Zgjedhim nje hark te kontrolluar dhe e ciklojme si te kontrolluar. E ndjekim harkun per ne nyjen tjeter pasardhese dhe rikthehem i ne hapin 3.
6. Kemi arritur ne nje nga fundet e ngjarjeve. E heqim kete nyje dhe rikthehem i tek nyja qe kishim ne shqyrtim para kesaj dhe i rikthehem i hapit 3. Nese kjo nyje eshte nyja fillestare atehere grafi nuk ka cikle dhe algoritmi perfundon.

Ajo cka ben ky algoritem eshte trajtimi i sejciles nyje si rrrenje te asaj qe mund te jete nje peme dhe ben nje kerkim te thelluar ne te. Ne se i rikthehet ndonje nyjeje qe ka hasur me pare atehere identifikon nje cikel. Ne se ezauron te gjitha nyjet pasardhese te nje nyje te dhene rikthehet ne nyje paralele me te dhenen. Ne se tenton te rikthehet ne nje nyje paralele me rrrenjen pa identifikuar nje cikel dhe nese i ka kaluar te gjitha nyjet atehere grafi nuk permbar cikle si rrjedhim sistemi nuk permbar ngjarje deadlock.

Per ta pare se si funksionon ky algoritem ne praktike le te perdorim grafin e figures 3-5 (a) . Rradha e procesimit te nyjeve eshte arbitrale keshtu qe le ti inspektojme nga e majta ne te djathte dhe nga lart poshte. Fillojme ekzekutimin e algoritmit tek

.... *\

Dhe me pas tek

..... ,4, /?, C, S, O, T, £', F

Dhe keshtu me rradhe. Nese hasim nje cikel y algoritmi ndalon.

Le te fillojme me R dhe inicializojme L si nje liste boshe. Shtojme R ne liste dhe levizim drejte mundesise se vetme A. Kete te fundit e shtojme ne L duke bere $L = fi(A)$. Nga A shkojme ne hapin 5 duke i dhene L vlore $L = RAS$. S nuk ka harqe te tjere dales ndaj mund te themi se kemi arritur ne nje fund nyjesh, cka na detyron te rikthehem i tek A. Meqenese A nuk ka harqe te tjera dalese, rikthehem i tek R duke cilesuar si te plotesuar inspektimin e pikes A.

Rifillojme ekzekutimin e algoritmit duke filluar nga A dhe duke boshatisur listen L. Ky kerikm gjithashtu eshte i thjeshte si dhe kerkimi i meparshem. Keshtu qe kalojme tek B. Nga B ndjekim harjet dalese deri sa arrijme tek D, ne kete rast lista L ka vlore $L =$

[B.T. E. V, (7. L.L Z>]..... Tani na duhet te bejme nje zgjedhje te rastit. Nese zgjedhim S arrijme ne nje fund nyjesh ndaj duhet te rikthehem i tek D. Heren e dyte zgjedhim T dhe lista L mer vlere **L=[£, T, Ey V, O, U, D, 7]**....., ne kete pike shikojme qe ne graf kemi nje cikel dhe ndalojme algoritmin.

Ky algoritdem eshte shume larg optimales. Per nje algoritdem me te detajuar shikon (Even 1979). Megjithate demostron ekzistencen e algoritmeve per identifikimin e ngjarjeve deadlock.

3.4.2 DETEKTIMI I DEADLOCKEVE ME BURIME TE SHUMFISHTA

Ne rast se kemi ekzistencen e shume tipeve te te njejtes paisje (burim) nevojitet te ndiqet nje tjeter menyre per identifikimin e ngjarjeve deadlock. Tani do te trajtojme nje algoritdem matricor per identifikimin e ngjarjeve deadlock midis n proceseve, **P_i** deri ne **P_t**. Le te jete numri klasave te paisjeve (burimeve) m, me E [burime te klases]. **F_i** burime te klases 2 dhe E burime te klases i ($1 < i < m$). E eshte nje vektor me burime ekzistente. Ne te paraqiten totalitet e instancave per sejcilin burim ekzistent. Per shembull nese klasa 1 eshte nje tape drive E t - 2 do te thote qe sistemi ka dy tape drives.

Ne nje cast te caktuar disa nga burimet jane te zena dhe nuk jane te lira per shfrytezim. Le te jete A nje burim i lire i vektorit, me 4 paraqitet numri i instancave te burimeve / qe jane aktualisht te lira per perdorim (te pa zena). Ne se te dy tape drives do te ishin te zena ahene 4 do te kishte vleren 0.

Tani na duhen me teper vektore, **C** matrica aktuale e perdorimit dhe R. Matrica e kerkesave.

Rrjeshti Z-te i C tregon se sa instanca te klases se burimeve Pf mbajne aktualisht *. Keshtu **C,7** eshte numri i instancave te burimit / qe mbahen nga procesi i. Ne menyre te ngjashme, **Ktj** eshte numri i instancave te burimit / qe kerkohen nga P. Keto kater struktura te dhenash paraqiten ne figuren 3-6.

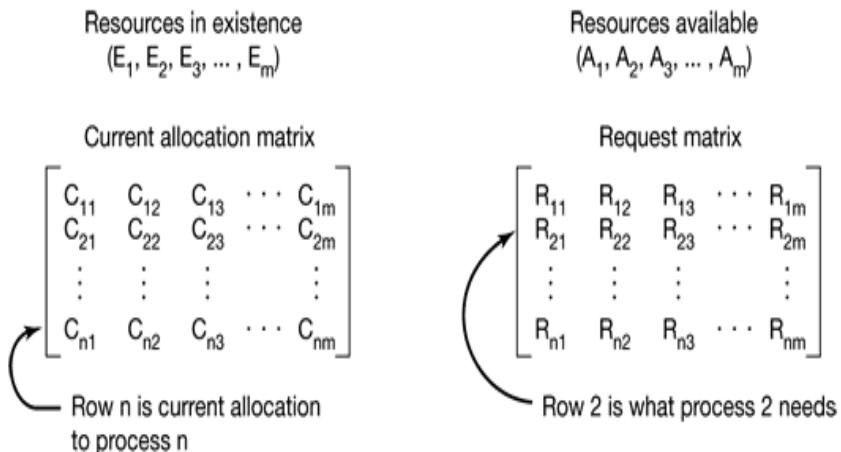


Figure 3-6. The four data structures needed by the deadlock detection algorithm.

Ne vencanti, sejcili burim eshte ose i bllokuar per shfrytezim ose i lire per shftytezim. Kete e tregon dhe struktura e te dhenave.

Algoritmi i identifikimit te ngjarjeve deadlock eshte i bazuar ne krahasimin e vektoreve. Le te percaktojme relacionin $A < B$ mbi dy vektoret A dhe B , d.m.th. çdo element i A qe eshte me i vogel ose barazim me elementin korespondent ne B . Matematikisht $A < B$ qendron vetem nese $A; \leq B$, per $I < / < m$.

Çdo proces fillimisht qendron si i pa shenuar. Gjate ekzekutimit te algoritmit proceset do te shenohen, duke treguar qe mund te plotesohen totalisht dhe nuk jane pjese e nje ngjarjeje deadlock. Kur algoritmi perfundon proceset e pashnuar njihen si procese pjesmarrese ne ngjarje deadlock.

Algoritmi i identifikimit te ngjarjeve deadlock mund te jepet si me poshte:

1. Kerkojme per proces te pashnuar, *Pit* per te cilin rrjeshti *i-th* i R eshte me i vogel ose barazim me A ,
2. Nese nje proces i tille gjendet, shtojme rrjeshtin */-th* te C tek ,4, shenojme procesin dhe i rikthehem i hapit 1.
3. Nese nje proces i tille nuk ekziston algoritmi perfundon.

Kur algoritmi perfundon te gjithe proceset e pashnuar, nese ka, jane pjese te ngjarjeve deadlock.

Ajo cka ben algoritmi ne hapin e pare eshte kerkimi per procese qe mund te arrine perfundimin e vetvetes. Keto procese karakterizohen nga kerekza per burime te cilat mund te plotesohen nga burimet aktuale te lira per shfrytezim. Procesi i zgjedhur ekzekutohet dhe deri sa mbaron, kohe ne te cilin ai kthen dhe te gjitha burimet qe mban per shfrytezim, burime te cilat kthehen ne hapsiren e burimeve te gatshme per shfrytezim dhe se fundi procesi shenohet si i kompletuar. Nese te gjitha proceset mund te arrijne perfundimin e tyre atehere asnjeri prej tyre nuk ben pjese ne nje ngjarje deadlock.

Megjithate dhe ky algoritem nuk eshte percaktues (determinues) sepse ai mund te ekzekutoje proceset ne menyra jo te paracaktuara.

Si shembull per funksionimin e nje algoritmi per identifikimin e nje ngjarjeje deadlock le te konsiderojme figuren 3-7. Ketu kemi tre procese dhe kater klasa burimesh te cilat i kemi emertuar arbitrarisht tape drive, ploter, skaner dhe CD-ROM. Procesi 1 ka nje skaner. Procesi 2 ka dy tape drivers dhe nje CD-ROM. Procesi 3 ka nje ploter dhe dy skanera. Sejcilit proces i nevojiten burime shtese, si paraqitet nevoja per burime shtese, si paraqitet dhe ne matricen R.

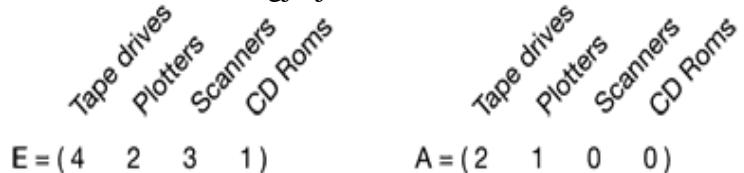
Per ekzekutimin e algoritmit te identifikimit te ngjarjeve deadlock, kerkojme per procese kerkesat e te cileve per burime mund te plotesohen. Kerkesa e pare nuk mund te plotesohet sepse nuk kemi CD-ROM te lire per shfrytezim. Kerkesa e dyte nuk mund te plotesohet gjithashtu sepse nuk kemi skaner te lire per shfrytezim. Fatmiresisht, kerkesa e trete mund te plotesohet keshtu procesi 3 ekzekutohet dhe pas perfundimit rikthen te gjitha burimet e zena per shfrytesim duke dhene

$$A = (2 \ 2 \ 2 \ 0)$$

Ne kete pike procesi 2 mund te ekzekutohet dhe ne perfundim kemi

$$A = (4 \ 2 \ 2 \ \downarrow)$$

Dhe se fundi ekzekutojme procesin 3. Nuk ka ngjarje deadlock ne sistem.



Current allocation matrix	Request matrix
$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$	$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$

Figure 3-7. An example for the deadlock detection algorithm.

Le te konsiderojme nje ndryshim ne situaten e paraqitur ne Figuren 3-7. Supozojme qe procesit 2 i nevojitet nje CD-ROM si dhe dy tape-drive dhe nje ploter. Asnje nga keto kerkesa nuk mund te plotesohen keshtu mund te themi se procesi 2 eshte pjese e nje ngjarjeje deadlock.

Tashme qe dime se si mund te identifikojme nje deadlock, pyetja qendron se kur mund te kerkojme per ndodhjen e tyre. Nje mundesi eshte te kerkojme per to sa here qe behet nje kerkesa per nje burim. Ne kete rast kemi gjetje te sigurt por kjo menyre eshte shume e shtrejnte (ne kohe perdorimi te CPU). Nje alternative tjeter strategjike eshte

kerimi per to çdo K minuta ose gjate kohes kur perdorimi i CPU ka rene nen nje nivel te caktuar.

Arsyea e konsiderimit te kohes se CPU eshte sepse nese shumica e proceseve jane pjesa e ngjarjeve deadlock mund te kete shume pak procese te mundeshme per ekzekutim dhe CPU mund te jete e lire per ekzekutimin e algoritmeve te identifikimit te proceseve deadlock.

3.4.3 RIFITIMI I GJENDJES

Supozojme qe algoritmi yne i identifikimit te ngjarjes deadlock pati sukses duke identifikuar nje ngjarje te tille. Me pas ne ndonje menyre do na duhet te zhbllokojme kete gjendje te sistemit ne menyre te tille qe ky i fundit te vazhdoje punen. Ne kete seksion do te diskutojme per menyrat e rekuperimit te gjendjes se sistemit pas nje ngjarjeje deadlock.

Recovery through Pre-emption

Ne disa raste mund te jete e mundur ne menyre te perkohshme te marim nje paisje burim nga perdoruesi aktual dhe ti'a japim per perdorim nje procesi tjeter. Ne shume raste nevojitet nderhyrje manuale vecanerisht gjate batch processing (grumbullimit te programeve) te sistemeve operative qe ekzekutohen ne mainframe (zhvillimet kryesore). Per shembull, per marrjen e printerit nga nje perdorues operatori duhet te rezervoje pjesen e skedarit te paprintuar ta vendose ne nje magazine te perkoheshme, ne kete moment te shenoje procesin si te ndaluar per momentin dhe ne kete pike tia jape printerin per perdorim nje procesi tjeter. Kur ky i fundit te kete mbaruar pune te nxjerre pjesen e paprintuar te skedarit e cila ndodhet ne magazinen e perkoheshme, ta vendose ate ne rradhen e printimit dhe me pas ta deklaroje procesin original si te filluar. Ky veprim i marrjes se paisjes burim nga perdorimi per nje proces te caktuar dhe kalimin per perdorim ndaj nje procesi tjeter varet shume nga natyra e burimit.

Rekuperimi ne kete menyre eshte shpesh shume i veshtire ose i pamundur. Zgjedhja e procesit qe duhet te ndalohet eshte ne vartesi se kush nga keto procese kane ne perdorim burime te cilat mund te rimeren ne perdorim ne menyre te thjeshte.

Recovery through Rollback

Nese projektuesit e sistemit dhe operatoret e makinave kane njohuri mbi ngjarjet deadlock, mund te realizojne kontroll gjendjeje periodik te proceseve. Kontrolli i gjendjes se nje procesi ka te beje me shkrimin e gjendjes se nje procesi specifik ne nje skedar ne menyre te tille qe ta rikthjeme procesin ne gjendjen e meparshme. Per te qene me efektiv gjendjet e reja te proceseve nuk duhet ti mbivendosen atyre te vjetrave por duhet te rishkruhen ne skedare te rinje ne menyre te tille qe te formojme nje sekunce gjendjesh.

Kur identifikohet nje ngjarje deadlock mundet te percaktohen lehte paisjet burim te cilat nevoiten. Per realizimin e rikthimit te gjendjes normale procesi i cili ka paisjen e nevojshme ne perdom kthehet ne nje gjendje perpara se te merte kete paisje ne perdom duke perdom skedaret e kontrollit te gjendjes. E gjithe puna e bere pas rikthimit te gjendjes se procesit humbet. Ne te vertete, procesi kthehet ne nje gjendje te meparshme gjate te ciles nuk e kishte akoma ne perdom paisjen burim e cila ne kete moment i jepet per perdom procesit qe ben pjese ne nje ngjarje deadlock. Nese procesi i cili ka rimar gjendje te me parshme kerkon serish perdomin e kesaj paisjeje duhet te prese qe kjo paisje burim te behet e mundshme per perdom.

Rikuperim nepemjet vrasjes se procesit

Megjithate menyra me e shkurter per daljen nga nje gjendje deadlock eshte shkatrimi i proceseve qe bejne pjese ne kete ngjarje. Me pak fat procesi deadlock mund te vazhdoje por nese kjo nuk mundeson vazhdimin e ketij procesi atehere shkatrimi i proceseve qe perbejne ciklin deadlock vazhdon deri sa cikli te shperbehet.

Nje alternative tjeter eshte shkaterrimi i nje procesi qe nuk ben pjese ne cikel ne menyre qe te liroje paisjet burim ne perdom. Ketu duhet te zgjedhim me kujdes procesin qe duhet shkateruar ne menyre te tille qe ky proces te kete ne perdom paisje te nevojshme per proceset ne ciklin deadlock. Per shembull supozojme nje proces qe po perdom nje printer dhe i nevoitet nje ploter dhe nje proces tjeter qe perdom ploterin por i nevoitet nje printer. Keto dy procese bejne pjese ne nje cikel deadlock. Nje proces i trete mund te mbaje ne perdom nje printer dhe nje ploter identik me ata aktualish ne perdom nga proceset ne deadlock. Shkatrimi i procesit te trete dhe lirimi nga perdomi i paisjeve ne perdom mund te kenaqe kerkesat e dy apo me shume proceseve ne deadlock.

Kur eshte e mundur, eshte me mire te shkatrrosh nje proces te tille dhe ta riekzekutosh pa efekte anesore.

Per shembull nje proces kompilimi mund te riekzekutohet sepse ajo cka ben eshte leximi nga nje skedar burim dhe prodhimi i nje skedari objekt. Nese ky proces shkatrrohet gjate ekzekutimit riekzekutimi i tije nuk eshte apsolusht i influencuar nga eksekutimi i pare. Nga ana tjeter dhe nje proces update i nje baze te dhenash nuk mund te ekzekutohet per here te dyte ne menyre te sigurt. Nese ky proces shton vleren 1 tek nje rekord i bazes se te dhenave nese ky proces shkatrrohet ekzekutimi per here te dyte i tije mund te shtoje me 2 vleren e keti rekordi, cka perben nje gabim.

3.5 SHMANGJA E DEADLOCK

Gjate diskutimit per identifikimin e ngjarjeve deadlock, supozuam se nje proces gjate kerkeses per paisje burim munde te kerkoste per shume te tilla ne te njejtene kohe (matrica R ne figuren 3-6) ne shumecen e sistemeve burimet kerkohen ne rradhe.

Sistemi duhet te jete ne gjendje te vendose nese ofrimi paisjes burim nga nje proces siguron vazhdimesi dhe te beje rezervimin ne rast se kjo gje eshte e sigurt.

Ketu lind pyetja: "A ekziston nje algoritem qe i shmang gjithmone gjendjet deadlock duke bere gjithmone zgjedhjen e duhur?"

Pergjigja eshte: “ po mund ti shhangim deadlock por kur kemi informacionin e duhur perpara ”.

Ne kete seksion do te ekzaminojme shhangien e gjendjeve deadlock duke zgjedhur me kujdes ofrimin e paisjeve burim per dororim.

3.5.1 Trajektoret e burimeve

Algoritmet kryesore per realizimin e shhangieve te gjendjeve deadlock jane te bazuara ne konceptin e gjendjeve te sigurta. Perpara se te peshkruajme keto algorime le te hedhim nje sy mbi koceptin gjendje e sigurt te peshkruar ne menyre grafike.

Megjithese ana grafike nuk perkthehet drejtoperdrejte ne nje algoritem te perdorshem, ajo jep nje pamje te qarte te natyres se problemit.

Ne figuren 3-8 paraqitet nje model qe trajton dy procese dhe dy burime psh. Nje priter dhe nje ploter. Akset horizontale paraqesin instruksionet e ekzekutuara nga procesi A. Akset vertikale perfaqesojne instruksionet e ekzekutuara nga procesi B.

Ne I_1 , A kerkon nje printer; ne I_4 i nevoitet nje ploter. Priteri dhe ploteri lirohen nga perdorimi ne I_1 dhe I_4 , respektivisht. Prosesit B i nevoitet nje ploter nga I_1 deri ne I_7 dhe nje printer nga *ify* deri ne K .

Cdo pike ne diagram perfaqeson nje status te bashkuar te dy proceseve.

Si fillim statusi eshte ne p , ku asnje proces nuk ka ekzekutuar asnje intruksion. Nese skeduleri vendos te ekzekutoje ne fillim A, ne arrijme ne piken q . Ne te cilen, A ka ekzekutuar disa instruksione, kurse B asnje. Ne piken q trajektorja behet vertikale, duke treguar keshtu qe skeduleri ka zgjedhur te ekzekutoje B. Me nje procesor te vetem te gjitha trajektoret duhet te jene horizontale ose vertikale por asnjeher diagonale.

Per me teper orientimi eshte gjithmone nga veriu ose lindja, asnjeher nga jugu apo perndimi. (proseset nuk ekzekutohen asnjeher mbrapsht). Kur A te kaloje trajektoren nga r te s ai kerkon printerin dhe pranohet. Kur B arrin piken t, ai kerkon plotter.

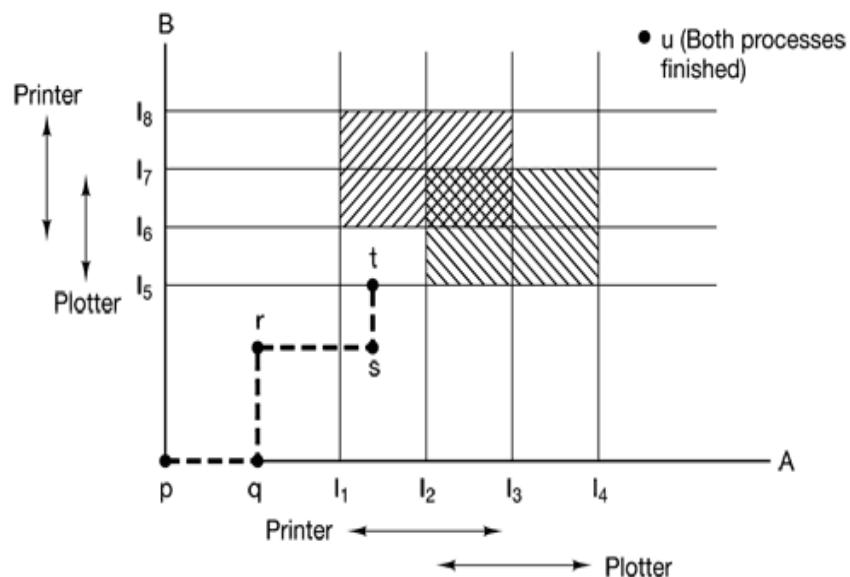


Figure 3-8. Two process resource trajectories.

Hapesirat e hijezuara jane shume interesante. Pjesa me vija te drejtuara nga jug-perendimi ne veri-lindje perfaqesojne te dyja proceset se bashku qe zoterojne printerin. Rregulli i perjashtimit ben te pamundur hyrjen ne kete zone. Ne te njejten menyre, zona e hijezuar ne formen tjeter perfaqeson te dy proceset qe zoterojne njekohesishte plotterin, dhe eshte po ashtu e pamundur.

Nese sistemi futet ne hapsiren e rrethuar nga /r dhe/: ne anesore, dhe nga! /₃ dhe /₆ larte e poshte, ai do te futet ne deadlock kur te arrije ndermjet ol'/₂ dhe /₄. Ne kete pike A po kerkon ploterin dhe B printerin, dhe te dyja jane tashme ne perdom. E gjithe hapesira eshte e pasigurt dhe ne kete zone !,olA!,4 dhe u me dalin te paqarta sipas meje per siguri shiko gjith interpretimin e grafikut sistemi nuk duhet te futet. **AL** tregon qe e vetmja gje qe duhet te behet eshte te ekzekutohet procesi 4 derisa te arrije ne /.t. Pas kesaj çdo trajktore per tek u do te vleje.

Nje gje e rendesishme per te vene re eshte qe pika B po kerkon nje burim. Sistemit i takon te vendose nese ta pranoje ate apo jo. Nese behet pranimi, sistemi futet ne nje zone te pasigurt dhe ne nje deadlock. Per te shmangur deadlockun B duhet te pezullohet deri sa A te kete kerkuar dhe mbaruar pune me plotterin.

3.5.2 Gjendjet e sigurta dhe te pasigurta

Algoritmin e shmangjies se deadlock-eve (deshtimeve) ne mund ta studiojme duke perdonur informacionin e Fig. 3-6. Ne çdo cast te kohes eshte nje gjendje qe konsiston tek f. A. C\ dhe R. Nje gjendje quhet e sigurt nese nuk ndodhet ne gjendje deadlock-u (deshtimi) dhe nese ka ndonje rregull te percaktuar ne te cilin çdo proces mund te ekzekutohet plotesisht edhe nese te gjitha papritur kerkojne menjehher numrin maximal te burimeve. Eshte e thjeshte te ilustrosh kete koncept me nje shembull duke perdonur nje burim ne Fig. 3-9(a) ku shikohet se A ka 3 instanca te burimit por mund te kete nevoje per 9. B ka momentalisht 2 por me vone do i duhen 4. Po njesoj edhe C ka 2 por mundti ti duhen 5. Ekzistojne 10 instanca te burimit, keshtu qe me 7 instanca te grumbulluara mbeten akoma 3 te lira.

Has Max														
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	-	B	0	-	B	0	-
C	2	7	C	2	7	C	2	7	C	7	7	C	0	-
Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		
(a)			(b)			(c)			(d)			(e)		

Figure 3-9. Demonstration that the state in (a) is safe.

Gjendja e Fig. 3-9 eshte e percaktuar sepse ekziston nje sekunce grumbullimesh qe lejojne te gjitha proceset te plotesohen. Nameiy, skeduleri thesht mund te ekzekutoje B ekskluzivisht dersa ai te kerkonte dhe te marre 2 instanca te tjera te burimit, duke shkuar ne gjendjen e Fig. 3-9(b). Kur te jetë plotesuar B ne marrim gjendjen e Fig. 3-9(c). Pastaj skeduleri mund te ekzekutoje C, duke u drejtuar te Fig. 3-9(d). Kur C te plotesohej shkojme tek Fig. 3-9(e). Tani A mund te marre gjashte instancat e burimit qe kishte nevoje dhe te plotesohet. Keshtu gjendja e Fig. 3-9(a) eshte e sigurte fale sistemit qe mund te skeduloje ne menyre te kujdeshme dhe te parandaloje deadlock-un.

Tani supozojme qe kemi gjendjen fillestare te treguar ne Fig. 3-10(a), por kete rradhe -4 kerkon dhe merr nje tjeter burim, duke na dhene keshtu Fig. 3-10(b).

A mund te gjejme nje sekunce qe eshte e garantuar te punoje? Le ta provojme. Skeduleri mund te ekzekutoje B derisa ai te kerkonte per te gjitha burimet te treguara ne Fig. 3-10(c).

Has Max		
A	3	9
B	2	4
C	2	7

Free: 3
(a)

Has Max		
A	4	9
B	2	4
C	2	7

Free: 2
(b)

Has Max		
A	4	9
B	4	4
C	2	7

Free: 0
(c)

Has Max		
A	4	9
B	-	-
C	2	7

Free: 4
(d)

Figure 3-10. Demonstration that the state in (b) is not safe.

Normalisht B perfundon dhe ne arrijme ne situaten e Fig. 3-10 (d). Ne kete pike ne kemi ngecur. Kemi tani vetem kater instanca te lira te burimit, kurse secila nga proceset aktive ka nevoje per pese te tilla. Nuk ka ndonje sekuese qe te garantoje perfundimin e ekzekutimit. Keshtu qe vendimi i grumbullimit qe levizi sistemin nga Fig. 3-10 (a) ne Fig. 3-10 (b) kaloi nga gjendja e sigurt ne ate te pasigurt. As te ekzekutosh A ose C pas fillimit ne Fig. 3-10 (b) nuk funksionon. Nga ana tjeter kerkesa /Ts nuk do ishte pranuar, **h \s** tani eshte nje gjendje deadlock. Po te filloje m Fig. 3-10(b) sistemi mund te ekzekutohet per nje cast. Ne fakt nje proces mund edhe te perfundoje. Per me teper, eshte e mundur qe A mund te liroje nje burim perpara se te kerkonte per te tjera duke lejuar keshtu C qe te perfundoje dhe gjithashtu te shmange deadlock-u. Keshtu pra ndryshimi midis nje gjendje te sigurt dhe nje te pasigurt eshte se ne nje gjendje te sigurt sistemi mund te garantoje qe te gjitha proceset te perfundojnë gje qe nuk garantohet ne nje gjendje te pasigurt.

3.5.3 Algoritmi i Bankierit per nje burim te vetem.

Nje algoritem rradhitjeje qe mund te shmange deadlock-et eshte fale Dijkstra (1965) dhe njihet si baker's algorithm, dhe eshte nje zgjerim i algoritmit te detektimit te deadlockeve te dhene ne 3.4.1. Eshte modeluar sipas menyres qe nje banke e vogel mund te nderveproje me klientet e saj te cileve u garanton linja krediti. Ajo cka ben ky algoritem eshte kontrolli per te pare nese pranimi i kerkeses con ne nje gjendje te pasigurt

. Nese ndodh, kerkesa mohohet. Nese pranimi te con ne nje gjendje te sigurt ateher ajo mbahet. Ne Fig. 3-11(a) shohim kater kliente, A, B, C dhe D, ku secilit i eshte pranuar nje numer i caktar kreditesh.

Bankieri e di qe jo te gjithe klientet do te kene nevoje per maksimumin e krediteve njehersh, keshtu qe ai ruajti vetem 10 njesi per tu sherbyer dhe jo 22.

	Has Max	
A	0	6
B	0	5
C	0	4
D	0	7
Free: 10		
(a)		

	Has Max	
A	1	6
B	1	5
C	2	4
D	4	7
Free: 2		
(b)		

	Has Max	
A	1	6
B	2	5
C	2	4
D	4	7
Free: 1		
(c)		

Figure 3-11. Three resource allocation states: (a) Safe. (b) Safe (c) Unsafe.

Klientet ecin me prespektivat e tyre, duke bere vazhdimeshit kerkesa. Ne nje moment te caktuar situata eshte si ne Fig. 3-11(b). Kjo gjendje eshte e sigurt sepse me 2 kreditet e mbetura bankieri mund te mohojte çdo kerkese pavec C1, dhe me C1 te mbaruar lirohen te katerta burimet e tij. Me kater kredite ne dore, bankieri mund te lejoje D ose 8 te marrin kreditet e nevojshme e keshtu me rradhe. Mendoni pak se cfare do te ndodhete nese nje kerkese nga B per nje kredit me shume do te ishte pranuar ne Fig. 3-11(b). Do te kishim situaten e Fig. 3-11(e), e cila eshte e pasigurte. Nese te gjithe klientet papritmas do te kerkonin maksimumin e sasise, bankieri nuk do mundte tia plotesonte kerkesen e asnjerit, dhe do te kishim nje deadlock. Nje gjendje e pasigurt sdo te thote qe sjell nje deadlock, meqe nje klient nuk ka nevoje per gjithe sasine e krediteve te disponueshme, por bankieri nuk mund te mbeshtetet tek kjo sjellje.

Algoritmi i bankierit i konsideron te gjitha kerkesat ti vijne, dhe shikon nese pranimi con ne nje gjendje te sigurt. Ne se po, kerkesa pranohet; perndryshe ajo shtyhet per me vone.

Ne piken H; i gjendja eshte e sigurt, bankieri kontrollon nese ka burime te mjaftueshme per te kenaqur klientet. Ne se po, sasia e krediteve supozohet te rikthehet dhe klienti me afer limitit chekohet, e keshtu me rradhe. Dhe nese sasia mund te rikthehet, do te thote qe gjendja eshte e sigurt dhe kerkesa fillestare mund te plotesohet.

3.5.4 Algoritmi i bankierit per burime te shumefishta

Ky algoritem mund te per gjithsohet qe te suportoje burime shumefishe dhe kete e tregon Figura 3-12.

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still needed

Figure 3-12. The banker's algorithm with multiple resources.

Ne Fig. 3-12 ne shohim dy matrica. Ajo ne te majte tregon sa burime jane momentalisht perdonur nga secili nga 5 proceset. Matrica ne te djathht tregon sa burime te tjera i duhen secilit proces qe te perfundoje. Keto matrica sjane gje tjeter pavec C dhe R nga Fig. 3-6. Ashtu si ne rastin me nje burim, proceset duhet te percaktojne totalin e burimeve qe kane nevoje para ekzekutimit, qe sistemi te llogarise matricen ne te djathte ne çdo rast.

Te tre vektoret ne te djathht te figures tregojne burimet ekzistuese, \underline{f} burimet ne perdonim, \underline{f} dhe burimet e lira, A. Nga \underline{f} ne shohim qe sistemim ka gjashte **tape drives**, tre ploteran kater printer, dhe dy CD-ROM

Nga keto, pese **tape drivers**, tre plotera dhe dy CD-ROM jane momentalisht te percaktuara. Ky fakt vihet re qarte tek kater kolonat ne matricen e majte.

Vektori i burimeve te gatshme eshte thjesht diferenca midis asaj cka ka sistemi dhe asaj cka eshte momentalisht ne perdonim.

Tani mund te tregojme algoritmin e kontrollit nese gjendja eshte e sigurt apo jo.

1. Gjej rreshtin R nevojat e te cilit per burime jane te barabarta apo me te vogla se A. Nese nje rresht i tille nuk ekziston sistemi do te shkoje ne gjendje deadlock meqe asnje proces nuk mund te plotosohet.
2. Supozojme qe procesi i atij rreshti te zgjedhur kerkon te gjitha burimet dhe perfundon. E shenojme kete proces si te perfunduar dhe i shtojme te gjitha burimet e tij vektorit A.
3. Perserisim hapin 1 dhe 2 derisa te gjitha proceset te jene shenuar te perfunduar, ne secilin rast qe gjendja fillestare eshte e sigurt, dhe derisa te kemi deadlock ne rastet kur kjo gjendje nuk eshte e sigurt.

Nese ne rastin 1 kemi shume raste zgjedhjesh, nuk ka rendesi se ke marrim te parin: Sasia e burimeve te gatshme ose behet me e madhe ose mbetet njesoj ne rastin me te keq.

Tani te kthehem i mbrapsht te shembulli i Fig. 3-12. Gjendja e castit eshte e sigurt. Supozojme qe procesi B tani kerkon nje printer. Kjo kerkese mund te pranohet sepse gjendja perfundimtare eshte perseri e sigurt (Procesi D mund te perfundoje, pastaj A ose E, e te tjerat).

Tani imaginoni qe pasi ti jepet B nje ose dy printerat e mbetur, E kerkon printerin e fundit. Pranimi i kerkeses mund te reduktoje vektorin e burimeve te vlefshme ne (1 0 0 0), gje qe con ne deadlock. Pra eshte e qarte qe kerkesa e E-se nuk duhet te pranohet per momentin.

Algoritmi i bankierit u publikua per here te pare ne 1965. Qe ateher pothuajse çdo liber per sistemet operative e ka peshkruar ate ne detaje. Me mijera faqe jane shkruajtur per shume aspekt te ndryshme te ketij algoritmi. Por fatkeqsisht pak autor kane shprehur qe panvaresisht se teorikisht ky algoritmet eshte i mrekullueshem, praktikisht eshte i pavlefshem sepse rralle ndodh qe proceset ta dine me pare se sa do jete maksimumi i burimeve qe do i nevojitet. Per me teper, numri i proceseve nuk eshte fikse, por dinamike duke variuar nga hyrja e dalja e shume perdoruesve. Gjithashtu burimet qe jan te vlefshme mund papritur te shkeputen nga sistemi. Keshtu qe ne praktike shume pak nga sistemet ekzistuese perdorin kete algoritmet per shmangien e deadlock-eve.

3.6. PARANDALIMI I DEADLOCK-EVE

Kemi pare qe shmangia e deadlock-eve ne thelb eshte e pamundur sepse duhet informacion rrreth kerkesave ne te ardhmen te cilat nuk njihen. Si jane sistemet reale te shmangin deadlock-et? Pergjigjja eshte duke u kthyer prapa ne kater kushtet te shpallura nga Coffman et al.(1971) per te pare nese ato mund te sigurojne ndonje te dhene. Ne qofte se mund ta sigurojme ate qe te pakten nje nga keto kushte te mos jete I realizueshem atehere deadlock-et do te jene strukturisht te pamundur (Havender,1968).

3.6.1 KRITIKAT NDAJ KUSHTIT TE PERJASHTIMIT TE NDERSJELLTE

Ne fillim le te kritikojme kushtin e perjashtimit te ndersjellte. Ne qofte se asnje burim nuk i caktohet nje procesi te vetem ne nuk do te kemi asnjeherë deadlock (bllokim). Megjithate ,eshte e qarte se te lejosh dy procese te printojne ne te njejten kohe do te coj ne kaos .Pas spooling output-in e printerit, disa procese mund te gjenerojne output-et e tyre ne te njejten kohe. Ne kete model, procesi i vetem qe porosit materialin ne printer eshte **deamon** (printer deamon). Meqenese deamon nuk do te kerkoj kurre burime te tjera ne mund te eleminojme deadlock-et per printerin .

Fatkeqesisht jo te gjitha pajisjet mund te bejne spooled (tabela e procesit nuk mund te jap veten plotesisht per te bere spooled). Vec kesaj konkurenca per hapsiren ne disk mund te coj vete ne deadlock. Çfare do te ndodhte ne qofte se dy procese do te mbushnin secili gjysmen e hapsires ne dispozicion per spooling me output-in e tyre dhe asnjeherë te mos mbaroj prodhimin e output-it ?Ne qofte se deamon eshte programuar te fillojne printimin ,perpara se i gjithe output-I te jete spooled, printeri mund te mbetet pa pune ne qofte se procesi i output-it vendos te pres disa ore perpara se te nxjerr output-in e pare. Per kete arsyen deamons jane programuar normalisht te printojne vetem pasi file perfundimtar te jete i gatshem. Ne kete rast ne kemi dy procese qe kane perfunduar

pjeserisht por jo komplet output-in (prodhimin) e tyre dhe nuk mund te vazhdojne. Asnjeri proces nuk do te perfundoje, pra kemi nje deadlock ne disk.

Megjithate ketu ka njefare ideje qe eshte shpesh e zbatueshme. Menjano caktimin e burimeve kur ato nuk jane absolutisht te nevonshme dhe perpiqu te sigurohesh se sado pak procese te jene ne te vertete mund te kerkojne burime

3.6.2 KRITIKAT NDAJ KUSHTIT TE MBAJTJES DHE TE PRITJES

Kushti i dyte i caktuar nga Coffman duket pak me premtues. Ne qofte se ne mund te parandalojme procese te cilet mbajne burime dhe presin per me shume burime ne mund te eleminojme deadlock-e. Nje menyre per te arritur kete qellim eshte tu kerkosh te gjithe proceseve te kerkojne te gjithe burimet qe u nevoiten perpara se te fillojne ekzekutimin. Ne qofte se çdo gje eshte ne dispozicion procesi mund te alokoj cfardo gjeje qe te kete nevoje dhe mund te ekzekutohet deri ne fund. Ne qofte se nje ose disa nga burimet jane te zena, asje gje nuk mund te alokohet dhe procesi duhet vetem te pres.

Nje problem qe del menjehere ne kete trajtim eshte se shume procese nuk e dine se sa burime u nevoiten deri sa ato te kene filluar ekzekutimin. Ne fakt, nese ato e dine, mund te jete perdor algoritmi banker. nje problem tjeter eshte se burimet nuk do te kene nje perdorim optimal ne kete trajtim. Marrim si shembull, nje proces qe lexon te dhena hyrese nga nje shirit (tape) hyres dhe i analizon ato per nje ore dhe pastaj shkruan shiritin dales (output tape) me se miri duke thurur rezultatin. Ne qofte se te gjitha burimet do te ishin kerkuar me perpara, procesi do te bllokonte diskun e shiritit dales (output tape drive) dhe plotter-in per nje ore.

Megjithate shume mainframe, grumbull sistemesh i kerkojne perdoruesit te bejne nje liste me te gjitha burimet ne linjen e pare te seciles pune. Sistemi atehere siguron menjehet te gjitha burimet dhe i mban ato derisa te mbaroj pune. Nje menyre pak e ndryshme per te thyer kushtin e mbajtjes dhe pritjes eshte duke i kerkuar nje procesi te bej kerkesen per nje burim por me pare te leshoj perkohesisht te gjitha burimet qe po mban. pastaj te provoj te, marre gjithcka ka nevoj pernjehersh.

3.6.3 KRITIKAT NDAJ KUSHTIT TE MOSZEVENDESIMIT

Te kritikosh kushtin e trete (moszevendesimin) eshte me pak premtues se te kritikosh e dytin. Ne qofte se nje proces i eshte caktuar printeri dhe eshte ne mes te printimit te output-it te tij i merret menjehet forcerisht printeri sepse ploteri qe eshte i nevojshe nuk eshte i disponueshem eshte i zene (nderlikuar) ne rastin me te mire dhe i pamundur ne rastin me te keq.

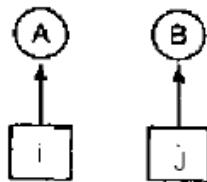
3.6.4 KRITIKAT NDAJ KUSHTIT TE PRITJES RRETHORE

Na ka mbetur vetem nje kusht. Rrethi i pritjes mund te eleminohet me disa menyra. Nje menyre eshte thjesht te kesh nje rregull qe thote qe nje proces eshte i lidhur me nje

burim te vetem ne nje moment te caktuar. Ne qofte se i duhet nje burim tjeter duhet te leshoj te parin. Per nje proces qe ka nevoje te kopjoj nje file te madh nga shiriti (tape) ne printer, kjo pengese eshte e papranueshme. Nje menyre tjeter per te shmangur rrithin prites eshte te sigurojme nje numerim te te gjitha burimeve sic tregohet ne figuren 3-13 (a). tani rregulli eshte ky: proceset mund te kerkojne burime kur te duan port te gjitha kerkesat duhet te behen ne nje rend numerik. Nje proces mund te kerkoi ne fillim nje printer dhe me pas nje disk shiriti (tape drive) por ai nuk mund te kerkoi ne fillim ploter-in dhe me pas nje printer.

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



(b)

Figure 3-13, (a) Numerically ordered resources, (b) A resource graph.

Me kete rregull, grafiku i alokimit te burimeve nuk mund te kete kurre nje cikel. Le te shohim se pse kjo eshte e vertete per rastin e dy proceseve, ne figuren 3-13 (b) Ne mund te kemi deadlock-e vetem nese A kerkon burimin j dhe B kerkon burimin i .Duke pretenduar qe **i** dhe **j** jane burime te ndryshme ata mund te kene numra te ndryshme. Ne qofte se **i>j** atehere procesit nuk i lejohet te kerkoi **j** sepse ai eshte me i paket se cfar ai ka.Ne qofte se **i<j** **B** nuk i lejohet te kerkoi **i** sepse ai eshte me i paket se cfare ai ka. Ne kete menyre deadlock-et jane te pamundur te ndodhin.

Me shumfishimin e proceseve, ndiqet e njejtja logjike. ne qdo moment nje nga burimet do te jete me larte. Procesi qe mban ate burim nuk do te pyes per burimin qe i eshte caktuar. Ai gjithashtu mund te perfundoj ose me keq mund te kerkoje nje numer me te larte burimesh te gjitha ato qe jane te disponueshme .Perfundimisht ai do te perfundoje dhe te liroj burimet e zena prej tij. Per kete shume prej proceseve te tjera mund te kapin burimet me larte dhe gjithashtu mund te perfundojne dhe ata. Me pak fjalë ekziston nje plan ne te cilin te gjithe proceset mbarojne, pra asnje deadlock nuk rezulton.

Nje ndryshim i vogel i ketij algoritmi eshte se hedh kerken qe burimi te jete saktesisht nje sekune rritese, dhe thjeshte insiston qe asnje proces te mos kerkoje nje burim me te ulet se ate qe mban. Ne qofte se procesi kerkon 9 dhe 10 dhe me pas i liron qe te dy eshte efektive qe te fillosh nga fillimi keshtu qe nuk ka arsyje qe te ndaloj kerkesen per burimin 1. Megjithese renditja numerike e burimeve eleminon problemin e deadlock-ve mund te jete e pamundur te gjesh nje rregull renditje per te kenaqur gjithkend. Kur burimet perfshijnë, proces table slots, hapsiren spooler ne disk, blokimin e rekordeve ne database dhe burimet e tjera abstrakte, numri i burimeve mund te jete aq i madhe saqe asnje renditje e mundshme nuk funksionon.

Mundesite e ndryshme per te parandaluar deadlock-e jane permblehdhurne fig 3-14

3.7 ÇËSHTJE TË TJERA

Në këtë pjesë do të diskutojmë pak çeshtje të lidhura më deadlock-un (bllokimet). Këta përfshijnë dy faza- bllokimi, deadlock-et (bllokim) pa burim dhe starvacioni (vdjekje nga uria).

3.7.1 DY FAZA BLLOKIMI (MBYLLJEJE)

Megjithse menjanimi dhe parandalimi, që të dy nuk jane tmerrësish premtues në çeshtje kryesore, për aplikime specifike, njihen shume algoritma perfekt për qellime speciale. Si për shembull, ne shume sisteme baza të dhenash (database) një operacion që ndodh herë pas here është kërkese bllokimi në disa rekordet dhe pastaj duke i rifreskuar të gjitha rekordet e bllokuara. Kur proceset e shumtë janë në ekzekutim në të njëjtën kohë, aty është një rrezik real për tu krijuar një deadlock (bllokim).

Afrimi me shpesh i përdorur është quajtur **dy-faza bllokimi** (mbyllje). Ne fazën e parë procesi provon të bllokoj të gjitha rekordet që ka nevojë, një nga një. Në qoftëse kjo ndodh, fillon faza e dytë performuar update (rifreskim) e tij dhe zbllokuar ato. Në fazën e parë muk është bërë një punë e vërtetë.

Nëse gjatë fazës së parë disa rekorde që ishin të nevojshme ishin tanimë të bllokuara procesi i zhbllokon ato dhe fillon fazën e parë nga fillimi përseri. Në një farë sensi ky afrim është si të kërkosh të gjitha burimet më përpara, ose të paktën përpara se diçka e pakthyeshme të ketë ndodhur. Në disa versione të dy faza bllokimi (mbylljeje), nuk ka zhbllokim dhe ristarto nese bllokimi është ndeshur gjatë fazës së parë. Ne këto versione mund të ndodh deadlock.

Megjithatë kjo strategji në per gjithësi nuk është e aplikushme. Në sistemet në kohe reale dhe sistemet e kontrollit të proceseve, për shembull, është e papranueshme që thjeshtë të përfundosh një proces në mes sepse burimi nuk është i disponueshëm dhe te fillosh gjithçka nga fillimi. Gjithashtu nuk është e pranuashme të fillosh përseni në qoftëse procesi gjen mesazhet e shkruara ose te lexuara ne një rrjet, update-on (rifreskon) fajlat ose gjithçka tjeter që nuk mund te përsëritet. Algoritmi punon vetëm në ato situata ku programuesi ka rregulluar me shume kujdes gjerat që programi mund të ndalojë në çdo pikë gjatë fazës së parë dhe të ristartoj. Shumë aplikacione nuk mund të jene strukturuar në këtë mënyrë.

3.7.2 DEADLOCK-ET PA BURIME

E gjithë puna jonë e gjertanishme është e përqëndruar në burimet e deadlock-ut. Një proces dëshiron diçka që një process tjeter ka dhe duhet të pres derisa i pari ta lëshoj atë. deadlock-u mund te ndodh edhe ne situata të tjera, gjithsesi, duke përfshirë këta duke mos ngatërruar fare burimet. Për shembull, mund të ndodh që dy procese deadlock të presin

njëri tjetrin për të bërë diçka. Kjo ndodh shpesh me semaforët. Në kapitullin 2 ne treguam shembuj në të cilën ka për të kaluar poshtë ne dy semaforë, tipiku ***mutex*** dhe nje tjetër si ai në qofte se këto mund te bëhen në një rend të gabuar, mund të rezultoj një deadlock (bllokim).

3.7.3 STARVACIONI

Një problem i lidhur drejtpërdrejt me deadlock-un është **starvacioni**. Në një system dinamik, kërkesa për burime ndodhin gjatë gjithë kohës. Disa politika janë të nevojshme për të marrë një vendim rreth kush të marrë, cilin burim, dhe kur. Kjo politik, megjithese duket e arsyeshme mund të çoj që shumë proceseve të mos u shërbehet kurrë megjithese ato nuk janë deadlock-e. Si për shembull, konsideroni alokimin e printerit. Imagjinoni që sistemi përdor shumë lloj algoritmash per te siguruar që alokimi i printerit te mos çoj në deadlock (bllokim). Tani supozoni që disa procese e duan atë në të njëjtën kohë. Cili prej tyre do ta marrë atë?

Nje algoritmom alokimi i mundur është tia jap atë procesit me file-t më të vegjël per tu printuar (duke spozuar që informacioni është i gatshëm). Ky trajtim maksimizon numrin e klientëve (proceseve) te gjuevar dhe duket i drejte.

Tani gjykon se çfarë ndodh në një system të zënë ku një proces ka një file gjigand për tu printuar. Çdo herë që printeri është i lirë, sistemi zgjedh procesin me file-t me te shkurter. Në qoftë se aty është një nivel constant me procese me file të shkurtër, procesit me file-n gjigand nuk do ti caktohet kurre e drejta për të printuar. Ai thjesht paracaktohet të ngordh urie deri në vdekje, të pres deri në fund(të zgjas për një kohë të papercaktuar, sadoqë nuk është i bllokuar).

Starvacioni mund të shmanget nga përdorimi,i pari që vjen i pari shërbehet, politikat e alokimit të burimeve. Me këtë trajtim proceset presin që me i gjati te sherbehet herës tjetër. Në rrjedhën e duhur të kohës, secili proces i dhënë vendos përfundimisht të bëhet më i vjetër dhe kështu merr burimet e nevojshme.

Condition	Approach
Mutual exclusion	Spoof everything
Hold&Wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 3-14. Summary of approaches to deadlock prevention.

KAPITULLI I KATËR

MENAXHIMI I MEMORJES

Memorja eshte nje burim qe duhet menaxhuar me shume kujdes. Me kalimin e kohes, kompjuterat per shtepi, ne kohet e sotme, kane njemije here me shume memorje se IBM7094, kompjuteri me i madh ne bote i viteve 1960, programet po behen me te shpejte se memorja. Le te citojme ligjin e Parkinsonit: Programet shtrihen per te mbushur memorjen e gatshme per t'i mbajtur. Ne kete kapitull do te studiojme se si sistemi operativ menaxhon memorjen.

Ajo qe çdo programues do te donte, eshte nje memorje pafundesisht e madhe dhe e shpejte, qe nuk i humb te dhenat gjate mungeses se energjise. Meqe jemi te memorja, pse nuk kerkojme qe ajo te jete edhe e pakushtueshme? Fatkeqesisht teknologjia nuk na siguron memorje te tilla. Si rrjedhoje, pjesa me e madhe e kompjuterave kane nje hierarki te memorjes me nje sasi shume te vogel dhe te shtrenjte te memorjes te paqendrueshme dhe te shpejte cache, dhjetera MB shpejtesi mesatare, nje çmim mesatar, memorje te paqendrueshme RAM dhe dhjetera e qindra GB te nje disku te lire dhe te ngadalte te qendrueshem. Eshte puna e sistemit operativ te kordinuje punen e seciles prej ketyre memorjeve. Pjesa e sistemit operativ qe ben te mundur menaxhimin e memorjes quhet **menaxhues i memorjes (memory manager)**. Puna e tij eshte te regjistroje se cila pjesa e memorjes po vihet ne pune si dhe cila jo, per ta lidhur ate me proceset dhe per ta shkeputur prej tyre kur kane perfunduar si dhe per te bere swaping midis memorjes vetjake dhe diskut kur ajo eshte teper e vogel per te perballuar proceset.

Ne kete kapitull do te studiojme varietet menaxhimi memorjesh duke u nisur nga me te thjeshtat tek me te sofistikuarat.

Sic e theksuan ne kapitullin e pare, historia e kompjuterave perserit veten.

4.1 Menaxhimi baze i memorjes

Sistemi i menaxhimit te memorjes ndahet ne dy klasa: ato qe transportojne proceset midis memorjes vetjake dhe hardiskut dhe ato qe nuk e bejne nje gje te tille. Keto te fundit jane me te thjeshta, ndaj do te studjohen te parat. Me vone do te studojme swap-in dhe paging. Keto te fundit shkaktohen prej pamundesisë se memorjes vetjake per te perballuar

procëses shume te medha njekohesht. Ne qofte se memoria kryesore zgjerohet aq shume sa nuk ka me vend, argumentat ne favor te nje lloji te skemes se menaxhimit te memories ose te tjera mund te zhduken.

Nga ana tjeter softwar-et po behen edhe me te shpejte se vete memorja, duhet nje menaxhim i memorjes shume eficient. Ne vitet 80, pati shume universitete qe perdoren sistemin e ndarjes se kohes me nje dyzine perdoruesish me 4 MB VAX. Tani Microsofti rekomandon qe te kete te pakten 64 MB per nje perdorues te vetem te sistemit Windows 2000. Trendi drejt multimedias kerkon akoma me shume memorje, keshtu qe nje menaxhim akoma me i mire do te duhet ne dekadat qe vijojne.

4.1.1 Monoprogramimi pa swap dhe paging

Skema me e thjeshte e menaxhimit te memorjes eshte procedimi i nje programi te vetem ne nje kohe te caktuar, duke e ndare memorjen midis atij programi dhe sistemit operativ. Ne figuren 4-1 jane treguar tre variante te kesaj skeme. Sistemi operativ mund te ndodhet mbi RAM, tek ROM ose driver-at e pajisjeve ndodhen ne ROM dhe pjesa tjeter ne RAM. Modeli i pare eshte implementuar ne kompjuerat mainframe si dhe tek minikompjuterat. Modeli i dyte eshte perdorur tek kompjuterat palmtop si dhe tek ata embeded. Modeli i trete eshte perdorur tek kompjuterat e hershem desktop(per MS-DOS), ku pjesa e sistemit e ndodhur ne RAM quhej *BIOS (Basic Input Output System)*

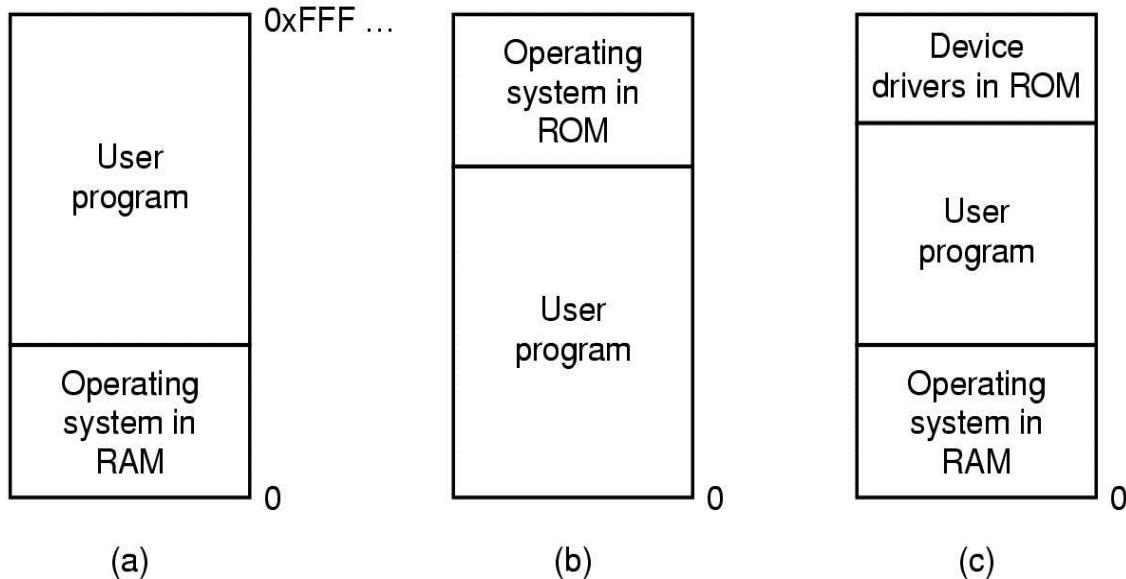


Figura 4-1.Tre menyra te organizimit te memororjes me nje system operativ dhe me nje proces perdorues.

Kur sistemi eshte organizuar ne nje menyre te tille, vetem nje proces ne nje kohe te caktuar mund te proçedoje. Ne çastin kur perdoruesi shtyp nje komande, sistemi operativ kopjon programin e kerkuar nga hard disku ne memorje dhe e ekzekuton ate. Kur procesi perfundon sistemi operativ gjeneron menjehere nje karakter dhe pret per nje komande tjeter. Me te marre komanden, ai e ngarkon programin e ri ne memorje duke ia mbishkruar te parit.

4.1.2 Multiprogramimi me particione fikse

Per veçse ne sistemet te nderfutura, monoprogramimi veshtere se mund te perdoret. Sistemet moderne lejojne procese te shumefishta qe veprojne ne te njejten kohe. Te kesh procese te shumefishta qe veprojne ne te njejten kohe do te thote qe, kur nje proces eshte duke pritur pajsjen I/O te perfundoje, nje tjeter eshte duke perdorur CPU. Keshtu, multiprogramimi rrit perdorimin e CPU. Serverat e rrjetit, gjithnje kane aftesine qe te veprojne me procese te shumefishta (per kliente te ndryshem) ne te njejten kohe. Tashme edhe makinat “klient” e kane kete aftesi.

Menyra me e thjeshte per te realizuar multiprogramimin eshte duke e ndare memorjen ne n pjesë(mundesisht jo te barabarta). Kjo ndarje per shembull.mund te behet manualisht kur hapet sistemi.

Kur nje pune eshte per t'u realizuar, ajo vendoset ne radhen e inputeve ne pjesen nje pjesë te vogel, por me madhesi te mjaftuesheme per ta mbajtur. Duke qene se pjeset kane nje madhesi fikse, ajo qe nuk eshte perdorur per ndonje pune te caktuar, shkon dem. Ne fig.4-2 (a) shohim sesi duket nje sistem i tille. Disavantazhi i vendosjes se puneve ne pjesë fikse behet i dukshem kur nje radhe me pjesë te medha eshte bosh dhe nje tjeter me pjesë te vogla eshte e mbushur plot si ne rastin e pjesave 1 dhe 3 ne fig 4-2(b). Sa here qe pjesa lirohet, puna me e afert qe ndodhet perballë radhes mund te ngarkohet ne pjesen bosh dhe te realizohet. Duke qene qe nuk eshte e deshirueshme qe te shkoje dem nje pjesë per nje pune te vogel, ndiqet nje tjeter strategji qe te kerkohet neper te gjithe radhen e inputeve per nje punë të mjaftueshem te madhe per tu ngarkuar ne pjesen e madhe. Vihet re qe algoritmi i fundit veçohet kundrejt puneve te vogla si te ishte e pavlere patja e nje particioni te plote, nga ana tjeter zakonisht pelqehet qe punes me te vogel (shpesh punet interaktive) ti jepet sherbimi me i mire, jo me i keqi.

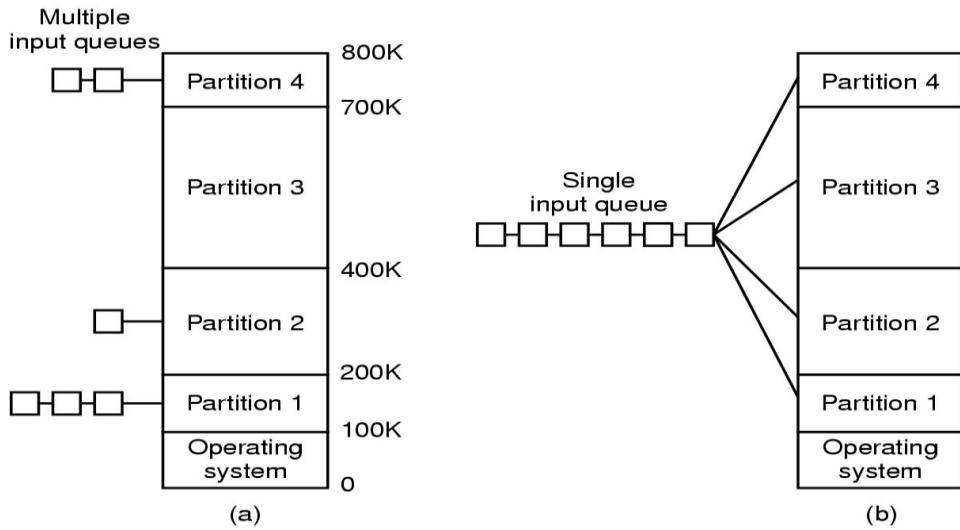


Figura 4-2.(a) Memorje me pjese fikse me radha inputesh te ndara per cdo pjese.(b) Memorje me pjese fikse me nje radhe te vetme inputesh.

Nje menyre zgjidhjeje per kete problem eshte te kesh ne gjendje nje pjese shume te vogel. Nje pjese e tille e vogel do te lejonte qe aty te procedoheshin si pune te vogla, pa pasur nevoje per te alokuar pjese me te medha.

Nje perafrim tjeter eshte te kesh nje rregull i cili nuk e lejon nje pune te caktuar te proçedoje me shume se k here.Sa here qe ajo proçedon, ajo merr nga nje pike. Kur ajo ploteson k pike, ajo nuk proçedon me.

Ky sistem me particione(pjese) fikse, i vendosur nga operatori “ne mengjes” dhe qe nuk ndryshon me, eshte perdorur gjeresisht nga OS/360 ne kompjuterat e medhenj mainframe.U quajt MFT (Multiprograming with Fixed number of Tasks). Nuk ka veshtiresi per t'u kuptuar dhe eshte i thjeshte, gjithashtu, per t'u implementuar: puna e ardhur vendoset ne nje radhe derisa nje pjese e pershtatshme te jete e gatshme per te. Aty puna ngarkohet dhe procedon derisa te perfundoje. Ne ditet e sotme pak sisteme operative e implementojne kete model.

4.1.3 Multiprogramimi me modelim

Kur perdoret multiprogramimi, perdorimi i CPU mund te jetë i qarte. Ne qofte se rapporti i procedimit te puneve ne kohe eshte rreth 20%, me pese procese ne kohe te caktuar,CPU do te ishte e zene gjate gjithe kohes. Ky model eshte ne menyre jo shume optimiste, pasi jo te peste proceset do te presin per pajisjet I/O ne te njeften kohe.

Ne model me mire eshte ta shohesh perdonimin e CPU nga nje pikepamje me probabilistike. Supozojme qe nje proces pret gjate nje fraksioni p te kohes se tij per

procedimin e pajisjeve I/O. Me n procese ne nje kohe te caktuar ne CPU, probabiliteti i pritjes se tyre per pajisjet I/O eshte p^n . Perdorimi i CPU jepet nga formula:

$$\text{Perdorimi i CPU} = 1 - p^n$$

Figura 4-3 tregon grafikisht perdorimin e CPU ne funksion te n e quajtur **shkalla e multiprogramimit**.

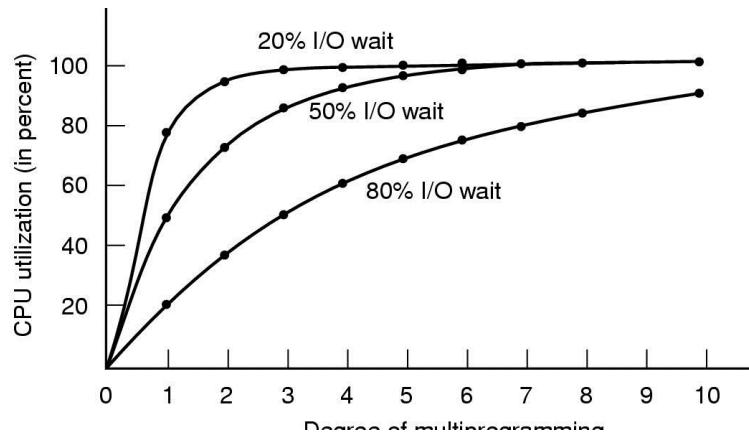


Figura 4-3. Perdorimi i CPU si funksion i numrit te proceseve ne memorje

Nga figura eshte e qarte qe proceset shpenzojne 80% te kohes se tyre per te pritur pajisjet I/O. Duhet te jene me pak se 10 procese ne memorje ne menyre qe CPU te harxhoje me pak se 10%. Kur kupton qe nje proces interaktiv eshte duke pritur per nje perdonues te shtype dicka nga nje terminal e ai eshte ne gjendjen e pritjes per I/O, duhet te jete i qarte fakti qe harxhimi i 80% te kohes eshte mese normal. Edhe ne sistemet batch, proceset qe perdonin sasi te madhe te diskut perqindja e tyre eshte po aq, madje edhe me shume.

Duhet te behet e ditur qe modeli probabilistik i sapo pershkruar eshte vetem nje perafrim. Ai thjesht nxjerr perfundimin qe n procese jane te pavarura, cka do te thote qe eshte e pranueshme qe ne 5 procese te ngarkuara ne memorje qe veprojne njekohesisht, realisht 3 te jene duke vepruar dhe dy te tjere duke pritur. Por me nje CPU te vetme nuk mund te kemi 3 procese qe veprojne njekohesisht, keshtu qe nje proces qe behet gati nderkohe qe CPU eshte e zene, do t'i duhet te prese. Keshtu proceset nuk jane te pavarura. Mund te ndertohet nje model me i pershtatshem duke perdonur teorine e radheve por ne momentin kur bejme multiprogramim lejohet qe proceset te perdonin CPU edhe pse ajo do te operoj me minimumin e shpejtjesise, sigurisht, per sa kohe eshte i vlefshem, edhe pse linjat e fig 4-3 jane paksa te ndryshme.

Edhe pse modeli i mesiperm i fig.4-3 eshte i thjeshte, ai gjithsesi mund te perdoret per te bere parashikime te perafruara per performancen e CPU. Supozojme, per shembull qe kemi nje memorje prej 32 MB, me nje sistem operativ qe merr 16 MB dhe secili program i perdonuesit qe merr rreth 4 MB. Keto permasa lejojne 4 programe te perdonuesit qe te

alokohen ne memorje njekohesht. Me nje sasi prej 80% te pritjes per pajisje I/O, ne kemi nje perdonim te CPU prej 1-0.8⁴ ose gati 60%. Duke shtuar rreth 60% te memorjes qe e lejojne sistemint te kaloje nga nje model me kater rruge per multiprogramim ne nje tjeter me 8 rruge, performanca e CPU rritet ne 83%. Ne fjale te tjera shtimi prej 16 MB do ta rrise throughput-in me 38%.

Duke shtuar edhe 16MB te tjera do ta çoje throughput-in ne 93%, pra nje rritje me 12%. Duke perdonur kete model, perdonuesi do te mendoje qe shtimi i pare ne memorje do te ishte nje investim i mire por i dyti, jo.

4.1.4 Analiza e performances se sistemit me multiprogramim

Modeli i diskutuar me siper eshte perdonur per te analizuar sistemet batch. Le te marrim ne konsiderate, per shembull,nje kompjuter qendror ku ne punet e te cilit, 80% eshte pritje per pajisjet I/O. Ne nje dite te veçante, punet jane mbivendosur si ne fig 4-4(a). Puna e pare, e cila vjen ne oren 10⁰⁰ kerkon 4 min nga koha e CPU.Me 80% pritje per pajisjet I/O ,puna e ardhur perdon vetem 12 sek nga koha e CPU perçdo minute qe ajo pret ne memorje, edhe pse s'ka pune te tjera qe konkurojne me te per kohen e CPU. 48 sek e tjera jane shpenzuar duke pritur per procedimin e pajisjeve I/O. Keshtu nje pune e caktuar duhet te qendroje se paku per 20 min ne memorje qe te perfitoje 4 min nga koha e CPU, edhe ne mungese te nje konkurrence

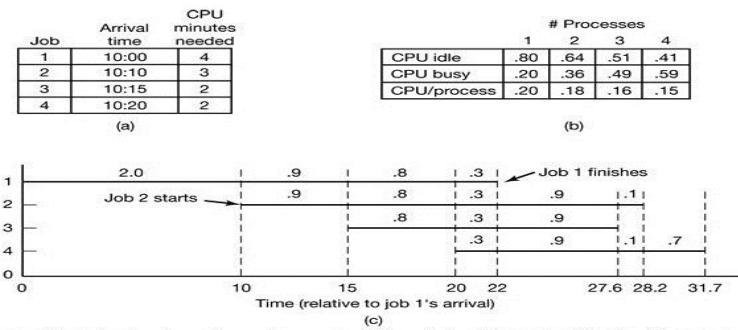


Figure 4-4. (a) Arrival and work requirements of four jobs. (b) CPU utilization for 1 to 4 jobs with 80 percent I/O wait. (c) Sequence of events as jobs arrive and finish. The numbers above the horizontal lines show how much CPU time, in minutes, each job gets in each interval.

Prej ores 10:00 deri ne oren 10:10 puna e pare eshte e vetme ne memorje dhe perfiton vetem 2 min per procedimin e saj. Kur puna e dyte ngarkohet ne memorje ne oren 10:10, perdonimi i CPU rritet nga 0.20 ne 0.36, fale shkalles se larte te multiprogramimit (shiko fig 4-3).Gjithsesi me ane te skedulimit (round-robin), secila pune shfrytezon gjysmen e

CPU. Keshtu secila pune merr 0.18 min nga puna e CPU e caktuar per cdo min qe ajo eshte ne memorje.Duhet nenvizuar qe cdo shtese e nje pune te dyte ne memorje i kushton punes se pare vetem 10% te performances se saj. Nga 0.2 per cdo min nga koha e CPU ajo tashme perfiton 0.18 per minute nga koha e saj.

Ne oren 10:15 ngarkohet ne memorje puna e dyte. Ne kete çast puna e pare ka perfituar 2.9 min nga koha e CPU dhe puna e dyte ka perfituar 0.9 min. Me nje multiprogramim me 3 rruge, secila pune perfiton 0.16 min nga koha e CPU per cdo min nga koha reale, sic eshte treguar ne fig 4-4(b).Nga ora 10:15 ne oren 10:20 secila nga tre punet perftojne 0.8 min nga koha e CPU. Ne oren 10:20 ngarkohet puna e katert.Fig 4-4(c) tregon sekuencen e plote te ngjarjeve.

4.1.5 Rivendosja dhe mbrojtja

Multiprogramimi na shfaq dy probleme esenciale qe mund te zgjidhen rivendosja dhe mbrojtja. Shiko fig 4-2. Nga figura duket qarte qe pune te ndryshme do te procedojne ne adresa te ndryshme. Kur nje program eshte linkuar(per shembull. programi kryesor, programi i shkruar nga perdoruesi dhe librari procedurash te kombinuara ne nje hapesire te vetme adresash), linkuesi duhet ta dije se ne cfare adrese duhet te filloje programi.

Per shembull.supozojme qe instruksioni i pare eshte nje thirrje procedure ne adresen absolute 100 ne file-in binar te gjeneruar nga linkuesi. Ne se programi eshte ngarkuar ne pjesen e pare(ne adresen 100K), ky instruksion do te kerceje ne adresen absolute 100, e cila eshte ne brendesi te sistemit operativ. Ajo çka nevojitet eshte nje thirrje nga 100K+100K. Ne qofte se programi eshte ngarkuar ne pjesen e dyte, ai do te mbahet si nje thirrje ne 200K+100 e keshtu me radhe. Kjo situate eshte e njobur si problemi i rivendosjes.

Nje zgjidhje e mundshme eshte te modifikohen instruksionet sa here qe nje program eshte ngarkuar ne memorje. Programi i ngarkuar ne pjesen e pare ka 100K plus adresen ku eshte vendosur, ata te vendosur ne pjesen e dyte kane 200K plus adresen e keshtu me radhe. Per te realizuar rivendosjen duke ngarkuar programe ne kete menyre, linkuesi duhet te prefshije ne programin e tij binar nje liste te bitmap i cili tregon se cilat fjale te programit jane adresa qe duhen perfshire ne rivendosje dhe cilat jane opcode-e, konstante apo fjale te tjera qe nuk kane nevoje per rivendosje.OS/MFT punon ne kete menyre.

Rivendosja gjate ngarkimit te programeve nuk e zgjidh problemin e mbrojtjes. Nje program dashakeqes mund te ndertoje nje instruksion te ri duke kercyer me pas tek ai. Edhe pse programet preferojne adresat absolute ne memorje sesa ato relative ne regjistra, nuk asgje qe ta pengoje nje te tille qe te ndertoje nje instruksion qe lexon ose shkruan nje fjale ne memorje. Ne nje sistem me shume perdorues, eshte shume e padeshirueshme qe te shkruhen apo te lexohen ne memorje gjera qe i perkasin perdoruesve te tjere.

Zgjidhja qe IBM gjeti per te mbrojtur 360 ishte ta ndante memorjen ne blloqe prej 2KB dhe te shenonte ne secilin prej tyre nje kod mbrojtjeje prej 4 bit. PSW (Program Status

Word) permbante nje celes prej 4 bit. Hardware-i 360 zbulonte cdo perpjekje te proceseve ne veprim qe aksesonte memorjen kodi i te cilit ndryshonte nga ai PSW. Duke qene se vetem sistemi operativ ka te drejte te ndryshoje kodet e mbrojtjes, proceset e perdoruesit jane parandaluar te interferojne me njeri-tjetrin dhe me sistemin operativ.

Nje zgjidhje alternative per te dyja problemet: rivendosjen dhe mbrojtjen eshte mobilizimi i makines me dy regjistra hardware-ik special, te quajtura regjistrat baze dhe limit. Kur procesi eshte skeduluar, regjistri baze ngarkohet me adresen e fillimit te pjeses se tij dhe regjistri limit ngarkohet ne gjatesine e pjeses se tij. Cdo adrese e memorjes qe gjenerohet automatikisht ka permbajtjen e regjistrat baze e cila eshte shtuar aty para se te dergohej ne memorje. Keshtu, ne se vlera e permbajtjes se regjistrat baze eshte 100K, nje instruksion thirrje 100 eshte kthyer efektivisht ne nje instruksion thirrjeje 100K+100, duke mos e modifikuar vete instruksionin. Adresat gjithashtu kontrollohen prej regjistrat limit per t'u siguruar qe nuk do te tentojne te adresojne memorjen jashte pjeses se percaktuar. Hardware-i i mbron keto regjistra nga modifikimi prej programeve te perdoruesit.

Disavantazhi i kesaj metode eshte shtimi dhe krahasimi ne çdo reference ne memorje. Krahasimet jane te shpejta, por shtimi eshte i ngadalte fale kohes qe kerkon mbartja. Vetem ne qofte se jane perdorur cikle speciale te shtimit.

Nje CDC-6600-superkompjuteri i pare ne bote- ka perdorur kete skeme. INTEL 8088 i perdorur per kompjuterat PC IMB, nje version me te dobet te kesaj skeme, regjistrat baze, por jo ato limit. Tashme po perdoret nga pak kompjutera.

4.2 SWAPING

Te organizosh memorjen ne nje sistem batch ne particione fikse, eshte e thjeshte dhe efektive. Çdo pune ngarkohet ne pjesen perkatese kur vjen ne krye te radhes. Ajo qendron ne memorje derisa te kete mbaruar veprimin e saj. Per sa kohe ka mjaftueshem pune qe qendrojne ne memorje dhe qe ta mbajne te zene CPU, nuk ka arsy qe te perdorim ndonje metode tjeter me te komplikuar. Ne sistemet me ndarje te kohes ose me PC qe jane te orientuar grafikisht, situata eshte me ndryshe. Ndonjehere nuk ka mjaftueshem memorje per t'i mbajtur ata ne nje gjendje aktive. Keshtu proceset ekstra vendosen ne disk per t'u sjelle me pas ne menyre dinamike.

Mund te perdoren dy perafrime te per gjithshme ne menaxhimin e memorjes, ne varesi te gatishmerise se hardware-it. Strategjia me e thjeshte, e quajtur swaping, konsiston ne sjelljen e dy proceseve ne teresine e tyre ne menyre qe te veprojne dhe me pas te kthehen serish ne disk. Strategjia tjeter, e quajtur memorje virtuale, i lejon programet te procedojne edhe kur ato jane pjeserisht ne memorjen kryesore. Me poshte do te studojme swaping, kurse ne seksionin 4.3 do te studojme memorjen virtuale.

Menyra e operimit e sistemit swaping eshte treguar ne fig.4-5. Fillimisht vetem procesi A eshte ne memorje. Me pas proceset B dhe C Jane krijuar ose "swaped" nga disku. Ne

fig.4-5(d) eshte procesi A qe shkon ne disk. Me pas vjen D dhe eshte B qe largohet. Se fundi eshte A qe rikthehet perseri ne memorje. Meqene se A eshte kthyer tashme ne nje pozicion tjeter, adresa qe e permban do te ripozicionohet. Kjo behet si nga ana e software-it kur programi merret nga disku, ashtu sikurse nga hardware-i kur ai ekzekutohet.

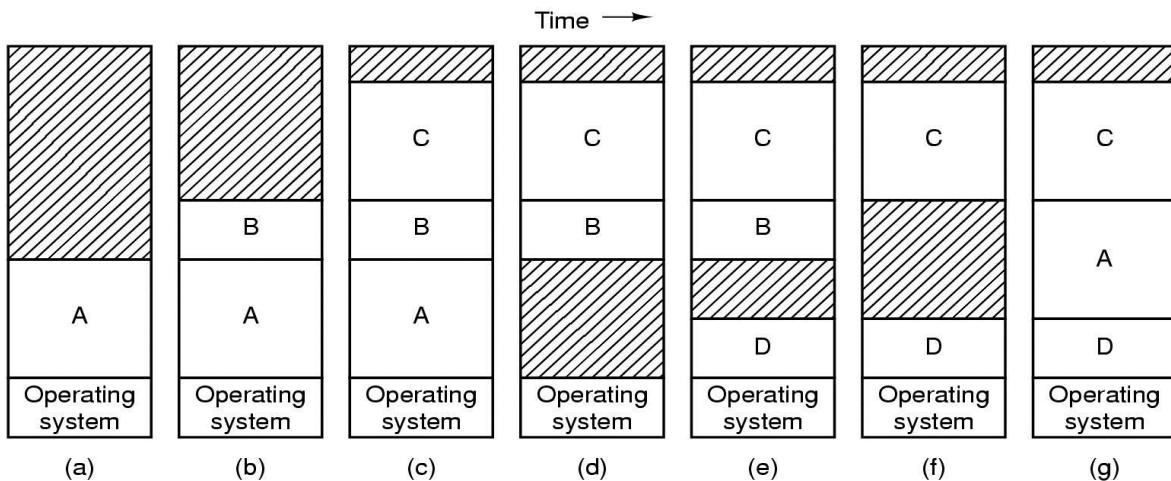


Figura 4-5. Alokimi i memorjes ndryshon kur proceset vendosen ne te dhe largohen prej saj. Pjeset e vijezuara tregojne memorjen e paperdorur.

Ndryshimi kryesor midis particioneve fikse ne fig.4-2 dhe atyre te ndryshueshme ne fig.4-5, eshte qe numri, pozicioni dhe madhesia e particioneve ndryshon ne menyre dinamike sa here qe proceset shkojne e vijne, nderkohe ne formen me particione fikse madhesia e tyre nuk ndryshon. Fleksibiliteti i te mos pershtaturit ne particione fikse te cilat mund te jene shume me te vogla apo shume me te medha sec duhet, permireson perdonimin e memorjes, por gjithashtu komplikon ngarkimin dhe shkarkimin ne memorje si dhe gjithashtu edhe regjistrimin ne te.

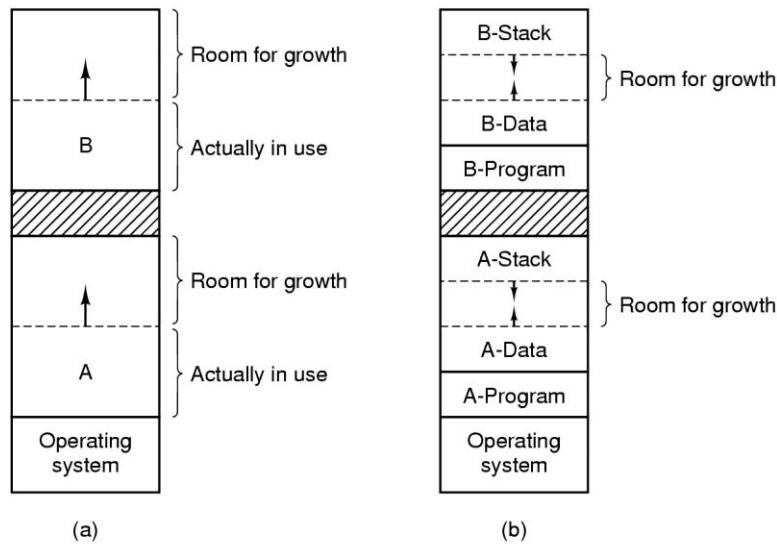
Kur swaping krijon shume holles ne memorje eshte e mundur qe t'i kombinosh ato ne nje me te madh duke i shtyre proceset sa me poshte qe te jete e mundur. Kjo teknike eshte e njojur si kompakteimi i memorjes(memory compaction). Kjo zakonisht nuk perdoret sepse kerkon nje kohe shume te madhe CPU. PER SHEMBULL: nje makine prej 256MB, qe kopjon 4B ne cdo 40 n/sek, i duhen 2.7 sek per kompakesuer te gjithe memorjen.

Nje çeshtje qe ia vlen te perqendrohet eshte se sa memorje duhet alokuar per proceset e krijuara ose qe largohen(bejne swaping). Ne se proceset jane krijuar ne nje madhesi fikse qe nuk ndryshon asnjeher, alokimi eshte i thjeshte: sistemi operativ alokon ekzaktesisht ate çka duhet, as me shume e as me pak.

Ne se, sidoqofte segmentet e te dhenave mund te zgjerohen, per shembull, duke e alokuar memorjen ne menyre dinamike nga nje heap, si ne çdo gjuhe programimi, problemi lind sa here qe procesi tenton te rritet. Ne qofte se gjendet nje hole fqinje me procesin, atehere

ajo mund te alokohet duke e lejuar procesin te zgjerohet pertej permasave te tij perigate hole-it. Nga ana tjeter, ne qofte se procesi eshte fqinje me nje proces tjeter, ky duhet te zhvendoset ne nje hole te madh mjaftueshem per te, ose proceset e tjera duhet te bejne swaping ne menyre qe te lirojne vend per te ne memorje. Ne qofte proceset nuk mund te zgjerohen ne memorje ose swaping nuk eshte i mundur pasi nuk ka vend mjaftueshem ne memorje, atehere proceset duhet te presin ose do “te vriten”.

Ne qofte se pritet qe proceset te zgjerohen gjate kohes qe ata veprojne, mund te jetë nje ide e mire qe te alokosh nje memorje te vogel ekstra sa here qe procesi eshte larguar (swaping) apo zhvendosur, per te reduktuar kohen shtese qe lidhen me zhvendosjen apo me proceset swaping qe ndodhin kur nuk ka pershatje ne alokimin ne memorje. Gjithsesi, kur swaping ndodh ne disk, ky proces ndodh vetem ne memorjen qe eshte perdonur aktualisht, pasi eshte e demshme per memorjen shtese. Ne fig.4-6(a) paraqitet nje konfigurim i memorjes ne te cilin hapesira shtese eshte alokuar nga dy procese.



Fjigra 4-6.(a) Alokimi i nje hapesire per nje segment shume te gjiat te dhenash.(b) Alokimi i nje hapesire per nje stack dhe nje segment te dhenash te ndryshueshem.

Ne qofte se nje procese kane dy segmente qe rriten, per shembull. nje segment te dhenash qe perdoret si heap per variablat qe alokohen dhe leshohen me pas nga memorja dhe nje segment stack per variablat normale lokale si dhe adresat qe kthehen, nje alternative kjo e sugjeruar nga fig.4-6(b). Ne kete figure ne shohim se çdo proces ka nje stack ne fillim te memorjes se alokuar, i cili rritet per poshte si dhe nje segment te dhenash fqinje me programin tekst, i cili rritet per nga siper. Memorja midis tyre mund te perdoret edhe per segment. Ne qofte se perfundojne se ekzekutuari, edhe proceset do te vendosen ne nje hole me nje hapesire te mjaftueshme, duke u larguar nga memorja derisa te krijohet nje hapesire e mjaftueshme ne memorje, ose ne te kundert do te vriten.

4.2.1 Menaxhimi i memorjes me *bitmap*

Kur memorja shenohet ne menyre dinamike, sistemi operativ duhet ta menaxhoje ate. Ne terma te per gjithshme, ka dy menyra per te regjistruar perdorimin e memorjes: *bitmap* dhe *listat e lira*. Ne kete seksion dhe ne tjetrin qe vijon do te shohim keto dy metoda.

Me ane te bitmap-it memorja eshte e ndare ne pjesa alokimi unike, ndoshta te vogla sa disa fjale ose sa disa kilobyte. Per cdo pjesa te alokuar ka nje bit ne bitmap, i cili eshte 0 ne qofte se ajo pjesa eshte e lire si dhe 1 ne se ajo eshte e zene. Figura 4-7 tregon pjesa te memorjes dhe bitmap-in korrespondues.

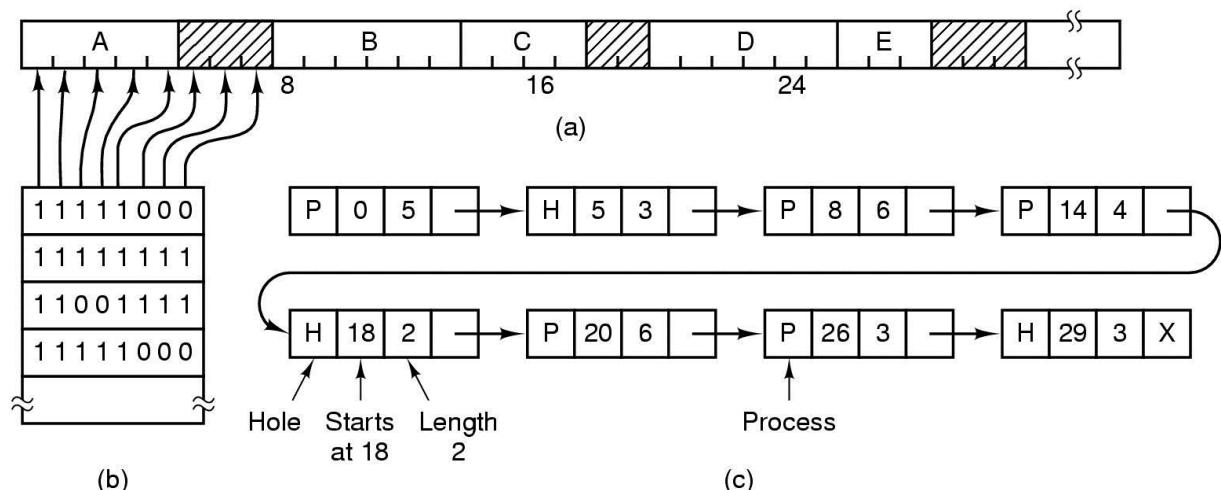


Figura 4-7.(a) Nje pjesa e memorjes me pese procese dhe tre hole. Shenjat tick tregojne pjesa te alokimit te memorjes. Pjeset e vijezuara(0 te bitmap-i) jane te lira.(b)Bitmap-I korrespondues.(c) Po i njejti informacion ne nje liste.

Madhesia e pjeses se alokuar eshte nje çeshtje shume e rendesishme e diznjimit. Sa me e vogel te jete pjesa e alokuar aq me i gjere eshte bitmap-i. Gjithsesi, edhe me nje pjesa te alokuar prej 4B, 32 bit memorje do te kerkonte vetem nje bit te map-it. Nje memorje prej $32n$ bit do te perdorte n bit te map-it, keshtu qe bitmap-i do te merrte persiper vetem 1/33 te memorjes. Ne qofte se pjesa e alokuar eshte zgjedhur e madhe, bitmap-i do te jete me i vogel, por do te çohej dem nje pjesa e rendesishme e memorjes ne pjesen e fundit te procesit kur madhesia e ketij te fundit nuk eshte nje shumefish i sakte i pjeses se alokuar.

Metoda bimap tregon nje menyre te thjeshte per te regjistruar fjale ne nje sasi fikse te memorjes, sepse madhesia e bitmap-it varet nga madhesia e memorjes dhe madhesia e pjeses se alokuar. Problemi kryesor ne kete rast eshte qe kur vendoset per te sjelle k pjesa procesi ne memorje, nje menaxhues i memorjes duhet te kerkoje per bitmap i cili duhet

te gjeje k bitet 0 qe vijojne ne map. Kerkimi per bitmap per nje proces te nje gjatesie te caktuar, eshte nje proces i ngadalte. Ky eshte nje argument kunder bitmap-it.

4.2.2 Menaxhimi memorjes me listat e linkuara

Nje tjeter metode per te mirembajtur memorjen eshte ajo e listave te linkuara, apo segmentet e alokuara apo te memorjes se lire, ku nje segment mund te jete nje proces apo edhe nje hapesire(hole) midis dy proceseve. Memorja e figures 4-7 eshte perfaquesuar me modelin e listave te linkuara ne fig.4-7(c). Çdo entry ne liste specifikon nje hole(H) ose nje process(P), adresen ku fillon, gjatesine si dhe nje pointer ne adresen ku ndodhet entry tjeter.

Ne shembullin me poshte lista e segmenteve gjenerohet prej adresave. Kjo menyre ka avantazhin qe kur nje proces perfundon ose largohet eshte shume i vlefshem update-im i listes. Nje proces qe perfundon, normalisht ka dy fqinje (me perjashtim te rasteve kur ndodhet ne krye apo ne fund te memorjes). Keto mund te jene ose procese ose hole, çka na çon ne kater kombinimet e fig.4-8. Ne fig.4-8(a), update-imi i listes kerkon zevendesimin e P nga nje H. Ne fig.4-8(b) dhe (c) dy entry Jane perfshire ne nje te vetme dhe lista behet me nje entry me te shkurter. Ne fig.4-8(d) tre entry Jane shkrire ne nje dhe dy grupe Jane larguar na lista. Meqene se tabela e proceseve qe slot per proceset qe perfundojn, ajo do te pointoje ne listen entry per vete procesin, eshte me e pershatshme ta kesh listen si nje liste dyshe te linkuar sesa nje liste te vetme si ne fig.4-8(c). Kjo strukture e ben me te lehte gjetjen e entry-t te ri si dhe mund te dallohet lethesiht ne se eshte e nevojshme nje shkrirje.

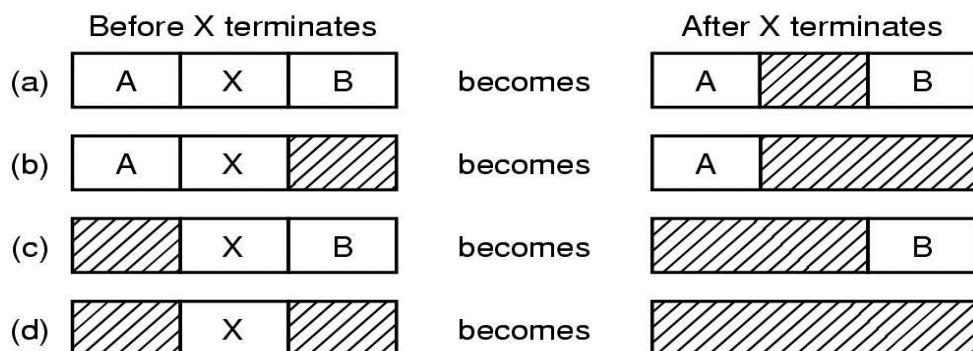


Figura 4-8. Kater kombinime fqinje per nje proces X qe perfundon.

Kur proceset dhe hole-t Jane vendosur ne nje liste qe gjenerohet prej adresave, mund te perdoren nje sere algoritmash per te alokuar memorjen gjate krijimit te nje procesi te ri(ose nje procesi te vjeter qe shkeputet nga disku). Nxjerrim perfundimin qe menaxheri i memorjes e di se sa memorje duhet alokuar. Algoritmi me i thjeshte eshte **first fit**. Menaxheri i memorjes skanon per gjate listes se segmenteve derisa te gjeje nje hole qe

eshte mjaftueshem i madh. Ky i fundit ndahet ne dy pjesa, nje per proceset dhe nje per memorjen e paperdorur, perveç rastit te nje pershatjeje ekzakte. **First fit** eshte nje algoritem shume i shpejte pasi kerkon sa me pak qe te jete e mundur.

Nje variant me i vogel i algoritmit me siper eshte **next fit**. Ai funksionon ne te njejtmenyre si **first fit** me perjashtim te faktit qe ai vezhgon derisa te gjeje nje hole te pershatatshem. Heren e ardhshme kur ai thirret per te per nje kontroll ai e nis atje ku e kishte lene, ne vend qe ta filloje nga e para, si ne rastin e algoritmit te pare. Simulimi nga Bays (1977) tregon qe algoritmi i dyte ka performance me te keqe se i pari.

Nje tjeter algoritem i mire eshte **best fit**. Ai kerkon neper te gjithe listen dhe gjen hole-in me te vogel te pershatatshem. Ne vend te te ndaje nje hole te madh qe mund te duhet me vone ai kerkon nje te tille qe te jete sa me i vogel dhe, gjithashtu i pershatatshem.

Si nje shembull te **first fit** dhe te **best fit** merrni ne konsiderate shembullin e fig.4-7. ne qoftese kerkohet nje bllok i nje madhesie 2, **first fit** do te alokonte hole-in 5 ndersa **best fit** hole-in 18.

Best fit eshte me i ngadalte se **first fit** pasi i duhet te kerkoje neper te gjithe listen sa here qe thirret. Ne nje fare menyre, çuditerisht rezulton te harxhoje me shume memorje se sa **first fit** apo **next fit** pasi tenton ta mbushe memorjen me hole te vegjel dhe te panevojshme. Ndersa **first fit** gjeneron hole te medha.

Ne qofte se duam te kemi te bajme me problemen e ndarjes ne pjesa ekzakte te pershatatshme, mund te perdorim algoritmin **worst fit**, i cili merr hole-in me te madh te mundshem dhe te gatshem ne menyre qe ta ndaje ate ne pjesa me te vogla, perseri te medha. Simulimi ka treguar qe as kjo nuk eshte ndonje ide e mire.

Te kater keto algoritme mund te pershpetojohen duke i ndare listat e proceseve nga ato te hole-ve. Ne kete menyre ata do te dedikonin te gjithe energjine e tyre ne kerkimin e hole-ve dhe jo te proceseve. Çmimi qe duhet te paguhet per kete rritje te shpejtesise gjate alokimit eshte kompleksiteti shtese si dhe ngadalesia gjate dealokimit te memorjes duke qene se nje segment i tere duhet te largohet nga lista e proceseve dhe te vendoset ne ate te hole-ve.

Ne qofte se mbahen lista te ndryshme per prçeset dhe hole-et, ajo e hole-ve duhet regulluar ne madhesi per ta bere me te shpejte algoritmin **best fit**. Kur ky algoritem kerkon nje hole ne liste, nga me i vogli tek me i madhi, ne çastin qe do te gjeje nje te pershatatshem, ai e di qe hole-i eshte me i vogli i gjetur i pershatatshem per punen. Nuk eshte me i nevojshem nje kerkim i metejshem, si ne rastin e nje liste te vetme. Me nje liste te hole-ve **best fit** dhe **first fit** jane njelloj te shpejte ndersa **next fit** eshte pointless.

Kur hole-t mbahen ne lista te veçanta nga proçeset, eshte i mundur nje optimizim i vogel dhe i mire. Ne vend qe te kemi nje strukture te veçante te dhenash per te mbajtur listen e hole-ve, si ne fig.4-7(c), mund te perdoren vete hole-t. Fjala e pare e secilit hole eshte madhesia e tij, ndersa e dyta do te jete nje pointer per hyrjen qe vijon. Nyjet e listes se fig.4-7(c), te cilat kerkojne tre fjale dhe nje bit (P/H) nuk nevojiten me.

Nje tjeter algoritmen alokimi eshte **quick fit**, i cili permban lista te vecanta per madhesite qe perdoren me shume. Per shembull.mund te kete nje tabele me n entry, ne te cilat e para eshte nje pointer tek koka e listes me hole prej 4 KB, e dyta nje pointer ne listen me hole prej 8KB dhe e treta nje pointer ne listen me hole prej 12 KB e keshtu me radhe. Hole-t per shembull, 21 KB mund te futen ne nje liste prej 20 KB, por edhe mund te futen ne nje liste te vecante me nje numer tek hole-sh. Me ane te ketij algoritmi eshte shume e shpejte te gjesh hole te nje madhesie te pershatshme, por ka ate avantazhin e njejte qe kane te gjitha skemat qe gjenerohen nga madhesia e hole-ve. Ne fjale te tjera, kur nje proces perfundon apo largohet, eshte e kushtueshme te kerkosh per fqinjin e tij ne se mund te behet ndonje shkrirje e mudshme. Por, ne se nuk behet nje shkrirje, memorja do te ndahej ne fragmente te vogla hole-sh, ne te cilat nuk do te pershtatej asnje proces.

4.3 Memorja virtuale

Shume vjet me pare, njerezit jane perballur me programe te cilat ishin shume te medha per memorjen e dhene. Zgjidhja qe eshte perdonur me shpesh eshte ndarja e programit ne pjesa te quajtura **nenshtresa**. Nenshtresa 0 do te fillonte te vepronte e para. Pasi te kishte perfunduar, ajo do te therriste nje tjeter nenshtrese. Disa sisteme nenshtresash kishin nje kompleksitet te larte, duke lejuar njekohesish nenshtresa shumefishe ne memorje. Nenshtresat mbaheshin ne disk dhe hynin e dilnin nga memorja me ane te sistemit operativ ,ne menyre dinamike ne se ishte e nevojshme.

Edhe pse puna e mesiperme behej nga sistemi, ndarje e progarmit ne pjesa duhej bere nga vete programuesi. Ndarja e programeve te medhenj ne pjesa me te vogla kerkonte shume kohe si dhe ishte shume e merzitshme. Nuk kaloi shume kohe derisa u gjet menyra per t'ia kaluar kete pune kompjuterit. Metoda qe u paraqit (Fotheringham, 1961) u be e njobur si **memorja virtuale**. Idea kryesore qe qendronte pas kesaj metode ishte fakti qe kombinimi i programit, i te dhenave dhe e stakut kerkon nje memorje shume te gjere .Per kete arsyte sistemi, pjeset e programit qe perdoren aktualisht, i mban ne memorjen kryesoren ndersa pjesen tjeter ne disk. Per shembull, nje program prej 16MB mund te veproje ne nje makine prej 4MB, mjafton te dime te zgjedhim pjeset e duhura (prej 4MB) te programit per memorjen kryesore, duke e lene pjesen tjeter per ne disk.

Memorja virtuale mund veproje edhe ne sisteme me multiprogramim, me bite dhe pjesa te shume programeve ne memorje njekohesish. Kur nje program eshte duke pritur per nje pjesa te tij te sillet ne memorje, ai pret per pajisjet I/O, ndaj nuk mund te veproje,keshtu qe CPU-ja i dorezohet nje programi tjeter, ashtu si ne cdo proces me multiprogramim.

4.3.1 Faqosja (Paging)

Shume sisteme te memorjes virtuale perdorin tekniken e quajtur **paging**, e cila do te pershkruhet tani. Ne shume kompjutera, ekziston nje set adresash ne memorje te prodhuara nga programet. Kur nje program perdor nje instruksion te tipit:

MOV REG,1000

kjo behet me qellim qe te kopjoje permbajtjen e memorjes me adrese 1000 tek regjistri REG (ose anaçjelltas, ne varesi te kompjuterit). Adresat mund te gjenerohen duke perdorur indeksimin, registrat baze, registrat segmente si dhe metoda te tjera.

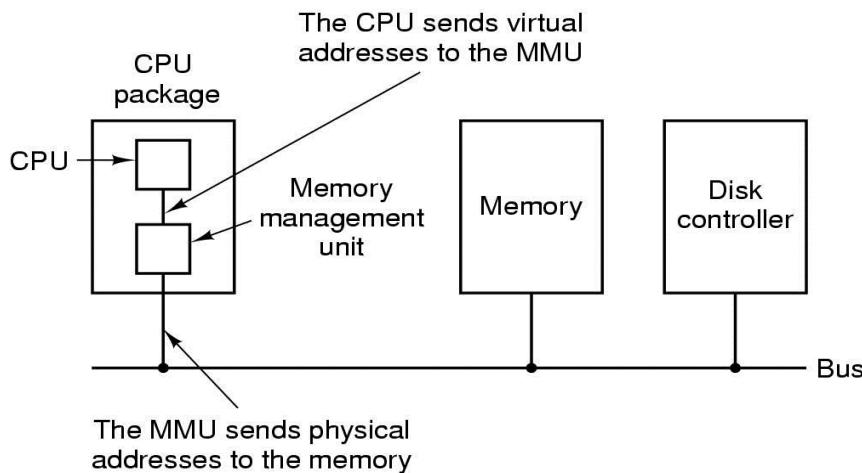


Figura 4-9. Pozicioni dhe funksionim i MMU-se. Ketu MMU-ja eshte shfaqur si te jete pjesa e qarkut te CPU-se, ashtu sic eshte ditet e sotme. Gjithsesi, mund te jete ne nje qark me vete si vite me pare.

Keto adresa te gjeneruara prej programeve jane quajtur adresa virtuale dhe formojne hapesiren e adresave virtuale. Ne kompjuterat pa memorje virtuale, adresa virtuale vendoset direkt ne bus-in e memorjes dhe ben qe fjala me te njejten adresë ne memorjen fizike te shkruhet apo te lexohet. Kur perdoret memorja virtuale, adresa virtuale nuk shkon direkt ne bus-in e memorjes. Ato ne fakt shkojnë ne MMU (Memory Management Unit) e cila harton adresat virtuale mbi memorjen fizike sic eshte ilustruar ne fig.4.9.

Ne fig.4-10 eshte treguar nje shembull shume i thjeshte se si funksionon kjo faqosje. Ne kete shembull kemi nje kompjuter qe gjeneron adresë 16-bit, nga 0 ne 64 KB. Keto janë adresë virtuale. Ky kompjuter ka vetem 32KB te memorjes fizike, keshtu, megjithese programet mund te shkruhen, ato nuk mund te ngarkohen plotesisht ne memorje dhe te

veprojne me pas. Nje pamje teresore e programit duhet, gjithsesi, te jete prezente ne disk, ne menyre qe pjese te tij te merren sa here te jete e nevojshme. Hapesira e adresave virtuale ndahet ne pjese me te vogla te quajtura pages (faqe). Pjeset koresponduese ne memorjen fizike quhen page frame (kampjone faqesh). Page-t dhe page frame-t kane gjithnje te njejten madhesi. Ne kete shembull ato jane 4 KB, por madhesia e faqeve nga 512 B ne 64 KB eshte perdonur ne sistemet reale. Me 64 K hapesire adresash virtuale dhe 32 K te adresave fizike, ne perftojme 16 faqe virtuale dhe 8 page frame. Transferimet midis RAM-it dhe diskut gjithmone behen ne njesi faqesh.

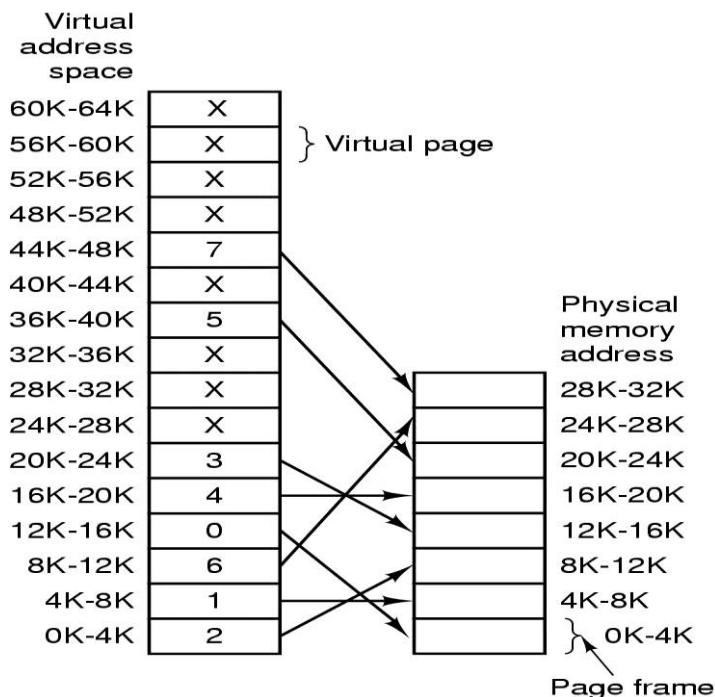


Figura 4-10. Lidhja midis adresave virtuale dhe atyre te memorjes fizike jepet nga tabela e faqeve.

Kur programi tenton te aksesoje adresen 0 duke perdonur instruksionin:

MOV REG, 0

Adresa virtuale 0 dergohet ne MMU. MMU-ja sheh ne se ka vendoşe te adreses virtuale ne faqen 0 (nga 0 ne 4095), e cila perputhet me harten e saj ne ne page frame 2 (nga 8192 ne 12287). Pra transformohet adresa ne 8192 dhe nxirret duke u vendoşur me pas ne bus. Memorja nuk di asgje ne lidhje me MMU dhe vetem sheh kerkesen per lexim apo shkrim te adreses 8192. Nderkohe MMU-ja ka hartuar te gjitha adresat virtuale nga 0 ne 4095 ne adresat fizike nga 8192 ne 12287.

Ne menyre te ngjashme nje instruksion i tille:

MOV REG, 8192

transformohet ne :

MOV REG, 24576

sepse adresa virtuale 8192 ndodhet ne faqen e dyte virtuale dhe kjo faqe i korrespondon page frame-it 6 (nga 24576 ne 28671). Si shembull i trete adresa virtuale 20500 ndodhet 20B nga fillimi i fakes virtuale 5(me adresa virtuale nga 20480 ne 24575) dhe korrespondon adreses fizike $12288+20=12308$.

Ne vetvete, kjo aftesi per t'i permblehdhur keto 16 faqe virtuale ne çdo njerен prej page frame-ve duke ndryshuar harten e MMU-se nuk e zgjidh problemin e qe perben fakti qe hapesira e adresave virtuale eshte me e gjere se memorja fizike. Duke qene se kemi vetem 8 page frame, vetem 8 nga faqet virtuale te fig.4-10 hartohen ne memorjen fizike. Te tjeret te shfaqur me kryq ne figure, nuk hartohen. Ne hardware-in aktual nje bit present/absent (i pranishem ose jo) tregon se cilat faqe jane prezente ne memorje.

Çfare ndodh ne se programi perdor nje faqe te pahartuar duke perdorur instruksionin:

MOV REG, 32780

i cili eshte biti i 12 midis fakes virtuale (duke filluar nga 32768)? MMU-ja e ve re qe faqa e 12 nuk eshte e hartuar (ne figure eshte e treguar me kryq) dhe ben qe CPU te kape sistemin operativ. Ky veprim quhet **page fault**. Sistemi operativ merr nje page frame pak te perdorur dhe e shkruan permajtjen e saj ne disk. Me pas merr faqen virtuale qe i korrspondon page framet te sapo marre, ndryshon harten dhe rinis instruksionin e kapjes.

Per shembull.ne qofte se sistemi operativ ka vendosur te evituar page frame-in e pare, ai do te ngarkoje faqen virtuale 8 ne adresen fizike 4K dhe ben dy ndryshime ne harten e MMU-se. Fillimisht do te shenoje sit e pa hartuar hyrjen e fakes se pare virtuale, ne menyre qe te kape çdo akses ne adresat virtuale midis 4K dhe 8K. Me pas do te mund te zevendesoje kryqin ne hyrjen e fakes virtuale 8 me nje 1, keshtu qe kur te ri ekzekutohet nje tjeter instruksion kapjeje, do te mund te vendose një lidhje midis adreses virtuale 32780 me adresen fizike 4180.

Tani le te shqyrtojme brendesine e MMU-se qe te shohim se si funksionon ajo dhe te tregojme se perse kemi zgjedhur te perdorim nje madhesi faqeje qe eshte fuqi e 2-it. Ne fig.4-11 ne shohim nje shembull te adreses virtuale 8196 (001000000000100 ne forme binare), e cila eshte hartuar duke perdorur harten e MMU-se te fig.4-10. Adresat 16-biteshe qe vijne, ndahan ne nje numer faqesh prej 4-bit dhe 12-bit offset. Me 4-bit per numrin e faqeve, ne kemi 16 faqe, dhe me 12-bit per offset-in, ne mund t'i adresojme te 4096 bytet ne nje faqe. Numri i faqeve eshte perdorur si nje indeks ne tabelen e faqeve (page table) duke i shtuar numrin e page frame-it qe i korrespondon asaj faqeje virtuale. Ne qofte qe biti present/absent eshte 0, behet nje kapje ne sistemin operativ. Ne qofte se biti eshte 1 numri i page frame-it i gjetur ne tabelen e faqeve ne rendin rrites 3-bit nga regjistrat e output-it se bashku me 12-bitet offset, te cilat kopjohen te pamodifikuara nga adresa virtuale. Se bashku ato formojne nje adrese fizike prej 15-bitesh. Regjistri output-it vendoset ne busin e memorjes ashtu si dhe adresa fizike e memorjes.

4.3.2 Tabela e faqeve (Page tables)

Ne rastin me te thjeshte, hartimi i adresave virtuale ne ato fizike eshte siç e pershkruam me pare. Adresat virtuale ndahen ne nje numer faqesh virtuale (ne bite sipas rendit rrites) dhe nje offset (me bite sipas rendit rrites). Per shembull. me nje adrese prej 16-bit dhe me nje madhesi te faqes prej 4 KB, kater bitet e mesiperme do te specifikojnë nje nga 16 faqet virtuale dhe 12 bitet e meposhteme do te specifikojnë offset-in ne byte(nga 0 ne 4095) neper faqen e selektuar. Gjithsesi nje ndarje ne 3, 5 apo ne cdo numer bitesh per faqe eshte gjithashtu i mundur. Ndarje te ndryshme çojne ne madhesi te ndryshme faqesh.

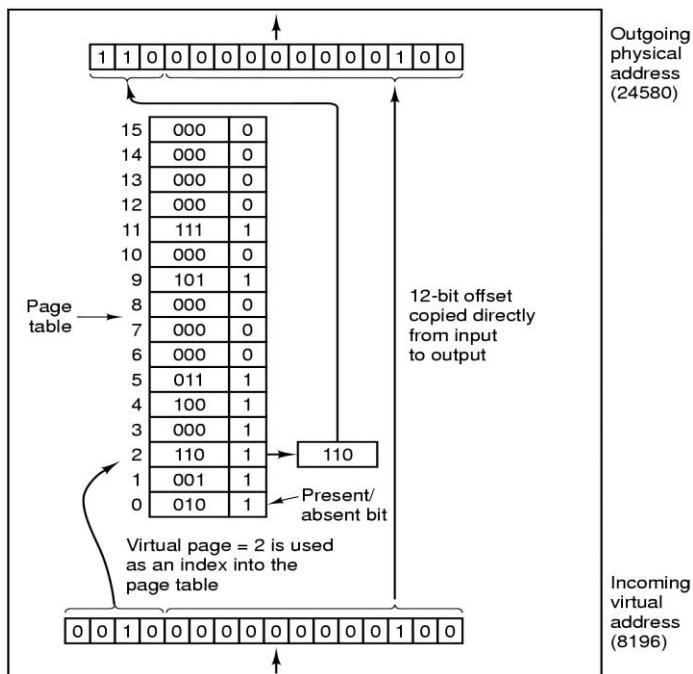


Figura 4-11. Operacioni i brendshem i MMU-se me 16 faqe 4-kB

Numri i faqve virtuale eshte perdorur si Indeks ne tabelen e faqeve per te gjetur entry-n per ate faqe virtuale. Prej entry-t ne tabelen e faqeve gjendet numri i kampjoneve te faqeve. Ky numer i bashkangjitet sipas rendit rrites numrit te offset-it, duke zevendesuar numrin e faqeve virtuale ne menyre qe te formoje nje adrese fizike e cila me pas dergohet ne memorje.

Qellimi i tabeles se faqeve eshte hartimi i faqeve virtuale ne kampjone faqesh. Matematikisht, tabela e faqeve eshte nje funksion me numrin e faqeve virtuale si argument dhe me rezultat numrin fizik te kampjoneve. Duke perdorur rezultatin e ketij funksioni, fusha e faqeve virtuale ne adresen virtuale mund te zevendesohet me nje fushe me kampione faqesh duke formuar nej adrese fizike ne memorje.

Përveç ketij pershkrimi te thjeshte, duhen perballur dy çeshtje te rendesishme.

1. Tabela e faqeve mund te jete ekstremisht e gjere.
2. Hartimi mund te jete shume i shpejte.

Pika e pare rrjedh nga fakti qe kompjuterat moderne perdonin adresa virtuale me se paku 32 bit. Le, le te themi, 4 KB madhesi faqeje, nje hapesire adresash me 32 bit ka 1 milion faqe dhe nje hapesire adresash 64-bit ka akoma me shume se 135 çmung te konceptojme ne. Me 1 milion faqe ne hapesiren e adresave virtuale, tabela e faqeve mund te kishte 1 milion entries. Kujtoni që çdo proces ka tabelen e tij te faqeve(pasi ka hapesiren e tij te adresave virtuale).

Pika e dyte eshte nje rrjedhoje e faktit qe hatrimi nga virtual ne fizik duhet bere ne cdo reference ne memorje. Nje instruksion tipik eshte ajo fjale dhe shpesh nje operand i memorjes. Si rrjedhoje jane te nevishme 1,2 apo me shume referencia ne tabelat e faqeve per çdo instruksion . Ne qofte se nje instruksion kerkon, te themi 4 n/sek, referencia ne tabelen e faqeve duhet bere se paku cdo 1 n/sec për te shmanjur pikën e vdekjes.

Nevoja per te pasur nje hartim te shpejte dhe te gjere eshte nje kufizim shume i rendesishem ne menyren se si ndertohen kompjuterat. Megjithate problemi eshte akoma me serioz me makinat e nivelit me te larte, eshte gjithashtu nje çeshtje jo shume e kushtueshme, kur kosto dhe çmimi/shpejtesi e performances jane kritike.

.Ne kete seksion dhe ne ata qe vijojne do te trajtojme ndertimin e tabelae te faqeve ne menyre me te detajuar si dhe nje numer zgjidhjesh hardware-ike te cilat jane perdonur ne kompjuterat e sotem.

Ndertimi me i thjeshte (se paku konceptual) eshte te kemi nje tabele te vetme faqesh qe konsistojne ne nje sasi regjistrash te shpejte hardware-ik, me nje entry per çdo faqe virtuale, te indeksuar me numrin e faqes virtuale, sic tregohet ne fig.4-11.Kur fillon nje proces, sistemi operativ ngarkon regjistrat me ane te tabelles se faqeve, i marre ky nga nje kopje e mbajtur ne memorje. Gjate ekzekutimit te procesit, nuk jane me te nevojshme referencat ne memorje per tabelen e faqeve. Avantazhet e kesaj metode jane qe kjo eshte e drejtperdrejt si dhe nuk kerkon referencia ne memorje gjate hartimit. Disavantazhi i saj eshte fakti i te qenurit potencialisht e kushtueshme (ne qofte se tabela e faqeve eshte e gjere). Ngarkimi i nje tabele te tere faqesh ne çdo konteks te sajin ndikon ne performance.

Po te kalojme ne ekstremin tjeter, tabela e faqeve mund te ngarkohet e gjitha ne memorjen kryesore. I gjithe hardware-ri, kerkon nje regjister te vetem qe pointon ne fillim te tabelles se faqeve. Ky model lejon qe harta e memorjes te jete e ndryshueshme duke ringarkuar vetem nje regjister. Sigurisht qe ekziston disavantazhin e te kerkuarit me shume se nje reference ne memorje per te lexuar çdo entry ne tabelen e faqeve gjate ekzekutimit te çdo instruksioni. Per kete arsy, ky perafrim perdoret rralle ne formen e tij te paster, por me poshte mund te shqyrtojme disa perafrime qe sjellin nje performance me te mire.

Tabelat e faqeve me shume nivele

Ne menyre qe te mund te perballen me problemin e magazinimit te shume tabelave te faqeve, shume kompjutera perdonin tabelat e faqeve me shume nivele. Nje shembull i thjeshte eshte treguar ne fig.4-12. Ne fig.4-12 (a) kemi te bezme me nje adrese virtuale prej 32-bitesh e cila eshte ndare ne 10-bit fushe PT1, 10-bit fushe PT2 si dhe 12-bit fushe offset-i. Duke qene se offset-i eshte 12-bit atehere faqet jane 4 KB, ndaj kemi nje total prej 2^{20} faqesh gjithesej.

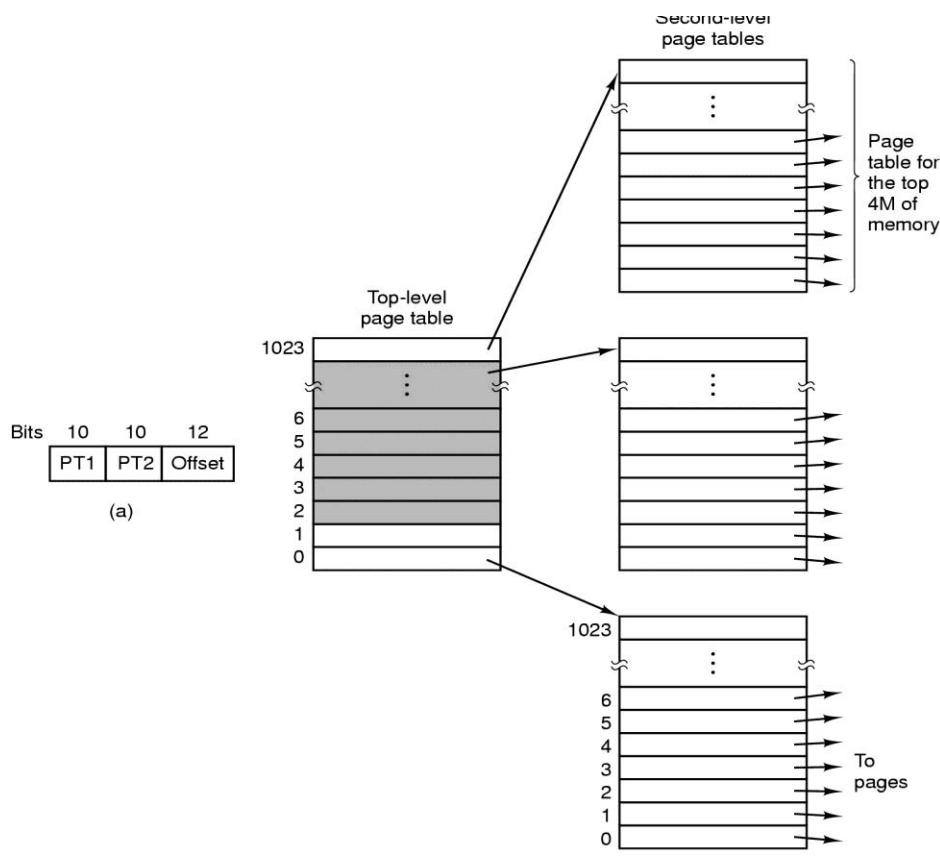


Figura 4-12.(a) Nje adrese 32-bit me dy fusha te tabeles se faqeve.(b)Tabela faqeve me dy nivele.

Sekreti i kesaj metode eshte shmangia e te mbajturit te te gjithe tabelave te faqeve ne memorje. Ne veçanti, ato qe nuk jane te nevojshme nuk duhen mbajtur aty. Supozojme,

qe nje procesi i nevojiten 12 MB dhe 4 MB me te siperm nevojiten per stack-un, ato qe vijojne per te dhenat dhe 4 te fundit per programin tekst. Midis pjeses se siperme te te dhenave dhe fundit te stack-ut ndodhet nje hole i madhe qe nuk perdoret.

Ne fig.4-12 (b) shohim se si nje tabele faqesh me dy nivele funksionon ne kete shembull. Ne te majte kemi nje tabele faqesh me nivel te siperm me 1024 entry, qe i korrespondojne fushes 10-bit PT1. Kur nje adrese virtuale prezantohet ne MMU, ajo nxjerr fillimisht fushen PT1 dhe e perdon fushen e saj si nje indeks ne tabelen e faqeve me nivel te siperm. Secila nga keto 1024 entry perfaqueson 4 M pasi e gjithe hapesira e adresave virtuale prej 4 G eshte copetuar ne pjesa prej 1024 bytes.

Entry e lokalizuar nga indeksimi ne tabelen e faqeve me nivel te siperm i bashkangjitet adreses apo numrit kampjon te faqes se nje tabele me nje nivel te dyte. Entry 0 i nje tabele me nivel te siperm pointon ne tabelen e faqeve per nje program tekst, entry 1 per te dhena ndersa ajo e 1024 per stack-un. Entry-t e tjera nuk perdoren. Fusha PT2 eshte tashme e perdonur si nje indeks ne tabelen e selektuar te faqeve me nje nivel te dyte ne menyre qe te gjeje numrin kampjon te asaj faqeje.

Si shembull, do te marrim ne konsiderate nje adrese virtuale prej 32-bitesh 0x00403004 (4,206,596 ne decimal), i cili eshte 12,292 bytes te dhena. Kesaj adrese virtuale i korrespondon PT1=1, PT2=2 dhe offset=4. MMU fillimisht perdon PT1 si nje indeks per tabelen me nje nivel te siperm dhe perfoton nje entry 1, e cila i korrespondon adresave nga 4M ne 8M. Me pas ajo perdon PT2 per te indeksuar tabelen me nivel te dyte te sapo gjetur dhe perfoton nje entry 3, e cila i korrespondon adresave nga 12288 ne 16383 midis atyre 4M **chunk** (me adresa absolute nga 4,206,592 ne 4,210,687). Kjo entry permban numrin kammpjon te faqes me adrese virtuale 0x00403004. Ne qofte se ajo faqe nuk ndodhet ne memorje, biti *present/absent* ne entry-n e abeles se faqeve do te behet zero duke shkaktuar nje mungese te faqes (page fault). Ne qofte se faqja do te ndodhet ne memorje numri i kampjonit te faqes i marre nga tabela e faqeve me nivel te dyte kombinohen me offset-in (4) duke ndertuar nje adrese fizike. Kjo adrese vendoset ne bus dhe dergohet ne memorje.

Ajo cka eshte per t'u shenuar nga fig.4-12 eshte fakti qe, edhe pse hapesira e adresave permban mbi 1 milion faqe, vetem 4 tabela faqesh nevijiten tamam: tabela me nivel te siperm, tabela me nivel te dyte nga 0 ne 4M, ajo nga 4 deri ne 8M dhe maksimumi me 4M. Bitet *present/absent* 1021 entry-t te tabelas me nivel te siperm vendosen ne 0, duke shkaktuar nje mungese faqej, sigurisht ne se ajo aksesohet ndonjehere. Me t'u shfaqur kjo, sistemi operativ do te vereje qe procesi po tenton t'i referohet memorjes, gje cka ai nuk eshte i autorizuar ta beje dhe sistemi do te ndermarre nje veprim te tille sic eshte dergimi i nje sinjali apo vrasja e procesit. Ne ketë shembull ne kemi zgjedhur nje sere numrash per madhesi te ndryshme dhe e kemi marre PT1 te barabarte e PT2 por ne jeten e perditshme jane te mundshme edhe vlera te tjera.

Tabelat e faqeve me dy nivele te fig.12 mund te zgjerohen deri ne nivele, kater apo me shume. Nivelet shtese sjellin me shume fleksibilitet, por eshte i dyshimte kompleksiteti shtese kur kalohet ne mbi tre nivele.

Struktura e nje entry ne nje tabelle faqesh

Le te kthehem i tani nga struktura e nje tabelle faqesh ne gjeresi, ne detajet e nje entry ne nje tabelle faqesh. Shtrirja ekzakte e nje entry eshet ne varesi te larte nga makina, por struktura e informacionit aty eshte praktikisht e njejte ne çdo makine. Ne fig.4.13 jepet nje shembull i nje entry ne tabelen e faqeve. Madhesia varet nga kompjuteri ne kompjuter, por 32-bit eshte nje madhesi normale. Fusha me e rendesishme eshte numri kampion i faqes (*page frame number*). Ne fund te fundit qellimi i hartimit eshte lokalizimi i ketij numri. Prane tij kemi bitin *present/absent*. Ne qofte se ky bit eshte 1, entry eshte e vlefshme dhe mund te perdoret. Ne qofte se eshte 0, faqja virtuale se ciles i perket entry nuk eshte aktualisht ne memorje. Te aksesosh nje entry ne nje tabelle faqesh ku ky bit eshte 0 do te shkaktoje nje *page fault*.

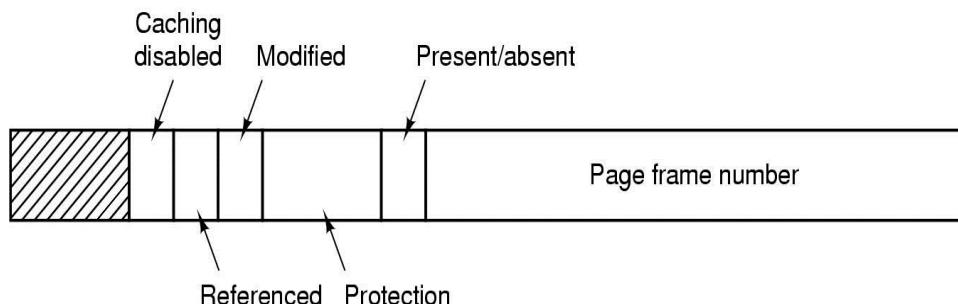


Figura 4-13. Nje entry tipike ne nje tabelle faqesh.

Biti i *mbrojtjes* tregon se çfare lloj aksesi eshte i lejuar. Ne rastin me te thjeshte, kjo fushe permban nje bit, me 0 ne se kemi te bezme me operacion shkrim/lexim dhe 1 ne se kemi te bezme vetem me operacion lexim. Nje rregullim me i sofistikuar eshte te kesh e besh me tre bite, dy te parat per aftesim te shkrimit dhe leximit dhe fundit per ekzekutim te faqes. Bitet e *modifikuar* dhe te *referuar* regjistrojnë çdo perdonim te faqes. Kur eshte shkruar mbi nje faqe, hardware-i autmatikisht ndryshon bitin e *modifikuar*. Ky bit ka vlore kur sistemi operativ vendos te kerkoje nje kampion faqeje. Kur faqja nuk eshte e modifikuar (eshte e piset), ajo do te rishkruhet perseri ne disk. Ne se nuk eshte e modifikuar (eshte e paster), thjesht mund te braktiset perderisa kopja ne disk eshte e vlefshme. Biti zakonisht eshte quajtur *dirty bit*, pasi tregon gjendjen e faqes.

Biti i *referuar* ndryshon sa here ne i referohemi nje faqeje, per shkrim apo per lexim. Vlera e tij e ndihmon sistemin operativ te shmange ndonje page fault te mundshem. Faqet qe nuk jane perdonur ndonjehere jane kandidate me te mire se ato qe jane perdonur me pare. Ky bit luan nje rol te rendesishem ne algoritmat e zevendesimit te faqeve, te cilat do te studiohen me vone ne kete kapitull.

Me ne fund, biti i fundit lejon qe fshehja te jetë disa bite per faqen.

Kjo forme eshte e rendesishme per ato faqe qe hartohen ne pajisjet regjistra me shume se ne memorje. Ne se sistemi operativ perfshihet ne nje cikel duke pritur per pajisje I/O qe t'i per gjigjen komandes se sa po dhene, eshte e rendesishme qe hardware-i te marre fjale nga pajisja dhe jo te perdore nje kopje te vjeter. Me ane te ketij biti, [caching](#) nuk aftesohet me. Makinat qe kane hapesira I/O te ndryshme dhe nuk perdonin hartimin I/O te memorjes kete bit nuk e kane te nevojshem.

Mbaj mend qe adresa e diskut e perdonur per ta mbajtur kete faqe kur ajo nuk gjendet ne memorje nuk eshte pjese e tabelles se faqeve. Ky arsyetim eshte i thjeshte. Tabela e faqeve mban vetem informacionin qe i nevojitet hardware-it per te perkthyer adresat virtuale ne adresa fizike. Informacioni qe i nevojitet sistemit perativ per te perballuar page fault-et gjendet ne tabelat software ne brendesi te sistemit operativ. Hardware-i nuk ka nevoje per te.

4.3.3 TLB s

Ne shume skema te hartimit, tabelat e faqeve mbahen ne memorje, fale permasave te medha te tyre. Potencialisht, ky ndertim ka nje ndikim te madh ne performance. Merrni ne konsiderate nje instruksion qe kopjon nje regjister tek nje tjeter. Ne mungese te hartimit, ky instruksion i referohet vetem memorjes per te marre nje tjeter instruksion. Me ane te hartimit, jane te nevojshme referime te tjera ne memorje per te aksesuar tabelen e faqeve. Meqene se shpejtesia e ekzekutimit varet nga shpejtesi me te cilin CPU kap instruksionet dhe te dhenat nga memorja, te besh nga dy referime ne tabelat e faqeve per memorje e redukton performancen me 2/3. Ne kushte te tilla, askush nuk do ta perdorte kete metode.

Dizenjuesit e kompjuterave ishin ne dijeni te ketij problemi dhe gjeten nje zgjidhje. Zgjidhja u bazua ne fatin qe shume programe tentonin te benin nje numer te madh referimesh ne nje numer te madh faqesh te vogla. Keshtu vetem nje numer i vogel tabelash faqesh lexoheshin dhe pjesa tjeter thjesht mbetej jashte loje.

Zgjidhja ishte vendosja ne kompjutera i nje pajisjeje te vogel hardware-ike ne menyre qe hartimi i adresave virtuale ne ato fizike te behej aty, pa pasur nevoje per aksesim ne tabela te faqeve. Pajisja e quajtur TLB (Table Lookaside Buffer) ose, si zakonisht *memorje shoqeruese*, eshte ilustruar ne fig.4.14. Ajo gjendet zakonisht ne brendesi te MMU-se dhe konsiston ne nje numer te vogel entry-sh, tete ne kete shembull, por rralle me shume se 64. Cdo entry permban informacion mbi faqen duke perfshire ketu edhe numrin e faqes virtuale, nje bit qe ndryshon kur faqja modifikohet, kodin e mbrojtjes (leje mbi leximin, shkrimin dhe ekzekutimin) dhe numrin kampion te faqes fizike ne te cilin kjo faqe eshte lokalizuar. Nje bit tjeter tregon nese nje entry eshte e vlefshme (ne perdonim) apo jo.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 4-14.Nje TLB pe te pershpejtuar faqosjen

Nje shembull qe gjeneron nje TLB si ne fig.4-14 eshte nje proces qe gjendet ne nje cikel qe pershon faqet virtuale 19, 20 dhe 21, keshtu keto TLB entry kane kode mbrojtjeje per lexim dhe ekzekutim. Te dhenat qe perdoren ne ate kode ndodhen ne faqet 129 dhe 130. Faqja 140 permban indekset e perdorura ne llogaritjet e te dhenave. Se fundi, stack-u ndodhet ne faqet 860 dhe 861.

Le te shohim tani se si funksionon TLB. Kur ne MMU prezantohet nje adrese virtuale per perkthim, hardware-i kontrollon per te pare ne se numri i faqes se saj virtuale ndodhet ne TLB duke e krahasuar ate me te gjitha entry-t njekohesisht (ne parallel). Ne se ka perputhje dhe aksesimi nuk i dhunon bitet e mbrojtjes, kampioni i faqes merret direkt nga TLB, pa qene e nevojshme te shkojme tek tabela e faqeve. Ne se faqja virtuale ndodhet ne TLB por instruksioni kerkon te shkruaje mbi nje faqe vetem te lexueshme, atehere kemi nje dhurim te bitit te mbrojtjes dhe nuk lejohet aksesi, njelloj si te tabela e faqeve.

Rast interesant eshte kur numri i faqes virtuale nuk gjendet tek TLB. MMU-ja e zbulon nje gje te tille dhe ben nje kontroll te zakonshem ne tabelen e faqeve. Ajo perjashton nje entry ne TLB dhe e zevendeson ate me nje entry ne tabelen e faqeve te sapo zbuluar. Keshtu, ne se ajo faqe eshte perdorur me pare, heren e dyte ajo rezulton si nje goditje ne memorje dhe jo si nje mungese. Kur nje entry eshte purged nga TLB, biti i modifikuar kopjohet ne entry-n e tableles se faqeve ne memorje. Vlerat e tjera jane, sakaq atje. Kur TLB ngarkohet nga tabela e faqeve, te gjitha fushat merren nga memorja.

Menaxhimi software-ik imemorjes

Deri tani, kemi supozuar qe memorjet me faqe virtuale kane tabela faqesh te njohura nga hardware-i, plus nje TLB. Me kete model, menaxhimi I TLB dhe perballja me gabimet ne te do te merreshin persiper nga hardware-i i MMU-se. Nderhyrja ne software ndodh vetem kur kemi nje mungese faqeje ne memorje.

Ne te kaluaren, nje supozim i tille ishte i vertete. Gjithsesi, shume makina moderne te RISC duke perfshire edhe SPARC, MIPS, ALPHA dhe HPPA bejne te gjitha perafresisht nje menaxhim software-ik. Ne makina te tilla entry-t ne TLB behen vetem nga sistemi operativ. Kur ndodh nje mungese ne TLB, ne vend qe te ngarkohet MMU-ja me detyre e gjetjes se faqes perkate se memorje, tashme gjenerohet nje sinjal (TLB fault) dhe problemi i kalohet sistemit operativ. Sistemi duhet te gjeje faqen, te zhvendose nje entry nga TLB, ta zevendesoje me nje tjeter dhe te rinise instruksionin qe deshtoi me pare. Sigurisht, te gjitha keto duhet te behen ne menyre manuale pasi sinjalet e gabimit ne TLB ndodhin me shpesh sesa ne tabelat e faqeve.

Çuditerisht, ne se TLB eshte mjafueshem e gjere (64 entry le te themi) për të reduktuar mungesat, menaxhimi i software-it te TLB rezulton te jete eficent. Perfitimi me i madh ketu është nje MMU me e thjeshte e cila liron nje hapesire te konsiderueshme ne qarkun e CPU-se per cache apo per forma te tjera qe kerkojne performance. Menaxhimi software-ik i TLB eshte diskutuar nga Uhlig et al. (1994).

Jane zhvilluar strategji te ndryshme per te permiresuar performancen ne makina qe aplkojne menaxhimin software-ik te TLB. Nje prej tyre tenton te zvogeloje mungesat ne TLB si dhe pasojat qe shkakton kjo mungese (Bala et al. 1994). Per të redukruar mungesat, ndonjehere sistemi operativ perdor intuiten e tij per te gjetur se cila do te jete faqja qe do te perdoret ne vijim dhe per te ngarkuar paraprakisht entry-t nga tabela e faqeve ne TLB. Per shembull, kur nje proces klient i dergon nje mesazh procesit server ne te njejtën makine, kerkohet qe serveri te veproje menjehere. Duke e ditur kete, nderkohe qe kryhet procesi i dergimit, sistemi mund te kontrolloje ku ndodhen kodi i serverit, te dhenat dhe faqet e stack-ut duke i hartuar ato perpara se te ndodhe ndonje gabim ne TLB.

Nje menyre normale per te vepruar me nje mungese ne TLB, si ne hardware ashtu edhe ne software, eshte te shkosh ne tabelen e faqeve dhe te kryesh nje operacion indeksimi per te lokalizuar faqen qe mungon. Problemi ne nje kerkim e tille ne software eshte se faqja qe mban tabelen e faqeve mund te mos ndodhet ne TLB duke shkaktuar nje tjeter gabim ne TLB gjate procesit keto gabime mund te reduktohen duke mbajtur nje cache software ne entry-t e TLB ne nje vendodhje fiks, faqja e se ciles ndodhet gjithnjë ne TLB. Duke kerkuar me pare ne kete cache, sistemi operativ mund te reduktoje mungesat ne TLB.

4.3.4 Tabelat e invertuara te faqeve

Tabelat tradicionale te faqeve te pershkruara deri tani kerkojne nje entry per çdo faqe virtuale, duke qene se jane indeksuar nga numri i faqes virtuale. Ne se hapesira e adresave konsiston ne 2^{32} bytes, me 4096 byte per faqe, atehere nevojiten mbi 1 milion entry te tabelave te faqeve. Duke marre minimumin, tebela e faqeve duhet te kishte te pakten 4MB. Ne sisteme me te gjera, kjo shifer eshte ndoshta e mundshme.

Gjithsesi, duke qene se kompjuterat 64 bit-ësh po bëhen më te zakonshem, situata ndryshon ne menyre drastike. Ne se hapesira e adresave tashme eshte 2^{64} byte, me faqe prej 4KB, do te na duhej një tabele faqesh me 2^{52} entry. Ne se cdo entry eshte 8 byte, tabela do te kishte mbi 30million GB. Bllokimi i 30 milion GB vetem per page table nuk mund te behet, as tani dhe as per shume vite te tjera.

Si rrjedhoje do ta na duhet një zgjidhje tjeter per hapesirat e faqeve virtuale prej 64bit.

Nje zgjidhje e tille do te ishte *tabela e invertuar e faqeve*. Ne nje model te tille kemi nje entry per cdo kampion faqeje ne memorjen reale, ne karahasim me rastin kur kemi nje entry per cdo faqe te hapesires se adresave virtuale. Per shembull, me 64 bit adresa virtuale, një faqe prej 4 KB dhe me nje RAM prej 256 MB, një tabele faqesh e invertuar kerkon vetem 65.536 entry. Entry regjistron se cili proces apo faqe virtuale eshte lokalizar ne kampionin e fakes. Edhe pse tabelat e invertuara te faqeve shmangin shperdorim te madh te memorjes, te pakten, kur hapesira e adreses virtuale eshte me e gjere se memorja fizike, ata kane një ane te erret: perkthimi virtual-fizik behet shume me i veshtire. Kur procesi n i referohet fakes virtuale p , hardware-i nuk mund ta gjeje me faqen fizike vetem duke perdorur p si indeks ne tabelen e faqeve. Nga ana tjeter, ai mund te kerkonte ne gjithe tabelen e invertuar te faqeve per nje entry (n,p) . Akoma me tej, ky kerkim mund behet gjate çdo referimi ne memorje, jo vetem gjate një page fault. Duke kerkuar një tabele me 64 K ne çdo referim ne memorje nuk eshte një menyre per ta bere te shpejte makinen.

Zgjidhja e kesaj dileme eshte perdorimi i TLB-se. Ne se TLB mund te mbaje te gjitha faqet e renda te perdorura, perkthimi mund te ndodhe aq shpejte sa me tabela normale faqesh. Ne nje mungese tek TLB, gjithsesi, tabela e invertuar e faqeve duhet te kerkohet ne software. Nje menyre e mire per te realizuar kete kerkim eshte te kesh tabela hash te ndodhura ne adresat virtuale. Te gjitha faqet virtuale te ndodhura aktualisht ne memorje qe kane te njejtën vlerë hash janë te vendosura zinxhir se bashku, si ne fig.4-15. Ne se tabela hash ka aq slote sa një makine ka faqe fizike, zinxhiri do te ishte i gjate sa një entry, duke pershpejtuar me se miri hartimin. Me t'u gjetur numri kampjon i fakes, një pale e re (virtuale, fizike), do te gjendet ne TLB.

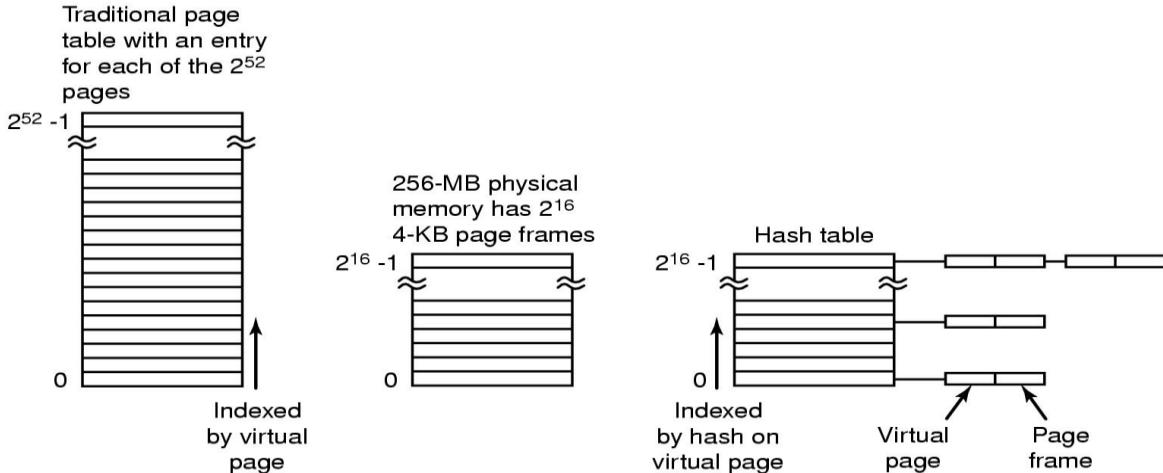


Figura 4-15. Krahasimi i tabelave klasike te faqeve me tabelat e invertuara te faqeve.

Tabelat e invertuara te faqeve perdoren ne disa IBM si dhe ne disa workstation Hewlett-Packard dhe do te behen akoma me te zakonshme ashtu sikurse po behen makinat 64-biteshe. Perafrime te tjera qe perballojne memorje te gjera virtuale mund te gjenden ne (Huck & Hays, 1993; Talluri & Hill, 1994 dhe Talluri et al., 1995).

4.4 ALGORTMAT E ZËVENDËSIMIT TË FAQEVE

Kur ndodh një page fault, sistemi operativ duhet të fshijë një faqe nga memorja për të lënë vend për faqen tjetër që duhet të çojë atje. Në qoftë se faqja që do të fshihet është modifikuar gjatë kohës që ka qenë në memorje duhet të rishkruhet në disk në menyrë që disku të azhornohet. Por në qoftë se ajo nuk është ndryshuar (per shembull, përmban një program text), disku është i azhornuar, kështu që rishkrimi nuk është i nevojshëm. Faqja që do të lexohet në memorje thjesht do të shkruhet mbi faqen që do të fshihet.

Edhe pse do të ishte e mundur të zgjidhej rastësisht një faqe për t'u fshirë gjatë çdo page fault-i, performanca e sistemit do të ishte më e mirë nëse do te hiqej një faqe që nuk përdoret shpesh. Në se fshihet një faqe që përdoret shpesh, ajo shpejt do të duhej të sillej në memorje, duke rezultuar në extra overhead. Është bërë shumë punë teorike dhe eksperimentale mbi algoritmat e zëvendësimit të faqeve. Më poshtë do të përshkruajmë disa nga algoritmat më të rëndësishëm.

Duhet thënë se problemi i “zëvendësimit të faqeve” shfaqet edhe në pjesë të tjera të ndërtimit të kompjuterit gjithashtu. Per shembull shumë kompjutera kanë një ose më shumë memorje cache të përbëra nga blloqe memorjeje 32-byte dhe 64-byte te perdorur se fundmi. Kur cache-ja është plot, një bllok duhet të fshihet. Ky problem është i njëjtë me atë të zëvendësimit të faqeve, me përjashtim të intervalit kohor më të shkurtër (kjo gjë duhet të bëhet në disa ns dhe jo në disa ms si në rastin e faqeve). Arsyeja e intervalit më

të shkurër kohor është se miss cache-të e blloqeve plotësohen nga memorja kryesore, që nuk ka kohë kërkimi dhe asnje vonesë racionale.

Shembulli i dytë është ai i një web serveri. Serveri mund të përmbajë një numër të caktuar web faqesh shpesh të përdorura në cache-në e tij. Por kur cache-ja është plot dhe ne i referohemi një faqeje të re, duhet të fshihet një web faqe. Logjika është e njëjtë me atë të faqeve të memorjes virtuale, me përjashtim të faktit që faqet web nuk modifikohen në cache, prandaj ka gjithmonë një kopje tjeter në disk. Në sistemin e memorjes virtuale, faqet në memorjen kryesore mund të janë clean ose dirty.

4.4.1 ALGORITMI OPTIMAL

Algoritmi më i mirë i mundshëm është i thjeshtë për t'u përshkruar, por i pamundur për t'u implementuar. Ai funksionon kështu: në momentin që ndodh një page fault, një grup faqesh është në memorje. Ne do të adresojmë një tërë nga këto faqe në instruksionin pasardhës (faqen që përbën instruksionin). Faqet e të tjera mund të mos i adresojmë deri pas 10, 100 apo i(x) 0 instruksione më pas. Çdo faqe mund të etiketohet me numrin e instruksioneve që do të ekzekutohen para se t'i referohemi asaj faqeje.

Sipas algoritmit optimal duhet fshirë faqja me etiketë më të madhe. Nëse një faqe nuk do të përdoret për 7 milion instruksione dhe një tjetër nuk përdoret për 6 milion instruksione, duke fshirë të parën e shtyn page fault sa më larg në të ardhmen të jetë e mundur. Kompjuterat, ashtu si njerëzit, përpilen t'i shtyjnë ngjarjet e pakëndëshme sa më shumë kohë të munden.

Problemi i vetëm i këtij algoritmi është se është i parealizueshëm. Në momentin e page fault, sistemi operativ nuk ka mundësi të dijë kur do t'i referohemi në vazhdim secilës faqe (Pamë të njëtin situatë tek algoritmi shortest job first scheduling – Si mund ta dijë sistemi cili proces është më i shkurtri?). Gjithsesi, duke ekzekutuar një program në një simulator dhe duke ruajtur gjurmët e të gjitha adresave të faqeve, është e mundur të implementohet zëvendësimi optimal i faqeve në ekzekutimin e dytë, duke përdorur informacionin mbi gjurmët e adresave të faqeve të marrë gjatë ekzekutimit të parë.

Në këtë mënyrë është e mundur të krahasohet performanca e algoritmitave të realizueshëm me performancën më të mirë të mundshme. Nëse një sistem operativ arrin një performancë prej, të themi 1% më pak se algoritmi optimal, mundi i harxhuar për kërkimin e një algoritmi më të mirë do të japë të shumtën 1% fitim.

Për të shprehur çdo konfuzion të mundshëm, duhet të bëhet e qartë se ky aksesim i adresave të faqeve i referohet vetëm një programi apo të matur dhe me një input specifik. Algoritmi që derivon prej këtej është kështu specifik për këtë program dhe të dhënët hyrëse. Edhe pse kjo metodë është e përdorshme për vlerësimin e algoritmit, nuk përdoret në sistemet praktike.

Më poshtë do të studiojmë algoritmat e përdorshëm në sistemet reale.

4.4.2 ALGORITMI NOT RECENTLY USED

Në mënyrë që të lejojnë sistemin operativ të mbledhë statistikat e nevojshme se cilat faqe po përdoren dhe cilat jo, shumë kompjutera me memorje virtuale kanë 2 bite gjendjeje që i asociohet çdo faqeje. R vendoset kur faqja adresohet (shkruhet ose lexohet), M vendoset kur faqja shkruhet (pra modifikohet). Bitet ndodhen në çdo page table entry, sic tregohet në Fig 4-13. Është e rëndësishme të kuptohet se këto bite mund të azhornohen (update) në çdo aksesim memorjeje, prandaj është esenciale të vendosen nga hardware. Pasi biti setohet 1, qëndron kështu derisa sistemi operativ e reseton 0 në software.

Nëse hardware nuk i ka këto bite, ato mund të simulohen si më poshtë. Kur një proces fillon, të gjitha page table entrie te tij shënohen sikur nuk janë në memorje. Sapo një faqe adresohet do të ndodhë një page fault. Atëherë sistemi operativ seton bitin R (në tabelat e tij të brendshme), ndryshon page table entry të pointojë te faqja që duhet me mënyrën READ ONLY dhe restarton instruksionin. Nëse faqja më pas do të shkruhet atëherë do të ndodhë një tjetër page fault, duke e lejuar sistemin operativ të ndryshojë mënyrën e faqes në READ/WRITE.

Bitet R dhe M mund të përdoren për të ndërtuar një algoritëm të thjeshtë si më poshtë:

Kur procesi fillon, të dy bitet e faqeve për të gjithë faqet e tij zeroohen nga sistemi operativ. Periodikisht (pra për cdo interrupt clock-u) biti R zerohet për të dalluar faqet që nuk janë adresuar nga ato që janë adresuar.

Kur ndodh një page fault, sistemi operativ kontrollon të gjitha faqet dhe i ndan ato në 4 kategori, bazuar në vlerat aktuale të biteve R dhe M:

Kategoria 0 → jo e adresuar, jo e modifikuar

Kategoria 1 → jo e adresuar, e modifikuar

Kategoria 2 → e adresuar, jo e modifikuar

Kategoria 3 → e adresuar, e modifikuar

Edhe pse kategoria e parë duket, në pamje të parë e pamundur, kjo ndodh kur një faqe e kategorisë 3 e ka bitin R të resetuar nga një sinjal clock-u. Këto sinjale nuk resetojnë bitin M, sepse ky informacion nevojitet për të ditur nëse faqja është rishkuar në disk apo jo. Duke resetuar R, por jo M, ato çojnë në faqet e kategorisë 1.

Algoritmi NRU fshin rastësisht një faqe nga klasa jobosh me numër më të vogël. E mira e ketij algoritmi është se është më mirë të fshihet një faqe modifikuar që nuk është adresuar në të paktën një impuls clock-u (zakonisht 20ms), se të fshihet një faqe që është në përdorim. Avantazhi kryesor i NRU është se është i lehtë për tu kuptuar, mesatarisht eficent për tu implementuar dhe jep një performancë që, edhe pse jo shumë optimale, mund të jetë e mjaftueshme.

4.4.3 ALGORITMI FIRST-IN, FIRST-OUT (FIFO)

Një tjetër algoritëm low-overhead është algoritmi FIFO. Për të kuptuar se si funksionon, mendoni një supermarket që ka mjaft rafte për të reklamuar ekzaktsisht k produkte të ndryshme. Një ditë një kompani prezanton një ushqim të çastit të ri me leverdi, të ngrirë, kos organik që mund të rigjenerohet në një furrë me mikrovalë. Është një sukses i menjehershëm, kështu supermarketi ynë i kufizuar duhet të heqë një produkt tjetër në mënyrë që të fusë këtë produkt të ri.

Një mundësi është që të gjendet një produkt që supermarketi e ka mbajtur për më gjatë (per shembull diçka që ka nisur të shitet 20 vjet më parë) dhe për të cilin askush nuk interesohet më. Në fakt, supermarketi mban një listë të lidhur të gjithë produkteve që ai shet, në radhën që ato janë futur. Produkti i ri shkon në fund të listës, ai i pari fshihet prej listës.

Në algoritëm zbatohet e njëjtë ide. Sistemi operativ mban një listë të gjitha faqeve që ndodhen në memorje, me faqen më të vjetër në fillim të listës dhe me më të renë e ardhur në fund të saj. Në një page fault faqja në fillim fshihet dhe faqja e re shtohet në fund të listës. Kur aplikohet në magazinë, FIFO mund të fshijë parafine, por gjithashtu mund të fshijë miell, kripë ose gjalpjë. Kur aplikohet në kompjutera shfaqet i njëjtë problem. Per këtë arsy, FIFO përdoret rallë në formën e tij të pastër.

4.4.4 ALGORITMI SECOND CHANCE

Një modifikim i thjeshtë tek FIFO që shhang problemin e të fshirit të një faqeje të përdorshme, është të kontrollohet biti R i faqes më të vjetër. Nëse është 0, faqja është e vjetër dhe e papërdorur, kështu që zëvendësohet menjeherë. Nëse është 1, biti resetohet, faqja vendoset në fund të listës dhe koha e ngarkimit të saj azhornohet si të kishte ardhur tani në memorje, më pas kërkimi vazhdon.

Veprimi i këtij algoritmi, të quajtur Second Chance, tregohet në Fig 4-16. Në Fig 4-16 (a) shohim faqet nga A në H të mbajtura në një listë të lidhur dhe të radhitura sipas kohës në të cilën kanë arritur në memorje.

Supozojmë se ndodh një page fault në kohën 20. Faqja më e vjetër është A, që ka ardhur në kohën 0, kur procesi ka filluar. Nëse A e ka birin R 0, ajo fshihet nga memorja, edhe duke qenë shkruar në disk (nëse është dirty) apo thjesht “braktisur” (nëse është clean). Nga ana tjetër, nëse R është i setuar, A vihet në fund të listës dhe koha e ngarkimit të saj rivendoset në kohën aktuale (20). Biti R gjithashtu zerohet. Kërkimi për një faqe të përshtatshme vazhdon.

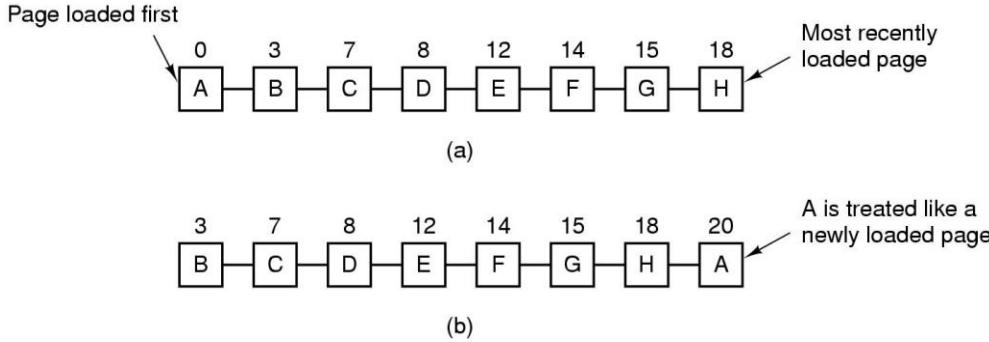


Fig. 4.16. Veprimi i Second Chance : (a) Faqet të renditura në rendin FIFO. (b) Lista e faqeve nëse ndodh një page fault në kohën 20 dhe A e ka bitin R të setuar. Numrat sipër faqeve janë kohët e tyre të ngarkimit.

Ajo që Second Chance bën, është të kërkojë një faqe të vjetër që nuk është adresuar në intervalin e mëparshëm të clock-ut. Ne qofte se të gjitha faqet janë adresuar, Second Chance degjeron në FIFO të pastër. Konkretisht, imagjinoni që të gjitha faqet në Fig 4-16 (a) e kanë bitin R 1. Një nga një sistemi operativ i çon në faqet në fund të listës, duke zemruar bitin R sa herë që shton një faqe në fund të listës. Në fund kthehet tek faqja A që tashmë ka bitin R 0. Në këtë pikë ajo fshihet. Kështu, algoritimi përfundon.

4.4.5 ALGORITMI THE CLOCK

Edhe pse algoritmi Second Chance është i logjikshëm, ai nuk është eficent për shkak të lëvizjeve të tij konstante rreth faqeve të listës. Një mënyrë më e mirë është që të mbahen të gjitha frame-t në një listë qarkulluese në formë ore, siç tregohet në Fig 4-17. Një tregues pointon në faqen më të vjetër.

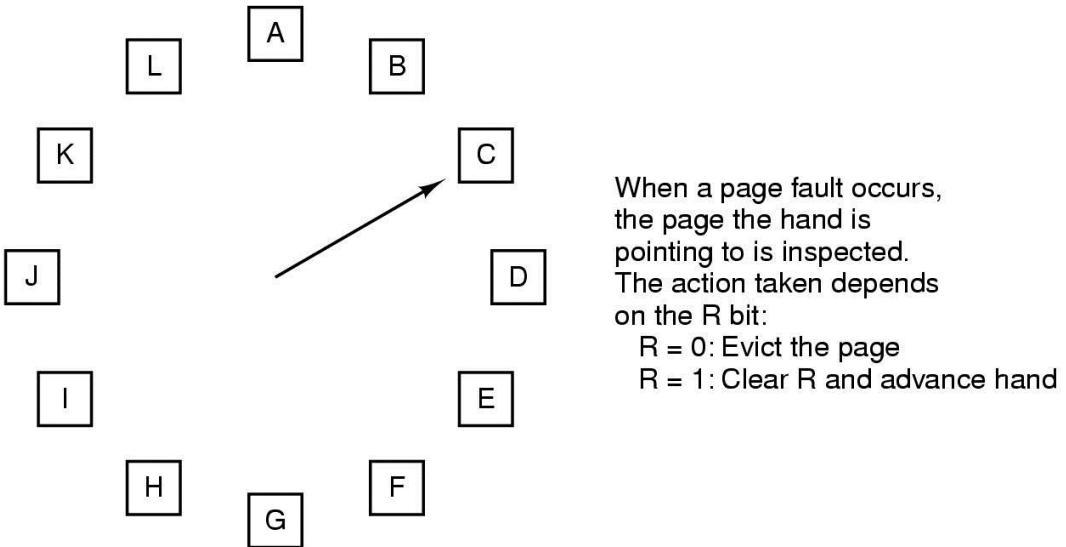


Fig. 4-17. Algoritmi The Clock

Kur ndodh një page fault, faqja tek e cila pointohet nga treguesi, kontrollohet. Nëse biti R i saj është 0, ajo fshihet, faqja e re shtohet në orë në vend të saj dhe treguesi avancon me një pozicion. Nëse biti R është 1, ai zerohet dhe treguesi avancon në faqen tjetër.

Ky proces përsëritet derisa gjendet një faqe me R=0. Nuk është çudi që algoritmi quhet The Clock. Ai ndryshon nga Second Chance vetëm nga implementimi.

4.4.6 ALGORITMI LEAST RECENTLY USED

Një përafrim i mirë i algoritmit optimal bazohet në hipotezën se faqet që janë përdorur në instruksionet e fundit ndoshta do të përdoren përsëri në instruksionet pasardhëse. Ndërsa faqet që nuk janë përdorur për vite, ndoshta nuk do të përdoren për një kohë të gjatë. Kjo ide sugeron një algoritëm të realizueshëm : kur ndodh një page fault, hiqet faqja që nuk është përdorur për kohën më të gjatë. Kjo strategji quhet LRU (Least Recently Used).

Edhe pse LRU është teorikisht i realizueshëm, nuk është i lirë. Për tu implementuar plotësisht LRU është e nevojshme të mbahet një listë e lidhur të gjitha faqeve në memorje, me faqen most recently used në fillim dhe atë least recently used në fund. Vështirësia është se lista duhet të azhornohet në çdo aksesim të memorjes. Të gjendet një faqe në listë, të fshihet dhe më pas të lëvizet në fillim është një veprim që harxhon shumë kohë, edhe në hardware (duke supozuar se një hardware i tillë mund të ndërtohet).

Megjithatë, ka mënyra të tjera për të implementuar LRU me një hardware të veçantë. Marrim në fillim mënyrën më të thjeshtë. Kjo mënyrë kërkon pajisjen e hardware me një numërues 64-bitësh, C, që automatikisht inkrementohet pas çdo instruksioni. Për më

tepër,çdo page table entry duhet të ketë një fushë aq të gjerësa të mbajë numërueshin. Pas çdo aksesimi memorjeje, vlera e C ruhet në page table entry për faqen apo të adresuar. Kur ndodh një page fault, sisemi operativ ekzaminon të gjithë numratorët në page table për të gjetur më të ulëtin prej tyre. Ajo faqe është least recently used.

Tani le të shohim algoritmin e dytë LRU të hardware. Për një makinë me n page frame, LRU mund të mbajë një matricë prej mxn bitesh, fillimi si tē gjitha 0. Sa herë që page frame k adresohet, hardware në fillim vendos tē gjitha bitet kolonës k në 0. Në çdo moment, rreshti, vlera binare e të cilit është më e lartë, është më pak recently used e kështu me radhë. Puna e këtij algoritmi tregohet në Fig. 4-18 për katër page frame dhe adresa faqesh në radhën : 0 1 2 3 2 1 0 3 2 3.

Pasi adresohet faqja 0, kemi situatën në Fig. 4-18(a). Pasi adresohet faqja 1, kemi situatën në Fig. 4-18(b), e kështu me radhë.

4.4.7 SIMULIMI I LRU NË SOFTWARE

Edhe pse të dy algoritmat LRU e mëparshëm janë të realizueshëm në parim, pak nëse ekzistojnë, makina e kanë këtë hardware, kështu që kanë përdorim të pakët në dizajnimet e sistemit operativ, që nuk janë sisteme për makina që nuk kanë këtë hardware. Në vend të saj nevojitet një zgjidhje që mund të implementohet në software. Një mundësi është quajtur algoritmi NFU (Not Frequently Used). Kjo kërkon një numrator software të asociuar çdo faqeje, në fillim 0. Në çdo sinjal clock-u, sistemi operativ skanon tē gjitha faqet në memorje. Për çdo faqe, biti R që është 0 ose 1, skanon tē gjitha faqet në memorje. Në fakt, numratorët janë një përpjekje për të ruajtur gjurmët se sa shpesh është adresuar çdo faqe. Kur ndodh një page fault, faqja me numratorin më të ulët, shpesh zgjidhet për zëvëndësim.

Page			
0	1	2	3
0	0	1	1
1	0	0	0
2	0	0	0
3	0	0	0

(a)

Page			
0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

(b)

Page			
0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

(c)

Page			
0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

(d)

Page			
0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	1
1	1	0	0

(e)

0	1	2	3
0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0
1	1	0	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0
1	1	1	0

(i)

0	1	0	0
0	0	0	0
1	1	0	0
1	1	0	0
1	1	1	0

(j)

Fig. 4-18. LRU duke përdorur një matricë kur faqet adresohen sipas radhës 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Problemi kryesor me NFU është që asnjeherë nuk harron asgjë. Per shembull në një kompilator me shumë kalime, faqet që u përdorën shpesh gjatë kalimit 1, mund të kenë një numërim të lartë edhe në kalimet e mëvonshme. Në fakt, nëse ndodh që kalimi 1 të ketë kohën më të gjatë të exekutimit të të gjithë kalimeve, faqet që mbajnë kodin e kalimeve të mëvonshëm mund të kenë gjithmonë numërimë më të ulëta se kalimi i faqeve. Si pasojë e kësaj, sistemi operativ do të fshijë faqe të nevojshme në vend të faqeve që nuk përdoren.

Fatmirësisht, një modifikim i vogël i NFU bën të mundur të simulohet mjaft mirë LRU. Modifikimi ka 2 pjesë. Së pari, numratorët zhvendosen, secili prej tyre, djathtas me një bit, përpala se të shtohet biti R. Së dyti, biti R i shtohet bitit më të majtë e jo atij më të djaththë.

Fig. 4-19, ilustron se si algoritmi i modifikuar, i njohur si **Aging**, funksionon.

Supozoni se pas impulsit të parë të clock-ut bitet R përfaqet nga 0 në 5, kanë përkatesisht vlerat: 1, 0, 1, 0, 1 dhe 1 (faqja 0 është 1, fakja 1 është 0 etj). Me fjalë të tjera, midis sinjaleve 0 dhe 1, u adresuan faqet 0, 2, 4 dhe 5, duke i vendosur bitet R të tyre në 1, ndërkohë që të tjerat qëndrojnë 0. Pasi gjashtë numëruesat korrespondues janë zhvendosur dhe biti R është shtuar më të majtë, ata kanë vlerat e treguar në Fig. 4-19(a). Katër kolonat e tjera tregojnë gjashtë numëratorët pas katër impulseve pasardhëse.

Kur ndodh një page fault, fakja, numratori i së cilës është më i ulët, fshihet. Është e qartë se një faqe që nuk është adresuar përf, të themi, 4 impulse clock-u, do të ketë 4 bitet e para 0 në numratorin e saj dhe kështu do të ketë vlerë më të vogël se një numërues që nuk është adresuar përf 3 impulse clock-u. Ky algoritëm ndryshon nga LRU në dy aspekte.

Shikoni faqet 3 dhe 5 në Fig. 4-19(e). Asnjëra nuk është adresuar për dy impulse clock-u, të dyja ishin adresuar në impulsin pararendëse. Sipas LRU, nëse një faqe duhet të zevendësohet, duhet të zgjedhim njëren nga këto të dyja. Problemi është se nuk e dimë se cila nga këto të dyja ishte adresuar e fundit midis impulsive 1 dhe 2. Duke regjistruar vetëm një bit për interval kohe, nuk e kemi aftësinë t'i dallojmë se cila nga dy qafet u adresua më shpejt në intervalin e kohës se tjetra. Gjithçka që mund të bëjmë është të fshijmë faqen 3, duke qenë se faqja 5 ishte adresuar gjithashtu 2 impulse më parë, ndërsa faqja 3 jo.

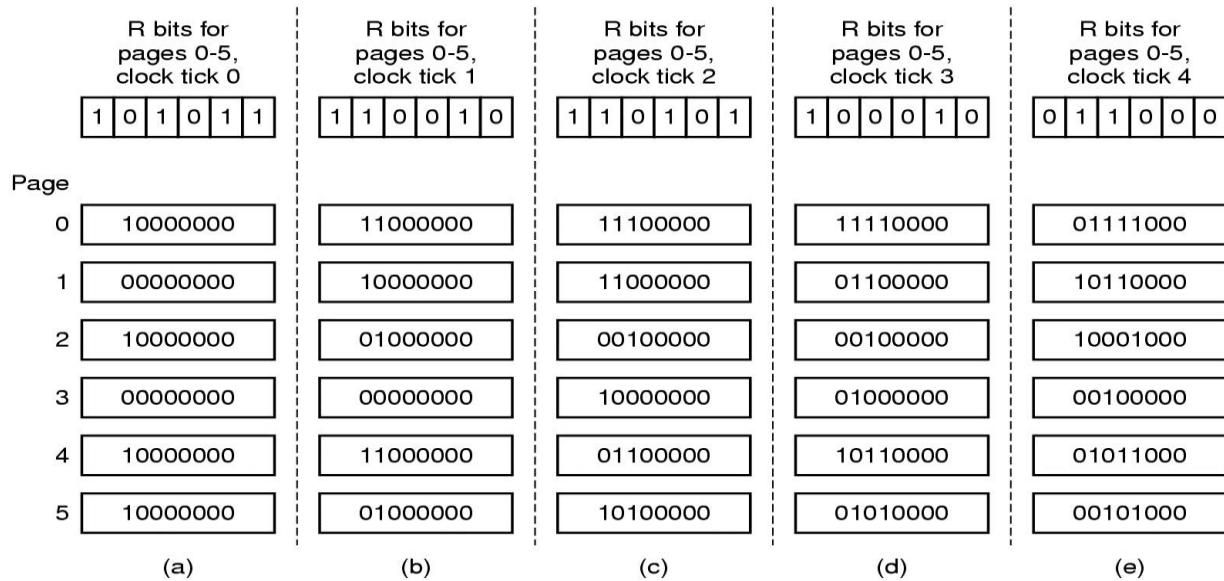


Fig. 4-19. Algoritmi Aging simulon LRU në software. Janë treguar gjashtë faqe për pesë impulse clock-u. Impulset e clock-ut paraqiten nga (a) te (e).

Ndryshimi i dytë midis Aging dhe LRU është se në aging numëruesh kanë një numër limit bitesh, 8 në rastin tonë. Supozojmë se dy faqe kanë secila vlerë të numërueshit 0. Ajo që mund të bëjmë është të zgjedhim rastësisht njëren prej tyre. Në realitet, mund të ishte një nga ato faqe që ishte adresuar per herë të fundit 9 impulse me parë, ndërsa tjetra 1000 impulse më parë. Nuk kemi si e shohim këtë gjë. Megjithatë, në praktikë 8 bit janë zakonisht të mjaftueshëm nëse një impuls clock-u është zakonisht 20 ms. Nëse një faqe nuk është adresuar në 160 ms, ndoshta ajo nuk është shumë e rëndësishme.

4.4.8 ALGORITMI WORKING SET

Në formën më të pastër të punimit me faqe, proceset fillojnë me asnjë nga faqet e tyre në memorje. Sapo CPU përpinqet të kapë instrukcionin e parë, ai merr një page fault, duke bërë që sistemi operativ të sjellë në page mbajtesin e instrukzionit te parë. Kjo ndiqet shpejt nga të tjera page faults për variabla globalë dhe stack-un. Pas njëfarë kohe, procesi ka shumicën e faqeve që i duhen dhe normalizohet duke u ekzekutuar relativisht pak page faults. Kjo strategji quhet **demanding paging** sepse faqet ngarkohen vetëm me kërkeshë dhe jo me përparësi.

Sigurisht, është mjaft e thjeshtë të shkruhet një program test që lexon sistematikisht të gjitha faqet në një hapësirë të madhe adresash, duke shkaktuar aq shumë page faults, sa nuk ka memorje të mjaftueshme për t'i mbajtur të gjitha. Fatmirësisht, shumica e proceseve nuk punojnë kështu. Ata paraqesin një lokalitet adresash, që do të thotë se gjatë çdo faze ekzekutimi procesi adreson vetëm një fraksion relativisht të vogël të të gjitha faqeve të veta. Çdo kalim i një kompiatori shumë-kalimësh, për shembull, akseson vetëm një fraksion të të gjitha faqeve, dhe një fraksion të ndryshëm nga ai.

Seti i faqeve që një proces është duke përdorur quhet **working set**-i i tij (Denning, 1968, a; Denning, 1980). Nëse i gjithë working set-i është në memorje, procesi do të ekzekutohet pa shkaktuar shumë page faults dhe do të ekzekutohet ngadalë, duke qenë se ekzekutimi i një instruksioni kërkon disa ns, ndërsa leximi i një faqeje nga disku zakonisht kërkon 10 ms. Në një shkallë prej disa 1 ose 2 instruksionesh për 10ms, do të duhej shumë kohë për të mbaruar. Një program që shkakton page fault çdo disa instruksione thuhet se është **thrashing** (Denning, 1986b).

Në një sistem me multiprogramim, proceset çohen shpesh në disk (pra të gjitha faqet e tyre fshihen nga memorja), për t'ua lënë radhën proceseve të tjera në CPU. Lind pyetja se ç'do të ndodhë kur një proces duhet sjellë përsëri në memorje. Teknikisht nuk duhet bërë asgjë. Procesi do të shkaktojë page faults, derisa working set-i i tij të jetë ngarkuar. Problemi është se duke patur 20,100 apo edhe 1000 page faults sa herë që ngarkohet një proces, ngadalëson dhe gjithashtu humbet një kohë të konsiderueshme të kohës së CPU, duke qenë se sistemit operativ i duhen disa ms të CPU për të procesuar një page fault.

Prandaj shumë sisteme që punojnë me faqe përpinqen të mbajnë gjurmë të working set të çdo procesi dhe të sigurohet se është në memorje para se të lejojnë që procesi të ekzekutohet. Kjo mënyrë quhet **working set model** (Denning, 1970). Ajo është projektuar për të reduktuar shumë shkallën e page fault. Ngarkimi i faqeve para se të lejohet ekzekutimi i proceseve quhet gjithashtu **prepaging**. Kini parasysh se working set ndryshon në lidhje me kohën.

Dihet prej kohësh se shumë programe nuk e aksesojnë adresën e tyre të adresave uniformisht, por adresat kanë tendencën të mblidhen në një numër të vogël faqesh. Një aksesim memorjeje mund të kapë një instruksion, një të dhënë ose mund të ruajë një të dhënë. Në çdo moment të kohës t, ekziston një grup i të gjitha faqeve nga k adresimet të pak mëparshme. Ky grup, w{k, t}, është working set. Meqënëse k=1, shumica e adresave

të fundit (recent) duhet t'i kenë përdorur të gjitha faqet e përdorura nga $k>1$ adresat më të fundit (most recent) dhe mbante të tjera, $w\{k, t\}$ është funksion monoton jo zbritës i k. Limiti i $w\{k, t\}$, duke qenë se k rritet, është i fundëm, sepse një program nuk mund të adresojë më shumë faqe nga sa mban hapësira e tij e adresimit dhe pak programe do të përdorin çdo faqe. Fig. 4-20 paraqet madhësinë e working set si funksion i k.

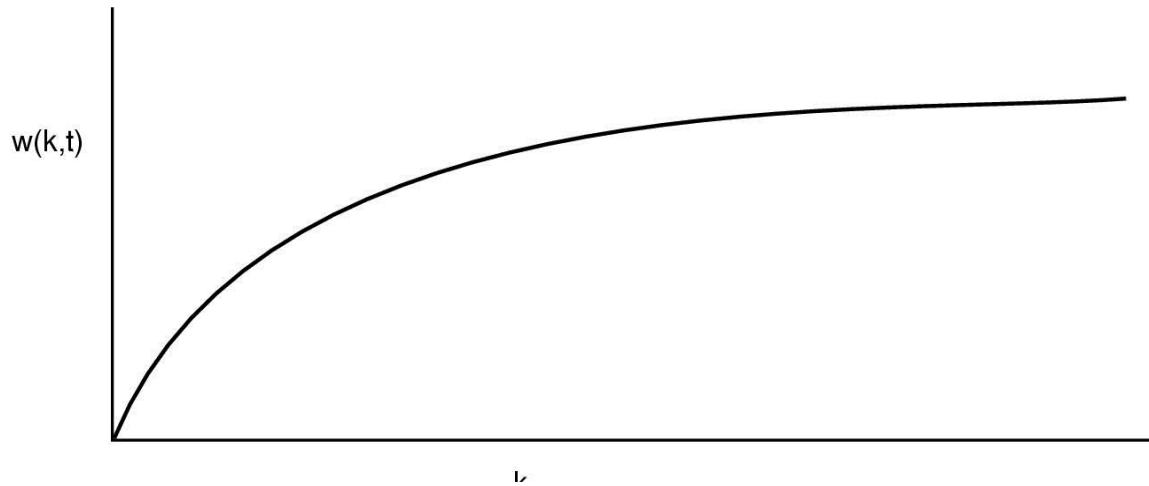


Fig. 4-20. Working set është grapi i faqeve të përdorura nga k adresimet më të fundit të memorjes. Funksioni $w\{k, t\}$ është madhësia e working set në kohën t.

Fakti se pjesa më e madhe e programeve aksesojnë një numër të vogël faqesh, por që ky grup ndryshon ngadalë, shpjegon rritjen e shpejtë fillestare të kurbës dhe më pas rritjen e ngadalësuar për k të mëdha. Per shembull, një program që është duke ekzekutuar një cikël që ze dy faqe, që përdorin të dhëna për katër faqe, mund të adresojë të gjashtë faqet çdo 1000 instruksione, por adresa më e fundit (most recently) në një faqe tjetër mund të jetë 1000 instruksione më shpejt, gjatë fazës së inicializimit. Për shkak të kësaj sjelljeje asimptodike, përbajtja e working set nuk është e ndjeshme ndaj vlerës së k-së së zgjedhur. Për ta thënë ndryshe, ekziston një varg i gjatë vlerash të k për të cilat working set është i pandryshuar. Meqënëse working set ndryshon ngadalë në lidhje me kohën, është e mundur të bëhet një hamendje e logjikshme se cilat faqe do të duhen kur programi të rifillojë mbi bazën e working set-it kur ai u ndalua për herë të fundit. Prepaging konsiston në ngarkimin e këtyre faqeve përpara se procesi të lejohet të exekutohet përsëri.

Për implementimin e modelit working set është e nevojshme që sistemi operativ të ruajë gjurmët e se cilat faqe janë në working set. Të paturit e këtij informacioni gjithashtu çon menjeherë në një algoritëm të mundshëm të zëvendësimit të faqeve: kur ndodh page fault, gjendet një faqe që nuk është në working set dhe fshihet. Për të implementuar një algoritëm të tillë na duhet të prezkojmë mënyrën e përcaktimit se cilat faqe janë në working set dhe cilat nuk janë në një moment kohe të dhënë.

Siç përmendëm më lart, working set është bashkësia e faqeve të përdorura në k adresat më të fundit (disa autorë përdorin 7 adresat më të fundit, por zgjedhja është arbitrale). Për të implementuar çdo algoritëm working set, disa vlera të k duhet të zgjidhen me parë. Pasi është zgjedhur vlera, pas çdo adresimi, grupi i faqeve i përdorur nga k adresat e mëparshme është i përcaktuar në mënyrë unike.

Pa dyshim, të paturit e një përkufizimi funksional të working set nuk do të thotë se se ka një mënyrë eficiente për ta monitoruar atë në kohë reale gjatë ekzekutimit të programit. Mund të imagjinohet një regjistër rrëshqitës me gjatësi k_y , që çdo adresim memorjeje e zhvendos atë në pozicionin majtas dhe në të djathtë shtohet nr i fakes të adresuar më në fund (most recently). Grupi i k numrit të faqeve në regjistrin rrëshqitës do të ishte working set. Megjithatë, të mbahet rregjistri rrëshqitës dhe dhe të procesohej në një page fault do të ishtë shumë e kushtueshme, prandaj kjo teknikë nuk përdoret.

Në vend të saj, janë përdorur përafrime të ndryshme. Një përafrim që përdoret zakonisht është realizimi i idesë së numërimit përsëri të k adresave dhe në vend të tyre të përdoret koha e ekzekutimit. Per shembull, në vend të përcaktimit të working set si faqet e përdorura gjatë 10 mln adresimeve të mëparshme, mund ta përcaktojmë atë si grupi i faqeve i përdorur gjatë 10 m/sec e kohës së ekzekutimit. Në praktikë, ky përcaktim është po aq i mirë, sa edhe më i thjeshtë për tu përdorur. Kini parasysh se për çdo proces, ka rëndësi vetëm koha e tij ekzekutimit. Kështu, nëse një proces fillon ekzekutimin në kohën T dhe ka pasur 40 m/sec nga koha e CPU në kohën reale $T+100$ m/sec, për qëllimet e working set, koha e tij është 40 m/sec. Pjesën e kohës së CPU që procesi ka përdorur që në fillim të tij, shpesh quhet **koha virtuale aktuale** (current virtual time) e tij. Me këtë përafrim, working set i një procesi është grupi i faqeve që ai ka adresuar gjatë T sekondave të kaluara të kohës virtuale.

Tani le të shohim një algoritëm të bazuar në working set. Ideja bazë është të gjendet një faqe që nuk është në working set dhe të fshihet. Në Fig. 4-21 shohim një pjesë të page table të një kompjuteri. Duke qenë se vetëm faqet që ndodhen në memorje konsiderohen si kandidate për tu fshirë, faqet që mungojnë injorohen nga ky algoritëm. Çdo entry përmban të paktën dy elementë: kohën e përafërt kur faqja është përdorur për herë të fundit dhe bitin R (të referencës). Drejtkëndëshi i bardhë bosh përfaqëson fushat e tjera, të panevojshme për këtë algoritëm, të tilla si numri i page frame-it, bit i mbrojtjes dhe biti M (i modifikimit).

2204 Current virtual time

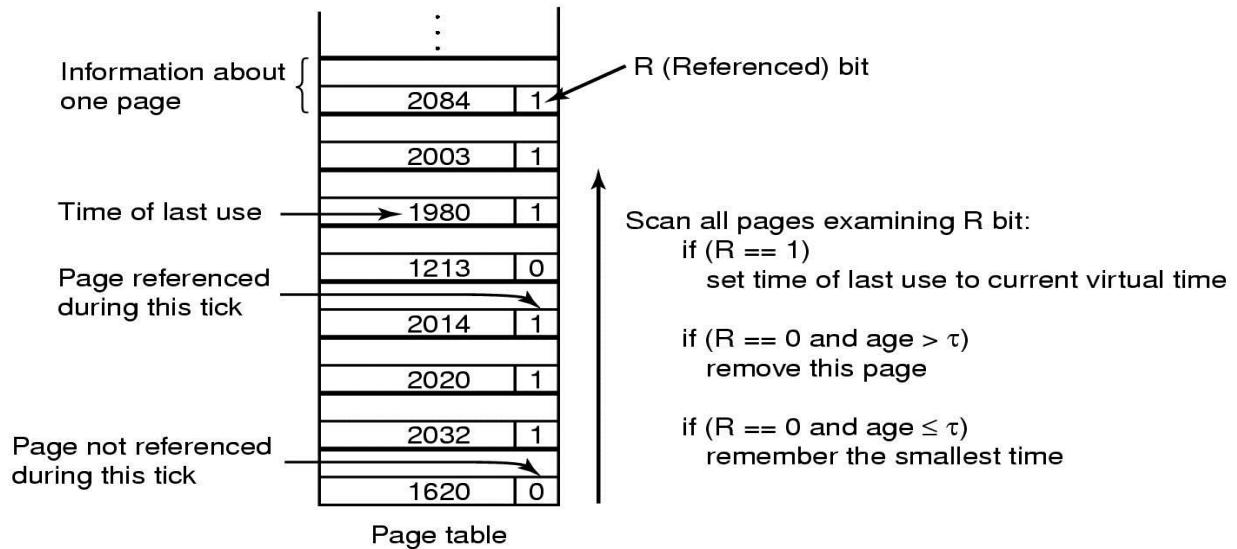


Fig. 4-21. Algoritmi Working Set

Algoritmi funksionon si më poshtë: hardware duhet të sertojë bitet R dh M, siç kemi thënë më parë. Gjithashtu, një inetrrupt clock-u periodik duhet të bëjë të ekzekutohet software që zeron bitin e referencës në çdo sinjal clock-u. Në çdo page fault, page table skanohet për të kërkuar një faqe të përshtatshme për të fshirë. Duke qenë se çdo entry procesohet, biti R ekzeminohet. Nëse është 1, koha virtuale aktuale shkruhet në fushën e *kohës së përdorimit të fundit* në page table, që tregon se faqja ishte në përdorim kur ndodhi page fault. Qëkur faqja u adresua gjatë sinjalit aktual të clock-ut, është e qartë se në working set nuk është kandidate për tu fshirë (pranohet se τ përfshin shumë sinjale clock-u).

Nëse R është 0, do të thotë se faqja nuk është adresuar gjatë sinjalit aktual të clock-ut dhe kështu mund të jetë kandidate për tu fshirë. Për të parë nëse ajo duhet të fshihet, mosha e saj, që është "koha virtuale aktuale - koha e përdorimit të fundit", llogaritet dhe krasoheret me τ . Nëse mosha është më e madhe se τ faqja nuk është më në working set. Ajo hiqet dhe faqja e re ngarkohet këtu. Megjithatë, skanimi vazhdon të azhorojë entries që ngelen.

Sidoqoftë, nëse R është 0, por mosha është më e vogël ose e barabartë me τ , faqja është ende në working set. Kjo faqe përkohësisht nuk fshihet, por faqja me moshën më të madhe (vlerën më të madhe të Time of last use) do të shënohet. Nëse gjithë tabela skanohet pa gjetur një kandidate për të fshirë, kjo do të thotë se të gjitha faqet janë në working set. Në ketë rast, nëse një ose më shumë faqe me R=0 do të ishin gjetur, ajo me moshën më të madhe do të ishtë fshirë. Në rastin më të keq, të gjitha faqet janë adresuar gjatë sinjalit aktual të clock-ut (kështu që të gjitha kanë R=1), kështu zgjidhet rastësisht një për tu fshirë, mundësisht një faqe clean, nëse ekziston një e tillë.

4.4.9 ALGORITMI WSCLOCK

Algoritmi bazë working set është i ngarkuar, duke qenë se e gjithë page table duhet skanuar në çdo page fault derisa lokalizohet një faqe e përshtatshme. Një algoritëm i përmirësuar, që bazohet në algoritmin clock, por përdor gjithashtu informacionin e working set, është quajtur WSClock (Carr and Henne sey, 1981). Për shkak të thjeshtësisë së tij për tu implementuar dhe performancës së tij të mirë, ky algoritëm përdoret gjërësist në praktikë. Struktura e të dhënave që nevojitet është një listë rrethore page frame-sh si në algoritmin clock, siç tregohet në Fig. 4-22(a). Fillimisht lista është bosh. Kur ngarkohet faqja e parë, ajo i shtohet listës. Duke u shtuar më shumë faqe, ato shkojnë brenda listës për të formuar një unazë. Çdo entry përmban fushën e kohës së përdorimit të fundit, nga algoritmi bazë working set, dhe gjithashtu bitin R (të treguar në figurë) dhe bitin M (nuk tregohet në figurë).

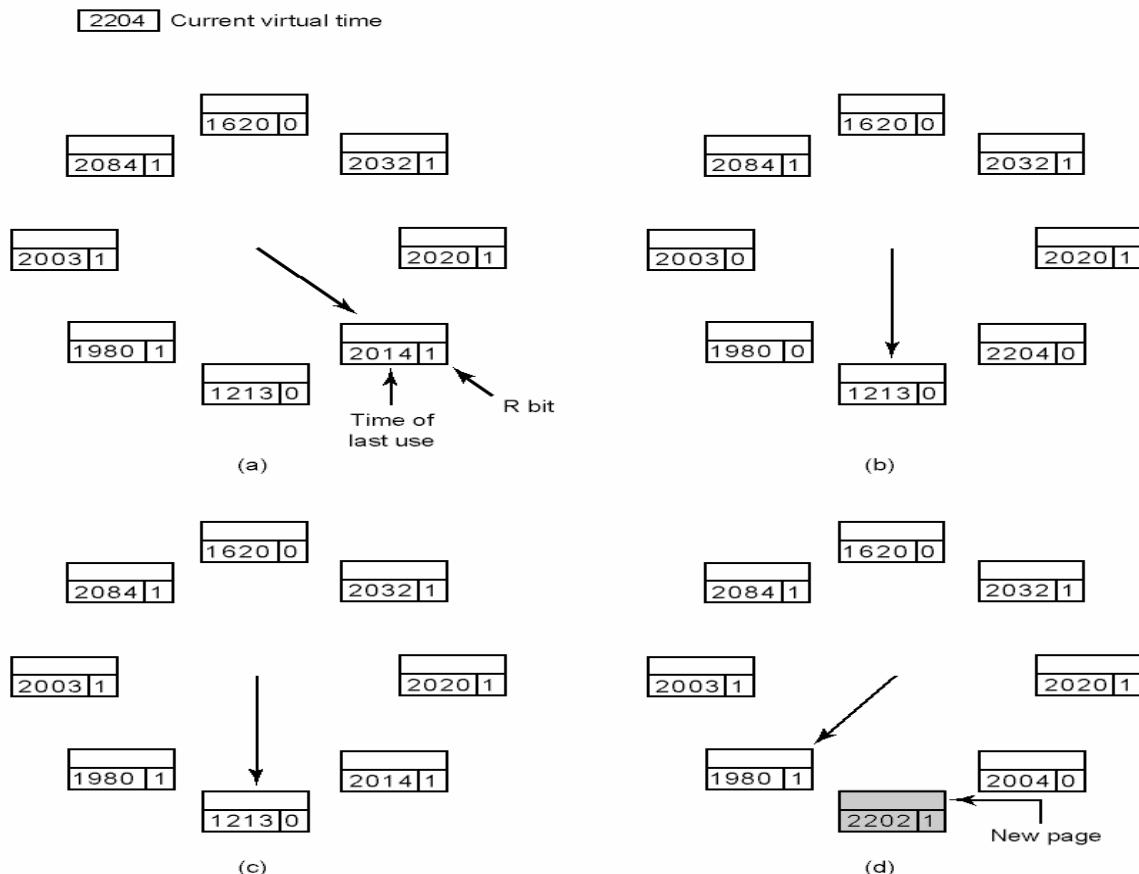


Fig. 4-22. Veprimi i algoritmit WSClock. (a) dhe (b) jepin një shembull se ç'ndodh kur R=1. (c) dhe (d) jepin shembull se ç'ndodh kur R=0.

Si tek algoritmi clock, në çdo page fault, faqja e pointuar nga treguesi, fillimisht ekzaminohet. Nëse biti R është 1, faqja është përdorur gjatë impulsit aktual të clock-ut, prandaj nuk është kanidate për tu fshirë. Biti R setohet në 0, treguesi avancon në faqen pasardhëse dhe algoritmi përsëritet për këtë faqe. Gjendja për këtë sekunçë ngjarjesh tregohet në Fig. 4-22(b).

Tani mendoni ç'ndodh nëse faqja tek e cila pointohet e ka R=0, siç tregohet në Fig. 4-22(c). Nëse mosha e saj është më e madhe se τ dhe faqja është clean, ajo nuk është në working set dhe një kopje e vlefshme e saj ndodhet në disk. Page frame thjesht hiqet dhe faqja e re vendoset atje, siç trgohet në Fig. 4-22(d). Në anën tjetër, nëse faqja është dirty, nuk mund të hiqet menjeherë, meqënjëse asnë kopje e vlefshme ekziston në disk. Për të shmangur një process switch, shkrimi në disk skedulohet, por treguesi avancon dhe algoritmi vazhdon me faqen tjetër. Në fund të fundit, mund të ketë një faqe të vjetër, clean më poshtë që mund të përdoret menjeherë.

Në parim, të gjitha faqet mund të skedulohen për disk I/O në rreth 1 cikël clock-u. Për të reduktuar trafikun në disk, mund të vendoset një limit që lejon një maksimum prej n faqesh të rishkuhen. Nëse ky limit arrihet nuk skedulohen më shkrime të reja.

Ç'ndodh nëse treguesi gjithë kohën vjen rrotull pikës së tij të filimit? Këtu dallohen dy raste :

1. Të paktën një shkrim është skeduluar.
2. Asnjë shkrim nuk është skeduluar.

Në rastin e parë, treguesi vetëm leviz, duke kërkuar një faqe clean. Duke qenë se një ose më shumë shkrime janë skeduluar, në fund një shkrim do të plotësohet dhe faqja e tij do të shënohet si clean. Faqja e parë clean që do të gjendet, do të fshihet. Kjo faqe nuk është domosdoshmërisht e para write skeduled sepse disk driver-i mund t'i regjistroje shkrimet në mënyrë që të optimizojë performancën e sistemit.

Në rastin e dytë, të gjitha faqet janë në working set, ndryshe të paktën njëra do të ishte skeduluar. Në mungesë të informacionit shtesë, gjëja më e thjeshtë për të bërë është të përdoret një faqe clean. Adresa e një faqeje clean mund të mbahet si gjurmë gjatë fshirjes. Nëse s'ka faqe clean, atëherë faqja aktuale zgjidhet dhe rishkuhet në disk.

4.4.10 PËRMBLEDHJE E ALGORITMAVE TË ZËVENDËSIMIT TË FAQEVE

Kemi parë disa algoritma zëvendësimi të faqeve. Në këtë seksion do t'i permblledhim ato shkurtimisht. Lista e algoritmave të diskutuar jepet në Fig. 4-23.

Algorimi optimal zëvendëson faqet e adresuara në fund më ato aktuale. Fatkeqësisht, nuk ka mënyrë përcaktimi se cila faqe do të jetë e fundit, kështu që në praktikë ky algoritëm nuk mund të përdoret. Gjithsesi, shërben si pikë referimi me të cilin mund të karahasohen algoritmat e tjera.

Algoritmi NRU i ndan faqet në 4 klasa, në varësi të biteve R dhe M. Zgjidhet rastësishët një faqe nga ato me numrin me te ulet. Ky algoritëm është i thjeshtë për tu implementuar, por është shumë i pa përpunuar. Ekzistojnë edhe më të mirë.

FIFO ruan gjurmët e radhës në të cilën faqet u ngarkuan në memorje duke i mbajtur ato në një list të lidhur. Duke fshirë faqen më të vjetër më pas bëhet i rëndomtë, por ajo faqe mund të jetë ende në përdorim, kështu që FIFO është një zgjedhje e keqe.

Second chance është një modifikim i FIFO që kontrollon nëse një faqe është në përdorim para se ta fshijë atë. Nëse po, atëherë faqja nuk fshihet. Ky modifikim e përmirëson shumë performancën. Clock është thjesht një implementim i ndryshëm i second chance. Ka të njëjtat karakteristika të performancës, por kërkon më pak kohë për ekzekutimin e algoritmit.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Fig. 4-23. Algoritmat e diskutuar në tekst.

LRU është një algoritëm i shkëlqyer, por nuk mund të implementohet pa hardware të veçantë. Nëse ky hardware nuk është i disponueshëm, LRU nuk mund të përdoret. NFU është një përpjekje e parëndësishme për të arritur LRU. Nuk është shumë i mirë. Gjithsesi, aging është një përafrim më i mire i LRU dhe mund të implementohet në mënyrë eficiente. Është një zgjedhje e mirë.

Dy algoritmat e fundit përdorin working set. Algoritmi working set ka performancë të arsyeshme, por në njëfarë mënyrë është i shtrenjtë për tu implementuar. WSClock është një variant që jo vetëm jep performancë të mirë, por është eficent për tu implementuar.

Si përfundim, dy algoritmat më të mirë janë aging dhe WSClock. Ato bazohen përkatesisht në LRU dhe working set. Të dyja janë performancë të mirë dhe mund të implementohen ne mënyrë eficiente. Ka dhe disa algoritma të tjera, por ndoshta këta të dy janë më të rëndësishëm në praktikë.

4.5 MODELI I ALGORITMIT ME ZËVËNDESIM FAQESH

Ne vite ështe bërë pune ne modelin e algoritmit me zëvendësim faqesh nga një perspektive teorike. Ne kete pjesë ne do te diskutojme disa nga keto ide, vetem per te pare se si punon procesi i modelimit.

4.5.1 ANOMALIA E BELADY

Llogjikisht, duket qe sa me shume Page frame ka memorja, me pak Page faults do te marre programi. Çuditerisht mjashtueshem, nuk eshte zbuluar gjithmone një shembull numerimi, ne te cilën FIFO shkaktoi me shume page faults me kater faqe frame se me tre. Kjo situate e çuditshme është bëre e njohur si ANOMALIA BELADY. Kjo është ilustuar ne figuren 4-24 për një program me pesë faqe virtuale, numeruar nga 0 ne 4. Faqet janë referuar ne rregullin:

(M 2 3 0 I 4 0 I 2 .1 4

Ne figuren 4-24.(a) ne shohim se si me tre page frame, një total me nente page faults kane ndodhur. Ne fig. 4-24 ne marrim dhjete page faults me kater page frame.

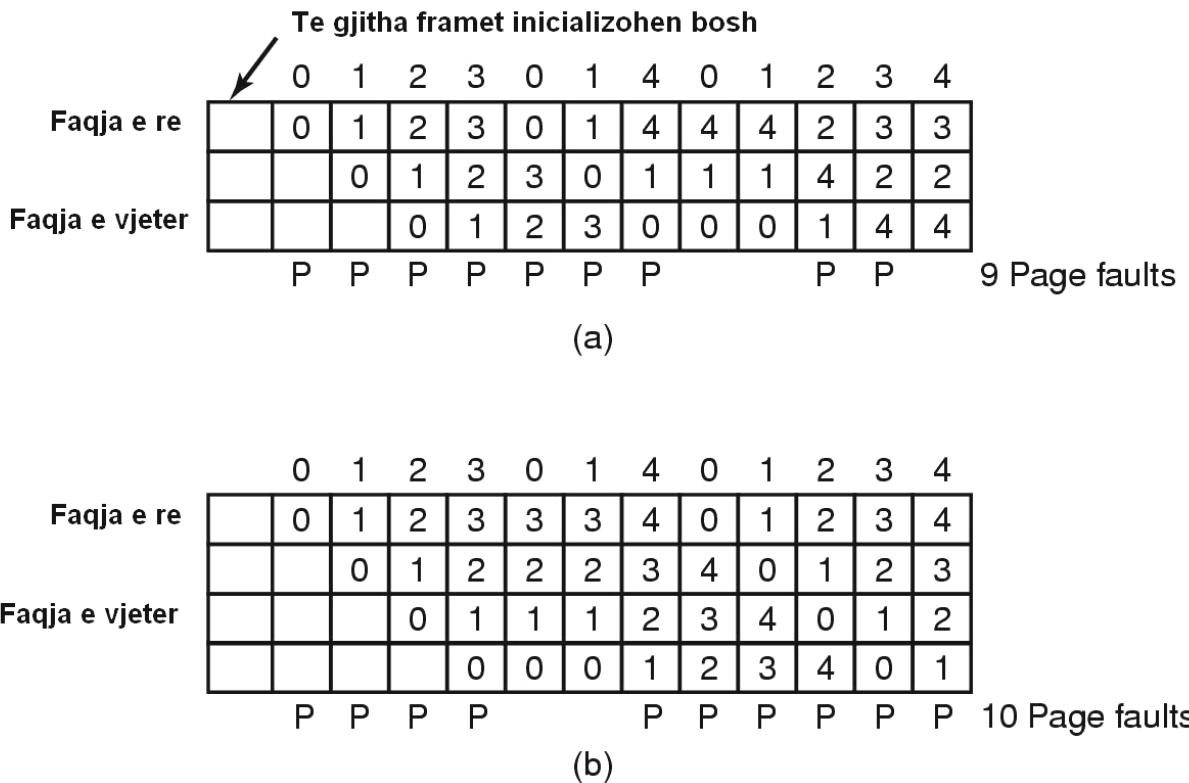


Figura 4-24 Anomalia Belady, (a) FIFO me tre page frames, (b) FIFO me kater page frame

4.5.2 ALGORITMET STACK

Shume kërkime në shkencen e kompjuterit ishin zbuluar nga anomalia Belady dhe filluan ta investigonin ate. Kjo pune çoi ne zhvillimin e një teorie të tёre te algoritave te faqeve dhe karakteristikave te tyre. Ndersa pjesa me e madhe e kesaj pune ishte jashtë ketij libri, ne do te japim një prezantim te shkurter me poshte. Për me shume detaje shiko (Maekawa et al 1987).

Protesimi i kesaj pune, fillon me vrojtimin qe çdo proces gjeneron një sekuence te referencave te memorjes, ndërkohe qe ekzekutohet. Çdo reference te memorjes i korespondon një faqe specifike virtuale. Keshtu qe, një proces i aksesimit ne memorje mund te karakterizohet nga një list e (renditur) e numrave te faqeve. Kjo list eshte quajtur stringa reference, dhe luan një rol qëndror ne teori. Per thjeshtesi, në vazhdim ne do te marrim ne konsiderate vetem rastin e një makine me një proces, keshtu qe çdo makine ka një proces, deterministic reference string (me shume procese ne do te duhej te merrnim ne llogari shtresezimin e stringave te tyre te references qe çojne ne multiprogramim).

Nje sistem numerimi karakterizohet nga tre gjera:

- 1. Stringa reference e procesit te ekzekutimit.**
- 2. Algoritmi i zevendesimit te faqeve.**
- 3. Numri i faqeve frame te mundshme ne memorje, m.**

Llogjikisht ne mund te imagjinojme nje interpretues abstrakt qe funksionon si me poshte. Kjo miremban nje grup te brendshem. A, qe mban gjurmet e gjendjes se memorjes. Ai ka aq shume elemente sa ka procesi faqe virtuale, te cilat ne i shenojme me n. Grupi M eshte ndare ne dy pjesë. Pjesa e siperme me **m hyrje**, permban te gjitha faqet qe jane momentalisht ne memorje. Pjesa e poshtme, me n-m faqe, permban te gjitha faqet qe kanë qëne referuar njehere por kane qene page OUT dhe nuk jane momentalisht ne memorje. M eshte set bosh per aq kohe sa asnje faqe nuk ka qënë referuar dhe nuk ka as nje faqe ne memorje.

Ndërkojë qe ekzekutimi fillon te lere jashte faqet ne stringen reference, nje ne cdo kohe. Nderkohe qe secila shfaqet, interpretuesi kontrollon ne se faqja eshte ne memorje (i.e., ne pjesen e siperme te M). Ne qofte se nuk eshte, ndodh nje Page Faults. Ne qofte se ndodhet nje vrime boshe ne memorje(i.e., pjesa e siperme e M permban me pak se m hyrje), faqja eshte ngarkuar dhe ka hyre ne pjesen e siperme te M. Kjo situate arrihet vetem ne fillim te ekzekutimit. Ne qofte se memoria eshte plote (i.e., pjesa e siperme e M permban m hyrje) algoritmi i faqes me zevendesim thirret per te zhvendosur nje faqe nga memorja. Në model ajo qe ndodh është që nje faqe ështe zhvendosur nga pjesa e siperme e M ne pjesen e poshtme, dhe faqja e nevojshme qe ka hyre ne pjesen e siperme. Per me teper, pjesa e siperme dhe tabani i poshtem mund te korrigohen te ndara. Per ti bere me te qarta veprimet e interpretuesit le te shikojme nje shembull konkret duke perdorur LRU faqe zevendesimi. Hapesira e adreses virtuale ka tete faqe, dhe memoria fizike ka kater Page frame. Ne fillim te figures 4-25 ne kemi nje string reference qe perbehet nga 24 faqe:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1

Poshte stringes se references, ne kemi 25 kolona me tete ITEM secila. Kolona e pare, e cila eshte bosh tregon gjendjen e M perpara se te filloje ekzekutimi. Secila kolone pasuese tregon M pasi nje faqe ështëlene jashte nga referanca, dhe ështe perpunuar nga algoritmi i faqosjes. Kufiri i rende tregon majen e AY, qe perben kater vrimat e para, te cilat iu korrespondojne page frame ne memorje. Faqet brenda kutise se rende jane ne memorje, dhe faqet poshte saj jane faqosur jashte diskut.

Modeli i algoritmit te zevendesimit te faqeve

Stringa e references	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	3	1	7	1	3	4	
		0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3		
			0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7		
				0	2	1	1	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5			
					0	2	2	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6		
						0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		
							0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
Page faults	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P		
Stringa e distances	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3							

Fig.4-25. Propozimi i memorjes tabele. M

Faqja e pare ne stringen reference eshte 0. keshtu qe ajo eshte futur ne fillim te memorjes, sic tregohet ne kolonen e dyte. Faqja e dyte eshte 2, RO kjo ka hyre ne fillim te kolones se trete. Ky veprim ben qe 0 te leviz poshte, hapesire bosh per shembull. Se fundmi faqja e ngarkuar gjithmone hyn ne fillim, dhe cdo gje akoma zhvendoset poshte si nevoje.

Secila prej shtate faqeve te para ne stringen reference shkakton nje page faults. Kater prej tyre mund te trajtohen pa zhvendosur nje faqe. HUT fillon me nje reference ne faqen 5, faqja e re e ngarkuar kerkon zhvendosjen ne faqen e vjeter.

Referanca e dyte ne faqen 3 nuk shkakton nje page fault, sepse eshte pothuajse ne memorje. Sidoqofte interpretuesi e leviz ate nga ku ajo eshte, dhe e vendos ate ne fillim sic tregohet. Procesi vazhdon per nje kohe derisa faqja 5 eshte referuar. Faqja eshte zhvendosur nga pjesa e fundit e M ne pjesen e siperme (i.e., ii eshte ngarkuar ne memorje nga disku). Sidoqofte nje faqe eshte referuar, keshtu qe nuk eshte brenda nje kutie te rende, ndodh nje page fault, sic shfaqet nga />'s poshte matrices.

Le te permblehdhim tani me pak fjale shkurtimisht disa nga karakteristikat e modelit gabim. Në mëndersa një faqe eshte referuar, ajo eshte zhvendosur gjithmone në hyrje të M. Se dyti, ne qoftë se faqja e adresuar eshte tashme në Af, gjitha faqet siper saj zhvendosen një pozicion poshtë.

Ky kalim nga brendesia e kutise jashtë saj i korrespondon një faqeje ekzistuese te debuar nga memorja. Se treti, faqet qe ishin poshtë fakes se adresuar nuk jane zhvendosur. Ne kete menyre, permbajtja e M saktësish paraqet permbajtjen e algoritmit LRU.

Ndene se ky shembull perdor LRU, modeli funksionon njelloj me algoritmat e tjere. Posacerisht, ndodhet një klase algoritmash e cila eshte kryesisht e vecante; algoritmat qe kane karakteristikat

$$M(m, r) \leq M(m+1, r)$$

Ku m ndryshon gjate page frames, dhe r eshte një indeks ne stingen reference.

Çfare kjo thote eshte qe seti i faqeve te perfshira ne pjesen e fillimit te M per një memorje me m page frame pasi r referencat e memorjes, jane perfshire gjithashtu ne MTOR një memorje me $m+1$ page frames. E thene ndryshe, ne qoftë se ne rrësim madhesine e memorjes me një page frame, dhe riekzekuton procesin, i pare nga cdo pike veshtrim gjate ekzekutimit, gjitha faqet qe ishin prezente ne ekzekutimin e pare, jane gjithashtu prezente ne ekzekutimin e dyte, per gjate me një faqe shtese.

Nga ekzaminimi i fig 4-25 dhe një ide e vogel se si funksionon, eshte e qarte qe LRU ka kete karakteristike. Disa algoritma te tjere, gjithashtu e kane ate, por FIFO nuk e ka. Algoritmet qe kane kete karakteristike quhen Stack Algorithms. Keto algoritme nuk preken nga anomalia Beladys dhe jane ne kete menyre me te preferuara nga teoria e memorjes virtuale.

4.5.3 Stringa distance

Per Stackun e algoritmit, kjo eshte shpesh e pershtatshme per te paraqitur stringen reference ne një menyre me abstrakte sesa numrat aktual te faqeve. Një faqe reference do te paraqitet qe tani e tutje me ane te një distance nga maja e stakut ku faqja e adresuar ishte vendosur. Per shembull, referanca ne faqen 1 ne kolonen e fundit te Figures 4-25, eshte një reference drejt një faqeje ne distance 3 nga maja e stakut. (sepse faqja 1 ishte ne vendin e trete perpara references). Faqet qe akoma nuk jane adresuar, dhe si rrjedhim akoma nuk jane ne stack (i.e., akoma jo ne M) jane konsideruar te rrine ne distance <<. Stringa distance per figuren 4-25 eshte caktuar ne fund te figures.

Shenimi qe stringa distance varet jo vetem nga stringa reference por edhe nga algoritmi i faqosjes. Me te njejtien stringe reference origjinale, nje tjeter algoritem faqosje, duhet te beje zgjedhje te ndryshme rreth asaj se cilat faqe te nxjerr jashte. Si rezultat, rrjedh nje sekuese e ndryshme e stakut.

Karakteristikat statistikore te stringes distance kane nje ndikim te madh ne performancen e algoritmit. Ne Fig 4-26(a) ne shohim funksionin e densitetit propabilitar per hyrjen ne nje stringe distance, d. Pjesa me e madhe e hyrjeve ne string Jane ndermjet 1 dhe k.

Me nje memorje prej k page frame, ndodhin me pak page faults.

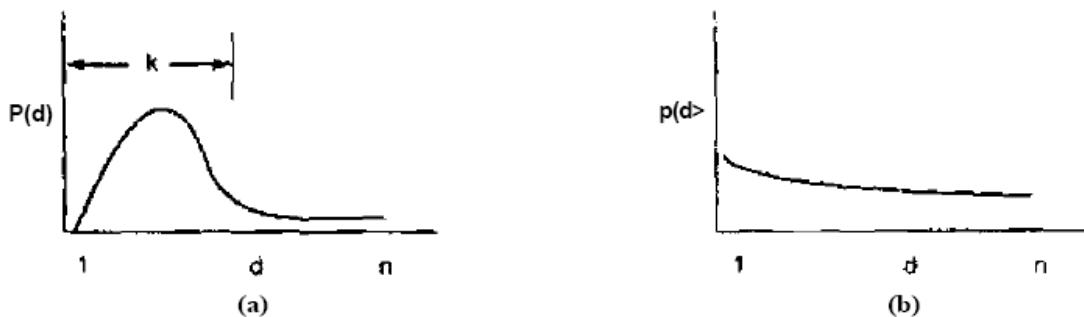


Figure 4-26. Probability density functions for two hypothetical distance strings.

Ne kundershtim, ne figuren 4-26(b) referencat Jane nxjerre jashte, keshtu qe e vetmja rruge per te menjanuar nje numer te madh te page faults eshte ti japesha programit aq page frame (aq) sa ka faqe virtuale. Te pasurit e nje programi si ky eshte tamam nje fat i keq

4.5.4 PARASHIKIMI I MASES SE PAGE FAULT

Nje nga karakteristikat me te mira te distances; stringa eshte qe ajo qe mund te perdoret per te parashikuar numrat e page faults qe do te ndodhin nga memorje te permasa te ndryshme.

Ne do te demostrojme se si kjo llogaritje mund te bazohet ne shembullin e figures 4-25. Qellimi eshte te besh nje kalim mbi stringen distance dhe, nga informacioni i mbledhur te aftesohesh per te parashikuar sa shume page fault, duhet te kete procesi ne memorjet me 1, 2, 3, ..., n. page frame, ku n eshte numri i faqeve virtuale ne procesin e hapesires se adreses.

Algoritmi fillon me skanimin e stringes distance, faqe per faqe. Ajo ruan gjurmet e numrit 0\, kohet 1 ndodhin, numri i koheve 2 ndodhin, dhe keshtu me rradhe. Le te jete C

numri i ndodhive te j. Per stringen distance te fig. 4-25 vektori C eshte ilustruar ne fig.4-27(a). Ne kete shembull, kjo ndodh kater here, keshtu qe faqja e referuar eshte pothuajse ne maje te stakut. Tre heret referencia eshte tek fillimi i fakes tjeter, dhe me tutje. Le te jete C_x numri i hereve qe ndodhin ne stringen distance.

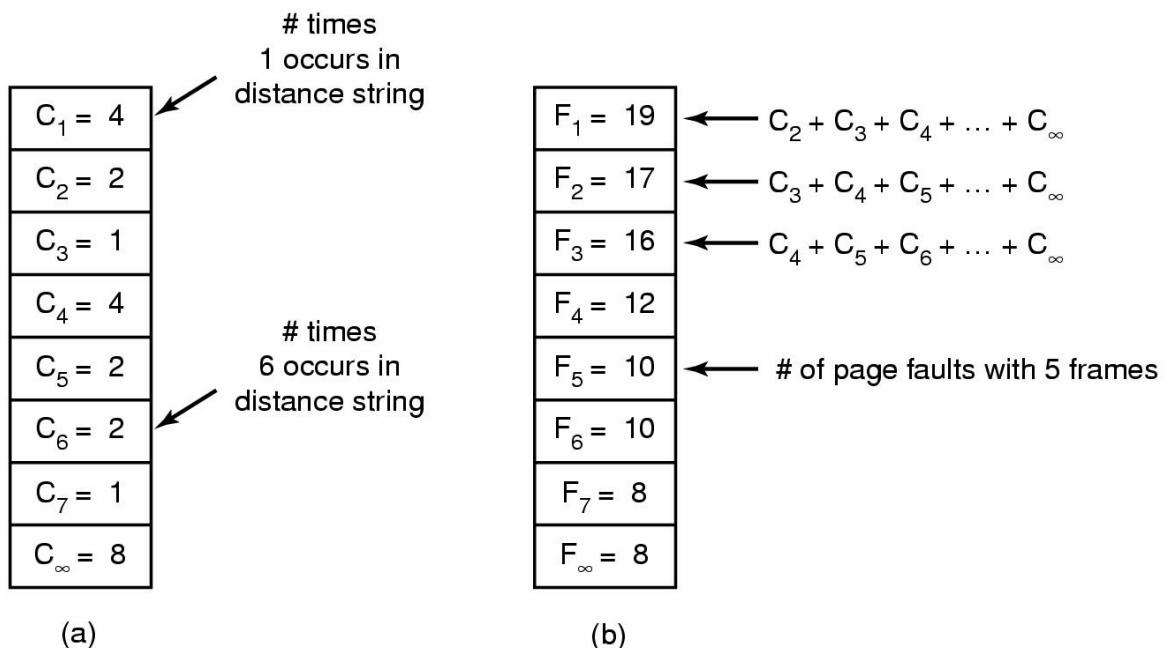


Fig.4-27 Njesimi i mases se page faults nga stringa distance, (a) vektori C, (b) Vektori F

Tani llogarit vektorin / sipas formules..

$$F_m = \sum_{k=m+1}^n C_k + C_\infty$$

Vlera e F_m është numri i page faults qe do të ndodhin ne distancen e stringes se dhene dhe m page frame. Për distancen e stringes të Fig. 4-25, Fig. 4-27(b) jep vektorin F . Për shembull, F është 20, dometheme qe ne memorje ajo ze vetëm 1 page frame, jashtë 24 referencave ne stringe, gjithe page fault përvèc katër qe jane të njëjtë si faqja reference e meparshme.

Për të pare pse kjo formule funksionon, kthehu tek kutia e rende ne fig. 4-25. Le të jetë m numri i page frame ne krye të M. Një page faults ndodh sa here një prishje e stringes distance është $m+1$ ose me shume. Shuma ne formulen e mesipërme përllogarit numrin e

hereve qe ndodhin crregullime. Ky model mund të përdoret për të bere parashikime të tjera. (Maekawa et ah, 1987).

4.6 Projektimi i faqeve të sistemeve.

Ne paragrafet e meparshem u tregua se si funksionojne faqet, u treguan disa nga algoritmet baze të zevendesimit të faqes dhe treguan dhe si ti modelojme faqet.

Për të projektuar një sistem, ju duhet të dini akoma me shume ne menyre qe ai të funksionoje sa me mire. Kjo është e ngjashme me një loje shahu, ku ju duhet të dini mire levizjet e torres, kaloresit, oficerit dhe gureve të tjere ne menyre qe të jeni një lojtar i mire.

Ne paragrafet qe vijne, ne do të shohim disa hapa të tjere qe duhen pasur parasysh me shume kujdes nga projektuesit e sistemeve operative ne menyre qe të kemi një performance sa me të mire të sistemit.

4.6.1 Krahasimi politikave të alokimit local kundrejt atij global.

Ne paragrafet paraprijes ne diskutuam disa algoritma qe bënин të mundur kapjen dhe zevenedsimin e një faqeje kur shkaktohej një gabim. Një ceshtje e rendesishme qe lidhet me ketë problem (të cilën ne e kemi shmangur deri tanë) është se si mund të alokohet memoria gjatë ekzekutimit të proceseve.

Shikoni figuren 4-28(a). Ne ketë figure tre proceset A, B, C Jane procese të ekzekutueshme. Supozojme se procesi A ka një gabim faqeje (page fault). A do të mundet algoritmi i zevendesimit të faqes të gjeje faqen qe është përdorur e fundit duke marre ne konsiderate vetëm 6 faqet qe jane alokuar për momentin tek A, ose duke i konsideruar të gjitha faqet ne memorie? Ne figuren 4-28(a), në qoftë se se shohim vetëm faqet A faqja me moshen (age) me të vogel është ajo me vlerë A5, keshtu kemi situatën si ne figuren 4-28(b).

Nga ana tjetër në qoftë se faqja me moshen me të ulet është hequr (removed) pa marre parasysh vleren e saj, atëherë do të merret faqja B3 dhe do të kemi situatën si ne figuren 4-28(c). Algoritmi ne figuren 4-28(b) quhet edhe algoritmi lokal i zevendesimit të faqeve, kurse algoritmi i figures 4-28(c) quhet algoritmi global i zevendesimit të faqes. Algoritmat lokal janë efektive kur alokojnë çdo proces ne një zone të caktuar të memories, kurse algoritmat globale alokojnë page frames ne menyre dinamike gjatë ekzekutimit të proceseve. Keshtu numri i faqeve qe i përket çdo procesi varion ne kohe.

Figure 4-28 consists of three tables labeled (a), (b), and (c). Each table has a column labeled "Age" on the left.

- (a) Konfigurimi fillestar:** Shows pages A0 through A9 and B0 through B6. The "Age" values are: A0 (10), A1 (7), A2 (5), A3 (4), A4 (6), A5 (3), B0 (9), B1 (4), B2 (6), B3 (2), B4 (5), B5 (6), B6 (12), C1 (3), C2 (5), C3 (6).
- (b) zevendesimi local i faqes:** Shows pages A0 through A6 and B0 through B6. The "Age" values are: A0 (10), A1 (7), A2 (5), A3 (4), A4 (6), A6 (3), B0 (9), B1 (4), B2 (6), B3 (2), B4 (5), B5 (6), B6 (12), C1 (3), C2 (5), C3 (6). The page A6 is circled.
- (c) zevendesimi global i faqes:** Shows pages A0 through A5 and B0 through B6. The "Age" values are: A0 (10), A1 (7), A2 (5), A3 (4), A4 (6), A5 (3), B0 (9), B1 (4), B2 (6), B3 (2), B4 (5), B5 (6), B6 (12), C1 (3), C2 (5), C3 (6). The page A5 is circled.

Figura4-28. Zevendesimi local kundrejt atij global. (a) Konfigurimi fillestar. (b) zevendesimi local i faqes. (c) zevendesimi global i faqes.

Zakonisht, algoritmat globale punojne me mire, ne vecanti kur hapesira e punes (working set) ndryshon gjatë gjithe kohes se proçesit. Në qoftë se përdorim një algoritem lokal dhe hapesira e punes rritet, “goditja” (thrashing) do të shfaqet në qoftë se kemi faqe të lira. Në qoftë se hapesira e punes zvogelohet, algoritmat lokal humbasin memorie. Në qoftë se përdorim një algoritem global, sistemi duhet të vendose ne menyre të vazhdueshme se sa faqe duhet ti caktoje çdo proçesi. Një menyre është të monitorojme hapesiren e punes nepërmjet biteve “aging”, por kjo menyre nuk e parandalon plotësisht “goditjen” (thrashing).

Një menyre tjetër është të kemi një algoritem për alokimin e faqeve të proceseve. Sipas ketij algoritmi përcaktohet periodikisht numri i proceseve qe po ekzekutohen dhe alokohet çdo proçes ne menyre të barabartë. Keshtu në qoftë se kemi 12.416 faqe dhe 10 proçese, çdo proçes merr 1241 faqe kurse 6 faqet e fundit përdoren kur shkaktohen page fault (gabime ne faqe).

Megjithese kjo metode nga njëra ane duket e mire (ndarja e faqeve ne menyre të barabartë për çdo proçes), nga ana tjetër nuk bën dot një dallim midis hapesires qe zene proçeset, për shembull ndermjet një proçesi 10-KB dhe 300-KB (i konsideron proçeset si të barabartë ne hapsire memorieje). Proçesi 300-KB është 30 here me i madh se ai 10-KB. Ne ketë menyre është me mire qe ti jepet çdo proçesi një numer minimal faqesh qe ai të ekzekutohet, pa u shqetësuar se sa i vogel mund të jetë proçesi. Ne disa makina, një instruksioni të vetëm me dy operative (burim dhe destinacion) mund ti duhet rrerh 6 faqe sepse vetë instruksioni, operandi burim dhe operandi destinacion mund të ndryshojne

kufijtë e faqes. Me një alokim prej vetëm 5 faqesh instrukzionet qe ndodhen ne një program nuk do të ekzekutohen dot.

Ne qoftë se përdorim një algoritem global, është e mundur të fillojme çdo proces me një numer të caktuar faqesh i cili është proporcional me madhesine e procesit, por alokimi duhet të beje update ne menyre dinamike apo procesi ekzekutohet. Një menyre për të menaxhuar alokinin është përdorimi i algoritmit **PFF** (Page Fault Frequency). Ky algoritem tregon se kur duhet të zmadhojme apo të zgjelojme alokinin e faqes se procesit por nuk na tregon se cilen faqe duhet të zevendesojme kur shkaktohet një gabim. Ky algoritem kontrollon vetëm madhesine e alokitit.

Për një klase të madhe algoritmash të zevendesimit të faqes, duke përfshire edhe LRU, është pare qe sa me i madh të jetë numri i faqeve qe alokohet aq me shume ulet mundesa e ndodhjes se gabimit. Kjo tregohet me ane të grafikut të figures 4-29.

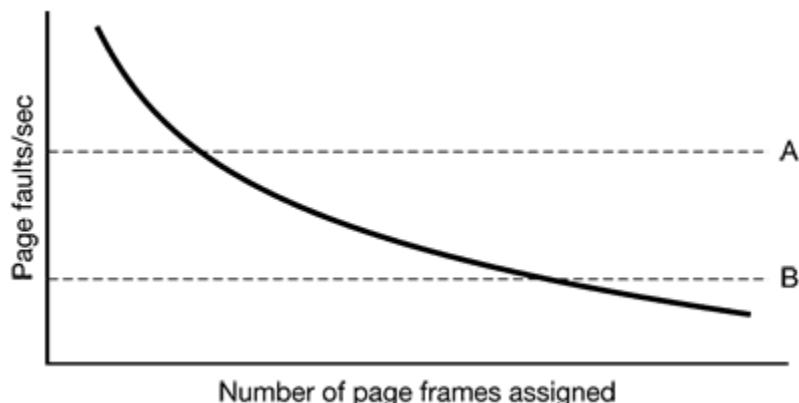


Figura 4-29. Gabimi i faqes ne funksion të numrit të faqeve

Për të matur vleren e gabimit të faqes duhet të numerojme numrin e gabimeve qe ndodhin për sekonde. Vija A, me nderprerje, qe paraqitet ne grafik, paraqet një vlere të lartë të gabimit të faqes, prandaj algoritmi rrit numrin e faqeve për të ulur vleren e gabimit. Vija B, paraqet një vlere të ulet të gabimit, por ne ketë rast arrijme ne përfundimin qe procesi ka marre një hapesire me të madhe memorieje. Prandaj numri i faqeve duhet zgjidhur i tille qe vlera e gabimit dhe memoria qe okupon procesi të jetë sa me e vogel (kurba midis vijes A dhe B), gje e cila kontollohet nepërmjet PFF qe mundohet të ruaje një normalitet të caktuar.

Është e rendesishme të theksojme se algoritmat e zevendesimit të faqes mund të përdorin njëren nga dy politikat: politiken e zevendesimit lokal ose politiken e zevendesimit global. Për shembull, FIFO mund të zevendesoje faqen e vjetër nga kujtesa (algoritm global) ose mund ta zevendesoje atë nga procesi qe e ka ne zotërim (algoritem local). Keshtu zgjedhja e politikes local apo global varet nga algoritmi dhe menyra se si funksionon ai.

4.6-2 Kontrolli i ngarkeses

Megjithese mund të kemi algoritmin me të mire për zevndesimin e faqes dhe një alokim global të përshtatshem për faqet e proceseve, mund të ndodhe qe sistemi të “goditet” (thrashes). Ne fakt, sa here qe hapesira e punes e të gjithe proceseve kalon kapacitetin e memories, “goditja” është e pa evitueshme. Një simptome e kesaj situate shfaq edhe algoritmi PFF i cili tregon se disa procese kane nevoje për me shume memorie dhe asnjëri prej proceseve të tjera nuk kerkon me pak memorie. Ne ketë situatë nuk ka rruge tjetër përvecse ti jepet memorie proceseve qe kerkojne me shume duke demtuar disa të tjera. Një zgjidhje e ketij problem është shpetimi i përkohshem i proceseve.

Për të reduktuar numrin e proceseve qe konkurojne me njëri tjetrin për memorie behet një **swap** (shkembim), duke i derguar disa nga proceset ne disk ne menyre qe të lironen faqet qe ata kishin zene. Keshtu për shembull në qoftë se një process dergohet ne disk, atëherë faqet qe ai kishte zene ne memorie do të ndahen midis proceseve të tjera, qe kane nevoje për me shume memorie. Në qoftë se kerkesa për memorie plotësohet për të gjitha proceset atëherë sistemi do të vazhdoje të punoje për aq kohe, derisa do të linde një proces i ri i cili kerkon serish memorie. Në qoftë se pas procesit të pare qe beme swap, kerkesa për memorie nuk plotësohet akoma, atëherë një proces i dytë do të kaloje ne disk duke bere prape swap. Kjo do të vazhdoje derisa të përfundojne “goditjet” (domethene derisa të plotësohen kerkesat e proceseve për memorie). Pra nepërmjet procedures swap, të shkembimit të proceseve midis memories dhe diskut, behet i mundur kontrolli i ngarkese ne memorie. Procedura swap behet sipas një plani (schedule) të caktuar. Sapo disa nga proceset mbarojne se ekzekutuari ne memorie, procedura i rikthen proceset nga disku ne memorie.

Një tjetër faktor qe duhet marre ne konsiderate është edhe shkalla e multiprogramimit. Sic e pame edhe ne figuren 4-4, kur numri i proceseve ne memorien kryesore është i vogel, CPU-ja mund të qendroje ne gjendje pritjeje për një periodë të konsiderueshme kohe. Ky faktor tregon qe përvet madhesise se procesit dhe vleres se faqeve, kur vendosim të bejme një proces swap, duhen marre ne konsiderate edhe karakteristikat e procesit si: në qoftë se procesi është i lidhur me CPU apo me pajisjet I/O, si dhe karakteristikat e proceseve të tjere.

4.6.2 Madhesia e faqes

Madhesia e faqes është një parametër i cili zgjidhet nga sistemi operativ. Në qoftë se hardware është projektuar për shembull me faqe 512-byte, sistemi operativ mund ti rendise faqet 0 dhe 1, 2 dhe 3, 4 dhe 5 dhe keshtu me rradhe, si faqe 1-KB duke alokuar gjithmone dy faqe të njëpasnjëshme 512-byte secila.

Duke përcaktuar sa me mire madhesine e faqes, ne balancojme shume faktore konkurues. Si rezultat, nuk ka një kufi të favorshem. Si fillim, ka dy faktore qe argumentojne një faqe të vogel. Një tekst, e dhene ose një stack i zgjedhur ne menyre random (të rastesishme) nuk mbushin një numer të plotë faqesh. Mesatarisht, gjysma e faqes nga fundi do të jetë bosh. Hapesira e vecantë ne ketë faqe është e humbur (waste). Kjo hapesire e humbur quhet fragmentim i jashtëm (internal fragmentation). Në qoftë se kemi n segmentë ne memorie dhe një madhesi faqeje prej p byte, $n * p / 2$ byte do të jene waste (do të humbin) ne fragmentimin e jashtëm. Ky arsyetim përdoret për një faqe me madhesi të vogel.

Një argument tjetër për një faqe me madhesi të vogel behet i dukshem në qoftë se marrim ne konsiderate një program me 8 sekuencia me 4 KB secili. Me një madhesi faqeje 32 KB, programi do të alokoje 32 KB gjatë gjithe kohes. Me një madhesi faqeje 16KB do të alokoje 16 KB. Me një madhesi faqeje 4 KB ose me të vogel kerkon vetëm 4 KB ne çdo moment. Ne përgjithesi, një faqe me hapesire të madhe mund të mbaje me shume programe të papërdorshme ne memorie sesa një faqe me hapesire të vogel.

Nga ana tjetër, duke qene se faqet jane të vogla, bejne qe programet të kerkojne me shume faqe, duke krijuar një page table (tabele faqesh). Një programi 32 KB i duhen katër faqe 8 KB, pothuaj 64 faqe 512 byte. Transferimi midis memories dhe diskut behet me nga një faqe ne çdo transfertë. Keshtu transferimi i një faqeje të vogel apo të madhe kerkon të njëjtën kohe. Duhet $64 * 10$ msec për të ngarkuar 64 faqe 512 byte qe është pothuaj ekuivalentë me $4 * 12$ msec qe duhen për të ngarkuar 4 faqe 8 KB.

Ne disa makina, tabela e faqeve ngarkohet ne registrat e hardware-it çdo here qe CPU-ja kalon nga një proces ne tjetrin. Keto makina duke patur faqe me madhesi të vogel bejne qe koha qe duhet për ngarkimin e regjistrave të faqeve të rritet kur madhesia e faqeve vjen e zvogelohet. Vec kesaj, hapesira qe okupohet nga tabela e faqes vjen e rritet kur madhesia e faqes zvogelohet.

Le ta analizojme ceshtjen me sipër nga ana matematikore. Shenojme madhesine e procesit me s byte dhe madhesine e faqes me p byte. Vec kesaj, supozojme se çdo faqe ne hyrje kerkon e byte. Numri i faqeve qe duhen për një process është s/p byte dhe okupohen $s * e / p$ byte nga tabela e faqes. Memoria wasted është $p/2$ byte. Keshtu shuma e tabeles se faqeve me fragmentimin e jashtëm (waste) shenohet overhead dhe jepet nga shprehja:

$$\text{overhead} = s * e / p + p / 2$$

Termi i pare se/ p (madhesia e tabeles se faqeve) ka vlere të madhe kur madhesia e faqes është e vogel. Termi i dytë, $p/2$ (fragmentimi i jashtëm - waste) ka vlere të madhe kur madhesia e faqes është e madhe. Duke derivuar shprehjen e mesipërme sipas p dhe duke e barazuar me zero marrim ekuacionin:

$$-se/p^2 + 1/2 = 0$$

Prej ketij ekuacioni ne mund të nxjerrim formulen e përshtatshme për të përcaktuar madhesine e faqes (duke konsideruar vetëm memorien waste dhe page table). Keshtu kemi:

$$p = \sqrt{2se}$$

Për $s = 1\text{MB}$ dhe $e = 8$ byte madhesia e faqes do të jetë 4 KB.

4.6.3 Vecimi i hapesirave të instruksioneve dhe të dhenave.

Pjesa me e madhe e kompjuterave kane një single address space (hapesira e adreses) e cila mban programe (instruksione - I) dhe të dhenat – D, sic tregohet ne figuren 4-30(a). Në qoftë se hapesira e adreses është e mjaftueshme, çdo gje do të punoje mire. Ne të kundert në qoftë se ajo është shume e vogel, do ti detyroje programuesit ti qendrojne mbi koke për të vendosur çdo gje ne hapesiren e adresave.

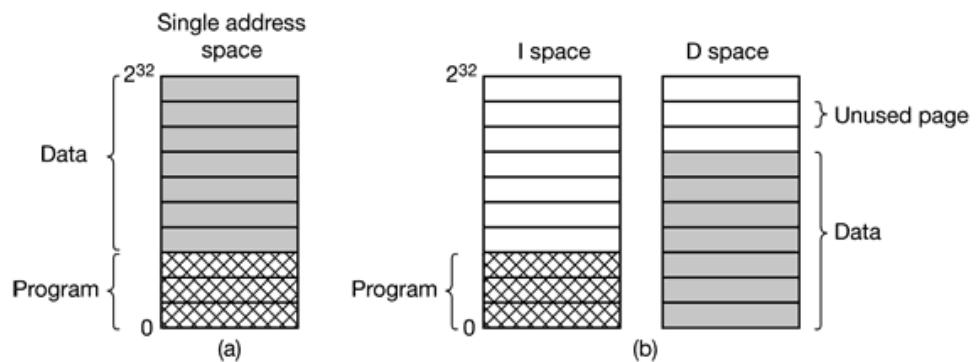


Figure 4-30. (a) One address space (b) Separate I and D spaces.

Një zgjidhje është qe të ndajme hapesiren e adresave nga hapesira e instruksioneve. Keto i shenojme me I-space për instruksionet dhe me D-space për të dhenat. Hapesira e adresave fillon nga zero 0 deri ne max i cili mund të jetë $2^{16} - 1$ ose $2^{32} - 1$, sic tregohet edhe ne figuren 4-30(b). Ne një kompjuter me ketë projektim, të dyja hapesirat e adresave mund të faqosen (paged), pavaresisht nga njëra tjetra. Secila faqe ka tabelen e vetë, me të dhenat e saj nga faqet virtual deri tek faqet fizike qe jane page frames. Kur hardware do të térheq (fetch) një instruksion, ai e di qe duhet të përdor I-space dhe tabelen e I-space.

Ne menyre të ngjashme, në qoftë se do të transferoje të dhena ai do ti drejtohet tableles D-space. Ne ketë menyre, duke përdorur dy adresa me hapesira të ndryshme, një për instrukzionet I-space dhe një për të dhenat D-space, ulet kompleksiteti i adresimit dhe shfrytëzohet me mire memoria e kompjuterit.

4.6.4 Ndarja e faqeve

Një tjetër problem ne projektimin e faqeve është dhe ndarja (share). Ne një sistem të madhe multiprogramimi shpesh here ndodh qe disa përdorues të përdorin të njëjtin program, ne të njëjtin cast të kohes. Keshtu është me eficientë të ndajme faqet dhe të shmangim dy kopje të një faqeje ne memorie ne të njëjtin cast të kohes. Por një problem tjetër është se jo të gjitha faqet mund te ndahen. Ne vecanti, faqet qe jane vetëm të lexueshme (read only), si tekstet e programeve mund të ndahen, kurse faqet e të dhenave (data) nuk ndahen.

Në qoftë se ndarja e hapesirave I-space dhe D-space është e mundur, atëherë është e lehtë të ndajme programet qe kane dy ose me shume procese dhe përdorin të njëjtën faqe ne I-space, dhe ato qe kane faqe të ndryshme ne D-space. Zakonisht, ne një implementim qe mundeson ndarjen ne ketë menyre, tabelat e faqeve janë struktura të dhenash të pavarura nga tabela e proceseve. Çdo proces ka dy pointerë ne tabelen e proceseve: njëri ndodhet ne tabelen I-space dhe tjetri ne tabelen D-space, sic tregohet ne figuren 4-31.

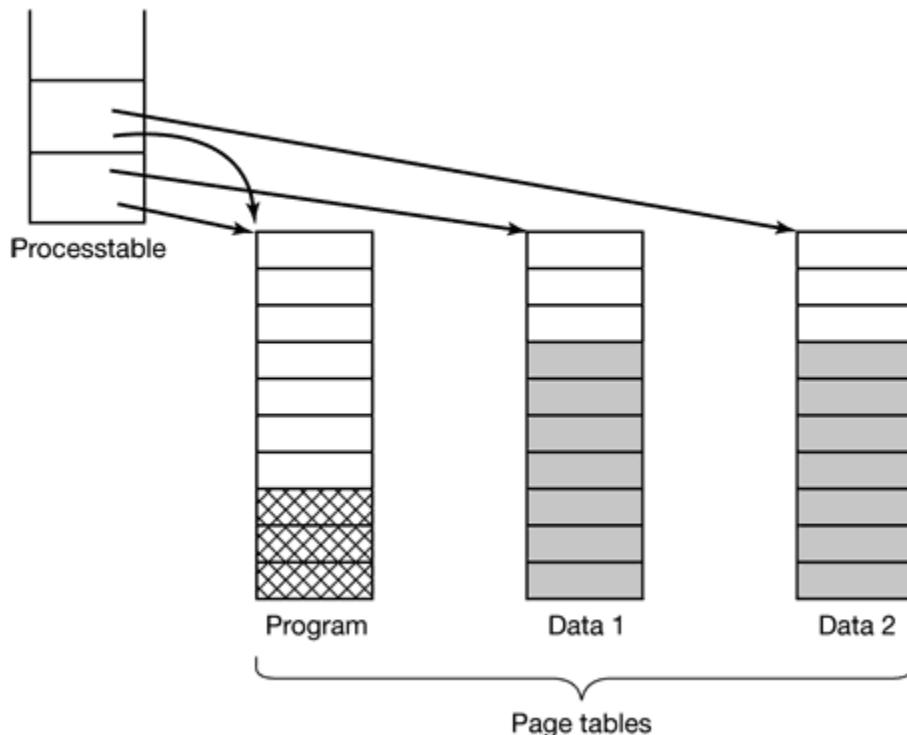


Figura 4-31. Dy procese të cilat ndajne të njëtin program, ndajne edhe tabelen e faqeve.

Kur skedulimi zgjedh një proces për tu ekzekutuar, përdor pointerat për të lokalizuar tabelat e faqes dhe therret MMU për ti përdorur ato. Në qoftë se nuk do të kishim një ndarje të hapesirave I dhe D, proceset mund të ndanin programet (libraritë), por mekanizmi do të ishte shume i komplikuar.

Kur dy ose me shume procese ndajne disa kode, atëhere shfaqet një problem me ndarjen e faqeve. Supozojme se proceset A dhe B jane duke përdorur të dy editorin dhe duke ndare faqet e tyre. Në qoftë se skeduleri kerkon të fshij procesin A nga memoria, atëhere ai debon të gjitha faqet dhe mbush të gjitha page framet bosh me programe të tjera, duke bere qe procesi B të gjeneroje një numer të madh gabimesh ne faqe (page faults), për ti risjelle ato serisht ne memorie.

Ne menyre të ngjashme, kur procesi A përfundon, është e rendesishme të dime ne se faqet jane duke u përdorur akoma, ne menyre qe hapesira e tyre ne disk të mos boshatiset nga ndonjë gabim. Kerkimi i të gjitha tabelave të faqes për të pare në qoftë se ndonjë faqe mund te ndahet, është tepër i kushtueshem, keshtu struktura speciale të të dhenave nevojiten për të mbajtur drejtimin e faqeve qe ndahan.

Ndarja e të dhenave është e ndryshme nga ndarja e kodeve, por kjo nuk është shume e rendesishme. Ne vecanti, ne UNIX, pas një thirrjeje sistem “fork”, babai dhe femija duhet të ndajne të dyja: program tekst dhe data. Ne një sistem, ajo c’ka duhet bere është ti jepet secilit prej proceseve tabela e tij e faqes dhe të kene të dy proceset të njëtin grup faqesh. Keshtu nuk nevojitet asnjë kopje e faqeve ne kohen “fork”. Megjithatë, të dyja faqet e të dhenave jane mapped ne të gjitha proceset si READ ONLY (vetëm të lexueshme).

Për sa kohe qe të dy proceset lexojne të dhenat e tyre pa i modifikuar ato, kjo situatë do të vazhdoje. Sapo ndonjëri prej proceseve i bën update një fjale memorije, atëhere shkaktohet një “trap” (kurth) ne sistemin operativ. Nderkohe është bere një kopje e faqes, keshtu çdo process ka kopjen e tij private. Të gjitha kopjet jane bashkuar si READ-WRITE, keshtu qe shkrimet pasuese ne secilen prej kopjeve behet pa shkaktur një “trap”. Kjo strategji nenkupton qe ato faqe qe nuk Jane shkruajtur asnjeherë (duke përfshire të gjitha faqet e programit) nuk kane nevoje të kopjohen. Vetëm faqet e të dhenave qe Jane shkruajtur aktualisht kane nevoje të kopjohen. Kjo menyre quhet **copy on write**, rrit performancec duke reduktuar kopjimin.

4.6.5 Cleaning Policy

Faqet punojne mire kur ato kane shume page frame të lira qe mund të deklarohen si shkaktare të gabimeve ne faqe. Në qoftë se çdo page frame është plot dhe vec kesaj

modifikohen, përapara se një faqe e re të sillet, një faqe e vjetër duhet të ruhet ne disk. Për të siguruar hapesire të madhe me page frames të lira, shume faqe sistemi kane një fushe proçesesh e cila quhet **paging daemon**. Paging daemon fle (është jo aktive) pjesen me të madhe të kohes dhe zgjohet (behet active) periodikisht për të inspektuar gjendjen e memories. Në qoftë se shume page frame jane të lira, paging daemon fillon selektimin e faqeve për të evituar përdorimin e algoritmit të zevendesimit të faqes, në qoftë se keto faqe jane modifikuar kur jane ngarkuar atëhere ato patjetër qe jane ruajtur ne disk.

Në qoftë se ndonjë nga faqet e nxjerra nevojitet serisht përparrë se frame i faqes të jetë mbishkruajtur, ajo mund të korrigohet duke e hequr atë nga zona e page frame-ve të lira. Mbajtja e një fushe rezerve me page frames rrit performance dhe eviton përdorimin e gjithe memories dhe kerkimin e një frame ne momentin qe duhet.

Një nga menyrat për të implementuar ketë cleaning policy është edhe ora me dy-shigjeta. Shigjeta e përparme kontrollohet nga paging daemon. Kur ajo pointon ne një faqe të mbushur, kjo faqe shkruhet ne disk dhe shigjeta e përparme vazhdon përparrë për të pointuar ne faqen e rradhes. Kur ajo pointon ne një faqe të bardhe (boshe) ajo vazhdon serisht përparrë dhe nuk kryen asnjë veprim me faqen. Shigjeta e prapme përdoret për zevendesimin e faqes, tamam si ne algoritmin standart të ores.

4.6.6 Nderfaqja e Memories Virtuale

Deri tani ne kemi diskutuar se memoria virtual është transparente ndaj proçeseve dhe programuesve. E gjitha cfare shohim është një hapesire e madhe adresash virtual ne një kompjuter me memorie fizike të vogel. Kjo është e vertetë për shume sisteme, por ne disa sisteme të avancuara, programuesit kane disa kontolle mbi hartën e memories dhe mund ta përdorin atë ne një menyre jo tradicionale për të rritur sjelljen e programeve.

Një nga shkaqet qe programuesit duhet të marrin kontrollin mbi hartën e memories është qe të lejojne dy ose me shume proçese të ndajne të njëjtën memorie. Në qoftë se programuesit mund të emertojne zonat e memories, është e mundur qe një proçes ti jape një proçesi tjetër emrin e një zone dhe qe ky proçess të mund të vendoset ne të. Me dy (ose me shume) proçese qe ndajne të njëjtat faqe përftohet një gjeresi brezi e lartë, ku njëri proçes shkruan ne memorien e ndare dhe tjetri lexon prej saj.

Ndarja e faqeve përdoret gjithashtu edhe për të implementuar një sistem me mesazhe-kaluese (message-passing) me performance të lartë. Normalisht, kur mesazhet kane ardhur, të dhenat jane kopjuar nga një adresë ne një tjetër, me një kosto të konsiderueshme. Në qoftë se proçeset do kontollonin hartën e faqes se tyre, një mesazh do të kalonte duke patur proçesin e derguar të pa vene ne hartën e faqes qe përmban mesazhin, dhe proçesin e marrjes të vene ne të. Ketu vetëm emrat e faqeve do të kopjohen, ne vend qe të kopjohet të gjithe të dhenat.

Një teknike tjetër e avancuar e menaxhimit të memories është **distributed shared memory** (Feely ne 1995; Li, 1986; Li dhe Hudak, 1989; dhe Zekauskas ne 1994). Ketu idea është qe të lejohen shume procese ne një network të ndajne mundesisht një bashkesi fakesh (ky nuk është kusht i nevojsphem), duke përdorur një linjë të vetme adresash. Kur një proces i referohet një faqeje, e cila nuk gjendet ne hartë, atëhere shkakohet një page fault (gabim ne faqe). Mbajtesi i page fault – it mund të gjendet ne kernel ose ne hapesiren user, pasi lokalizohet makina mbajtëse e fakes i dergohet asaj një mesazh qe i kerkon për të hequr faqen dhe dergohet faqja ne network. Kur faqja arrin, ajo vendoset ne hartë dhe instruksioni i gabimit ristartohet. Ne do të shikojme me me hollesi distributed shared memory ne kapitullin 8.

4.7 QELLIMET E IMPLEMENTIMIT

Implementuesit e sistemeve të memorjes virtuale duhet të zgjedhin midis algoritmeve teorike me kryesore si një rast i dytë kunder kohes, shpërndarjen e faqeve lokale kunder faqeve globale. Ata gjithashtu duhet të jene ne dijeni për një numer qellimesh implementimi. Ne ketë seksion do i hedhim një shikim disa problemeve dhe disa zgjidhjeve.

4.7.1 SISTEMI OPERATIV I PËRFSHIRE ME ‘PAGING’

Sistemi operativ kur është ne pune e përdor katër here ‘paging’: krijimi i procesit, ekzekutimi, koha e gabimit dhe mbarimi i procesit. Tani do i ekzaminojme të gjitha.

Kur një proces i ri krijohet ne sistemin ‘paging’, Sistemi operativ duhet të dije sa i madh do jetë programi dhe të dhenat qe të krijoje një tabele për të. Duhet të vendoset një hapesire ne memorje për faqen e tableles dhe duhet të fillohet. Kur procesi ekzekutohet, tabela e fakes duhet të jetë ne memorje. Hapesira duhet të vendoset ne një sipërfaqe të ndryshme të diskut qe kur të ndryshohet faqja të ketë ku të shkoje. Sipërfaqja e ndryshuar duhet të fillohet me një program tekst dhe të dhena qe kur procesi i ri të filloje të kape ‘page faulting’, faqet të hyjne brenda nga disku. Perfundimisht, informacioni rrëth tabelave të fakes dhe sipërfaqes se ndryshueshme ne disk duhet ta kemi ne tabelen e procesit.

Kur një process jepet për ekzekutim, MMU-ja duhet të risetohet për procesin e ri dhe TLB-ja të mbushet qe të jetë ne dijeni të procesit qe po ekzekutohej me pare. Tabela e fakes se procesit të ri duhet të behet paresore duke e kopjuar atë ose duke e shenjestruar

ne një regjistër hard-i. Zakonisht disa ose të gjitha fajt e procesit mund të sillen ne memorje për të ulur numrin e gabimeve fillestare.

Kur ndodh një page fault; SO duhet të lexoje ne regjistrat hardware, të gjeje se cila adrese virtuale e shkaktoi fault. Nga ky informacion, duhet të gjeje cila faqe është e nevojshme dhe ta vendose atë faqe ne disk. Pastaj, duhet të gjeje një page frame të disponueshme qe të vendose fajen e re, ndoshta dhe duke hequr disa faqe te vjetra ne se do te jete e nevojshme.

Pastaj duhet të lexoje fjalen e nevojshme ne page frame. Ne fund, duhet të beje një kthim mbrapshët te numruesit të programit qe të përqendrohet te instruksioni i gabimit dhe ta ekzekutoje ate perseri.

Kur një proce ekziston, SO duhet të liroje tabelen e fakes se tij, fajen e tij dhe hapesirene disk qe nxe faqja kur ato jane ne disk. Në qoftë se disa faqe jane të share-uara me procese te tjera, fajt ne memorje dhe ne disk mund të lironen vetëm kur procesi i fundit qe i përdore ato ka përfunduar.

4.7.2 FUNKSIONI I PAGE FAULT.

Me ne fund jemi ne një pozicion qe mund të përshkruajme cfare ndodh ne një ‘page fault’ ne detaje; Sekuенca e ngjarjeve është;

1. Pajisja bie ne kurthin e kernelit, duke e ruajtur numeruesin e programit ne stack. Ne shumicen e makinave, disa informacione rreth gjendjes se instruksionit të tanishem memorizohet ne regjistrat speciale të CPU-se.
2. Një kod i përbashket leshohet të regjistroje regjistrat e përgjithshem dhe informacione të nevojshme, qe SO te mos i shkaterroje ato. Kjo rutine therret sistemin operativ si një procedure.
3. SO zbulon qe ka ndodhur një page fault dhe mundohet të gjeje se cila faqe virtuale nevojitet. Zakonisht ky informacion mbahet ne njërin nga regjistrat hardware. Ne të kundert SO duhet të térheq numruesin e programit, të ngarkoje instruksionin dhe ta përshkruaje ne soft, qe të gjeje cfare po bënte kur ndodhi fault.
4. Kur adresa virtuale qe shkaktoi fault dihet, procesi kontrollon në qoftë se adresa ishte e rregullt. Në qoftë se jo, procesi dergon një sinjal për ta zhdukur atë. Në qoftë se adresa është e rregullt, sistemi kontrollon ne se korniza e fakes është e lire. Ne se s’ka korniza të lira algoritmi i zhvendosjes se fakes duhet të zgedhe një victim.
5. Ne se korniza e fakes se selektuar është pis, faqja vendoset për transferim ne disk dhe një switch cfaredo vendoset ne vend të saj, derisa të përfundoje

- transferimi. Ne cdo rast, frame shenohet si e zene, qe të mos përdoret për ndonjë qellim tjetër.
6. Kur pastrohet korniza e faqes, SO shikon sipër adreses se diskut ku ndodhet faqja e nevojshme dhe skedulon diskun qe do ta sjelle atë. Kur faqja po ngarkohet procesi i faulting është akoma i pezulluar dhe një tjetër proces ekzekutohet në qoftë se ai punon.
 7. Kur interrupti i diskut tregon qe faqja ka ardhur, tabelat e faqes jane update-uar të reflektojne pozicionin e tij dhe korniza është vendosur ne gjendje normale.
 8. Instruksioni i faulting është vendosur ne gjendjen qe kishte kur filloi dhe numruesi i programit është risetuar qe të përqendrohet te instruksioni.
 9. Procesi ‘faulting’ është vendosur dhe SO kthehet të rutina e gjuhes assemblér qe e therriti atë.
 10. Kjo rutine ringarkon regjistrat dhe informacione të tjera të gjendjes, dhe kthehet të hapsira e userit të vazhdoje ekzekutimin, në qoftë se nuk ndodh një fault tjetër.

4.7.3 INSTRUKSIONI BACKUP

Kur një proces i referohet një faqeje qe nuk është ne memorje, instruksioni qe shkakton ‘fault’ ndalon dhe ndodh një kurth te sistemi operativ. Pasi SO ka ngarkuar faqen e nevojshme, duhet të ristartoje instruksionin qe shkakton kurthin. Kjo është dicka e thjeshtë për tu thene. Per te pare natyren e ketij problemi nga ana negative e saj, shqyrtojme një CPU qe ka instruksione me dy adresa, si Motorola 680x0, të përdorura gjeresisht ne sistemet e nderfutura. Per shembull, instruksioni:

MOVE.L #6(A1), 2(A0)

ka 6 byte. Shiko fig 4-32. Qe të ristartosh sistemin, SO duhet të percaktoje se ku do të vendoset byte i pare i instruksionit. Vlera e PC ne kohen e kurthit varet se cili operand ka deshtuar dhe se si Jane implementuar mikrokodet e CPU.

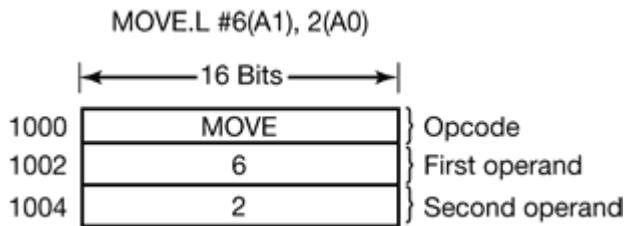


Fig 4-32- një ton instruktiv qe shkakton page fault

Ne fig 4-32 kemi një instruksion qe fillon të adresa 1000, qe perfshin tre referenca memorje: vetë fjalen e instruksionit dhe dy offset për operandet. Duke u varur të secila nga keto tre referenca shkaktoi ‘page fault’. PC mund të jetë 1000, 1002 ose 1004 ne kohen e gabimit. Është e pamundur për SO të përfundoje pikerisht aty ku filloj instruksioni. Në qoftë se PC është 1002 ne kohen e gabimit, SO nuk ka rruge për të treguar ne se fjala ne 1002 është adrese memorje e lidhur me një instruksion të 1000 apo një instruksion ‘opcode’.

Ky problem mund të kish shkuar dhe me keq. Disa tipe adresash 680x0 përdorin autoinkrementim, e cila do të thotë qe një efekt anesor i ekzekutimit të instruksionit është të inkrementosh një ose me shume regjistra. Instruksionet qe përdorin autoinkrementim mund të deshtojne. Ne varesi të detajeve të mikrokodit, inkrementimi mund të behet para referencave të memorjes, ku SO duhet të dekrementoje regjistrin ne soft para se të ristartoje instruksionin. Autoinkrementimi mund të behet edhe pas referencave të memorjes ku nuk mund të behet ne kohen e kurthit dhe nuk mund të kthehet nga SO. Ekziston edhe autodekrementimi i cili shkakton të njëjtin problem.

Detajet precise të autoinkrementimit dhe autodekrementimit qe jane ose jo para referencave të memorjes korresponduese, mund të ndryshojne nga instruksioni ne instruksion dhe nga një model e CPU-je të një tjetër.

Fatmiresisht, ne disa makina, dizenjuesit e CPU-se i japin një zgjidhje, zakonisht ne formen e një regjistri të fshehur ne te cilin PC është e kopjuar para se të ekzekutohet çdo instruksion. Keto makina mund të kene dhe një regjistër dytësor qe u tregon se cili regjistër është për momentin autoinkrementuar ose dekrementuar dhe me sa. Pasi jepet ky informacion, SO i kthen nga e para të gjitha efektet e instruksionit “fault”, keshtu qe mund të fillohet e gjitha nga e para. Në qoftë se ky informacion nuk është i disponueshem, SO duhet të sforcohet të zbulojë se cfare ndodh dhe si ta rregulloje atë, mendohet se dizenjuesit e pajisjeve hardwareishin të paaftë të zgjidhin problemin, keshtu qe ata hoqen dore dhe ja lane perdorueseve të SO të meren me të.

4.7.4 KYCJA E SISTEMEVE TË MEMORJES

Dhe pse nuk i kemi diskutuar I/O ne ketë kapitull, fakti qe një kompjuter ka një memorje virtuale s'do të thotë qe I/O mungon. Memorja virtuale dhe I/O nderveprojne ne rruge të organizuar.

Konsidero një proces qe ka si qellim një thirrje system, për te lexuar nga disa skedare ose pajisje brenda ne një buffer me hapesire e adreses se saj. Ndersa pret qe I/O të përfundoje, procesi pezullohet dhe një tjetër process fillon ekzekutimin. Ky process tjetër merr një ‘page fault’.

Në qoftë se algoritmi i faqes është global, është një shans i vogel, por jo zero qe faqja qe përmban buffer-at I/O të zgjidhet të hiqet nga memorja. Në qoftë se një pajisje I/O për momentin ndodhet ne proces duke bere një transfertë DMA të asaj faqe, ta heqesh atë shkakton qe një pjese e të dhenave të shkriven ne buffer-in qe I perkasin dhe një pjesa tjetër të shkriven mbi faqet e reja të ngarkuara. Një zgjidhje e problemit është kycja e faqeve të zena me I/O ne memorje. Keshtu qe ato nuk do të hiqen. Një zgjidhje tjetër është qe të gjitha I/O te behen ne kernel buffer dhe pastaj të kopojme të dhenat ne faqet e përdorura.

4.7.5 BACKING STORE

Ne diskutim pér rivendosjen e algoritmeve të faqes pame se si një faqe selektohet pér heqje. Nuk kemi thene se ku vendoset ne disk pasi nxirret jashtë. Le të përshkruajme disa nga qellimet ne lidhje me menaxhimin e diskut.

Algoritmi me i thjeshtë pér të shpérndare hapesirat e faqes ne disk është qe ne disk duhet të kesh një hapesire të vecantë. Kur operon sistemi, kjo sipërfaqe është bosh dhe përfaqesohet ne memorje si një hyrje e vetme, duke i dhene origjinën dhe masen e saj. Kur fillon procesi i pare, një pjese e sipërfaqes se procesit të pare rezervohet dhe pjesa tjetër zbritet me atë njësi. Kur fillojne procese të reja u caktohen sipërfaqe të reja, të njëjtë ne madhesi me imazhet e tyre. Kur ato mbarojne hapesira ne disk është e lire.

Adresa e sipërfaqes se diskut qe mbahet ne tabelen e procesit është e lidhur me secilin proces. Para se një proces të filloje, duhet të inicializohet sipërfaqja. Një rruge është të kopjosh të gjithe imazhin e procesit ne sipërfaqe, keshtu qe mund ta marim ashtu si duhet. Menyra tjetër është të ngarkojme të gjithe procesin ne memorje dhe të shfletohet ashtu si nevojitet.

Megjithatë ky model i thjeshtë ka një problem sepse proceset mund të zgjerohen pasi të fillojnë. Edhe pse programi zakonisht është i fiksuar, të dhenat nganjehere rriten dhe pila rritet gjithmone. Megjithatë do ishte me mire të rezervonim sipërfaqe të ndara pér tekstin, të dhenat dhe pilen dhe ti lejojme seciles nga keto sipërfaqe, qe të përbaje me shume se një pjese ne disk.

Ekstremi tjetër është të mos shpërndaje asgje me përparesi, dhe një hapesire disku për secilen faqe kur ndryshohet. Keshtu proçeset ne memorje nuk marrin ndonjë sipërfaqe të ndryshueshme. E keqja është se një adresë disku nevojitet ne memorje të mbaje pjese të seciles faqe ne disk. Me fjale të tjera duhet një tabelë për proçes për të treguar për secilen faqe se ku ndodhet ne disk. Dy alternativat jane treguar ne fig.4.33

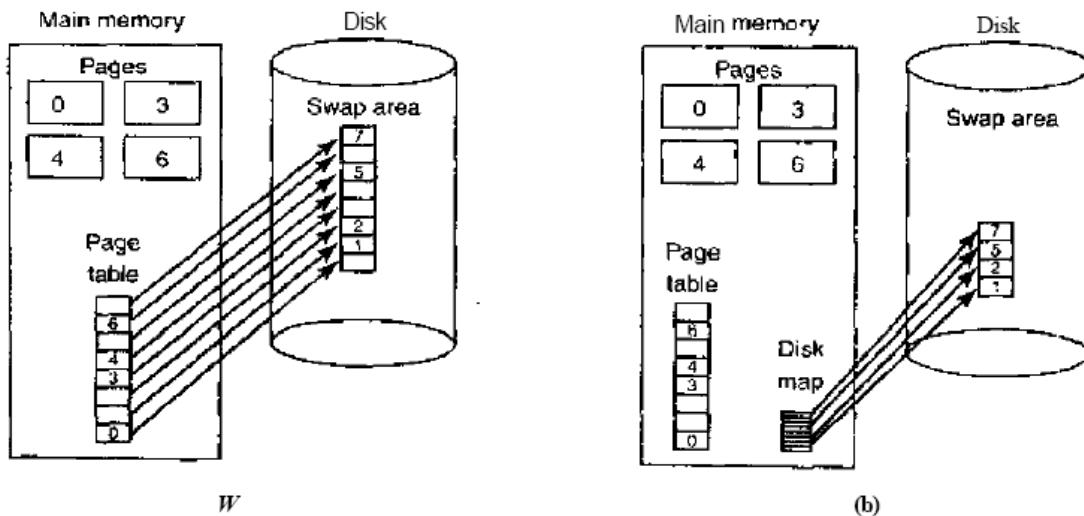


Fig. 4-33

Ne fig 4-33 (a) është ilustruar një tabelë me 8 faqe. Faqet 0, 3, 4 dhe 6 Jane ne memorjen kryesore. Faqet 1, 2, 5 dhe 7 Jane ne disk. Sipërfaqja ne disk është po aq e madhe sa sipërfaqja e adreses virtuale të proçesit (8 faqe), ku secila faqe ka një vendndodhje fikse e cila shkruhet kur hiqet nga memorja kryesore. Për të llogaritur ketë adresë duhet të dime se ku fillon sipërfaqja ‘paging’ e proçesit meqë faqet registrohen ne te vazhdimisht, sipas numrave te faqeve virtuale te tyre. Faqet qe Jane ne memorje e kane gjithmone një kopje (ne hije) ne disk, por kjo kopje nuk funksionon në qoftë se faqja modifikohet kur ngarkohet.

Të fig 4-33(b) faqet nuk kane adresë fikse ne disk. Kur një faqe shpërndahet, zgjidhet një faqe disku boshe dhe harta e diskut (e cila ka një dhome për një adresë disku) behet “update” ne varesi të saj. Faqet ne memorje nuk kane kopje ne disk. Hyrjet e tyre ne hartën e diskut përbajne një adresë disku jo te rregullt ose një bit qe i tregon ato sikur nuk Jane ne përdorim.

4.7.6 NDARJA E PLANIT DHE E MEKANIZMIT

E rendesishme për menaxhimin e kompleksitetit të një sistemi është të ndash planin nga mekanizmi. Ky princip mund të aplikohet të menaxhimi i memorjes duke patur ekzekutimin e memories si një proces të nivelit përdorues. Një ndarje e tille fillimisht u bë ne Mach (Young et al,1987) dhe ne MINIX (Tanenbaum,1987). Përshkrimi me poshtë bazohet ne Mach.

Një shembull i thjeshtë se si mund të ndahen plani dhe mekanizmi tregohet ne fig.4-34. Këtu sistemi i menaxhimit të memorjes ndahet ne tre pjese.

1. MMU e nivelit të ulet
2. një ‘page fault’ qe është pjese e kernel
3. një faqe të jashtme qe vepron ne hapesiren e përdoruesit

të gjitha detajet se si funksionon MMU janë të përbledhura të mbajtësja e MMU-se qe është një kod makine dhe qe duhet të rishkruhet për çdo platforme të re qe SO dergon. Mbajtësja e ‘page fault’ është një makine e pavarur dhe përmban shumicen e mekanizmit për paging. Plani përcaktohet gjeresisht për një faqe të jashtme, qe vepron si një proces përdorues.

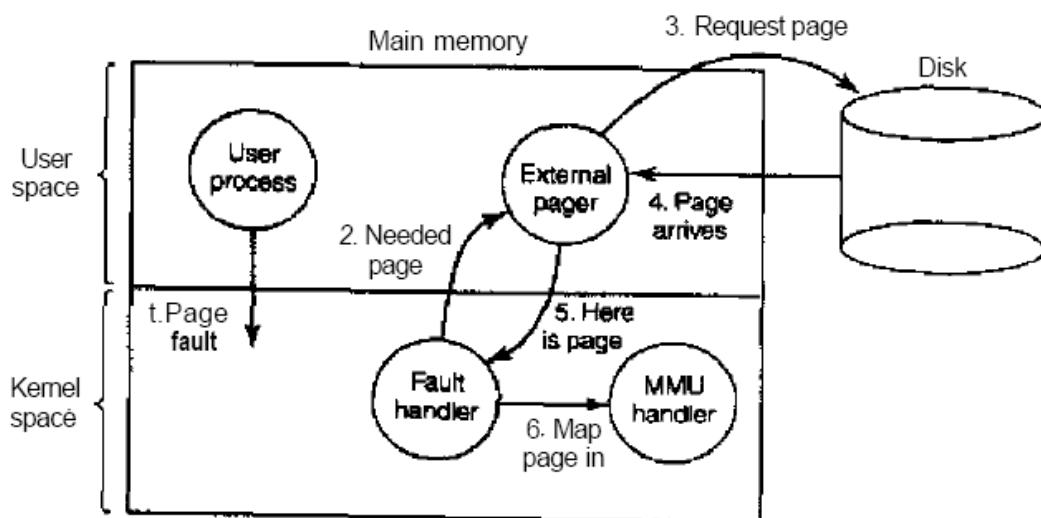


Fig.4-34. ‘page faulting’ me një faqe të jashtme

Kur fillohet një process, njoftohet faqja e jashtme qe të ndertoje hartën e faqes se procesit dhe të shpërndaje ‘backing store’ ne disk, në qoftë se është e nevojshme. Ndërsa procesi

vazhdon, mund të hartoje objekte të reja ne hapesiren e adreses se saj, keshtu qe njoftohet përseri faqja e jashtme.

Kur proçesi fillon, ndonjehere mund të marre një ‘page fault’. Mbajtësja e gabimit e dallon se cila faqe virtuale nevojitet dhe i dergon një mesazh faqes se jashtme, duke i treguar se ku është problemi. Faqja e jashtme lexon faqen e nevojshme nga disku dhe e kopjon atë ne një porcion te hapesira e adreses se saj. Dhe pastaj i tregon mbajtëses se gabimit se ku ndodhet faqja. Kjo e fundit e heq faqen dhe pyet mbajtësen e MMU-se qe ta vendose ne vendin e duhur te hapesira e adreses se përdoruesit. Pastaj proçesi i përdoruesit mund të ristartohet.

Ky implementim e le të hapur ku do të vendoset algoritmi i zhvendosjes se faqes. Do ishte me e pastër ta kishim atë te faqja e jashtme, por me ketë afrim do të ketë probleme. Principja midis ketyre është se faqja e jashtme nuk ka akses të bitet R dhe M e të faqeve të tjera. Keto bite luajne rol tek shume algoritme ‘paging’. Prandaj disa mekanizma duhet ta kalojne ketë informacion deri të faqja e jashtme ose algoritmi i zhvendosjes se faqes duhet të shkoje ne kernel. Ne raste të tjera faqja e gabimit i tregon faqes se jashtme se cilen faqe ka selektuar për ta hequr dhe prodhon të dhena duke i hartuar ato ne adresat e faqeve të jashtme ose me një mesazh. Menyre tjetër është se faqja e jashtme i shkruan të dhenat ne disk.

Avantazhi kryesor i ketij implementimi është: një kod i moduluar dhe me një fleksibilitet shume të madh. Disavantazhi është se i kalon se tepërmë kufinjtë e përdoruesve kernel si dhe sasia e madhe e mesazheve qe dergohen midis pjeseve të sistemit. Për momentin ky subjekt është shume i diskutueshem, por me rritjen e kompjuterave dhe programi behet me kompleks. Gjatë rruges duke sakrifikuar disa performanca për programe me të përshtatshme, ndoshta mund të behet me e pranueshme për shumicen e implementuesve.

4.8 Segmentimi

Memoria virtuale qe u diskutua deri tani është një-dimensionale sepse adresat virtuale shkojne nga 0 deri ne adresen maksimale, njëra pas tjetres. Për shume probleme, është me mire të kemi dy ose me shume hapsira adresash virtuale të vecanta sesa një të vetme. Për shembull: një kompilues ka shume tabela të cilat jane ndertuar nga kompilimi i procedurave, duke përfishhere:

1. Teksti burim është ruajtur për listën e printueshme (ne një batch sistem).
2. Tabela simbol përmban emrat dhe atributet e variablate.
3. Tabela përmban të gjitha konstantet integer (dhjetore) dhe floating-point (me presje levizese).
4. Pema e analizes, përmban analizen sintaksore të programit.
5. Stack-u përdoret për thirrjet e procedures brenda kompilatorit.

Secila prej katër tabelave të para zhvillohet ne menyre të vazhdueshme, ashtu si kompilimi i procedurave. Kurse tabela e fundit zhvillohet dhe tkurret ne menyre të paparashikueshme gjatë kompilimit. Ne një memorie një-dimensionale, keto katër tabela duhet të alokojnë vazhimisht pjese të hapesires se adresave virtuale sic tregohet edhe ne figuren 4-35.

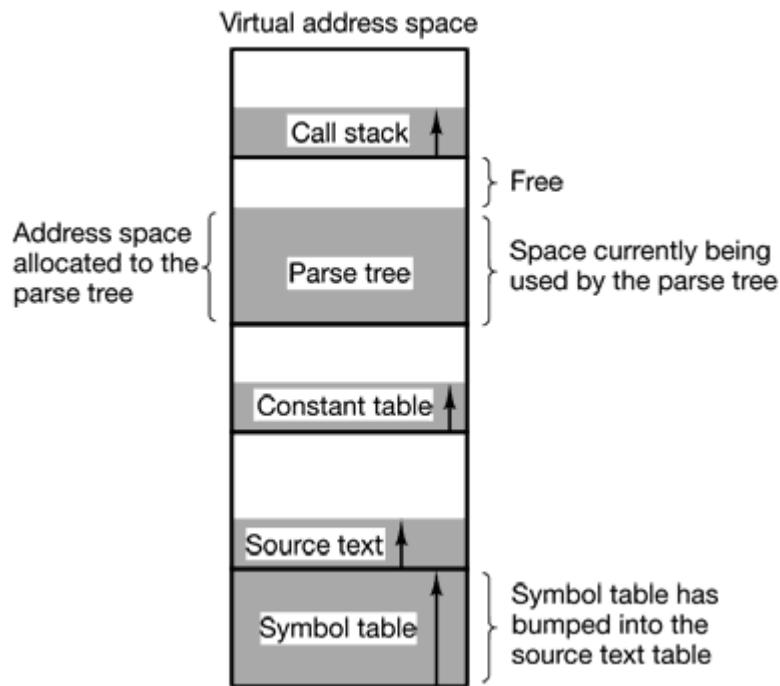


Figura 4-35. Ne një hapesire adresash një-dimensionale me zhvillimin e tabelave, një tabele mund të përplaset me një tjetër.

Konsideroni se cfare do të ndodhne ne se një program do të kishte një numer mjaft të madh variablesh por një sasi normale për gjerat e tjera. Pjesa e hapesires e alokuar për tabelen e simboleve mund të mbushet plot, megjithese mund të ketë hapesira të lira ne tabelat e tjera. Kompiluesi do të shfaqte një mesazh, i cili do të na thoshtë qe kompiluesi nuk mund të vazhdoje për shkak të numrit të madh të variablate, gje e cila nuk duket shume argetuese kur hapesira të lira jane të pranishme ne tabelat e tjera.

Një mundesi tjetër është të luajme si Robin Hood, duke marre hapesiren e lire nga tabelat qe e kane me teprice (të pasura) dhe ia dhurojme tabelave qe kane pak hapesire (të varfra).

Ajo cfare duhet ne të vertetë është një menyre për ta cliruar programuesin nga detyra e menaxhimit të zhvillimit dhe tkurries se tabelave, ne të njëtën kohe qe memoria virtuale eliminon problemen e organizimit të programit ne “overlays”.

Një zgjidhje është të pajisim makinën me shume hapesira adresash komplet të pavarura qe quhen segmentë. Çdo segment konsiston ne një sekunçe lineare adresash nga 0 deri ne max. Gjatësia e çdo segmenti mund të jetë cfaredo, nga 0 deri ne max e lejuar.

Segmentë të ndryshme zakonisht kane gjatësi të ndryshme, por gjatësia e segmentëve mund të ndryshoje gjatë ekzekutimit. Gjatësia e një segmenti stack rritet sa here qe dicka futet (push) ne stack dhe zvogelohet sa here qe nxjerrim (pop) dicka.

Çdo segment cakton një hapesire adresash, kurse segmente të ndryshme mund të rriten ose të zvogelohen ne menyre të pavarur pa demtuar njëri tjetrin. Në qoftë se një stack ne një segment të caktuar kerkon me shume hapesire adresash për tu rritur, ai do ta ketë atë, sepse nuk ka ndonjë rezik qe adresat të përplasen. Mund të ndodhe qe segmenti të mbushet plot, por meqe ata jane shume të medhenj kjo gje mund të ndodhe shume rralle. Për të specifikuar një adresë ne ketë segment ose ne ketë memorie dy-dimensionale, programi duhet të rezervoje numrin e segmentit dhe adresën e segmentit. Figura 4-36, ilustron një segment memorieje qe u përdor ne diskutimin për kompilimin e tabelave. Pese segmente të ndryshme jane treguar ne ketë figure.

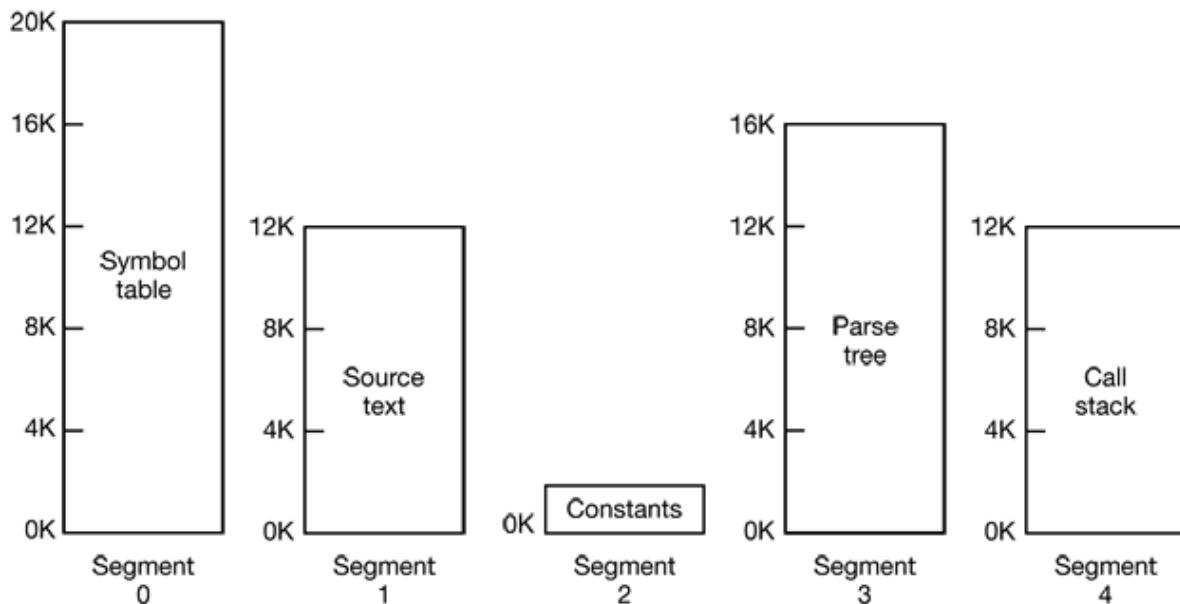


Figura 4-36. Një memorie e segmetuar lejon çdo tabelle të zhvillohet ose tkurret pavaresisht nga tabelat e tjera.

Theskojme se një segment është një entitet llogjik. Segmenti mund të përbaje një procedure, një tabelle, një stack ose një grup variabash, por asnjehere nuk përban një përzierje të tyre.

Ne një memorie, në qoftë se çdo procedure okupon një segment të caktuar, ku adresa e fillimit është 0, atëherë metoda e bashkmit dhe kompilimit të procedurave të vecanta është e thjeshtë. Pasi të gjitha procedurat qe formojne një program Jane kompiluar dhe bashkuar, një thirrje procedure ne proceduren e cila ndodhet ne segmentin n do të përdori

adresat e përbera (n, 0) ne fjalen me adresë 0 (pika e hyrjes). Në qoftë se procedura ne segmentin n me pas është modifikuar dhe rikompiluar, procedurat e tjera nuk mund të ndryshohen (sepse adresa e fillimit është modifikuar), vetëm në qoftë se versioni i ri është me i madh se i vjetri. Me një memorie një-dimensionale, procedurat Jane paketuar dhe nqjeshur me njëra tjetren pa asnjë hapesire adresash midis tyre. Si pasoje e ndryshimit se madhesise se një procedure mund të kemi demtimin e adreses fillestare për procedurat pasardhese. Kjo, si rregull kerkon modifikimin e të gjitha procedurave që therrasin ndonjë nga procedurat e spostuara, ne vend qe të inkorporojme adresat e tyre të fillimit. Në qoftë se një program përmban me qindra procedura, atëherë ky proces do të jetë tepër i kushtueshem.

Segmentimi gjithashtu ndihmon ne ndarjen e procedurave apo të dhenave midis shume proceseve. Një shembull i zakonshem është dhe shared library. Kompjuterat workstation qe punojne me sisteme të avancuara windows-i kane librari grafike ekstreme të kompiluarra ne çdo program. Ne një sistem të segmentuar, libraria grafike vendoset ne një segment dhe mund të ndahet midis shume proceseve duke eliminuar nevojen për ta patur atë vetëm nga një proces. Gjithashtu mund të kemi edhe ndarjen e librarive midis faqeve të sistemit por kjo është me e komplikuar. Ne fakt kjo behet nga sistemet duke stimuluar segmentimin.

Meqë çdo segment formon një entitet llogjik, programesi është i informuar se c'ka ne secilin prej segmentëve, si procedura, stack apo tabela, dhe segmente të ndryshme kane mbrojtje (protection) të ndryshme. Një procedure mund të caktohet vetëm si e ekzekutueshme, duke ndaluar përpjekjet për të lexuar apo shkruajtur ne të. Ne një tabele mund të shkruhet dhe lexohet, por nuk mund të ekzekutohet. Kjo lloj mbrojtje është e nevojshme ne parandalimin e gabimeve gjatë programimit.

Një pyetje qe shtrohet është: Pse mbrojtja përfshihet ne një memorie të segmentuar dhe jo ne një memorie një-dimensionale sic Jane faqet e memories? Ne memorien e segmentuar përdoruesi është i informuar se cfare ka ne çdo segment. Zakonisht një segment nuk mund të mbaje dhe një procedure dhe një stack, por vetëm njëren prej tyre. Keshtu, meqenese çdo segment përmban vetëm një lloj objekti, segmenti do të ketë një mbrojtje të përshtatshme për atë lloj objekti. Faqet dhe segmentet Jane krahasuar ne figuren 4-37.

Konsiderata	Faqet	Segmentët
Kerkon qe programuesi të jetë i informuar për tekniken qe është përdorur?	JO	PO
Sa hapesira adresash lineare ka?	1	Shume
A e kapërcen hapesira e adresave madhesine e memories fizike?	PO	PO

A Jane procedurat dhe të dhenat të dallueshme dhe të mbrojtura vecmas?	JO	PO
A minden tabelat madhesia e të cilave ndryshon të përshtaten lehtë?	JO	PO
Behet ndarja e procedurave?	JO	PO
Pse është krijuar kjo teknike?	Për të patur një hapesire të madhe lineare adresash pa patur nevoje për të blere me shume memorie fizike	Për të lejuar programet dhe të dhenat të vendosen ne hapesira adresash logjike dhe të pavarura si dhe për të ndihmuar ndarjen dhe mbrojtjen

Figura 4-37. Krahasimi i faqeve dhe segmentëve

Përbajtja e një faqeje është ne një fare menyre aksidentale. Programuesi është i pa informuar për krijimin e një faqeje. Ndonëse vendosim disa bite ne hyrje të çdo page table për të specifikuar aksesin e lejuar, për të përdorur ketë vecori programuesi duhet të ndjeke se ne cfare vendi ne hapesiren e adresave gjenden kufijtë e faqes. Kjo ishte pikërisht menyra e administrimit për faqet qe jane zbuluar për tu eliminuar. Për shkak të përdorimit të memories se segmentuar është krijuar iluzioni se të gjitha segmentët jane ne memorien kryesore gjatë gjithe kohes, se programuesi mund ti adresoje ato me mendimin qe ato ekzistojne, apo ai mund të mbroje çdo segment ne menyre të vecantë, pa u shqetësuar për atë qe ka të beje me administrimin e mbulimit të tyre.

4.8.1 Implementimi i segmentimit të pastër (pure segmentation)

Implementimi i segmentimit ndryshon nga krijimi i faqeve ne menyre thelbesore: faqet kane madhesi fikse kurse segmentet jo. Figura 4-38(a), tregon një shembull të memories fizike e cila fillimisht përmban pese segmente. Cfare do të ndodhët në qoftë se segment 1 nxirret jashtë dhe segmenti 7 i cili është me i vogel të vendoset ne vendin e tij. Ne arrijme ne konfigurimin e memories si ne figuren 4-38(b). Ndermjet segmentit 7 dhe 2 ka një zone boshe e cila quhet vrime. Me pas segmenti 4 zevendesohet nga segmenti 5, si ne figuren 4-38(c) dhe segmenti 3 zevendesohet nga segmenti 6, si ne figuren 4-38(d). Pasi sistemi është duke punuar për një fare kohe, memoria do të ndahet ne një numer të caktuar pjesesh (chunks), ku disa përbajne segmente dhe disa përbajne vrima. Ky

fenomen quhet ndarja ne kuadrate (**checkerboarding**) ose **fragmentimi i jashtëm**. Forma e kompaktësuar e ketij segmentimi jepet ne figuren 4-38(e).

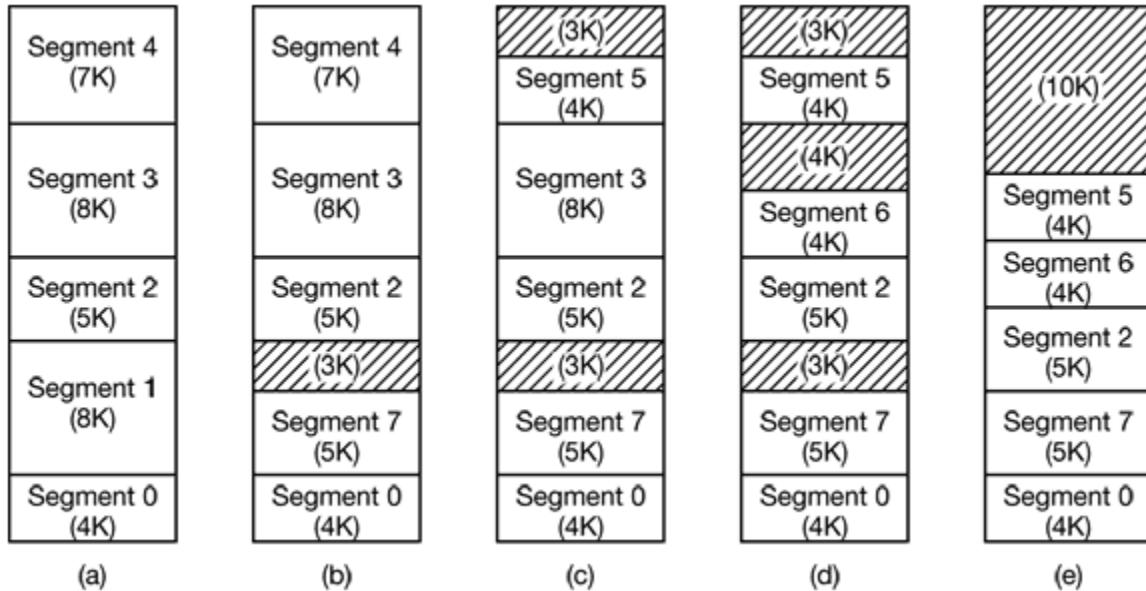


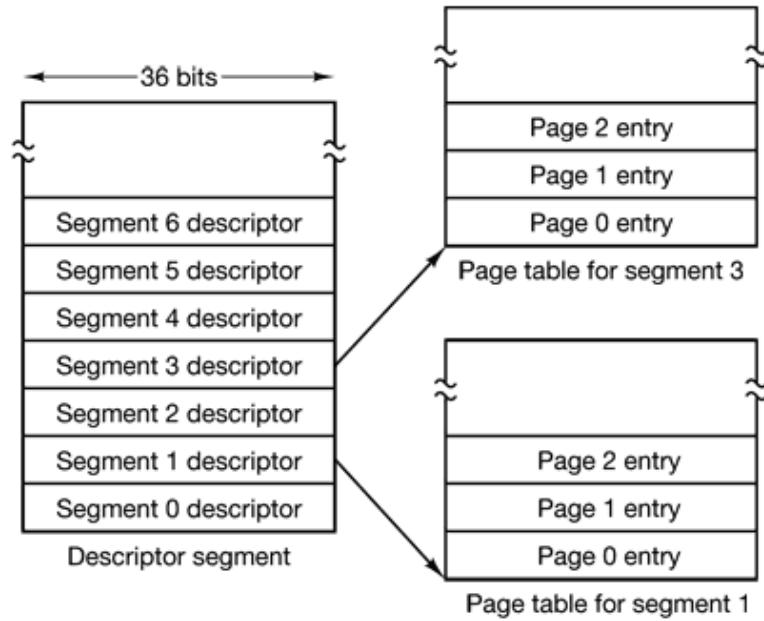
Figura 4-38 (a)-(d) Zhvillimi i checkerboarding. (e) eliminimi i checkerboarding nepërmjet kompaktësimit.

4.8.2 Segmentimi me faqe: MULTICS

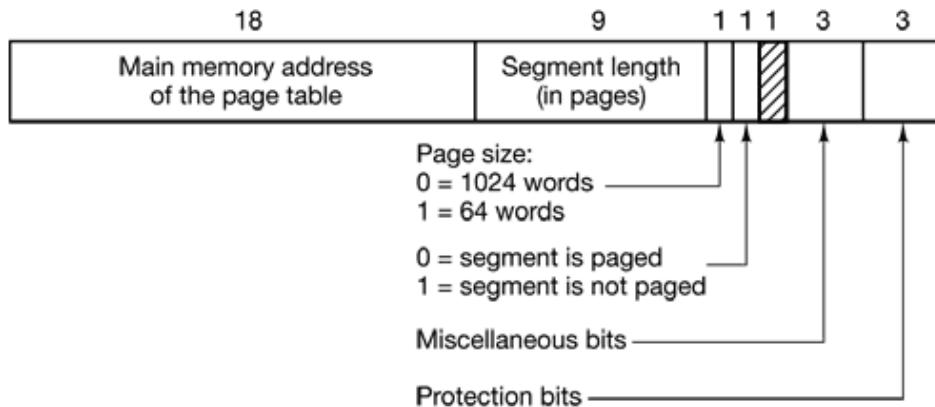
Në qoftë se segmentet jane të medhenj, mund të jetë e papërshtatshme ose gati e pamundur, për ti mbajtuar ato ne memorien kryesore. Kjo con ne idene e krijimit të faqeve të tyre, keshtu vetëm ato faqe qe duhen do të mbahen. Shume sisteme kane mbështetur segmentimin e faqeve. Ne këtë paragrafë ne do të pëershkrumë të parin: MULTICS. Me pas ne vijim do të diskutojme për një tjetër: Intel Pentium.

MULTICS, punojne ne makinat Honeywell 6000 dhe ne pasardhesit e tyre dhe pajisin secilin program me një memorie virtuale mbi 2^{18} segmentë (me shume se 250 000), secila prej të cilave mund të jetë mbi 65 536 (36-bit) fjale e gjatë. Për të implementuar ketë, projektuesit e MULTICS duhet ta trajtojnë çdo segment si një memorie virtuale dhe si faqe, duke kombinuar avantazhet e faqes (madhesia uniforme e faqes dhe mos mbajtja e gjithe segmentit ne memorie kur përdoret vetëm një pjese e tij) me avantazhet e segmentimit (lehtësia e programimit, modulariteti, mbrojtja dhe sharing).

Çdo program MULTICS ka një tabele segment, me një përshkrues për çdo segment. Përderisa mund të kemi me shume se një cerek milioni hyrje ne një tabele, tabela segment është ne vetvete një segment dhe një faqe. Përshkruesi i segmentit përmban një të dhene qe tregon ne se segmenti është ne memorie ose jo. Në qoftë se ndonjë pjese e segmentit ndodhet ne memorie, segmenti konsiderohet sikur është ne memorie, përshkruesi i tij përmban një pointer 18-bit ne tabelen e fakes se tij [shiko figuren 4-39(a)]. Meqenese adresat fizike jane 2 bitshe dhe faqet jane rradhitur ne kufij 64 byte, ku vetëm 18 bit nevojiten për përshkruesin qe të ruaje adresen e page table. Përshkruesi gjithashtu përmban madhesine e segmentit, bitet mbrojtës (protection bits) dhe disa të dhena të tjera. Figura 4-39(b), ilustron një përshkrues segmenti MULTICS. Adresa e segmentit ne memorien sekondare nuk gjendet ne përshkruesin e segmentit por ne një tabele tjetër e përdorur nga segment fault handler.



(a)



(b)

Figura 4-39. Memoria virtuale MULTICS (a). Përshkruesi (descriptor) i segmentit duke pointuar ne tabelat e faqeve (b). Një përshkrues segmenti. Numrat Jane gjatësive e segmenteve.

Cdo segment është një hapesire adresash virtuale e zakonshme, e ndare ne faqe ne të njëjtën menyre si faqet e memories të pa segmentuar, qe i shpjeguam me përparrë ne ketë kapitull. Madhesia normale e fakes është 1024 fjale.

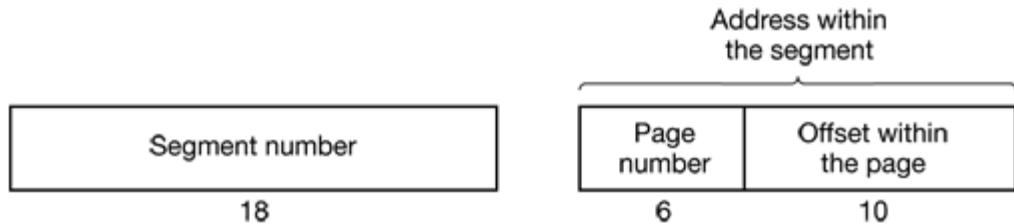


Figura 4-40. Një adrese MULTICS 34-bit.

Një adrese ne MULTICS përbhet nga dy pjese: nga segmenti dhe adresa brenda segmentit. Adresa brenda segmentit ndahet me tej ne një numer të caktuar fakesh dhe fjalesh brenda fakes, sic tregohet ne figuren 4-40. Kur ndodh një reference ne memorie, algoritimi qe ndiqet është carried out.

1. Numri i segmentit përdoret për të gjetur përshkruesin e segmentit.
2. Një verifikim behet për të pare në qoftë se tabela e fakes se segmentit është ne memorie. Në qoftë se tabela e fakes është ne memorie, atëhere ajo është lokalizuar. Në qoftë se jo, shkaktohet një segment fault. Në qoftë se ka një shkelje të sigurise, shkaktohet një fault (trap).
3. Në qoftë se është ekzaminuar kerkesa e fakes virtuale për të hyre ne tabelen e fakes. Kur faqja nuk ndodhet ne memorie, shkaktohet një page fault. Kur faqja ndodhet ne memorie, adresa e fillimit të fakes ne memorien kryesore është nxjerre nga page table entry.
4. Offset futet qe ne fillim të fakes për të dhene adresen e memories kryesore ku është lokalizuar fjala.
5. Leximi ose ruajtja përfundimisht zaptojne vend.

Ky proçes është ilustruar ne figuren 4-41. Për thjeshtësi, fakti qe përshkruesi i segmentit është dhe vetë i ndare ne faqe, është harruar. Cfare do të ndodhte ne të vertetë në qoftë se një regjistër (regjistri baze i përshkruesit), do të përdorej për të lokalizuar përshkruesin e segmentëve të tabelës se fakes, i cili ne parim pointon ne faqet e përshkruesit të segmentit. Pasi përshkruesi për segmentin e kerkuar është gjetur, adresimi i proceseve është treguar ne figuren 4-41.

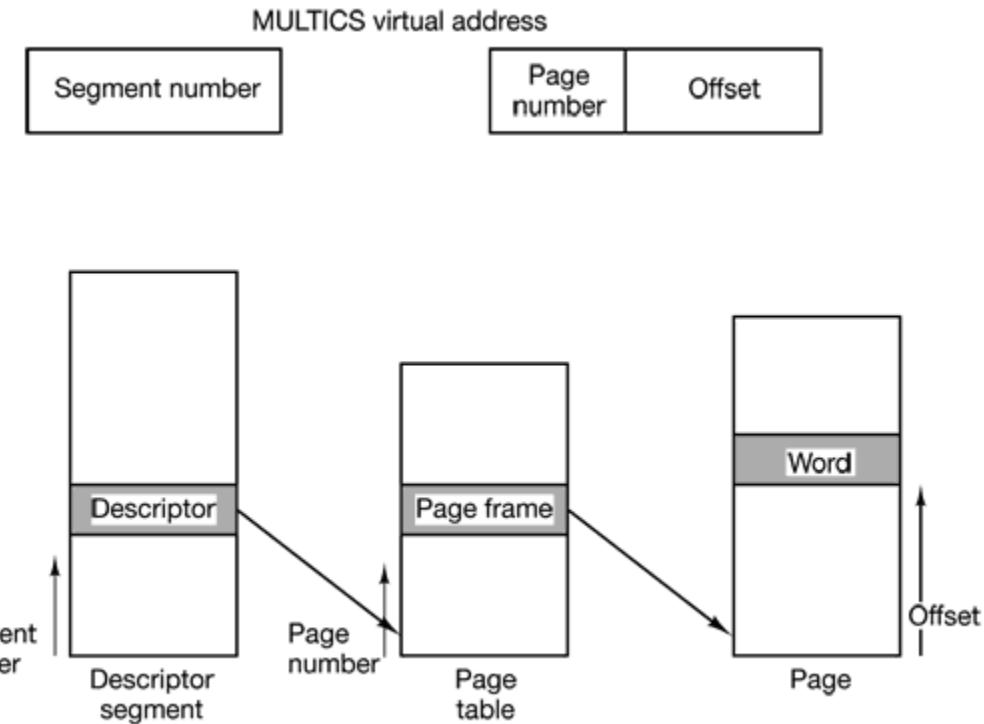


Figura 4-41. Shnderrimi i përbere i një adresese MULTICS ne një adrese se memories kryesore.

Në qoftë se procedura e algoritmit është kryer nga sistemi operativ instuksion pas instruksioni, programet nuk do të punojne shume shpejt. Ne realitet, hardware i MULTICS përmban një TLB 16-fjaleshe me shpejtësi të lartë qe mund të kerkoje të gjitha të dhenat e futura ne paralel për një celes (key) të dhene. Kjo është ilustruar ne figuren 4-42. Kur një adrese është prezente ne kompjuter, hardware i adresimit ne fillim kontrollon ne se adresa virtuale ndodhet ne TLB. Në qoftë se po, merr numrin e page frame direkt nga TLB-ja dhe formon adresen aktuale të fjales qe i referohet pa marre parasysh përshkruesin e segmentit apo page table (tabelen e faqes).

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figura 4-42. Një version i thjeshtë i TLB-se MULTICS. Ekzistanca e dy madhesive të faqes e bën TLB-ne aktuale me të komplikuar.

Programet qe kane një hapesire pune me të vogel se madhesia e TLB do të vijne duke u barazuar me adresat qe hyjne ne hapesien e punes se TLB, dhe keshtu do të punojne me me eficense. Në qoftë se faqja nuk është ne TLB, përshkruesi dhe tabelat e faqeve detyrohen të gjejne adresen e page frame-it dhe TLB-ja bën një update (freskim) për të përfshire ketë faqe, kurse faqja e fundit qe u përdor nxirret jashtë. Fusha e moshes (age field) tregon jetëgjatësine e faqeve duke përcaktuar ne ketë menyre dhe faqen qe është përdorur e fundit. Shkaku se pse kemi përdorur një TLB është për të krahasuar segmentin dhe faqen për të gjitha hyrjet ne paralel.

4.8.3 Segmentimi me faqe: Intel Pentium

Memoria virtuale tek Pentium-et ngjason me MULTICS, pasi kane të dy segmentimin me faqe. MULTICS ka 256 K segmentë të pavarur, secili mban 64 K fjale 32-bitshe, kurse Pentumi ka 16 K segmentë të pavarura, secili mban mbi 1 milion fjale 32-bit.

Zemra e memories virtuale tek Pentium përbehet nga dy tabela; **LDT** (Local Descriptor Table) dhe **GDT** (Global Descriptor Table). Çdo program ka LDT-ne e tij, kurse GDT është vetëm një dhe ndahet midis të gjitha programeve të kompjuterit. LDT-ja përshkruan segmentët lokale të çdo programi, duke përfshire kodet, data, stack, etj, ndersa GDT përshkruan segmentët e sistemit, duke përfshire ne vetvete sistemin operativ.

Për të aksesuar një segment, një program Pentium ne fillim ngarkon një përzgjedhes për atë segment ne një nga gjashtë registrat e makines. Gjatë ekzekutimit, regjistri mban përzgjedhesin për kodin e segmentit dhe regjistri DS mban përzgjedhesin për të dhenat e

segmentit. Registrat e tjere të segmentëve jane me pak të rendesishem. Çdo përgjedhes është një numer 16-bit, ashtu sic tregohet ne figuren 4-43.



Figura 4-43. Një përgjedhes (selector) Pentium-i

Një nga bitet e përgjedhesit tregon se një segment është lokal apo global (ndersa ai është ne LDT ose ne GDT). 13 bitet e tjere përcaktojnë numrin e hyrjeve të LDT-se ose të GDT-se, keshtu keto tabela Jane të përcaktuara që secila të mbaje nga 8 K përshkrues segmenti. 2 bitet e tjere kane të bejne me mbrojtjen (protection) dhe do të trajtohen me vone. Përshkruesi 0 (zero) mungon.

Sapo një përgjedhes ngarkohet ne një segment regjistër, përshkruesi qe i korrespondon atij regjistri është marre (fetch) nga LDT ose GDT dhe është ruajtur ne regjistrat e mikroprogramit, keshtu mund të aksesohet me lehtë. Një përshkrues përbhet nga 8 byte, duke përfshire adresen baze të segmentit, madhesine dhe informacione të tjera, sic paraqitet ne figuren 4-44.

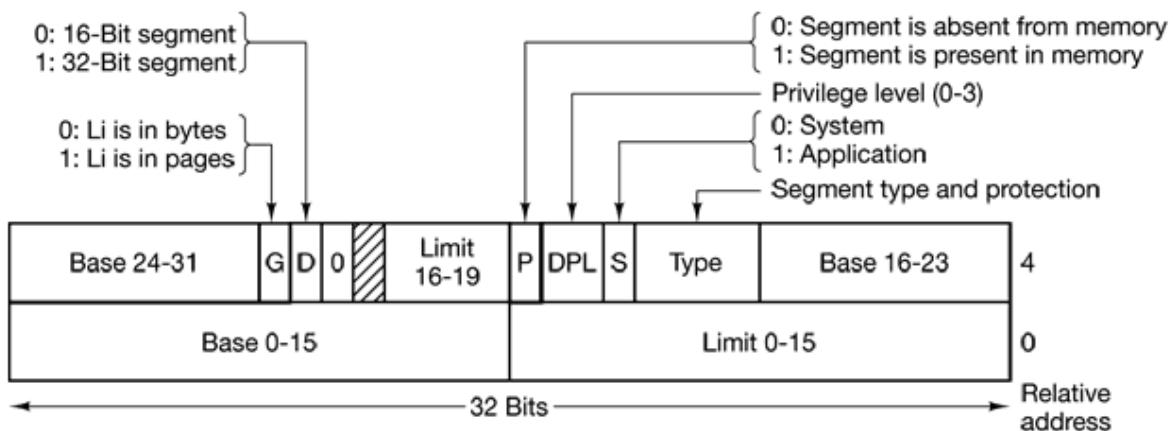


Figura 4-44. Pentium code segment descriptor. Segmentët e të dhenave ndryshojne pak.

Formati i përgjedhesit është marre i aftë qe të beje lokalizimin e përshkruesit sa me të lehtë. Ne fillim behet selektimi i LDT ose GDT. Me pas përgjedhesi kopjohet ne një

regjistër të jashtëm dhe tre bitet me peshe me të vogel behen 0. Ne fund adresat e secilit prej LDT ose GDT vendosen ne tabele për ti dhene direkt një pointer përshkruesit.

Për shembull, përzgjedhesi 72 i referohet hyrjes 9 ne GDT, i cili është vendosur ne adresen GDT + 72.

Le të shohim hapat nepër të cilat një cift (përzgjedhes, offset) është shnderruar ne një adresë fizike. Sapo mikroprogrami identifikon segment regjistrin qe është përdorur, ai mund të identifikoje përshkruesin qe i korrespondon përzgjedhesit ne regjistrin e tij të jashtëm. Në qoftë se segmenti nuk ekziston (përzgjedhesi 0) ose është paged out, shkaktohet një trap.

Llogjikisht duhet një fushe 32 biteshe ne përshkruesin qe përcakton edhe madhesine e segmentit, por jane vetëm 20 bit të mundshme. Në qoftë se fusha e Gbit (Granularity) është 0, atëherë fusha Limit është ekzaktësisht madhesia segmentit, mbi 1 MB. Madhesia e faqes Pentium është fiksë 4 KB, keshtu qe 20 bit mjaftojne për segmentët, mbi 2^{32} byte.

Duke pretenduar qe segmenti ndodhet ne memorie dhe offset-i është ne varg, atëherë Pentumi fut fushen Base 32 bit-she ne përshkruesin e offset-it për të formuar atë qe quhet **linear address** (adrese lineare), sic tregohet edhe ne figuren 4-45. Fusha Base është ndare ne tre pjesë: Adresa baze, fusha Limit dhe fushat e tjera. Ne të vertetë, tek fusha Baselejon çdo segment të filloje me një hapesire arbitrale, me 32-bit hapesire adresë lineare.

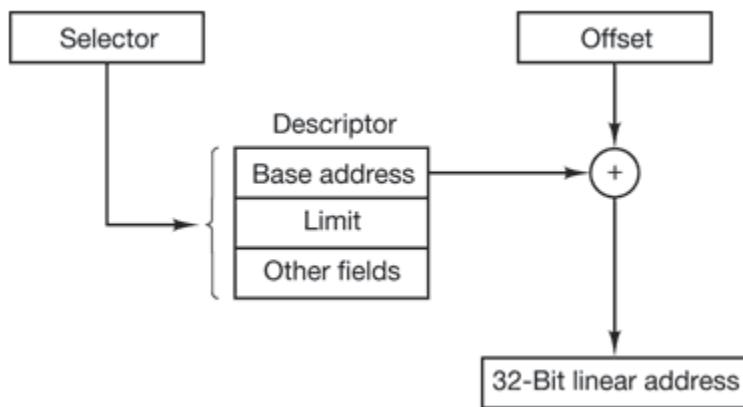


Figura 4-45. Shnderrimi i ciftit (selector, offset) ne një adresë lineare.

Në qoftë se faqet jane disabled (ndaluar nga një bit ne regjistrin e kontrollit global), adresa lineare interpretohet si adresë fizike dhe dergohet ne memorie për read ose write. Keshtu me faqet disabled (të ndaluara), ne marrim skemen e një segmentimi të pastër (pure segmentation), ku adresa baze e çdo segmenti është dhene ne përshkruesin e tij.

Nga ana tjetër, në qoftë se faqet jane lejuar (enabled), adresa lineare interpretohet si një adresë virtuale dhe mapped (krijon një hartë) mbi adresen fizike duke përdorur tabelat e faqes. Një veshtiresi qe haset është se me një adresë virtuale 32-bit dhe një faqe 4 KB, një

segment duhet të mbaje 1 milion faqe, keshtu një mapping me dy nivele përdoret për të reduktuar madhesine e tabelles se faqes për segmente të vegjel.

Çdo program ne ekzekutim ka një direktori faqeje (page directory) qe përmban 1024 hyrje 32 bit-she. Ai është i vendosur ne një adresë qe pointohet nga një regjistër global. Çdo hyrje ne ketë direktori pointon ne një tabelë faqeje e cila gjithashtu përmban 1024 hyrje 32-bitshe. Hyrjet ne tabelen e faqes pointoin ne page frame-t. Skema paraqitet si ne figuren 4-46.

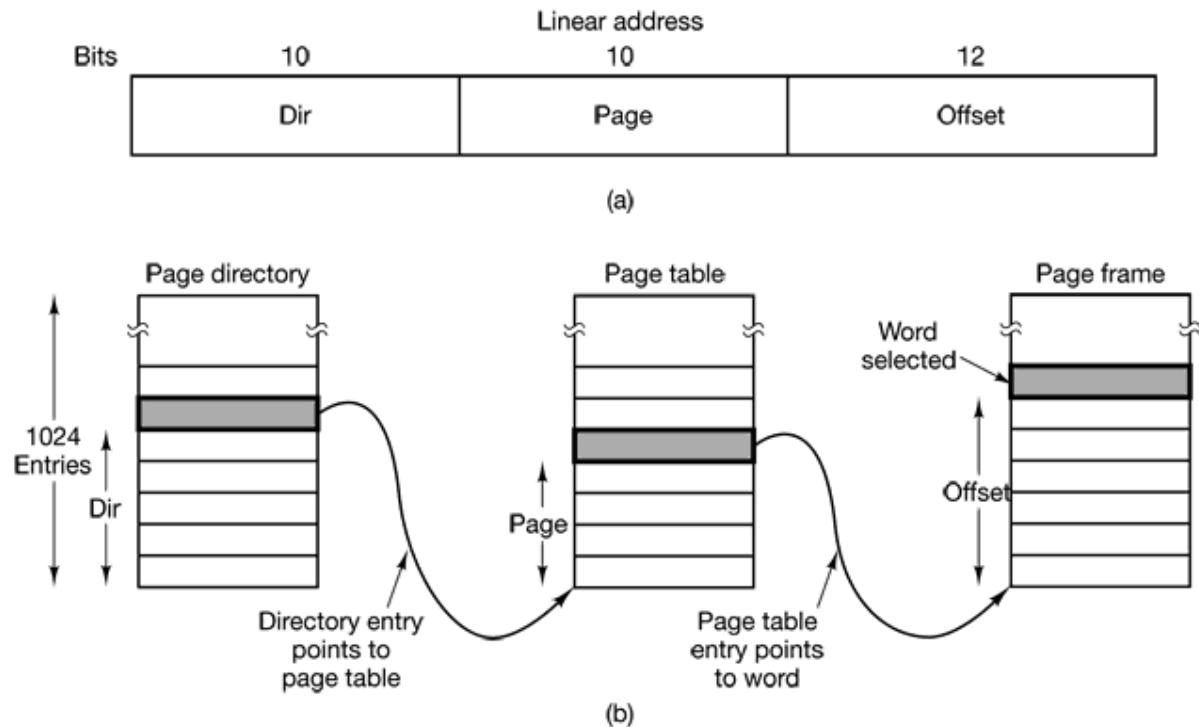


Figura 4-46. Mapping i një adrese lineare mbi një adrese fizike.

Ne figuren 4-46(a) shohim një adrese lineare të ndare ne tre fusha: Dir, Page dhe Offset. Fusha Dir përdoret si indeks ne direktorine e faqes për të lokalizuar një pointer ne tabelen e faqes. Fusha Page përdoret si një indeks ne tabelen e faqes për të gjetur adresen fizike e një page frame. Fusha Offset vendoiset ne adresen e page frame-it për të gjetur adresen fizike të byte-it apo word-it qe nevojitet.

Hyrjet e tabelles se faqes jane 32-bit secila, 20 prej të cilave përbajne një numer page frame-sh. Bitet e ngelur përbajne access dhe dirty bits, të caktuar nga hardware për qellimet e sistemit operativ, protection bits dhe bite të tjere të nevojshem.

Çdo tabelë faqeje (page table) ka 1024 hyrje 4-KB për page frame-t, keshtu vetëm një tabelë mban 4 MB memorie. Një segment me i shkurtër se 4 M do të ketë një direktori faqeje me një hyrje të vetme, qe është një pointer ne tabelen e faqes e cila është e vetme

(sepse kemi vetëm një hyrje), ne vend të 1 milion faqeve qe do të duheshin ne një tabelle fajeje me një nivel.

Për të shhangur përsritjet ne adresimin e memories, Pentiumi, po ashtu edhe MULTICS, kane një TLB të vogel e cila tregon kombinimet me të fundit të përdorura ne Dir-Page ne adresen fizike e page frame-it. Kur kombinimi i castit nuk është prezent ne TLB, është mekanizmi i figures 4-46 carried out dhe TLB bën një update. Kur deshtimet ne TLB rrallohen, përfomanca rritet.

Është e gabuar të themi se disa aplikacione nuk kane nevoje për segmentim, por ky model është i mundur vetëm kur segmenti përban vetëm një faqe të vetme me hapesire adresë 32-bitshe. Të gjithe regjistrat mund të ndertohen me të njëjtin përzgjedhes (selector), pëershkruesi (descriptor) i të cilit ka bazen = 0 dhe limitin deri ne maksimum. Offseti i instruksionit do të jetë adresa lineare vetëm me një hapesire të vetme, ne fakt e përdorur si normal paging. Të gjithe sistemet operative për Pentiumet punojne ne ketë menyre. OS/2 është i vetmi nga të gjithe qe përdor të gjithe fuqine e arkitektures se Intel MMU.

Për sa i përket mbrojtjes se sistemeve, Pentiumi ka 4 nivele mbrojtjeje, qe fillon me nivelin 0 dhe është me i privilegjuari dhe vazhdon deri ne nivelin 3 qe është me pak i privilegjuar. Keto nivele jane treguar ne figuren 4-47. Ne çdo cast të kohes, një program qe është ne ekzekutim, është ne një nivel të caktuar qe identifikohet nga një fushe 2 bitshe ne PSW e tij. Çdo segment ne sistem gjithashtu ka një nivel.

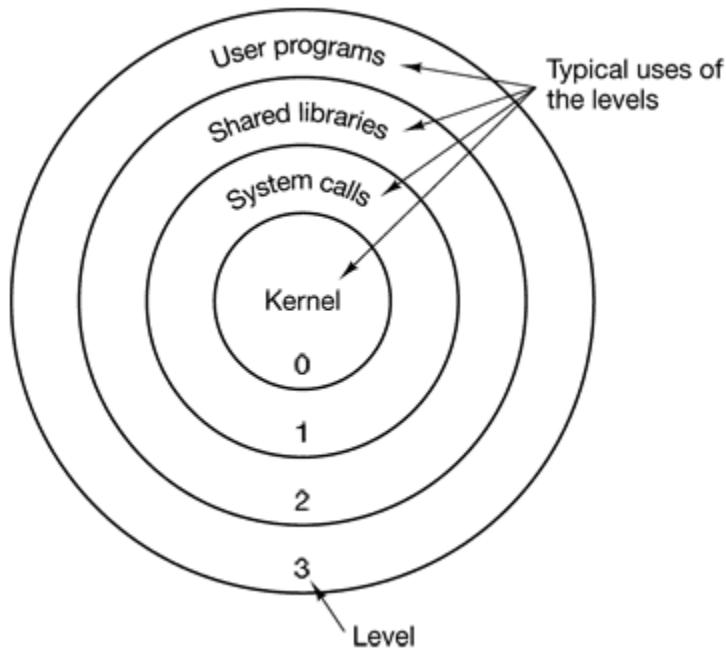


Figura 4-47. Mbrojtja tek Pentiumet

Për sa kohe qe një program kufizon veten ne përdorimin e segmenteve ne nivelin e tij, çdo gje punon ne rregull. Përpjekjet për të aksesuar të dhena ne nivele me të larta jane të lejuara. Kurse përpjekjet për të aksesuar të dhena ne nivele me të ulta jane ilegale dhe shkaktojne “trap”. Përpjekjet për të thirrur procedura nga nivele të ndryshme (nivele të larta ose të uleta) jane të lejueshme por të kontrolluara. Për të bere një call midis dy nivele, instruksioni CALL duhet të përbaje një përzgjedhes ne vend të një adrese. Ky përzgjedhes (selektor) përcakton një përshkrues (descriptor) qe quhet **call gate** (porta e call), i cili jep adresen e procedures qe do të thirret. Keshtu nuk është e mundur të kercejme ne mes të një kodi segment arbitrar ne një nivel tjeter. Vetëm hyrjet e lejuara mund të përdoren. Konceptet e mbrojtjes se niveleve dhe call gates jane përdorur tek MULTICS, ku ato jane pare si **protection rings** (unaza mbrojtëse).

Një përdorim tipik i ketij mekanizmi është sugjeruar ne figuren 4-47. Ne nivelin 0 kemi kernelin e sistemit operativ, i cili mban I/O, menaxhimin e memories dhe pjese të tjera kritike. Ne nivelin 1, system call handler është prezent. Programet user mund të themrin procedura për të patur systems calls carried out, por vetëm një listë specifike dhe e mbrojtur e procedurave mund të thirret. Niveli 2 përmban libraritë e procedurave qe ndahen midis shume programeve ne ekzekutim. Dhe i fundit, niveli 3 përmban programet qe ekzekutohen dhe është me pak i mbrojturi.

Traps dhe interrupt-et përdorin një mekanizem të ngjashhem me call gates. Ata, gjithashtu i referohen përshkruesve me mire sesa ne adresat absolute dhe keta përshkrues pointojne ne procedurat specifike për tu ekzekutuar. Fusha Type ne figuren 4-44, bën një dallim ndermjet code segments, data segments dhe një numri të caktuar gates.

KAPITULLI I PESTË

INPUT/OUTPUT

Një nga fuksionet kryesore të sistemt operativ është të kontrolloj të gjitha paisjet hyrese/dalese të kompjuterit. Ai duhet te trajtoj konditat për paisjet, të kapi interraptet, dhe të kujdeset për gabimet. Ai gjithashtu duhet të siguroj një nderfaqe midis paisjes dhe pjeses tjeter të sistemit, ne menyre qe ta bej atë sa me të thjeshte për tu përdorur. Ne zgjerimin e mundshem, nderfqja duhet të jetë e njëjtë për të gjitha paisjet (pavaresia e paisjeve). Kodi i paisjeve hyrese/dalese përfaqeson një pjese sinjifikative të të gjithe sistemit operativ. Subjekti i ketij kapitulli është se si sistemi operativ menaxhon paisjet I/O.

Ky kapitull është i ndare si me poshtë. Ne fillim do të shikojme disa principe të paisjeve I/O dhe me pas do shikojme software I/O. Softwaret I/O mund të strukturohen ne shtresa, ku çdo shtrese do të ketë një mision të mire përcaktuar për të kryer. Ne do të shikojme keto shtresa për të pare se cfare bejne ato dhe se si Jane ato të kombinuara se bashku.

Duke ndjekur ketë hyrje qe beme, ne do të shikojme disa paisje I/O të ndryshme:

disket, clock-et, tastierat dhe shfaqet. Për çdo paisje do shikojme pjesen hardware të tij dhe atë software. Dhe ne fund do të marrim parasysh power management.

5.1 PRINCIPET E I/O HARDWARE

Njerez të ndryshem i shikojne I/O hardware ne menyra të ndryshme. Inxhinieret elektrik i shikojne ato si chipe, tela, konvertuesa të tensioneve, motora dhe si çdo paisje tjeter fizike qe përdoren për të ndertuar hardware-et. Programuesit shikojne nderfaqen e paraqitur për software, komandat qe pranojne hardware, funksionet ne dalje dhe gabimet qe ato mund te kthejne mbrapsht. Ne ketë liber ne do të përqendrohem ne programimin e paisjeve I/O dhe jo ne dizjenimin e tyre, ndertimin ose mirembajtejn e tyre, pra interesit yne do të kufizohet se si hardware është programuar dhe jo si ai punon ne vetvete. Si do qoftë ne disa raste programimi i paisjeve I/O është i lidhur ngushtë me punen e tyre. Ne tre seksionet e ardhshme ne do të sigurojme një background të përgjithshem për I/O hardware, se si lidhen ato me programin.

5.1.1 PAISJET I/O

Paisjet I/O përafersisht mund të ndahan ne dy kategori: *block devices* dhe *character devices*. Një block devices është ai qe mbledh blloqe informacione me madhesi të caktuar, çdo informacion ka adresen e tij. Hapesira e bllokut të përbashket shkon nga 512 bytes ne 32,768 bytes. Karakteristika esenciale e një block device është qe ajo bën të mundur leximin ose shkrimin e çdo blloku ne menyre të pavarur nga njëri tjetri.

Ne qoftë se shikojme me kujdes, kufiri midis paisjeve qe jane blloqe të adresueshme dhe atyre qe nuk jane, nuk është i mire përcaktuar. Të gjithe bien dakort qe një disk është një paisje bllok e adresueshme se pavaresisht se ne cilin krah ndodhet, është gjithmone e mundur qe të arrij një cilinder tjetër dhe të presi deri sa blloku i kerkuar të rrotullohet posht kokes. Tani le të marrim ne konsiderat një tape drive për të bere një disk backup. Kasetat përbajne një sekuese të bllokut. Ne qoftë se një tape drive jep një komand për të lexuar bllokun N, ajo mund ta riktheje kasetën dhe të shkoj përpresa deri sa të arrij të blloku N. Ky operacion është i njëjtë me atë të kerkimit të një diskut, për vec se ai kerkon me shume kohe. Gjithashtu, jo gjithmone është e mundur për të riskruajtur ne një bllok ne mes të kasetës. Edhe pse ngadonjehere është e mundur të përdoren kasetë si block devices me akses random, kjo e zgjeron qellimin deri diku: normalisht keto nuk përdoren ne ketë menyre.

Pjesa tjetër e paisjeve I/O është character device. Një character device shpérndan ose merr një varg karakteresh, pa kerkuar asnje lidhe me struktura blloku. Nuk është e adresueshme dhe nuk ka asnje operacion kerkimi. Printerat, nderfaqet e networkut, maus-i, spiun (për eksperimente laboratorike psikologjike), dhe për shume paisje të tjera qe nuk i ngajnjë diskut mund të shihen si character devices.

Kjo skeme klasifikimi nuk është perfekte. Disa paisje nuk bajne pjesë ne to. Clock-et përshebull nuk jane blloqe të adresueshme. Ato as nuk gjenerojnë ose pranojnë vargje karakteresh. Ato vetëm shkaktojnë interrupte ne intervale të mire përcaktuara. As hartat e memories nuk bajne pjesë ne të. Akoma, modeli block dhe character device është mjaft i përgjithshem aq sa mund të përdoret si baze për të bere disa software sistemesh operative qe të punojne ne menyre të pavarur me paisjet I/O. Për shembull, file sistem merret vetëm me block device abstraktë dhe ja le pjesen e lidhjes me paisjen shtresave me të ulta software.

Paisjet I/O mbulojne një vije të gjere ne shpejtësi, gje qe shkakton një presion të konsiderueshem ne software qe të plotësoj sa me mire të gjitha kerkesat. Fig.5 tregon disa data rates për disa paisje të ndryshme. Shumica e ketyre paisjeve tentojne të shpejtojnë sa po koha nis.

Device	Data rate	
Keyboard	10	bytes/sec
Mouse	100	bytes/sec
56K modem	7	KB/sec
Telephone channel	8	KB/sec
Dual ISDN lines	16	KB/sec
Laser printer	100	KB/sec
Scanner	400	KB/sec
Classic Ethernet	1.25	MB/sec
USB (Universal Serial Bus)	1.5	MB/sec
Digital camcorder	4	MB/sec
IDE disk	5	MB/sec
40x CD-ROM	6	MB/sec
Fast Ethernet	12.5	MB/sec
ISA bus	16.7	MB/sec
EIDE (ATA-2) disk	16.7	MB/sec
FireWire (IEEE 1394)	50	MB/sec
XGA Monitor	60	MB/sec
SONET OC-12 network	78	MB/sec
SCSI Ultra 2 disk	80	MB/sec
Gigabit Ethernet	125	MB/sec
Ultrium tape	320	MB/sec
PCI bus	528	MB/sec
Sun Gigaplane XB backplane	20	GB/sec

Figure 5-1. Some typical device, network, und bus data rates.

5.1.2 KONTROLLUESIT E PAISJEVE

Paisjet I/O zakonishtë jane të përbera nga pjese mekanike dhe pjese elektronike. Shpesh është e mundur qe keto dy pjeset të ndahen pér të marre një kuptim sa me të përgjithshem ne ndertimin e tyre. Pjeset elektronike quhen *device controller* ose adaptor. Ne kompjuterat personal ato kane formen e një qarku të stampuar e cila mund të vihet ne një slot. Pjesa mekanike është vetë paisja. Kjo marreveshje është treguar ne fig.1-5.

Karta kontrolluese ka një konektor ne të cilin mund të lidhen telat qe dalin nga paisja. Në qoftë se nderfaqia midis kontrolluesit dhe paisjes është një nderfaqe standarte ose ASSI, IEEE ose standarti ISO atëhere kompanit e prodhimit të ketyre pjeseve mund të ndertojne kontrolluesa, ose paisje qe të mos krijojnë konflikte me njëra tjetren. Për shembull disa kompani prodrojne disk driver të cilat shkojnë me nderfaqet IDE ose SCSI.

Nderfaqa midis kontrolluesit dhe paisjes është shpesh here nderfaqe e nivelit të ulet. Një disk përshembull mund të ketë 256 sektor me nga 512 bytes për linjë. Ajo cfare merret nga drive megjithese nis me një varg serial bitesh, nis me një hyrje, me pas 4096 bit për sektor dhe nu fund është checksum ose sic quhet ndryshe **Error-Correcting code**. Hyrja shkruhet ne momentin qe formohet disku dhe përmban cilindrat dhe numrin e sektorit, përmast e sektorit dhe të dhenat krahasuese si një informacion të sinkronizuar.

Puna e kontrolluesit është qe të konvertoj një varg bitesh serial ne blloqe me byte dhe të bej çdo korrigjim gabim në qoftë se do të jetë e nevojshme. Blloku me byte asemblohet bit për bit ne një buffer brenda kontrollerit. Mbasi është kontrolluar checksum i tij dhe nuk është gjetur asnje gabim atëherë ai mund të kopjohet ne memorien kryesore.

Kontrolleri për një monitor punon edhe si një paisje me bite serial ne një nivel të ulet. Ajo lexon bytet të cilët përbajne karakteret qe do të meren nga memorja dhe qe gjenerone sinjalat qe do të përdoren për tu moduluar ne binar CRT për të shkruajtur ne ekran. Gjithashtu kontrolleri gjeneron sinjale qe detyron CRT të beje një rishikim horizontal pasi ai të ketë mbarur një skanim të një rreshti, ashtu si edhe sinjale për të bere një rishikim vertikal pasi i gjithe ekranit të jetë skanuar. Në qoftë se nuk do të kishim kontrollerin CRT, programatori I sistemit operativ do ti duhej ta programonte ne menyre eksplisite skanimin analog të tubit katodik. Sistemi operativ e inicializon kontrolleri me disa parametra, sic jane numri I karaktereve, ose pixel-at për rrjesht dhe numrat e rrjeshtave për ekran dhe le kontrolleri për tu kujdesur.

5.1.3 HARTA E MEMORIES I/O

Çdo kontroller ka disa regjistra të cilat përdoren për të komunikuar me CPU. Duke shkruajtur ne keto regjistra, sistemi operativ mund të komandoj paisjet për të cuar të dhena, për të mare të dhena, për tu aktivizur ose caktivizur ose për të ndermarre veprime të tjera. Duke lexur nepër keto regjistra, sistemi operativ meson se ne c'gjendje është paisja, në qoftë se është gati për të marre komanda të reja dhe keshtu me radhe. Përvèc regjistarve të kontrollit, shume paisje kane edhe të dhena buffer ku sistemi operativ mund të shkruaj ose të lexoj ne to. Përshembull një menyre e përbashket të kompjuterat për të nxjerr pixelat ne ekran, është qe duhet të kene një video RAM e cila ne parim është një data buffer, e gatshme për programin ose ku sistemi operativ mund të shkruaj ne të. Pra CPU komunikon me regjistrat e kontrollit dhe me paisjet data buffer. Ekzistojne dy alternativa. Ne trajtimin e pare, çdo regjistri kontrolli i është caktuar një numer portë I/O, 8 ose 16 bit integer. Duke përdorur një instruksion special I/O si :

IN REG,PORT,

CPU mund të lexoj ne regjistrin kontrollues PORT dhe të ruaj rezultatin ne regjistrin REG të CPU-se.

OUT PORT,REG

CPU mund të shkruaj përbajtjen e REG ne regjistrin kontrollues. Shumica e kompjuterave të atëhershëm, duke përfshire dhe mianframet, si IBM 360 dhe të gjithë pasardhesit e tij punojne ne të njëjtën menyre.

Ne ketë skeme hapesira e adreses për memorjen dhe I/O janë të ndryshme, sic duket edhe nga fig 5.2. Instruksioni

IN RO,4

dhe

MOV RO,4

jane komplet ndryshe ne ketë dizenjim. I pari lexon përbajtjen e portës 4 I/O dhe e vendos atë ne R0 kurse i dyti lexon përbajtjen e fjalës 4 ne mermorje dhe e vendos atë ne R0.

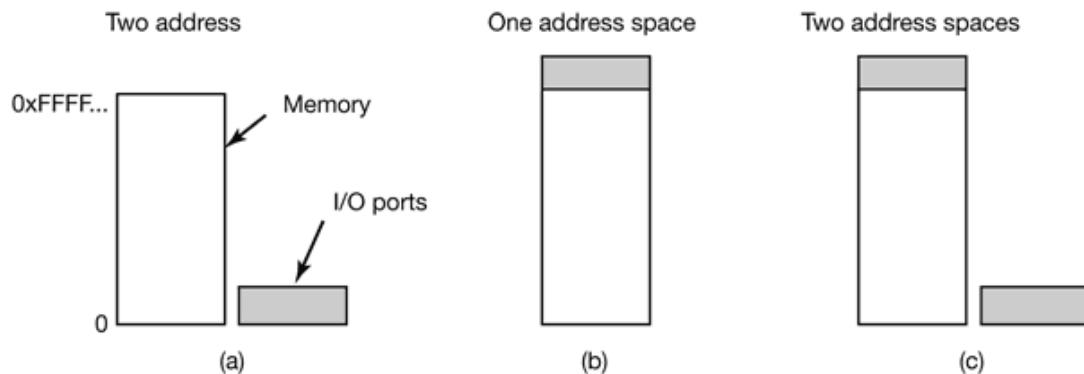


Figure 5-2. (a) Separate I/O and memory space. (b) Memory mapped I/O. (c) Hybrid.

Afrimi i dytë, qe ka të bej me PDP-11, është qe të skicoje të gjithe regjistrat e kontrollit ne hapesiren e memories sic tregohet ne fig.5-2(b). Çdo regjistri kontrolli I është caktuar një adresë e vetme ne memorie. Ky sistem quhet **memory-mapped I/O**. Zakonisht adresat e caktuara ndodhen ne fillim të hapesires se adresave. Një skeme hibride me memori-mapped I/O, me data buffer dhe me porta I/O të ndara për regjistrat e kontrollit tregohen ne fig.5.2(c). Pentium-et e përdorin ketë arkitekturë, me adresa 640K deri ne 1M të rezervuara për paisje data buffer.

Si funksionojne keto skema? Ne çdo rast, kur CPU kerkon të lexoj një fjale, nga memoria ose nga portat I/O, ajo vendos adresen qe duhet ne busin e linjave të adresave dhe aktivizon sinjali READ ne busin e linjave të kontrollit. Një sinjal i dytë përdoret për të treguar se sa hapesire I/O ose hapesire ne memorie duhet. Në qoftë se është për hapesiren ne memorie, memoria i përgjigjet kerkeses, në qoftë se është për hapesire I/O, paisja I/O i përgjigjet.

Në qoftë se ka vetëm hapesire memorie [si ne fig.5-2(b)], çdo modul i memories dhe çdo paisje I/O krahasojne linjën e adreses me range-in e adreses qe ato ofrojnë. Në qoftë se adresa bie ne range-in e tij, ajo i përgjigjet kerkeses. Të dy skemat për adresimin e kontrollerave kane të mira dhe të keqija të ndryshme. Le të fillojme me avantazhet e memory-mapped I/O. E para, ne se kerkohen instrukSIONE speciale I/O për të shkruajtur apo lexuar regjistrat e kontrollit të paisjeve, aksesimi i tyre do të kerkoj përdorimin e kodeve assembler përderisa është e pamundur për të ekzekutuar një instrukSION IN ose OUT ne C ose C++. Therritja e një procedure të tille do të shtonte komplikimin ne kontrollin e I/O. Ne të kundert, me memory-mapped I/O regjistrat e kontrollit jane thjeshtë variabla ne memory dhe mund të adresohen ne C ne të njëjtën menyre si çdo variabel tjeter. Keshtu, me memory-mapped I/O, një driver i një paisje I/O mund të shkruhet i téri ne gjuhen C. Pa memory-mapped I/O do të duhen disa komanda ne assembler.

E dyta, me memory-mapped I/O nuk na nevoitet ndonjë mekanizem special mbrojtjeje qe të ndaloj proceset e përdoruesit qe të përdori I/O. Gjithcka qe duhet të bej sistemi operativ është qe të shmang futjen e pjeses se hapesirave qe përbajne regjistrat e kontrollit ne hapesiren e adresave virtuale të përdoruesit. Me mire akoma, ne se çdo paisje ka regjistrat e kontrollit ne faqe të ndryshme të hapesires se adresave, sistemi operativ mund të japi një kontroll nga përdoruesi mbi specifikat e paisjeve thjesht duke përfshire faqet e deshirueshme ne page table e tij. Një skeme e tille mund të lejoj drivera të paisjeve të ndryshme të vendosen ne hapesira adresash të ndryshme, jo vetëm qe redukton përmasat e kernel por nuk lejon qe driverat të interferohen me njëri tjeterin.

E treta, me memory-mapped I/O çdo instrukSION mund të adresoj memorien mund edhe të adresoj regjistrat e kontrollit. Për shembull, ne se do kemi një instrukSION TEST qe teston një fjale të memories për 0, ajo gjithashtu mund të përdoret për të testuar një regjistër kontrolli për 0, e cila mund të jetë sinjali qe paisja nuk po rri koton dhe mund të pranoj një komand të re. Kodi ne gjuhen assembler do ketë ketë forme:

```
LOOP:    TEST PORT_4          //kontrollon ne se porta 4 është 0
```

```

BEQ READY           //ne se është 0 shko te ready
BRANCH LOOP        //ne të kundert vazhdo testin

```

READY:

Në qoftë se memory-mapped I/O nuk është presente, regjistrat e kontrollit duhet të jene gati ne CPU, me pas të testohen, duke kerkuar dy instruksione ne vend të një. Ne rastin e një loop, duhet të shtohet edhe një instruksion i katërt i cili bën qe të ulet ne menyre të avasht aftësia për të detektuar ne se paisja nuk po ben gje.

Ne dizenjimin e kompjuterave, çdo gje praktikisht ka të bez me shkembime. Memory-mapped I/O ka edhe kjo disavantazhet e saj. Se pari kompjuterat e sotëm kane disa forma kapje të fjaleve ne memorie. Të kapesht një paisje regjistër kontroll mund të jetë një shkatërim i vertet. Konsideroni kodin assembler për një loop të dhene me poshtë ne rastin kur të kemi një kapje. Adresimi i pare për PORT_4 mund të shkaktoj një kapje. Adresimet pasues do tu duhej ta merrnin vleren nga cache pa e pyetur fare paisjen. Me pas kur piasja të behet gati software nuk do të ketë mundesi ta zbuloj atë. Dhe cikli loop do të vazhdoj keshtu përfundimisht.

Për ti paraprire kesaj situate me memory-mapped I/O, hardware duhet të ketë aftësin qe të caktivizoj kapjen ne menyre selektive. Ky tipar do të rrissi kompleksitetin si për hardware-in ashtu edhe për sistemin operativ i cili ka për detyre të drejtoj kapjen selektive.

Se dyti, ne se do të kemi një hapesire adrese të vetme atëhere të gjithe modulet e memories dhe të gjithe paisjet I/O duhet të kontrollojnë të gjitha adresimet e memories për të pare se cilil i përgjigjen. Në qoftë se kompjuteri do të kishte vetëm një bus, si ne fig.5-3, kerkimi i adresave nga të gjithe behet ne menyre të “ndershme”.

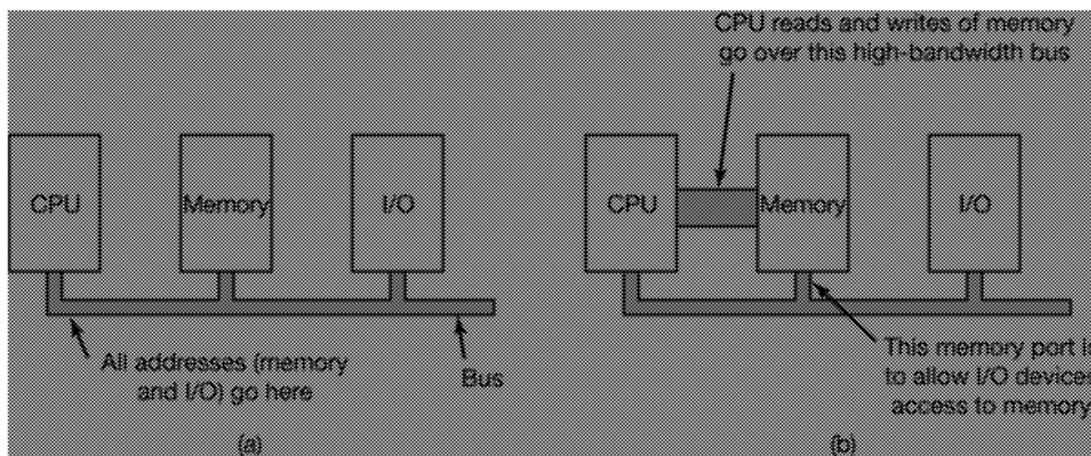


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

Tendenca për kompjuterat e sotëm personal është qe të kene një bus memorie me shpejtësi të madhe si ne fig.5-3(b), një karakteristikë e mainfraimeve. Ky bus i është dhene për të optimizuar performancen e memories pa u rezikuar nga shpejtësia e vogel e paisjeve I/O. Sistemet pentium kane 3 buse të jashtëm (memory, PCI, ISA), të treguara ne fig.1-11.

E meta e të paturit një memorie të ndare ne makinat memory-mapped është qe paisjet I/O nuk i shikojne adresat e memories pasi shkojne ne buset e memories, dhe keshtu qe nuk kane mundesi të përgjigjen. Prapo duhet të behen matje speciale për të bere të mundur memory-mapped I/O të punojne ne sisteme me shume buse. Një mundesi është qe të dergojme ne fillim të gjitha referencat e memories ne memorie. Në qoftë se memoria nuk na kthen përgjigje, atëherë CPU përdor buset e tjere. Kjo arkitekturë është bere qe të punoj por do kerkonte një kompleksitet të madh hardware-ik.

Një arkitekturë tjetër e mundshme është vendosja e një paisje zhbiruese ne busin e memories qe të bej kalimin e adresave të paisjeve I/O të interesuara. Problemi ne ketë rast qendron pasi paisjet I/O kane shpejtësi me të vogel se memoria.

Një arkitekturë tjetër, e cila përdoret sot te Pentium është vendosja e një filtri adresash ne chipin PCI. Ky chip përmban një linjë regjistrash të cilat preludohen ne kohen e butimit. Për shembull, 640 K deri ne 1 M mund të shenohet si një linjë jo e memories. Adresat të cilat nuk i përkasin linjave të memories shkojne te busi PCI ne vend të memories. Disavantazhi i kesaj skeme është qe duhet të dihet ne kohen e butimit se cilat nga adresat e memories janë vertet adresa të memories.

5.1.4 DIRECT MEMORY ACCESS (DMA)

Nuk ka rendesi ne se CPU ka apo jo memory-mapped I/O, asaj i i duhet të dergoj adresën paisjeve kontrolluese për të shkembyer të dhena me to. CPU kerkon të dhena nga paisjet I/O me nga një byte ne një cast të kohes gje qe do të sillte humbjen e kohes se CPU, për ketë ofrohet një skeme tjetër e quajtur **DMA (Direct Memory Access)**. Sistemi operativ mund të përdor DMA vetëm ne se edhe hardware ka një kontroller DMA, ku shumica e sistemeve e kane. Shpeshere ky kontroller është i integruar ne disk kontrollerin por arkitektura të ndryshme kerkojne kontrollera DMA të ndryshem për çdo paisje. Një DMA kontroller i vetëm është i gatshem për të rregulluar kalimet ne shume paisje, sipas një rregulli të caktuar.

Pavaresisht se ku ndodhet fizikisht, DMA kontrolleri ka akses me sistem bus-in duke mos u varur nga CPU, sic tregonet ne fig.5-4. Ajo përmban disa regjistra të cilët mund të shkruhen dhe të lexohen nga CPU. Keto përfshijnë një adresë regjistër për memorien, një byte për numrator regjistrin dhe një ose me shume rregjistra kontrolli. Rregjistrat e kontrollit specifikojnë portat I/O qe do të përdoren, drejtimin e transferimit (leximin nga

paisjet I/O ose shkrimin ne paisjet I/O), njësin e transferimit (byte ne një cast të kohes ose fjale ne një cast të kohes) dhe numrin e byteve, qe do transmetohen ne një vrull.

Për të shpjeguar se si punon DMA, le të shikojme ne fillim se si lexon një disk kur nuk përdoret DMA. Ne fillim kontrolleri lexon blloqet ne menyre seriale nga driveri, bit për bit, derisa i gjithe blloku është ne pjesen e jashtme të bufferit të kontrollerit. Me pas ai llogarit në qoftë se është bere ndonjë gabim gjatë leximit. Dhe me pas kontrolleri shkakton një interrupt. Kur sistemi operativ fillon, ai mund të lexoj disk bllokun nga bufferi i kontrollerit me byte ose fjale për një cast kohe duke ekzekutuar një loop, me çdo lexim ne menyre të përsëritur të byteve ose fjaleve nga rregjistrat e kontrollerit të paisjes ne memorien kryesore.

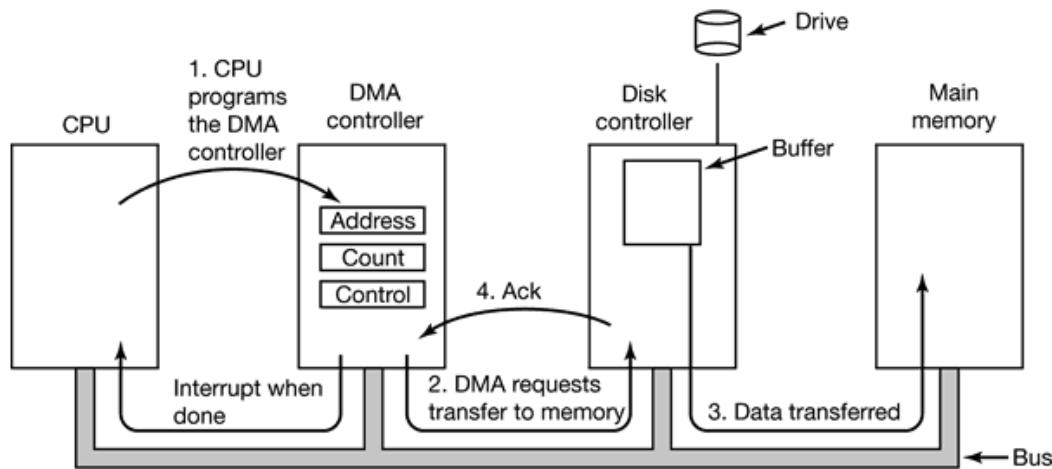


Figure 5-4. Operation of a DMA transfer.

Kur përdoret DMA, procedura ndryshon. Ne fillim CPU programon kontrollerin DMA duke i vendosur rregjistrat e saj dhe keshtu e di se ku dhe cfare do transportoj. Ajo gjithashtu i jep një komand disk kontrollerit duke i treguar atij të lexoj të dhenat nga disku ne bufferin e jashtëm dhe të verifikoj checksum-in. Kur të dhenat e vlefshme jane ne bufferin e disk kontrollerit, DMA mund të nisi.

Kontrolleri DMA nis transfertën duke dhene një kerkes leximi nepërmjet bus-it për dis kontrollerin (hani 2). Kjo kerkes për lexim duket njësoj si çdo kerkes tjetër për lexim dhe disk kontrolleri nuk e di ne se kjo kerkes po vjen nga CPU-ja ose nga kontrolleri DMA. Adresa e memories ku do të shkruhet është ne busin e adresave dhe keshtu kur disk kontrolleri kap fjalen tjetër për të shkruajtura nga bufferi i jashtëm, di se ku ta shkruaj atë. Për të shkruajtura ne memorie është një tjetër cikel standart busi (hani 3). Kur përfundon shkrimi, disk kontrolleri dergon një signal ack disk kontrollerit nepërmjet busit (hani 4). Kontrolleri DMA incrementon adresen e memories për të përdorur dhe decrementon numeruesin e byte-ve. Ne se numeruesi i byte-eve është me i madh se zero, përsëritet hani i dyte dhe i katërt derisa numeruesi behet zero. Ne ketë kohe, kontrolleri DMA i dergon një interrupt CPU për ti bere të ditur qe transferimi tashme u krye. Kur sistemi

operativ fillon, ai nuk ka pse të kopjoj disk bllokun ne memorie sepse ai gjendet tashme aty.

Kontrollerat DMA ndryshojne shume ne sofistikimin e tyre. Me i thjeshti prej tyre përballon një transfert ne një cast të kohes. Llojet e tjera me komplekse mund të programohen qe të përballojnë shume transferata njehersh ne një cast kohe. Kontrollera të tille kane paisje paralele të rregjistrave të jashtëm, nga një për çdo kanal. CPU fillon loading-un për çdo paisje të rregjistrave me parametrat e përshtatshem qe do të transferoj.

Pasi çdo fjale është transmetuar (hapi 2 deri ne 4) ne fig.5-4, kontrolleri DMA vendos se ciles paisje do ti sherbej. Ajo mund të përdori disa algoritma të caktuar si round-robin ose mund të përdorin prioritet ku një paisje ka prioritet me të lartë se tjetra e keshtu me rradhe. Kerkasat e shumta drejt kontrollerave të paisjeve të ndryshme mund të varen nga e njëjtë kohe, duke parashikuar se atje është një menyre e qartë për të ndare ack menjane. Zakonisht linja të ndryshme ack ne bus përdoren për çdo kanal DMA për ketë arsyen.

Shume buse mund të operojne ne dy forma: forma një fjale për një cast kohe dhe forma ne bllok. Disa kontrollera DMA mund të operojne ne forma të tjera. Ne formen e pare operacioni është sic përshkruhet me poshtë: kontrolleri DMA kerkon për transferten e një fjale dhe e merr atë. Ne se edhe CPU e kerkon busin atëhere atij i duhet të pres. Ky mekanizem quhet cikli I vjedhjes sepse kontrolleri i paisjes vjedh një cikel busi të rastesishem nga CPU. Ne formen me blloqe kontrolleri DMA i thotë paisjes të siguroj një bus, lehon një seri transfertash dhe me pas liron busin. Kjo forme e të vepruar quhet **burst mode**. Është me eficiente se cikli I vjedhjes sepse të marresh një bus kerkon kohe dhe shume fjale mund të dergohen me të njëjtin cmim si një marrje busi. Ana negative e kesaj metode është qe mund të bllokoj CPU dhe paisje të tjera për një kohe të konsiderueshme ne se një **burst** i gjatë mund të jetë duke u transportuar.

Ne ketë model qe po flasim, shpesh here i quajtur **fly-by mode**, kontrolleri DMA i tregon kontrollerit të paisjes të transferoj te dhena direkt e ne memorien kryesore. Një forme alternative qe përdorin disa DMA është qe të kene disa kontrollera paisjesh qe ti dergojne fjale kontrollerit DMA i cili me vone i cakton një kerkese për të shkruajtj fjalen atje ku është nisur për të shkuar. Kjo skeme kerkon një cikel busi ekstra për çdo fjale qe do transmetohet por është me fleksible për sa i përket kopjimit paisje me paisje dhe kopjimit memorie me memorie.

Shumica e kontrollerave DMA adresat e shtreses fizike për transfertat e tyre. Për të përdorur adresat fizike sistemi operativ duhet të bej përkthimin nga adresa virtuale ne atë fizike dhe ta shkruaj atë ne rregjistrat e adreses se kontrollerit DMA. Një skeme alternative e përdorur ne disa rregjistra adresash të kontrollerit DMA është qe të shkruaj adresat virtuale ne kontrollerin DMA. Dhe me pas kontrolleri DMA i duhet të përdori MMU për të bere përkthimin nga adresa virtuale ne atë fizike.

Me pare përmendem qe disku lexon të dhenat ne busin e tij të jashtëm përpëra se të filloj DMA. Ju mbante pyesni veten pse kontrolleri nuk i dergon menjehere bytet ne memorjen kryesore apo i merr ato nga diskut. Me fjale të tjera pse i duhet një buffer I jashtëm? Për ketë ka dy arsyen. Ne fillim duke bere bufferimin e jashtëm kontrolleri i diskut verifikon

checksumin përpëra se të nis një transfertë. Ne se checksumi është i parregullt, sinjalizohet një error dhe transferimi nuk kryhet.

Arsyeja e dytë është qe bitet vijne nga disku me të njëjtën shpejtësi pavaresisht ne se kontrolleri është gati për to ose jo. Ne se kontrolleri do mundohet të shkruaj të dhena direkt ne memorie ai do të ndryshonte sistemin e busit për çdo fjale të transmetuar. Ne se busi është i zene me disa paisje të tjera, kontrollerit i duhet të pres. Ne se fjala e rradhes vjen përpëra se fjala qe ka rradhen për të ardhur, kontrollerit i duhet qe ta ruaj atë diku. Ne se busi është i zene, kontrolleri mund të përfundoj marrjen e fialeve të vogla dhe të ketë një administrim të plotë për ta bere po aq mire. Kur bloku është bufferuar jashtë, nuk nevojitet busi derisa të nis DMA dhe keshtu ndertimi i kontrollerit është me i thjeshtë, sepse transferimi DMA për ne memorie nuk është time critical. (Disa kontrollera të vjetër, ne fakt shkojne direkt për ne memorie, por kur busi ishte shume i zene një transfertë mund të mos kryhej.)

Jo të gjithe kompjuterat përdorin DMA. Kjo shpjegohet se CPU është shume me e shpejtë se kontrolleri DMA dhe mund ta bej punen shume me shpejtë. Ne se nuk do të ketë pune të tjera për të bere, duke pasur një CPU të shpejtë dhe një kontrollues DMA të avashtë nuk do të kishte kuptim.

5.1.5 Interrupts Revisited

Ne shkurtimisht i përshkruam interruptet ne Sec. 1.4.3, por ka akoma shume për të thene. Ne një sistem të një kompjuteri personal, struktura e interrupteve jezet si ne Fig.5-5.

Ne niveli hardware interruptet punojne si me poshtë. Kur një paisje I/O përfundon punen qe i është caktuar, ajo shkakton një interrupt. Ajo e bën ketë duke kerkuar një sinjal ne një linje busi të caktuar. Sinjali kapet nga cipi i kontrollerit të interrupteve, i cili me pas vendos se cfare të bej.

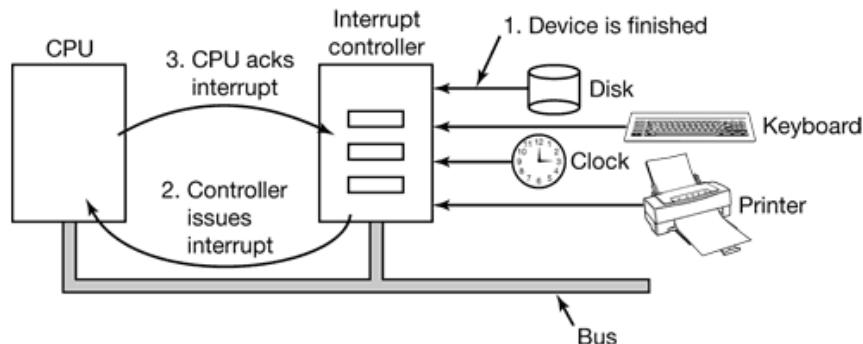


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Ne se nuk ka interrupte të tjere, kontrolleri i interrupteve përpunon menjehere interruptin. Ne se ndonjë tjetër është ne progres ose një paisje tjetër ka bere një kerkes me prioritet të lartë, paisja injorohet për momentin. Ne ketë rast ajo vazhdon të kerkoj një sinjal interrupt ne bus derisa ajo të sherbehet nga CPU.

Për të përballuar interruptet, kontrolleri vendos një numer ne linjën e adresave qe të specifikoj se cila paisje do kujdes dhe kerkon një sinjal qe të shkaktoj interrupt ne CPU.

Sinjalet e interruptit detyrojne CPU të nderpres atë qe po bënte për të bere dicka tjetër. Numri ne linjën e adresave përdoret si një indeks ne një tabel të quajtur **interrupt vector** qe të sjell një program counter të ri. Ky program counter shenjon ne fillimin e procedures se sherbimit të interrupteve korrespondues. Trap-et dhe interruptet përdorin të njëjtin mekanizem duke i pare nga kjo pike dhe zakonishtë ndajne të njëjtin vektor interruptesh. Vektori I interrupteve mund të ndodhet ne makine si një pjese hardware ose ne çdo vend të memories, me një rregjistër CPU qe pointon ne fillimin e tij.

Pak para se ajo të vazhdoj ekzekutimin, procedura e sherbimit të interrupteve njofton për interrupt duke shkrnuajtur një vlere të caktuar ne një nga portat I/O të kontrollerit të interrupteve. Kjo ack njofton kontrollerin qe është I lire për të leshuar një tjetër interrupt.

Disa kompjutera (të vjetër) nuk kane një chip kotrolleri interruptesh të centralizuar prandaj çdo kontroller paisjesh kerkon interruptet e tij.

Hardwaret gjithmon ruajne një informacion të caktuar përpра se të nisi procedura e sherbimeve. Se cfare informacioni është ruajtur dhe se ku është ruajtur varet shume nga CPU te CPU. Program counter duhet ruhet dhe keshtu qe procesi i interrupteve duhet mund të ristartohet. Ne anen tjetër të gjithe rregjistrat e dukshem dhe një numer i madh i rregjistrave të brendshem mund të ruhen po aq mire.

Një ceshtje tjetër është se ku do ruhet ky informacion. Një opsjon është qe ta vendosim atë ne rregjistrat e jashtëm dhe sistemi operativ mund ta lexoj atë sa here qe ti duhet. Një problem qe na lind është qe kontrolleri I interrupteve nuk mund të verifikohet derisa i gjithe informacioni I përshtatshem të lexohet, vetëm, në qoftë se një interrupt i dytë të rishkruej rregjistrin e jashtëm duke ruajtur gjendjen. Kjo strategji sjell një kohe të madhe të vdekur kur interruptet jane të caktivizuar dhe ndoshta interruptet mund të humbin gjithashtu edhe të dhenat.

Si pasoj shume CPU e ruajne informacionin ne stack. Gjithesesi edhe kjo menyre ka problemet e saj. Të nisim me pyetjen: staku i kujt? Ne qoftë se staku i rradhes është i përdorur ai mund të jetë një stack proces i përdorur. Stack pointeri mund të mos jetë i njëjtë, gje qe mund të sjell një gabim fatal kur hardware mundohet të shkruej fjale ne të. Gjithashtu ai mund të pontoj ne fund të faqes. Pas disa shkrimesh ne memorie, faqia tjetër mund të kalohet dhe një faqe gabim mund të gjenerohet. Duke patur faqe gabim qe ndodhin gjatë procesit të interrupteve hardware, krijon një problem të madh: ku ta ruaj gjendjen qe të merret me faqet gabim?

Në qoftë se stacku i kernel-it është i përdorur ka shume mundesi qe stacku i pointerit të jetë i njobur dhe të puntoj ne një pinned page. Gjithesesi switchimi ne kernel mode mund të sjell ndryshim ne kontekstet e MMU dhe mund të paralizoj shume ose të gjitha cache

dhe TLB. Duke riloaduar të gjithe ketë, ne menyre statike ose dinamike mund të rrisi kohen për të proçesar një interrupt dhe kjo do të sjell humbjen e kohes se CPU.

Një problem tjetër lind ngaqë shume CPU moderne punojne ne pipeline dhe zakonishtë jane superscale (paralel ne brendesi). Ne sistemet e vjetra, pas çdo instruksioni përfundonte ekzekutimi, mikroprogrami ose hardware shikonte ne se kishim ndonjë interrupt të ngelur pezull. Ne se po, program counteri dhe PSW shtyheshin për ne stack dhe fillonte sekuenca e interrupteve. Pasi ekzekutohet interrupti, proçesi i anasjelltë ndodh dhe PSW e vjetër dhe program counteri bejne pop nga stacku dhe proçesi i ardhshem vazhdon.

Ky model tregon qe, në qoftë se një interrupt ndodh mbas disa instruksioneve, të gjithe instruksionet duke përfshire dhe atë instruksion, ekzekutohen plotësisht, dhe asnje instruksion mbas atij nuk ekzekutohet. Ne një makine të vjetër kjo gje ka qene gjithmone e vlefshme. Ne makinat moderne mund të mos jetë me.

Për të nisur, konsideroni modeli pipeline ne fig.1-6(a). Cfare ndodh kur kerkohet një interrupt nderkohe qe pipeline është i ngarkuar (rasti I zakonshem)? Shume instruksione jane ne fazë të ndryshme ekzekutimi. Kur ndodh një interrupt, vlera e program counter nuk mund ta pasqyroje kufirin e saktë midis një instruksioni të ekzekutuar dhe të një instruksioni të pa ekzekutuar. Ajo pasqyron adresen e instruksionit tjetër pasardhes qe do të merret dhe do të vendoset ne pipeline, sesa adresen e instruksionit i cili sapo u ekzekutua nga njësia e ekzekutimit.

Rrjedhimisht mund të jetë një kufi i mire përcaktuar midis instruksioneve të cilat sapo jane ekzekutuar dhe të atyre të cilat presin për tu ekzekutuar, por hardware mund të mos e njobhi ketë. Kur sistemi operativ kthehet nga një interrupt, ai nuk mund të nisi direkt mbushjen e pipeline nga adresa e ruajtur ne program counter. Ajo duhet të tregoj se përfçfare duhej instruksioni qe sapo u ekzekutua, zakonisht është një pune komplekse qe duhet për të analizuar gjendjen e makines.

Ndone se kjo situate ishte e keqe, interruptet ne një makine superskalare, si ato ne fig.1.6(b) jane akoma me keq. Ngaqë instruksionet mund të ekzekutohen jo sipas një rradhe, nuk mund të ketë një kufi të mire përcaktuar midis një instruksioni të ekzekutuar dhe të një instruksioni të pa ekzekutuar. Mund të ndodh qe instruksionet 1,2,3,5, dhe 8 të jene të ekzekutuar dhe instruksionet 4,6,7,9 dhe 10 mund të mos jene ekzekutuar. Program counter mund të pointoj ne instruksionin 9, 10 ose 11.

Një interrupt qe le makinen ne një gjendje të mire përcaktuar quhet interrupt precis. Një interrupt ka katër karakteristika:

1. PC (program counter) ruhet ne një vend të njobur.
2. Të gjithe instruksionet para se të pointohet I pari nga PC, kane ekzekutim të plotë.
3. Asnjë instruksion pasi është pointuar I pari nga PC, ka qene i ekzekutur.
4. Gjendja e ekzekutimit të një instruksioni të pointuar nga PC është e njobur.

Shenojme qe nuk ka asnje ndalim për instruksionet pasi I pari është pointuar nga PC qe nga fillimi. Çdo ndryshim qe bejne për tu rregjistruar ose për tu memorizuar, duhet të jetë

I përfunduar përpara se të ndodh një interrupt. Është e lejuar qe instruksioni i pointuar të jetë i ekzekutuar. Gjithashtu është e lejuar qe instruksioni i pointuar të mos jetë i ekzekutuar. Gjithesesi, duhet të jetë e qartë se cila forme kerkohet. Zakonisht, në qoftë se kemi një interrupt për I/O, instruksioni nuk duhet të ketë nisur akoma. Në qoftë se interrupti është një trap ose page fault, atëherë PC do të pointoj te instruksioni qe shkaktoi fault-in keshtu qe ai mund të ristartoj me vone.

Një interrupt qe nuk plotëson keto kerkesa quhet interrupt jo preciz dhe ja nxin jetën atyre qe programojne sisteme operative, të cilët duhet të vrasin mendjen se cfare ndodhi dhe se cfare pritet të ndodhi. Makinat me interrupte joprecize nxjerrin një numer të madh gjendjesh të brendshme mbi stack-un për ti dhene sistemit operativ mundesi të zbuoje se cfare po ndodh. Duke ruajtur një numer të madh informacionesh ne memorie për çdo interrupt, do të kerkonte një kohe të madhe. Kjo do të sillte një situate ironike, pasi do të kishim CPU superskalare shume të shpejta të cilat jane të papërshtatshme për të punuar me interrupte të avashtë.

Disa kompjutera jane ndertuar ne menyre të tille ku disa interrupte dhe trap Jane precize dhe të tjere jo. Disa makina kane një bit i cili setohet për ti detyruar të gjithe interruptet të jene preciz. Ana e keqe e kesaj është se ajo do të detyroj CPU qe të log-oj çdo gje qe po bën dhe të ruaj një kopje të rregjistrave ne menyre qe të gjeneroj një interrupt preciz ne çdo moment. E gjithe kjo ka një ndikim të madh ne performance.

Disa makina superskalare, sic Jane Pentium Pro dhe të gjithe pasardhesit kane interrupte precize për të lejuar programet e vjetër të 386, 486 dhe Pentium 1, qe të punojne ne menyre të rregullt. Cmimi qe paguhet për të patur interrupte precize nuk është ne kohe, por ne kompleksitetin e chipeve dhe të arkitektures. Në qoftë se nuk do të përdoreshin interrupte precize mund të ndertoheshin chipe të tille qe do të sillnin një CPU me shpejtësi shume të madhe. Nga ana tjetër interruptet joprecize do të bënin sistemin operativ me kompleks dhe me të avashtë, keshtu qe e kemi të vesh tire për të treguar se cila nga keto është me e mire.

5.2 Parimet e software I/O

Le ta leme hardware I/O dhe të shikojme software I/O.

Filimisht do të shqyrtojme synimet e software I/O, pastaj menyrat e ndryshme qe I/O mund të beje nga pikepamja drejtuese.

5.2.1 Synimet e software I/O

Një celes kyc ne synimet e software I/O është i njobur si pajisje e pavarur. Kjo do të thotë qe do të jetë e mundur të shkruash programe qe kane mundesi hyrjeje ne ndonjë pajisjeje I/O pa dashur të përcaktosh pajisjen me të rendesishme.

Për shembull një program qe i cili lexon një dosje si informacion duhet të jetë ne gjendje ne disketë, ne hard disk ose ne CD ROM, pa dashur të ndryshoje programin për pajisje të ndryshme.

Gjithashtu duhet të jetë ne gjendje të shkruaje një komande të tille si:

Sort<input>output

Dhe të punoje me informacionin qe del nga një disketë, IDE disk, SCSI DISK ose nga tastiera, dhe përpunimi i të dhenave del nga cfaredo lloj disku ose ne ekran.

I takon sistemit operativ (drejtues) të merret me problemet e shkaktuara nga fakti se keto pajisjeje jane vertetë të ndryshme dhe kerkojne komanda sekuencash të ndryshme për ti lexuar apo shkruar.

Relativisht shume afer me pajisjen e pavarur është **uniform naming**.

Emri i një dosje ose i një pajisjeje do të jetë thjesht një varg ose një numer i plotë dhe nuk duhe të varet për asnjë arsyje nga pajisja.

Ne **UNIX** të gjithe disqet mund të integrohen ne hirarkine e sistemeve të larta ne menyre arbitrale, ne menyre qe përdoruesi të mos dij cfare emri i përket çdo pajisjeje.

Për shembull: një disketë mund të qendroje ne direktorine *usr/ast/backup* keshtu qe ne vend qe të kopjosh një dosje ne *usr/ast/backup/monday* e kopjon file-in ne disketë.

Ne ketë menyre të gjitha dosjet dhe pajisjet jane adresuar ne të njëtin vend, ne një emer të vetëm.

Një tjetër emertim për software I/O eshte **error handling** (trajtim gabimesh). Ne përgjithesi gabimet duhet të trajtohen sa me afer hardware qe të munden. Ne se kontrolluesi zbulon një gabim ne lexim, ne se mundet, duhet të përpinqet ta rregulloje vetë. Ne se jo atëhere duhet të merret pajisja drejtuese, ndoshta vetëm duke u përpjekur të lexoje përseri bllokimin (gabimin). Disa gabime ndodhin rastesisht, si për shembull: gabim leximi i shkaktuar ne fillim dhe fshihet ne se veprimi përsëritet. Vetëm ne se **lower layers** (pjesa e poshteme) nuk është ne gjendje ta zgjidhe problemin, i duhet të informoje pjesen e sipërme. Ne se disa raste gabimet mund të behen me shume ne nivelin e poshtëm, jo ne nivelin e sipërm edhe pse e di gabimin.

Një tjetër emertim i rendesishem është sinkron (bllokues) ose anasjellta, versioni i asinkronizuar (nderprerje-drejtimi) ne transferime. Ne të shumtën e rasteve I/O është i pasinkronizuar. CPU fillon transferimin dhe kalon të dicka tjetër, derisa të vij interrupti. Përdoruesit e programeve e kane me të lehtë të shkruajne ne se proceset e I/O janë të bllokua, pas një thirrje të sistemit lexues, programi automatikisht nderpritet derisa të dhenat ne dispozicion ruhen ne kujtese. I takon sistemit drejtues të bez veprime qe janë aktualisht nderprerje -drejtimi, të shohe bllokimet ne përdorimin e programit.

Një tjetër emertim për software I/O është kujtesa. Shpesh të dhenat qe vijnë ne pajisje nuk mund të ruhen menjehere ne vendin e tyre përfundimtar. Per shembull, kur një pak oj vjen ne rrjet, sistemi drejtues nuk di ku ta vendose derisa ta ruaje diku dhe ta verifikoje.

Gjithashtu disa pajisje kane shume detyrime për shembull: pajisjet dixhitale audio, keshtu qe të dhenat duhet të vendosen ne kujtesen e përpunimit të të dhenave, ne avantazh të lidhe kategori ne të cilën kujtesa eshe plotësuar, akumuluar nga kategori të tjera, si rregull, 10 është numri qe ka kujtesa. Kujtesa përmban kopje të konsiderueshme dhe shpesh ka një ndikim të madh ne kryerjen e I/O.

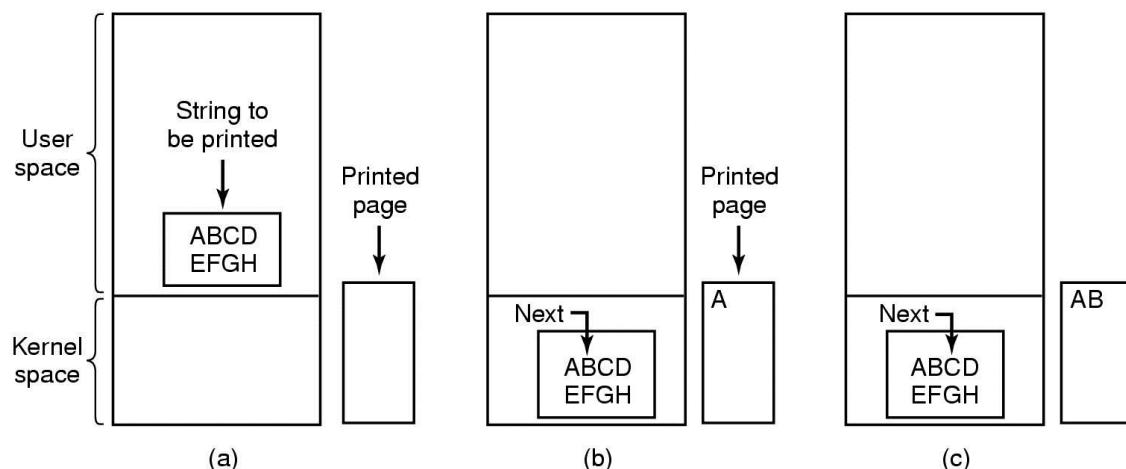
Koncepti i fundit qe do të përmendim i dedikohet pajisjeve. Disa pajisje I/O sic jane disqet, mund të përdoren nga përdorues të ndryshem ne të njëjtën kohe. Nuk shkaktohen probleme ne se përdorues të ndryshem kane hapur dosje ne të njëjtin disk, ne të njëjtën kohe. Pajisje të tjera sic jane: shiritat drejtues, duhet të kene një përdorues të vetëm derisa ai të ketë mbaruar. Pastaj, një përdorues tjetër mund të ketë shiritin drejtues. Dy ose me shume përdorues të shkrimit të bllokimeve të ndryshme rastesisht ne të njëtin shirit, përfundimisht nuk mund të funksionoje.

Njohja me pajisjet gjithashtu paraqet një sere problemesh sic jane pengesat. Përsëri sistemi drejtues duhet të jetë ne gjendje të kryeje të dyja funksionet, dhe ti përkushtohen pajisjes ne një menyre qe të shmange problemet.

5.2.2 Programimi I/O

Ka 3 menyra të rendesishme, të ndryshme me të cilat mund të shfaqet I/O. Ne ketë pjese të paren (**programmed I/O**). Ne të dy pjest e tjera do të analizojme të tjera nderprerje të pajisjeve I/O duke përdorur DMA

Forma me thjeshtë e I/O është qe të ketë CPU për të funksionuar. Kjo metode është quajtur programimi I/O. Është shume e thjeshtë të ilustrosh programin I/O nepërmjet një shembulli. Marrim një përdorues procesi qe do të printoje vargun prej 6 shkronjash "A B C D E F G" ne printer. Fillimisht grupon vargun ne një kujtese ne hapsiren përdoruese sic tregohet ne fig. 5.6 (a)



Pastaj përdoruesi i procesit ne seleksionimin e bere nga një një sistem qe përdoret për ta hapur atë.

Ne se printeri ne moment është ne përdorim nga një proces tjetër, ky proces do të deshtoje dhe të kthehet ne gabim kodi ose të bllokohet derisa printimi të realizohet duke u mbështëur ne sistemin drejtues dhe ne parametrat e kerkuara.

Kur printimi të behet, përdoruesi i procesit krijon një sistem për ti dhene komanden sistemit drejtues të printoje vargun ne printer.

Pastaj sistemi drejtues zakonisht kopjon kujtesen ne një informacion thotë «*p*» ne hapsiren kryesore, ku është me e lehtë hyrja (sepse hapsira kryesore i duhet të nderroje hartën e kujteses pér të marre një hapesire përdorimi).

Pastaj kontrollon ne se printeri momentalisht është gati. Ne se jo duhet të prese derisa të behet gati. Kur printeri të jetë gati sistemi drejtues kopjon shkronjën e pare ne manualin e të dhenave të regjistruar, ne ketë shembull duke përdorur hartën kujtese I/O. Ky veprim aktivizon printerin. Shkronja nuk mund të shfaqet akoma sepse disa printojne kujtesen ose faqen e rradhes përpëra se të printoje çdo gje tjetër.

Megjithatë ne fig 5.6 (b) shohim qe shkronja e pare është printuar dhe pastaj sistemi ka shenuar "B" si shkronjën pasardhese pér tu printuar.

Pasi ka kopjar shkronjën e pare ne printer, sistemi drejtues kontrollon ne se printeri është gati të printoje një shkronjë tjetër. Ne përgjithesi printeri ka një regjistër të dytë qe i jep atë gjendje. Kryerja e shkrimit ne të dhenat e regjistruar bën qe printeri të mos jetë gati.

Kur kjo të ndodhe, printohet shkronja tjetër, sic tregohet ne fig 5.6 (c).

Kjo procedure vazhdon derisa të printohet gjithe vargu. Pastaj kontrolluesi kthehet të procesi i përdoruesit veprimet e bera (kryera) nga sistemi drejtues janë përbledhur ne fig 5.7.

Fillimisht të dhenat kopjohen ne kernel. Pastaj sistemi operativ fillon procesin e përpunimit të te dhenave të çdo shkronjë me radhe. Aspekti kryesor i programit I/O, i ilustruar qartë ne ketë figure është se pasi përpunon një shkronjë, CPU kontrollon vazhdimesht pajisjen ne se është gati të pranoje një tjetër. Ky veprim shpesh është quajtur **polling** ose **busy waiting**.

```

copy_from_user(buffer, p, count);           /* p is the kernel bufer */
for (i = 0; i < count; i++) {               /* loop on every character */
    while (*printer_status_reg != READY) ;   /* loop until ready */
    *printer_data_register = p[i];           /* output one character */
}
return_to_user();

```

Programimi I/O është i thjeshtë por ka një disavantazh, CPU punon pa pushim derisa të gjithe I/O të kenë mbaruar. Nese koha per te “printuar” një karakter eshte shume e shkurtet (sepse gjithçka qe një printer ben eshte te kopjoje karakterin ne një buffer te brendshem), me pas busy waiting kalon ne maje. Gjithashtu ne një sistem të nderfutur, ku CPU nuk ka cfare të beje tjetër, busy waiting eshte me funksionale. Megjithatë ne shume sisteme të nderlikuara ku CPU bën pune tjetër busy waiting nuk funksionon. Është e nevojshme një metode me e mire I/O.

5.2.3 Nderprerjet drejtuese

Le te shqyrtojme rastin e printimit ne një printer i cili nuk con cdo karakter ne buffer por printon secilen kur vjen. Ne se printeri printon 100 karaktere/sekonde, çdo shkronjë i duhet 10 msec per tu printuar. Kjo do të thotë se pasi çdo shkronjë është shkruar ne regjistrin per printimin e te dhenave, CPU nuk punon për 10ms dhe pret konfirmimin qe të përpunoje të dhenat e shkronjës tjetër. Kjo është një kohe mese e mjaftueshme për të zgjuar dhe egzekutuar disa procese te tjera ne keto 10 msec, qe do te ishin gjithsesi te humbura.

Menyra me e mire qe CPU të punoje nderkohe qe pret printimin të behet gati është të përdore interruptet. Kur thirrja sistem per te printuar string eshte kryer, bufferi eshte kopjuar ne hapesiren kernel, sic e treguam me sipër dhe shkronja e pare kopjohet ne printer menjehere apo ai deshiron te pranoje një karakter. Ne ketë moment CPU therret scheduler dhe disa procese te tjera qe jane duke vepruar. Proçesi qe ka kerkuar printimin e vargut blokohet derisa i gjithe vargu të jetë printuar. Puna e kryer nga thirrja sistem është treguar ne fig 5.8 (a).

```

copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler();

if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();

```

(a)

(b)

Kur printeri ka printuar një shkronjë dhe është bere gati për të pranuar shkronjën tjeter gjenerohet një interrupt. Ky interrupt ndalon procesin qe eshte duke u egzektuar dhe e ruan ne atë gjendje. Pastaj egzekutohet procedura per sherbimin e interrupteve, një menyre paraqitjeje e papërpunuar e ketij rregulli është treguar ne fig 5.8 (b). Ne se nuk ka me shkronja për të printuar, zoteruesi i interrupteve kryen disa veprime per te bere aktiv perdoruesin. Përndryshe, ai nxjerr karakterin tjeter, pranon interruptet dhe kthehet tek procesi qe ishte duke vepruar para se te ndodhje interrupti, duke vazhduar nga aty ku ishte nderprere.

5.2.4 Programi I\O duke përdorur DMA.

Një dizavantazh i qartë i interrupt-driven I/O, është se një interrupt ndodh ne çdo shkronjë. Interruptet duan kohe, keshtu qe kjo skeme humbet njëfare kohe nga ajo e CPU-se. Një zgjedhje është qe të përdoresh DMA. Ideja është të lejosh kontolluesin e DMA-se të vendose shkronjat ne printer një nga një, pa ju dashur të nderhyje CPU. Ne thelb, DMA është program I\O, vetëm me kontolluesin e DMA behet puna, ne vend të CPU-se. Një skice e ketij rregulli është dhene ne fig 5.9.

<pre>copy_from_user(buffer, p, count); set_up_DMA_controller(); scheduler();</pre>	<pre>acknowledge_interrupt(); unlock_user(); return_from_interrupt();</pre>
(a)	(b)

Suksesi me i madh i DMA-se është ulje e numrit të interrupteve, nga një për çdo shkronjë, ne një ne bufferin e pare. Ne se ka disa shkronja dhe nderprerje ato jane të vogla, kjo mund të jetë një arritje e madhe. Nga ana tjeter kontolluesi DMA është shume me i ngadaltë se CPU. Ne se kontolluesi DMA nuk është ne gjendje të drejtoje pajisjen ne shpejtësine e duhur, ose CPU zakonisht nuk ka cfare të beje nderkohe qe pret per DMA-interrupt, pastaj interrupt-driven I\O ose edhe programi I\O mund të jetë me i mire.

5.3 SHTRESAT E SOFTWARE- it të I\O

Software-i I\O zakonisht merret si i përberë nga 4 shtresa, sic tregohet dhe në fig. 5-10. Çdo shtresë ka një funksion të mirë përcaktuar per te realizuar dhe një njerfaqe te mire përcaktuar me shtresat ngjitur. Funksionimi dhe ndërfaqja ndryshojnë nga një sistem në tjetrin, kështu që diskutimi që do pasoje, i cili shqyrton te gjithe shtresat qe nisin nga fundi nuk është specifik vetëm për një lloj makine.

5.3.1 Përgjegjësit e Interrupt-eve

Për pjesën më të madhe të paisjeve I/O, interruptet janë një fakt i padëshirueshëm që s'mund të shmanget. Ato duhet të fshihen në nivelet më të ulëta të sistemit operativ, në mënyrë që ky i fundit të dijë sa më pak për to. Mënyra më e mirë për ti fshehur ato është posedimi i një driver-i që ti bllokojë ato derisa I/O të përfundojë punën e vet dhe më pas interrupt mund të vazhdojnë. Driveri mund të bllokojë veten e vet në mënyrë të ngjashmë sic ndodh me semaforin, duke pritur në bazë të një variabli të kushtezuar ose te marrë një mesazh.

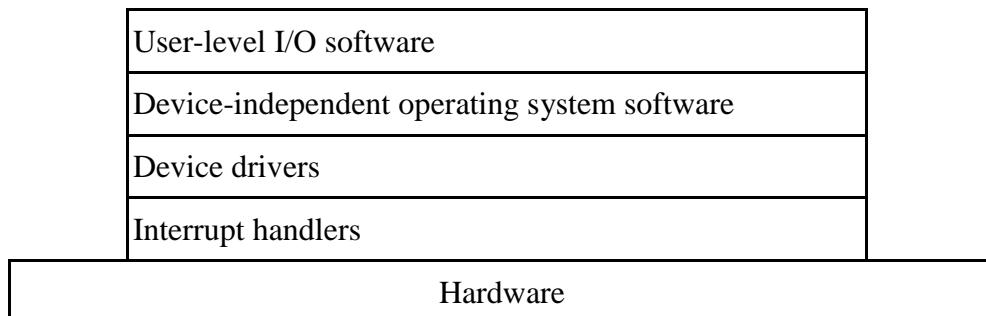


fig.5-10. Shtresat e sistemit software I/O.

Për shembull: Kur ndodh një interrupt, procedura e tij bën detyrën e vet në mënyrë që të arrijë qëllimin. Më pas ajo mund ta zbullokojë driverin që e filloj atë.

Në disa raste ai mund të veprojë sipas semaforit dhe në disa raste të tjera mund të përdorë një variabël të kushtezuar duke afishuar sinjalin e tij në një monitor. Dhe në raste të tjera dërgon një mesazh tek driveri i bllokuar. Në të gjitha rastet efekti i përgjithshëm i një interrupti është zbullokimi i driverit që ishte i bllokuar. Ky model është shumë i përshtatshëm kur driver-at janë strukturuar si procese kernel, me stivat, stak-et dhe program counter-in e tyre.

Sigurisht që në realitet nuk është aq e thjeshtë. Përpunimi i një interrupti nuk është të marrësh interruptin, ti bësh një "up" në semafor dhe më pas me anën e ekzekutimit të instruksionit **IRET** të kthehem nga interrupti tek procesi i mëparshëm. Është një punë shumë më e madhe e Sistemit Operativ për të menaxhuar interrupt-et. Tani le të japim një seri hapash duhet te kryhen ne software pasi interrupti hardware ka përfunduar. Duhet të kemi parasysh që detajet e mësiperme për shembull janë shume ne varesi te sistemit, kështu që disa nga hapat mund te mos jene te nevojshme ne makina të vecanta. Gjithashtu këto hapa do ti gjejmë të renditura ndryshe në makina të ndryshme.

- 1- Ruajtja e disa rregjistrave (përfshirë dhe atë PSW), që nuk janë ruajtur nga hardware i interrupteve.
- 2- Përcaktimi i një konteksti për procedurën e shërbimit të interrupteve. Këtu mund të përfshihet dhe përcaktimi i TLB, MMU dhe i page table.

- 3- Përcaktimi i një staku për procedurën e shërbimit të interrupte-ve
- 4- Njohja e kontrolluesit të interrupte-ve. Në se nuk ka kontroller interruptesh qëndror, bëhet riaktivizimi i interrupte-ve.
- 5- Kopjimi i rregjistrave nga ku ishin ruajtur (me shumë mundësi në ndonjë stak) tek tabela e proceseve.
- 6- Ekzekutimi i procedurës së shërbimit të interrupte-ve, e cila do të nxjerrë informacion nga rregjistrat e kontrollit të paisjes të interrupte-ve.
- 7- Zgjedh se cili proces do ekzekutohet në vazhdim. Në se interrupti kishte ndërprerë më përpara një proces me prioritet të lartë, është më e mundshme që ky proces të ekzekutohet pas interruptit.
- 8- Vendos kontekstin e MMU për procesin që do ekzekutohet në vazhdim. Mund të duhet dhe një “set up” për TLB.
- 9- Ngarkon rregjistrat e rinj të procesit duke përfshirë dhe PSW.
- 10- Fillon ekzekutimin e procesit të ri.

Sic mund të shihet procesi i interrupt-eve është i rëndësishëm. Ai kërkon gjithashtu një numër të konsiderueshëm instruksionesh CPU-je, sidomos në makinat ku memorja virtuale është prezantë. Në disa makina TLB dhe CACHE kanë nevojë të menaxhohen gjatë kohës së kalimit nga mode kernel në atë user, e cila kërkon cikle makinë shtesë.

5.3.2 Driver-at e paisjeve

Pak më sipër pamë se c'farë bënin kontrollerët e paisjeve. Pamë që çdo kontroller ka disa rregjistra paisjesh, që shërbejnë për të dhënë komandat e veta ose disa rregjistra paisjesh, që shërbejnë për të lexuar gjendjen e tij ose për ti bërë që të dyja njëkohësisht. Numri i regjistrave të paisjeve dhe natyra e komandave ndryshojnë në mënyrë radikale nga një paisje në tjetrën. Për shembull, driver-i i mouse duhet të marrë informacion nga vet mouse-i, që ti tregonë sa ka lëvizur dhe cilët janë butonat që janë shtypur. Në kontraste me këtë një driver disku duhet të dijë për sektorët, track-et, cilindrat, kokat, kohën e vendosjes së kokave, si dhe për shumë mekanizma të tjerë që bëjnë të mundur mirëfunkcionimin e diskut. Sic shihet, këto drivera do jenë të ndryshëm nga njëri-tjetri.

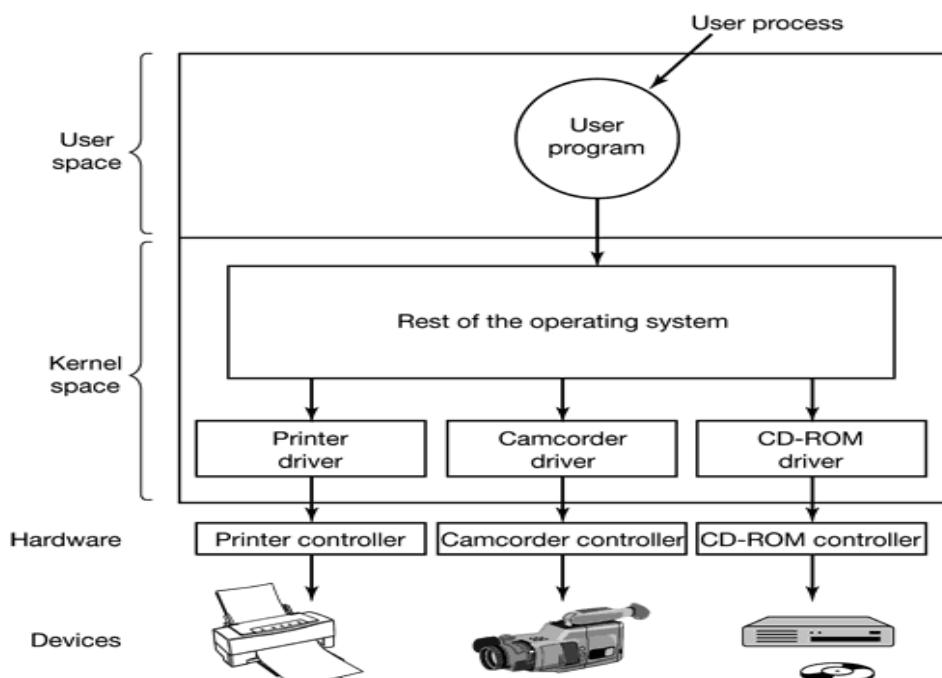
Si rrjedhim çdo paisje I/O e lidhur me një kompjuter kërkon një kod të vecantë për ta kontrolluar atë. Ky kod i quajtur driver paisjeje, përgjithësisht është shkruar nga prodhuesi i paisjes dhe është i bashkangjitur me paisjen. Duke qënë se çdo sistem operativ kërkon driver-at e vet, prodhuesit e paisjeve ndërtojnë drivera specifikë për sisteme të ndryshme operative (për ata që janë më popullorë).

Çdo driver normalisht përgjigjet vetëm për një lloj paisjeje, ose maksimumi për një bashkësi paisjesh të ngjashme me njëra-tjetrën. Për shembull, një driver disku SCSI mund të përgjigjet për një sasi të madhe disqesh SCSI të cilët mund të kenë madhësi dhe shpejtësi të ndryshme, dhe ndoshta të përgjigjet për një CD-ROM SCSI. Nga ana tjetër një mouse dhe një joystick janë aq të ndryshëm saqë duhen drivera komplet të ndryshëm

për secilin. Gjithsesi nuk ka kufizime teknike për sa i përket posedimit të një lloj driveri për paisje të llojeve të ndryshme. Thjesht nuk është një ide e mirë.

Në mënyrë që të aksesojë hardware-in e paisjes (rregjistrat e kontrollerit), driveri duhet të jetë pjesë e kernelit të sistemit operativ. Aktualisht është e mundur që të dizenjohen driver-a që punojnë në hapësirën user me thirrje sistemi për të lexuar dhe shkruar rregjistrat e paisjes. Në fakt ky dizenjim do ishte një ide e mirë, ku do izolohej kerneli nga driver-at dhe driver-at nga njëri-tjetri. Kjo do të eleminonte burimin ku do lindnin shumë ‘përplasje sistem’ (drivera që në një mënyrë ose një tjetër interferojnë me kernelin). Megjithatë duke qënë se në sistemet e sotme operative driverat ekzekutohen në kernel, ky është dhe modeli që ne do marrim në konsiderate.

Që në momentin që projektuesit e sistemeve operative kanë marrë parasysh faktin që mund të shtohen nga jashtë driver-a (kode) që më pas të instalohen në këtë sistem operativ, gjithashtu duhet krijuar një arkitekturë e përshtatshme për këtë lloj instalimi. Kjo do të thotë që duhet të kemi një model të mirëpërcaktuar se cfarë bën një driver dhe sesi ndërvepron ky me pjesën tjetër të SO. Driver-at e paisjeve zakonisht pozicionohen poshtë pjesës tjetër të SO sic mund ta shohim të ilustruar në fig.5-11.



Sistemet operative përgjithësisht i klasifikojnë driver-at në disa kategori. Kategoritë më të zakonshme janë ‘**block devices**’ (paisjet me bloqe), sic janë disqet që përbajnë të dhëna të shumefishta të adresueshme në mënyrë të pavarrur dhe ‘**character devices**’ (paisjet me karaktere), sic janë tastjerat apo printerat, të cilat gjenerojnë ose aksesojnë një sasi të madhe karakteresh.

Pjesa më e madhe e sistemeve operative përcaktojnë një standart ndërfaqje në mënyrë që të gjithe driverat me blloqe ta suportojnë, si dhe një standart të dytë ndërfaqje, në mënyra që të gjithe driverat me karaktere ta suportojnë. Këto ndërfaqe konsistojnë në një numër procedurash, të cilat thirren nga pjesa tjetër e SO në mëyrë që driveri të bëjë detyrën e vet. Procedura tipike janë ato që lexojnë një bllok (block device), dhe ato për të shkruar një karakter string (character device).

Në disa sisteme, SO është një program i vetëm binar, që përmban të gjithe driver-at që duhen të jenë të kompiluar në të. Kjo skemë ka qënë standarte përvite në sistemet UNIX, sepse ato ekzekutoheshin nga qëndra kompjuterike, paisjet I/O të të cilave shumë rrallë ndryshonin. Në se shtohej një paisje e re, sistemi administrator thjesht rikompilonte kernelin me driver-in e ri duke ndërtuar një kode binar të ri.

Në fillimet e kompjuterave personalë, paisjet I/O në to ishin të shumta në numër. Ky model nuk vazhdoi për shumë kohë. Pak përdorues janë të aftë të rikompilojnë kernelin, edhe pse ndoshta mund të kenë kodin burim apo modulet objekt. Më vonë sistemet operative, duke filluar me MS-DOS, kishin një model ku driver-at ngarkoheshin në mënyrë dinamike gjatë ekzekutimit. Sisteme të ndryshme kanë mënyra të ndryshme të ngarkimit të driver-ave.

Një driver paisjeje ka funksione të shumellojshme. Më i rëndësishmi është të pranojë lexim abstrakt dhe kërkesa shkrimi nga software-të pavarur nga paisja. Por janë dhe disa funksione të tjera që ai duhet të performojë. Për shembull, driver-i i duhet te inicializojë paisjen kur është e nevojshme. Gjithashtu driveri mund të menaxhojë kërkesat e fuqisë së paisjes ose mund të bëjë rregjistrimin e ngjarjeve.

Shumë drivera paisjesh kanë strukturë të ngjashme. Një driver përgithësisht e fillon punën e vet duke kontrolluar parametrat e hyrjes, për të parë në se ata janë të vlefshëm. Në të kundërt kthehet një “error”. Në se janë të vlefshëm, duhet bërë një përkthim në terma konkretë. Për një driver disku, kjo mund të kuptohet si marrja e një blloku numrash dhe vendosja e tyre në numrat kokë, track, sektor dhe cilindër sipas gjeometrisë së diskut.

Driver-i mund të kontrollojë në se paisja është në përdorim. Në se është në punë, kërkesa mund të vihet në rradhë përpunime të mëtejshme. Në se paisja nuk është në përdorim, ekzaminohet gjendja e hardware-it për të parë në se kërkesa mund të merret parasysh tani. Ndoshta mund të jetë e nevojshme që paisja të ndizet, para se transferimi të fillojë. Në momentin që paisja ndizet dhe është gati për punë, mund të fillojë kontrolli aktual.

Të kontrollosh një paisje do të thotë të ushtrosh një sekuncë komandash në të. Driver-i është vendi ku kjo sekuencë komandash përcaktohet, duke u varur dhe nga detyra që ka për të kryer. Pasi driver-i di se cfarë komandash ka për të ekzekutuar, ai fillon ti shkruajë ato në rregjistrat e kontrollerit të paisjes. Pas shkruarjes së komandave në kontroller, ndoshta duhet parë në se ai i ka pranuar këto komanda dhe në se është përgatitur për të pritur ato pasardhëse. Kjo sekuencë vazhdon derisa të gjitha komandat janë përcaktuar. Disa kontrollerave mund tu jepet një listë komandash nga memorja, që ti lexojnë dhe ti përpunojnë vet pa ndihmën e sistemit operativ.

Pasi përcaktohen të gjitha komandat, mund të ndodhë një nga dy situatat. Në disa raste driveri duhet të presë derisa kontrolleri të përfundojë ndonjë punë për të, dhe kështu bllokoni veten e vet derisa të vijë interrupt-i për ta zhbllokuar. Në raste të tjera operacioni përfundon pa vonesa dhe driveri nuk ka nevojë të bllokojë veten e vet. Si shembull i situatës vijuese, shndërrimi i ekranit në ‘character mode’ mund të bëhet duke shkruar disa bite në rregjistrat e kontrollerit. Nuk nevojiten veprime mekanike, kështu operacioni mund të përfundojë në disa nanosekonda.

Në rastin e parë driver-i i bllokuar do të ‘zgjohej’ nga interrupt-i. Në rastin e dytë driveri nuk do ishte kurrë “në gjumë”. Gjithashtu pasi operacioni përfundon, driver-i duhet të kontrollojë për gabime. Në se çdo gjë është në rregull, më pas driver-it mund ti duhet të kalojë të dhëna tek softi i pavarur nga paisja (një bllok apo u lexua). Në fund ai kthen një informacion gjendjeje (raport gabimesh) tek thirrësi i tij. Në se janë vënë në rradhë kërkesa të tjera, njëra nga to mund të zgjidhet dhe mund të fillojë. Në se nuk ka kërkesa të rradhitura, driver-i bllokoni veten dhe pret për kërkesën e rradhës.

Ky model i thjeshtë është vetëm një përshkrim në vija të trasha i realitetit. Shumë faktorë e bëjnë kodin tepër të komplikuar. Interrupt-i mund ta bëjë një driver të punojë. Gjithashtu ai mund të bëjë të punojë një driver dhe kur ai është për momentin në punë. Për shembull, kur driver-i i network-ut është duke përpunuar një paketë hyrëse, një paketë tjetër mund të vij. Për pasojë një driver në punë duhet të presë që të thirret disa caste më herët se përfundimi i thirrjes së parë.

Në një sistem fleksibël, paisjet mund të shtohen ose të hiqen dhe kur kompjuteri është në punë. Si rezultat kur një driver është i zënë duke lexuar nga ndonjë paisje, sistemi mund të informojë driver-in që përdoruesi apo e hoqi këtë paisje nga sistemi. Jo vetëm që duhet ndërprerë transferimi data I/O i momentit pa prishur strukturën e të dhënavë, por duhen hequr në mënyrë të kujdeshme dhe ato kërkesa në pritje ndaj paisjes që nuk është më, dhe thirrësve duhet “tu jepen lajmet e këqia”. Për më tepër shtimi i paisjeve mund të shkaktojë një konfuzion në kernel duke e bërë që të zhvendosë paisjet e vjetra nga driveri duke i dhënë atij ato të reja.

Driver-ave nuk u lejohet të bëjnë thirrje sistem, por ata shpesh mund të ndërvaprojnë me pjesën tjetër të kernelit. Përgjithësisht thirrjet drejt procedurave të caktuara lejohen. Për shembull, ka thirrje që shërbijnë për të alokuar dhe c’alokuar faqe hardware të memorjes, që përdoren si bufera. Thirrje të tjera të nevojshme janë ato për menaxhimin e MMU, kohuesve, kontrollerin e DMA, kontrollerin e interrupt-eve, etj.

5.3.3 Software-i I/O i pavarur ndaj paisjes

Megjithëse një pjesë e software-ve I/O janë specifikë ndaj paisjes, pjesë të tjera të tij janë të pavarura ndaj paisjes. Dallimi i vërtetë mes driver-ave dhe software-ëve të pavarur ndaj paisjes është varësia sistem, sepse disa funksione që mund të kryen në mënyrë të

pavarur nga paisja, për eficencë ose arsyе të tjera mund të kryen nga vet driver-at. Funksionet e treguara në fig.5-12 janë të kryera nga software të pavarur nga paisja.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Funksioni kryesor i softwar-it të pavarur nga paisja është të përformojë funksione I/O që janë të përbashkët për të gjitha paisjet, dhe të prodhojnë një ndërfaqe uniforme ndaj softwar-it të nivelit-user. Më poshtë do i trajtojmë në mënyrë më të detajuar temat e mësipërme.

Ndërfaqja uniforme ndaj driver-ave të paisjeve

Një detyrë e rëndësishme e një sistemi operativ është si ti bëjë paisjet I/O dhe driver-at të duken pak a shumë njëloj. Në se disqet, printerat, tastjerat, etj, janë të gjitha të ndërfaqura në mënyra të ndryshmë, sa herë që vendoset një paisje e re në punë, sistemi operativ duhet të modifikohet sipas kësaj të fundit. Detyrimi për të “hack-uar” sistemin operativ sa herë që vendoset një paisje e re, nuk është një ide e mirë.

Një aspekt i kësaj detyre është ndërfaqja ndërmjet driver-ave dhe pjesës tjetër të SO. Në fig.5-13 (a) jepet një ilustrim i situatës ku çdo driver ka një ndërfaqe të ndryshme me SO. Kjo do të thotë që funksionet e driverave për të thirrur në sistem ndryshojnë nga një driver tek tjetri. Kjo mund të nënkuftohet dhe që funksionet kernel që duhen mund të ndryshojnë në varësi të driver-it. Si përfundim mund të themi, që të ndërfaqësh një driver të ri kërkon programime shtesë.

Në kontrast me këtë, figura 5-13, (b) na tregon një pamje krejt tjetër, ku të gjithë driverat kanë të njëjtën ndërfaqe. Tani është shumë më e lehtë të vendosësh një driver të ri. Kjo do të thotë që projektuesi e driverave e dinë se cfarë pritet të përformohet nga këto driver-a. Në praktikë jo të gjitha paisjet janë identike, por janë një numër i vogël tipe paisjesh, të cilat kanë të dhëna të përgjithshme të njëjta. Për shembull, dhe paisjet me bloqe dhe karaktere kanë gjëra të përbashkëta.

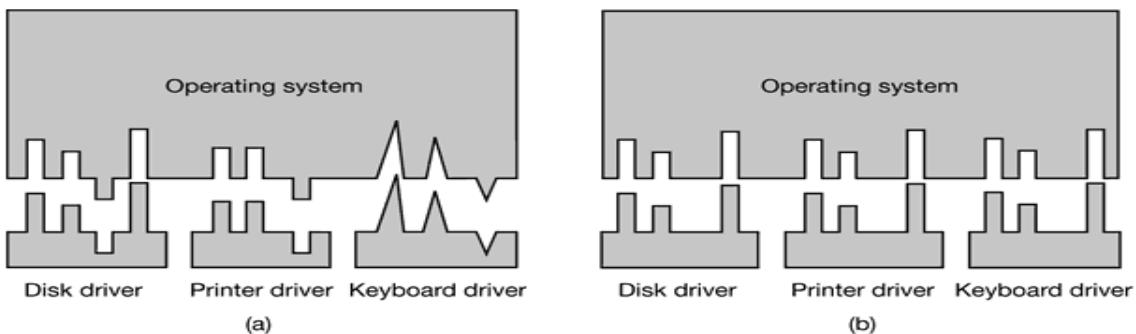


Fig.5.13

Një tjetër aspekt i posedimit të një ndërfaqje uniforme, është mënyra se si janë emërtuar paisjet I/O. Softwar-ët e pavarur nga nga paisja merren me vendosjen e emrave simbolikë të tyre në brendësi të driver-it përkatës. Për shembull, në UNIX emri i një paisjeje, si `/dev/disk0` specifikon i-node në mënyrë unike për një file specifik, dhe kjo i-node përmban numrin kryesor të paisjes, i cili shërben për të lokalizuar driver-in e përshtatshëm. Gjithashtu i-node përmban dhe numrin “minor”, i cili i kalohet si parametër driver-it për të specifikuar njësinë që do lexohet apo shkruhet. Të gjitha paisjet kanë numër kryesor dhe minor, dhe të gjithë driver-at aksesohen duke përdorur numrin kryesor të paisjes për të zgjedhur driver-in.

Shumë e lidhur me emërtimin është siguria. Si i parandalon sistemi përdoruesit që të aksesojnë paisjen që nuk është parapërcaktuar? Në UNIX dhe Windows 2000 paisjet shfaqen në ‘file system’ si objekte të emërtuara, që do të thotë që rregullat e zakonshme të mbrojtjes për ‘filet’ aplikohen dhe tek paisjet. Sistemi administrator më pas mund të përcaktojë leje të vecanta për çdo paisje.

Buffering

Buffering është gjithashtu një problem si për paisjet me blloqe, ashtu dhe për ato me karaktere për arsyet e ndryshme. Për të parë një prej tyre, le të marrim parasysh një proces që kërkon të lexojë të dhëna nga një modem. Një strategji e mundshme për të menaxhuar karakteret e hyrjes, është të bëjmë procesin e përdoruesit që të bëjë një thirrje sistem leximi dhe të bllokojë pritjen e një karakteri. Çdo karakter që vjen shkakton një interrupt. Procedura e shërbimit të interrupteve e kalon karakterin tek procesi i përdoruesit dhe e zhbllokon atë. Pasi e con karakterin diku, procesi lexon një karakter tjetër dhe bllokohet prapë. Ky model ilustrohet në fig5-14,(a).

Shqetësimi me këtë mënyrë të bëri ‘bisnes’ është që procesi i përdoruesit duhet te fillohet për çdo karakter që duke bërë që procesi të ekzekutohet shumë herë për ekzekutime shumë të shkurtra. Kjo nuk është eficiente, kështu që ky model nuk është një ide e mirë.

Një përmirësim tregohet në fig.5-14, (b). Këtu procesi përdorues ka një buffer me n-karaktere në hapësirën përdorues dhe bën një lexim prej n-karakteresh. Procedura e shërbimit të interrupteve i vendos karakteret në këtë buffer derisa të mbushet plot. Më pas ajo zgjon procesin përdorues. Ky model është shumë më tepër eficient se ai i pari, por dhe ky ka një problem: cfarë ndodh në se kur vjen karakteri, bufferi nuk gjendet në

memorje, por është zhvendosur? Buffer-in mund të kycim në memorje, por në se shumë procese fillojnë të kycin faqe në memorje, kërkimi në të pér faqe të gatshme do zvogëlohej, gjë që do sillte degradimin e performancës.

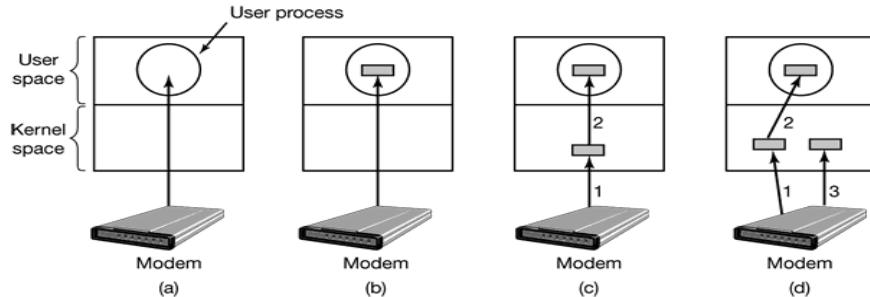


Figure 5-14. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Një përmirësim tjetër është në fig.5-14, (c) ku vendoset një buffer në kernel dhe përgjegjësi i interrupt-eve i con karakteret në të. Kur ky buffer mbushet plot, fajja me buffer-in e përdoruesit sillet aty dhe dë se nevojitet, këto të dhena kopjohen direkt në këtë buffer. Ky model është shumë më eficient.

Megithatë dhe kjo skemë e ka një problem: cfarë ndodh me karakteret që mbërrijne gjatë kohës që fajja e buffer- it përdorues po sillet nga disku? Duke qënë se buffer-i është plot nuk kemi ku ti cojmë këto karaktere. Një mënyrë është që të kemi një tjetër buffer në kernel. Pasi mbushet buffer-i i parë, para se ai të boshatiset, përdoret buffer-i i dytë, sic tregohet në fig.5-14, (d). Kur buffer-i i dytë mbushet, mund të kopjohet në hapësirën e përdoruesit. Gjatë kohës që po kopjohet buffer-i i dytë, ai i pari mund të presë karakteret e rind që po vijnë. Në këtë mënyrë të dy buffer-at punojnë më turne: gjatë kohës që njëri punon, tjetri akumulon. Kjo skemë është quajtur “double buffering”.

Buffering është gjithashtu i rëndësishëm në dalje. Për shembull, le të imagjinojmë sesi bëhet dalja në modem pa buffering duke përdorur modelin e fig.5-14, (b). Proçesi përdorues ekzekuton një thirrje sistem shkrimi pér të dhënë në dalje n-karaktere. Në këtë pikë sistemi ka dy zgjidhje.

1. Mund të bllokojë përdoruesin derisa të shkruhen të gjithë karakteret, por kjo do kërkonte shumë kohë në një linjë telefonike të ngadaltë.
2. Ose mund të clirojë përdoruesin menjeherë duke kryer një veprim I/O, ndërkohë që ai në këto momente mund të jetë dukë përpunuar dicka tjetër. Por kjo na con drejt një problemi më të madh: nga ta dijë proçesi përdorues që veprimi I/O ka përfunduar dhe që tanë mund të ripërdorë buffer-in? Sistemi mund të gjenerojë një sinjal ose interrupt software, por kjo mënyrë programimi është e vështirë. Një zgjidhje më e mirë pér kernelin është të kopjojë të dhënat në një buffer kerneli (analogë me fig.5.14, (c)) dhe të zhbllokojë thirrësin menjeherë. Tani nuk ka rëndësi se kur ka përfunduar veprimi I/O.

Buffering është një teknikë shumë e përdorur, por ka dhe anën e tij negative. Në se të dhënët buffero-hen shumë herë, performanca ulet. Le të marrim parasysh për shembull, network-un në fig 5.15. Këtu përdoruesi kryhen një thirrje sistem për të shkruar në network. Kerneli e kopjon paketën në buffer-in e kernelit dhe i jep të drejtë përdoruesit të vazhdojë menjeherë (hapi 1).

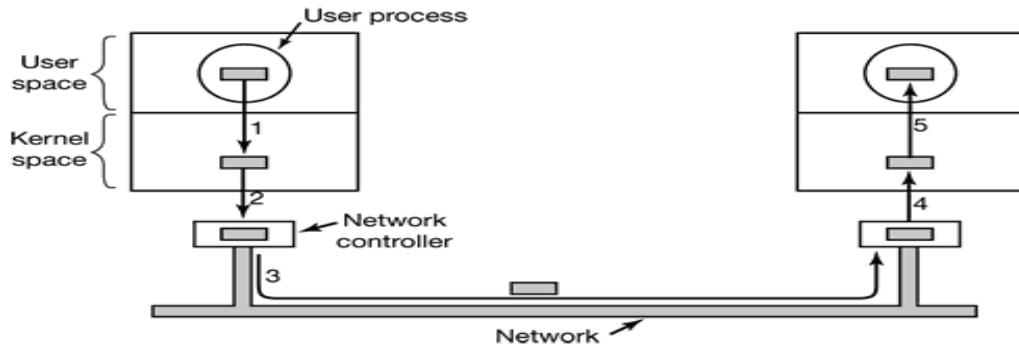


Fig.5.15

Kur thirret driveri, bufferi kopjon paketën tek kontrolleri për output (hapi 2). Arsyja pse ai nuk e nxjerr paketën për output nga memorja e kernelit direkt në linjën e transmetimit, është që kur një paketë fillon të transmetohet, ajo duhet të vazhdojë këtë me të njëjtën shpejtësi. Driver-i nuk na garanton që paketa shkon tek memorja me shpejtësi uniforme, sepse kanalet e DMA-së ose paisje të tjera I/O mund të vjedhin shumë cikle. Mos marrja në kohë e një fjale, mund të shkatërrojë komplet paketën. Duke e buffer-uar paketën brenda kontrollerit shmanget ky problem.

Pasi paketa kopjohet në buffer-in e brendshëm të kontrollerit, ajo kopjohet në network (hapi 3). Bitet mbërrijnë tek marrësi në fillim tek bufferi kontrollit të nework-ut. Më pas paketa kopjohet në buffer-in e kernelit të marrësit (hapi 4). Më në fund ajo kopjohet tek buffer-i i procesit përdorues të marrësit (hapi 5). Në përgjithësi marrësi kthen një paketë njohjeje tek dërguesi. Kur ky merr paketën e njohjes, ai ka të drejtë të dërgojë paketën tjeter. Është e qartë që ky kopjim do zvogëlojë shpejtësinë e transmetimit, pasi të gjithë hapat duhet të ekzekutohen në mënyrë sekuenciale.

Raportimi i gabimeve (error reporting)

Gabimet janë dicka shumë e zakonshme në kontekstin e I/O, si në asnjë kontekst tjeter. Kur ato ndodhin sistemi operativ duhet të merret me to në mënyrën më të mirë që të mundet. Shumë gabime janë specifikë sipas paisjeve, dhe duhet të merren në ngarkim nga driver-a të përshtatshëm, por framework-u që përgjigjet për gabimet është i pavarur nga paisja.

Një klasë gabimesh I/O janë gabimet në programim. Kjo ndodh kur procesi kërkon dicka të pamundur, si shkrimi në një paisje hyrëse (mouse, tastjerë, skaner, etj.) ose leximi në një paisje dalëse (printer, plottër, etj.). Gabime të tjera ndodhin kur gjenerojmë një adresë buffer-i të pavlefshme ose ndonjë parametër tjeter dhe specifikimi i një paisjeje të

pavlefshme (kërkojmë diskun 3 kur sistemi ka vetëm 2). Veprimi që kryhet kur kemi këto gabime është i thjeshtë: raportimi i një kodi gabimi drejt thirrësit.

Një klasë tjetër gabimesh I/O janë ato aktuale, për shembull përpjekja për të shkruar një bllok disku që është démtuar apo përpjekja për të lexuar nga një kamer që është e fikur. Në këto raste i takon driverit të vendosë cfarë të bëjë. Në se driveri nuk di cfarë të bëjë, ai mund t'ia kalojë si problem software-it të pavarur nga paisja.

Ajo cka bën ky soft varet nga ambienti dhe lloji i gabimit. Në se është një gabim i thjeshtë leximi, dhe në se është në dispozicion një përdorues interaktiv, mund të shfaqet një kuti dialoguese që pyet përdoruesin se cfarë do të bëjë. Në opsjon mund të përfshihet injorimi i gabimit dhe ripërpjekja disa herë, ose vrasja e proçesit të thirrjes. Në se nuk ka një përdorues në dispozicion ndoshta opzioni i vetëm është dështimi i thirrjes sistem me një kod gabimi.

Megjithatë disa lloje gabimesh nuk mund të trajtohen si më sipër. Për shembull, një strukturë të dhënash kritike si direktoria root mund të jetë shkatërruar. Në këtë rast sistemi mund të shfaqë një mesazh gabimi dhe të përfundojë.

Alokimi dhe clirimi i paisjeve të dedikuara.

Disa paisje si regjistrueshit e CD-ROM, mund të përdoren vetëm nga një proces në një moment të caktuar. I takon sistemet operativ të ekzaminojë kërkesat për përdorimin e paisjeve dhe ti pranojë ose jo ato, duke u varur dhe nga disponibiliteti i paisjes. Një mënyrë e thjeshtë për ti menaxhuar këto kërkesa është që ti kërkosh proçesit të hapë file të caktuara specifike vetëm për paisjet. Në se paisja nuk është e gatshme, hapja dështon.

Një përafrim alternativ do ishte që të kishim mekanizma specifike për kërkimin dhe clirimin e paisjeve të dedikuara. Përpjekja për të aksesuar një paisje që është e zënë e bllokon thirrësin, në vend që të dështojë. Proçeset e bllokuar vendosen në një rradhë. Herët a vonë paisja e bllokuar bëhet e disponueshme dhe proçesi që është në rradhë i pari, lejohet ta aksesojë atë dhe të vazhdojë ekzekutimin.

Përmasa e bllokut të pavarur nga paisja

Disqe të ndryshëm mund të kenë përmasa të ndryshme të sektorëve. I takon softit të pavarur nga paisja për të fshehur këtë fakt, duke performuar blloqe me përmasa të njëjtë ndaj shtresave më të larta, për shembull, duke trajtuar sektorë të ndryshëm si një bllok logjik unik. Në këtë mënyrë shtresat më të larta kanë të bëjnë me paisje abstraktë, të cilat përdorin blloqe logjikë me të njëjtat përmasa, të pavarur nga përmasat fizike të sektorëve. Për shembull, paisjet me karaktere i transferojnë të dhënat me një byte në njësinë e kohës (modem-ët), ndërkohë që të tjerat i transferojnë në njësi më të mëdha (ndërsaqet e network-ut). Edhe këto diferenca mund të fshihen.

5.3.4 Software I/O i hapësirës përdorues (user space)

Megjithëse pjesa më e madhe e software-it I/O ndodhet brenda SO, një pjesë e vogël e tij konsiston në librari të lidhura me prgramet user, apo të programeve të tjera jashtë hapësirës kernel. Thirrjet sistem, përfshirë dhe ato I/O, normalisht kryen nga procedura librari. Kur një program në C përmban:

```
Count = write (fd, buffer, nbytes);
```

Procedura e librarisë *write* do lidhet me programin dhe do i bashkangjitet këtij programi në memorje kur ky të jetë në ekzekutim. Koleksioni i gjithë këtyre procedurave librari është qartësisht pjesë e sistemit I/O.

Ajo që bëjnë këto procedura është pak a shumë vendosja e parametrave në vendin e duhur, ndërkohë që janë procedura të tjera që bëjnë punën e vërtetë. Vecanërisht formatimin e input-it dhe output-it e bëjnë procedurat librari. Një shembull i gjuhës C është *printf*, e cila merr si input një string dhe ndoshta disa variabla, ndërton një string në ASCII, dhe më pas thërret *write* për shfaqur stringën. Si shembull të *printf* le të shohim :

```
printf (" The square of %3d is %6d \n " i, i*i);
```

Kjo formaton stringën prej 14 karakteresh “ The square” duke bërë një linefeed.

Si shembull i një procedure të ngjashme, por për input, është *scanf*, i cili lexon input-in dhe e ruan atë në variabla me një sintaksë të njëjtë me atë të *printf*. Libraria standarde e I/O përmban një numër procedurash që përfshin I/O dhe gjithe pjesën e programeve user.

Jo i gjithë software I/O i nivelit user përmban procedura librari. Një kategori tjetër e rendësishme, është sistemi spooling. Spooling është një mënyrë e të menaxhuarit të paisjeve I/O të dedikuara (në një sistem me multiprogramim). Le të marrim parasysh një paisje që përdor këtë sistem: printeri. Megjithëse teknikisht do ishte e lehtë të lejohej një proces përdorues të hapte një file të vecantë karakteresh për printerin, supozojmë se një proces e ka hapur dhe nuk bën asnjë gjë për orë të tëra. Asnjë proces tjetër s’do mundt të printonte gjë.

Për këtë krijojmë një proces të vecantë, të quajtur **daemon** dhe një direktori të vecantë, të quajtur **spooling directory**. Vetëm daemon (që ka leje të vecantë për të përdorur printerin dhe asnjë proces tjetër), mund të printojë file-a në direktori. Duke e mbrojtur file-n e vecantë nga përdorimi direkt nga përdoruesi, problemi i mbajtjes së saj në mënyrë të panevojshme, eleminohet.

Spooling nuk përdoret vetëm për printerat. Ai përdoret dhe në situata të tjera. Për shembull, transferimi i file-ave në network përdor një daemon network-u. Për të dërguar një file diku, përdoruesi e vendos atë në një spooling directory network-u. Pak më vonë, daemon i network-ut e transmeton atë. Një pjesë e vecantë e përdorimit të transmetimit me anë të spooling, është USENET news system. Ky rrjet kosiston në miliona makina që komunikojnë në të gjithë globin, duke përdorur internet-in. Ekzistojnë me mijëra grupe

“news” rrreth temave të ndryshme. Për të postuar një mesazh news, përdoruesi përfshin një program news, i cili pranon mesazhin që do postohet dhe e depoziton në një spooling directory për ta transmetuar më pas. I gjithe sistemi news ekzekutohet jashtë sistemit operativ.

Fig. 5-16 përbledh sistemin I/O, duke na trguar të gjitha shtresat dhe funksionet kryesore të secilës. Duke filluar nga fundi, shtresat janë: hardware, përgjegjësit e interrupt-eve (interrupt handlers), driver-at e paisjeve, software i paisjeve të pavarura nga paisja, dhe në fund proceset përdorues.

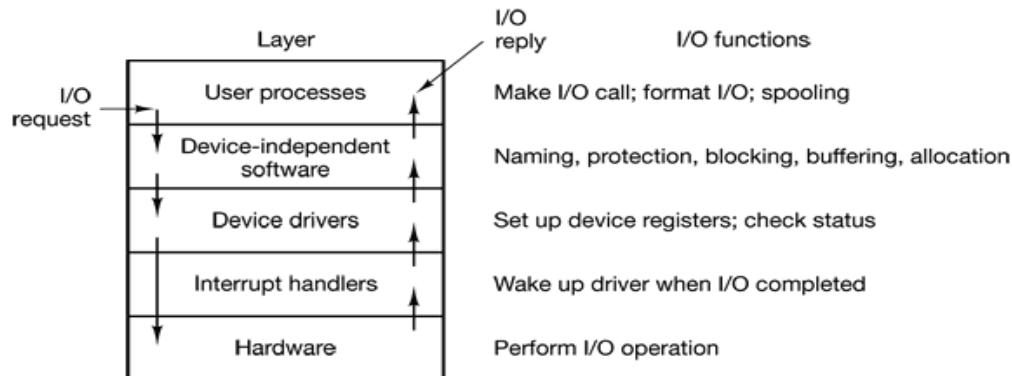


Figure 5-16. Layers of the I/O system and the main functions of each layer.

Shigjetat tregojnë drejtimin e kontrollit. Për shembull, kur një program user përpinqet të lexojë një bllok nga një file, përfshihet SO, në mënyrë që të përgjigjet për thirrjen e bërë. Software i pavarur nga paisja e kërkon në buffer cache. Në se blloku i kërkuar nuk ndodhet aty, thirret driveri i paisjes për ta transmetuar kërkesën tek hardware, që ky ta marrë bllokun në disk. Më pas procesi bllokohet derisa të përfundojë ky veprim.

Kur merret blloku, hardware gjeneron një interrupt. Përgjegjësi i interrupt-eve vihet në punë për të zbuluar cfarë ka ndodhur, që do të thotë se cila paisje ka rradhën në këtë moment. Më pas shikon statusin e paisjes dhe zgjon procesin në gjendje qetësie për të përfunduar kërkesën I/O, dhe të lejojë procesin user të vazhdojë.

5.4 DISQET

Tani do të studiojme disa pajisje hyrese/dalese (I/O devices) reale. Do të fillojme me disqet. Me pas do të shqyrtojme orat, tastierat dhe ekranet.

5.4.1 Hardware-i i Disqeve

Disqet jane të tipeve të ndryshem. Me të zakonshmit Jane disqet magnetike (hard disk dhe floppy disks). Keto mund të shkruhen dhe të lexohen me të njëjtën shpejtësi, gje qe i bën ideal si memorie sekondare (paging, file-a të sistemit, etj). Keto përdoren si ruajtës shume

të besueshem të informacionit. Për përhapjen e programeve, të dhenave dhe filmave përdoren lloje të ndryshem disqesh (CD-ROM, CD-Recordable dhe DVD). Ne seksionin qe vijon do të përshkruajme ne fillim hardware-in dhe me pas software-in e ketyre pajisjeve.

Disqet Magnetike

Disqet magnetike organizohen ne cilindra dhe secili prej tyre përmban gjurme. Gjurmët ndahen ne sektore, ku numri i sektoreve për një gjurme të mbyllur (rreth) varion nga 8 ne 32 për floppy disks dhe deri ne disa qindra për hard disqet. Numri i kokave varion nga 1 ne 16.

Disa disqe magnetike kane hardware të thjeshtë dhe japosin thjeshtë një seri të vazhdueshem bitesh ne dalje. Tek keto disqe vecanerisht disqet **IDE (Integrated Drive Electronics – driver elektroni i integruar)** vetë pajisja ka një mikrokontrollues (microcontroller) qe bën disa pune dhe i lejon kontrollerit të vertetë **të jape disa komanda të nivelit me të lartë**.

Një vecori e pajisjes, qe ka ndikim të rendesishem mbi driver-at e diskut është mundesa e kontrollerit për të bere dy ose me shume kerkime (seek) të sektoreve të disqeve. Keto njihen si **kerkime të mbivendosura (overlapped seek)**. Gjatë kohes qe një kontroller po pret për mbarimin e një kerkimi ne një vend, ai mund të filloje kerkimin ne një tjetër vend. Shume kontrollera mund të lexojne ose të shkruajne ne një disk gjatë kohes qe kerkojne ne një tjetër, por një kontroller i floppy disk nuk mund të lexoje ose të shkruaj gjatë kohes qe po kerkon ne të. (Të shkruarit apo të lexuarit kerkon levizjen e biteve ne një kohe të rendit të mikrosekondes, keshtu një kontrollues përdor pothuajse gjithe fuqine e tij llogaritëse për ketë transferim). Puna qendron ndryshtësia për hard disk me kontrollera të integruar dhe ne një sistem me shume se një hard disk. Ata mund të punojne njëkohesisht, të paktën ne transferimin nga disku ne bufferin e kontrollerit. Aftësia e kryerjes se dy operacionve njëkoheshisht mund të ule kohen e aksesit ne menyre të ndjeshme.

Figura 5-17 krahason parametrat e pajisjes ruajtëse për IBM PC, origjinale me parametrat e hard diskut modern, për të treguar sa kane ndryshuar disqet ne dy dekadat e fundit. Është interesante të dallosh se jo të gjithe parametrat janë përmiresuar njëlloj. Koha mesatare e kapjes se një sektori është përmiresuar shtatë here, shpejtësia e trasferimin është përmiresuar 1300 here, ndersa kapaciteti është rritur deri ne 50,000 here me shume. Kjo ka të beje me përmiresimin gradual të pjesave levizese, por rritjen e madhe të dendesise se biteve ne sipërfaqen e diskut.

PARAMETRAT	IBM 360-KB disku floppy	Hard disku WD 18300
Numri i cilindrave	40	10601
Gjurme për cilinder	2	12

Sektore për gjurme	9	281(mes)
Sektore për disk	720	35742000
Byte për sektore	512	512
Kapaciteti i diskut	360 KB	18.3GB
Koha kerkimit (cilindri ngjitur)	6 msek	0.8 msek
Koha kerkimit (rasti mesatar)	77 msek	6.9 msek
Koha rrotullimit	200 msek	8.33 msek
Koha ndalim/nisje të motorrit	250 msek	20msek
Koha për trasferimin e një sektori	22 msek	17 μ sek

FIGURA 5-17. Parametrat e floppy diskut origjinal të IBM PC dhe një hard disk origjinal të Werstern Digital WD

Duhet pasur kujdes kur shikohen specifikimet e hard disqeve modern sepse gjeometria e specifikuar, dhe e përdorur nga software-i i drive-it, mund të jetë e ndryshme nga forma fizike reale. Ne disqet e vjetër numri i sektoreve për gjurme ishte i njëjtë për të gjithe cilindrat. Disqet moderne jane ndare ne zona me shume sektore ne periferi të diskut se ne zonat e brendshme të tij. Fig 5-18 (a) ilustron një disk të vogel me dy zona. Gjurma me ne periferi ka 32 sektore për gjurme, ndersa ajo me ne brendesi ka 16 sektore për gjurme. Disqet reale, si për shembull WD 18300, shpesh kane 16 sektore, ku numri i sektoreve rritet me rrëth 4% për zone kur kalohet nga zona me ne brendesi ne atë me ne periferi.

Për të fshehur detajet e numrit të sektore qe ka një gjurme, shumica e disqeve modern kane një gjeometri virtuale qe i paraqitet sistemit operativ. Software-i është i bere për të vepruar sikur të ketë x cilindra, y koka, z sektore për çdo gjurme. Kontrolleri me pas riformulon kerkesen për (x,y,z) ne cilindrat, kokat dhe sektoret reale.

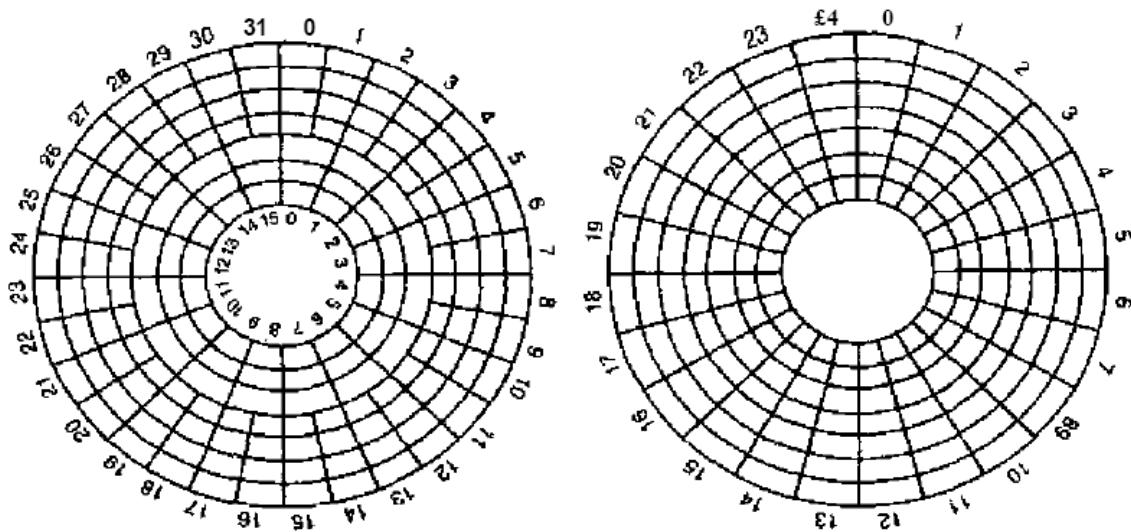


Figure 5-18. (a) Physical geometry of a disk with two zones, (b) A possible virtual geometry for this disk.

Figura 5-18. (a) Gjeometria reale e një disku me dy zona. (b) Një gjeometri virtuale e mundshme, për ketë disk

Një organizim gjeometrik i mundshem për diskun fizik të figures 5-18(a) tregohet ne fig 5-18 (b). Ne të dy rastet disku ka 192 sektore, vetëm se oraganizimi i tyre është ndryshe nga ai reali.

Për kompjuterat të bazuar ne Pentium, maksimumi i vleres për keto parametra është (65535, 16, dhe 63), për shkak të nevojes se përshtatjes me limitet e kompjuterave të meparshem **IBM PC**. Ne ketë makine, fusha prej 16-, 4- dhe 6- bit përdoreshin për të specifikuar keto numra, ku numerimi i cilindrave, sektoreve fillonte nga 1, ndersa numerimi i koka fillonte nga 0. Me keto parametra dhe me 512 byte për sektor, disku me i madh i mundshem ishte 31.5 GB. Për të anashkaluar ketë limit, shume disqe sot kane një sistem të quajtur **adresim llogjik të bloqeve (logical block addressing)**, ne të cilin sektoret e disqeve numerohen njëri pas tjetrit duke filluar nga 0, pa pasur parasysh gjeometrine e diskut.

RAID

Cilesite e CPU jane rritur ne menyre eksponenciale gjatë dekades se fundit, pothuajse duke u dyfishuar çdo 18 muaj. Kjo nuk ndodhi me disqet. Ne 1970 koha mesatare e kerkimit të një sektori ne disqet e minikompjuterave ishte 50 deri 100 msec. Tani koha kerkimit është pak me e vogel se 10 msec. Ne shumicen e industrive teknike (për shembull automobilistike, avionik, etj) një rritje prej 5 deri ne 10 here të cilesive të tyre ne dy dekada, do të ishte një lajm i madh, por ne industrine e kompjuterave kjo rritje është një "turpërim" ne krahasim me të tjerat. Me gjithe diferençen midis shpejtësise se CPU dhe shpejtësise se diskut, ajo është bere me e madhe me kalimin e kohes.

Sic po shohim, përpunimi ne paralel po përdoret gjithnjë e me shume për të rritur rendimentin e CPU. Gjatë viteve ka patur njerez qe menduan se pajisjet I/O ne paralel do

të ishin ide e mire. Ne studimin e tyre të 1988 Patterson et al, sugjeroi gjashtë organizime specifike të disqeve qe mund të përdoreshin për të përmiresuar cilesite e disqeve, sigurine ose të dyja (Patterson et al, 1988). Keto ide u zbatuan me shpejtësi ne industri dhe kane cuar ne një klase të re pajisjesh I/O të quajtura **RAID**. Patterson et al përcaktoi RAID si Redundant Array of Inexpensive Disk (rradhitje e përsëritur e disqeve jo të kushtushem), por idustria ndryshoi I-ne për të qene Indipendent-të pavarur ne vend të Inexpensive-jo të shtrenjtë (ndoshta ne ketë menyre mund të përdornin disqe të shtrenjta).

Konkuruesi i RAID ishte **SLED** (Single Large Expensive Disk – një disk i shtrenjtë, i vetëm, i madh), (njëloj sic është RISC me CISC, edhe kjo fale Patterson).

Idea baze mbrapa RAID është të instalosh një kuti plot me disqe prane kompjuterit, zakonisht një server, te zevendesosh kartën e kontrollit të diskut me një kontroller RAID, të kopjosh të dhenat tek RAID dhe të vazhdosh veprimet normale. Me fjalë të tjera RAID duhet ti duket sistemit operativ si një SLED, por të ketë rendiment dhe besueshmeri me të madhe. Meqe disqet SCSI kane rendiment të mire, cmim të ulet dhe aftësi për të patur deri ne 7 paisje, me një kontroller të vetëm (15 për SCSI-të e gjere) është e natyrshme qe shumica e RAID janë kontrollera RAID SCSI, plus një kuti me disqe SCSI qe i duken sistemit operativ si një disk i vetëm i madh. Ne ketë menyre, nuk ishte e nevojshme ndryshimi i software-it për të përdorur RAID.

Përvec paraqitjes si një disk i vetëm software-it, të gjithe RAID-et e shpërndajne informacionin nepër disa disqe, për të lejuar operacionet ne paralel. Patterson et al. beri disa skema të ndryshme për të bere ketë gje, dhe ato njihen si niveli (level) 0 i RAID deri ne nivelin 5 të RAID. Ka gjithashtu edhe disa nivele të tjera qe nuk do ti diskutojme. Termi “nivel” është thjesht një emertim jo i drejtë, sepse nuk përfshihen hierarki ne organizimin e tyre, por ka gjashtë organizime të ndryshme të mundshme.

Niveli 0 i RAID ilustrohet ne Fig. 5-19 (a). Disku i vetëm virtual i simluar nga RAID ndahet ne rripa (strip) me nga k sektore secili, ku sektoret nga 0 ne $k-1$ përbjegjne rripin e 0, sektoret nga k ne $2k-1$ i takojne rripit 1, e keshtu me rradhe. Për $k=1$ çdo rrip është një sektor, për $k=2$ çdo rrip ka 2 sektore, etj. Organizimi ne niveli 0 të RAID i shkruan rripa të njëpasnjëshem ne menyren round-robin, sic tregohet ne Fig. 5-19 (a) për një RAID me katër pajisje. Shpërndarja e te dhenave ne disa pajisje ne ketë menyre quhet **stripping**. Për shembull, ne se software jep një komande leximi të një bloku të dhenash ne katër rripa të njëpasnjëshem duke filluar nga fillimi, kontrolleri i RAID do të ndaje ketë komande ne katër komanda të vecanta, një për secilin nga katër disqet, dhe keshtu të veprojne paralelisht. Kemi keshtu I/O paralele pa njojhurine e software-it.

Niveli 0 i RAID punon me mire me kerkesa të medha, sa me e madhe aq me mire. Ne se një kerkesë është me e madhe se numri i rripave shumezuar me madhesine e një rripi, disa disqe do të marrin kerkesa të shumefishta, keshtu qe kur të mbarojne kerkesen e pare mund të fillojne të dytën. Është detyra e kontrollerit për të ndare kerkesat dhe për ti dhene diskut të duhur komanden e duhur, dhe me pas të bashkoje rezultatet ne memorie. Rendimenti është shume i mire dhe zbatimi i thjeshtë.

Niveli 0 i RAID punon me keq me sisteme operative qe zakonisht kerkojne të dhena ne një sektor të vetëm. Rezultati do të jetë korrekt por nuk do të kemi paralelizem dhe keshtu asnë rritle performance. Një tjetër disavantazh i ketij organizimi është fakti qe besueshmeria mund të jetë me e vogel ne krahasim me një SLED. Ne se një RAID përbehet nga 4 disqe, secili me një kohe mesatare pune pa deshtime prej 20,000 oresh, do

të thotë qe afersisht një here ne 5,000 ore njëri prej disqeve do të deshtoje dhe të gjithe të dhenat do të humbasin. Një SLED me kohe mesatare pune pa deshtime prej 20,000 oresh do të ishte katër here me i besueshem. Meqe ne ketë organizim nuk kemi përsritje ky nuk është një RAID i vertetë.

Variant tjetër, niveli 1 i RAID, i treguar ne Fig. 5-19 (b) është një RAID i vertetë. Ai duplikon të gjithe disqet, keshtu qe ka katër disqe primare dhe katër kopje. Për çdo shkrim, çdo rrip shkruhet dy here. Për një lexim, të dy kopjet mund të përdoren, duke shpërndare punen. Rrjedhimisht, rendimenti i shkrimit nuk është me i mire se ne rastin e një diskut të vetëm, por rendimenti i leximit mund të jetë deri ne dyfish me i mire. Toleranca e gabimeve është e shkelqyeshme, ne se një disk prishet, do të përdoret kopja e tij. Zgjidhja është instalimi i një diskut tjetër dhe kopjimi i diskut të rregullt ne të.

Ndryshe nga niveli 0 dhe niveli 1 qe punojne me rripa të sektoreve, niveli 2 i RAID punon me fjale, mundesisht edhe me byte. Imagjinoni të ndani çdo byte të diskut të vetëm virtual ne dy grupe me nga 4 bite, dhe me pas të shtosh një kod të Hamming-ut ne secilin grup për të formuar një fjale 7 bit të gjatë, nga të cilat bitet 1,2 dhe 4 Jane bite pariteti. Me tej, imagjinoni qe të shtatë pajisjet e Fig. 5-19(c) të ishin të sinkronizuar për nga pozicioni i krahut, e pozicioni i rrotullimit. Atëherë do të ishte e mundur të shkuanim një fjale 7 bit (e koduar sipas Hamming-ut) mbi shtatë pajisjet, një bit për pajisje.

Kompjuteri “The Thinking Machine CM-2” përdorte ketë skeme, duke patur 32 bit, duke shtuar 6 bitet e paritetit të Hamming-ut, formohej një fjale 38 bit, plus një bit tjetër shtese pariteti. Fjalet shpërndaheshin ne disqe. Throughput-i total ishte shume i madh, sepse ne kohen qe duhet për shkrimin e një sektori, mund të shkruheshin 32 të tille. Gjithashtu, prishja e një diskut nuk krijonte probleme, sepse kjo do të thoshte humbjen e njërit prej 39 biteve të fjalevë, po ketë gje e rregullon me lehtësi kodi i Hamming-ut.

SEC. 5.4

DISKS

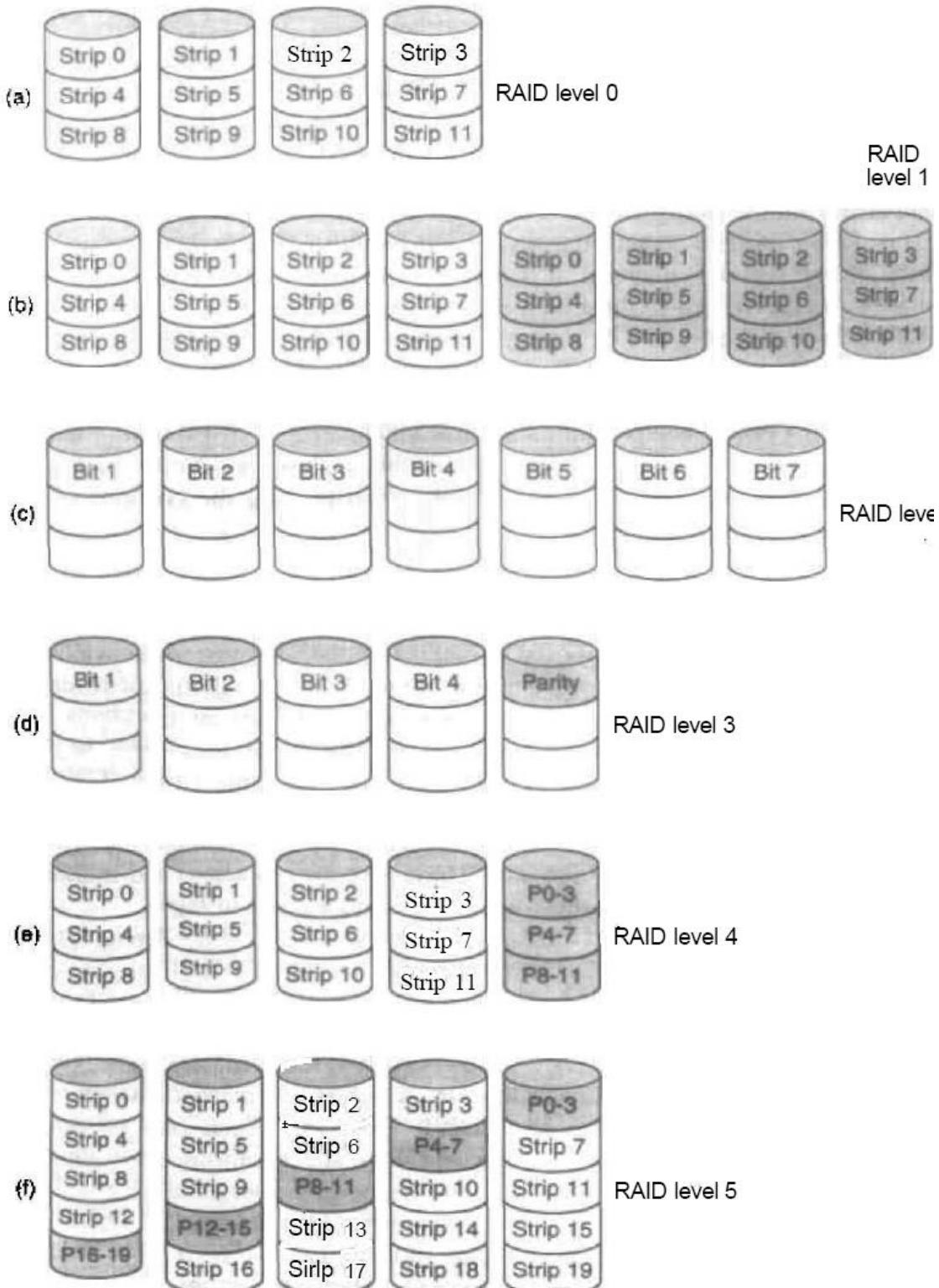
**Figure 5-19.** RAID levels 0 through 5, Backup and parity drives are shown shaded.

Figura 5-19 Nivelet e RAID nga 0 ne 5. Pajisjet qe mbajne kopjet dhe informacionin e paritetit jane shenuar me gri

Nga ana tjetër kjo skeme kerkon qe të githe disqet të jene të sinkronizuar, dhe ka kuptim për një numer të caktuar disqesh (edhe me 32 bit të dhena, 6 bite pariteti, gabimi është 19%). Gjithashu është një pune e madhe për kontrolluesin, sepse çdo here do të beje kontrollin e biteve të paritetit.

Niveli 3 i RAID është një version i thjeshtuar i nivelit 2. Ai ilustrohet ne Fig. 5-19(d). Ketu llogaritet vetëm një bit pariteti, i cili shkruhet ne diskun e paritetit. Ashtu si tek niveli 2, disqet duhet të jene saktësisht të sinkronizuara, meqë një fjale është e shpërndare nepër shume disqe.

Ne fillim mund të duket se një bit i vetëm pariteti jep vetëm detektim të gabimit dhe jo korigjim të tij. Për rastin e gabimeve të rastesishme të pa detektuar, kjo është e vertetë. Megjithatë ne rastin e deshtimit të një diskut të tere, ai na jep një korigjim të plotë të gabimeve 1 biteshe, meqë vendi i bitit të gabuar njihet. Kur një nga pajisjet prishet, kontrolleri pretendon se të gjithe bitet e tij janë 0. Ne se kemi një gabim pariteti, biti ne pajisjen e prishur duhet të ketë qene 1, keshtu qe korrigjohet. Megjithese të dy nivelet 2 dhe 3 të RAID-it, ofrojnë shpejtësi shume të medha, numri i kerkesave I/O qe mund të kryejne për sekonde nuk është me i mire se ai i një diskut të vetëm.

Niveli 4 dhe 5 i RAID,kane përseri rripa, s'kane me fjale me bite pariteti dhe nuk kane me nevoje për sinkronizim. Niveli 4 i RAID (Fig. 5-19(e)) është si niveli 0 i RAID me rripa pariteti për rripat, të shkruar ne një pajisje shtese. Për shembull ne se çdo rrip është i gjatë k byte, mbi të gjithe rripat aplikohet veprimi EXCLUSIVE-OR, duke rezultuar keshtu ne një rrip pariteti k bit të gjatë. Ne se një disk prishet, biti i humbur mund të rillogaritet nga diskut paritetit.

Ky lloj projektimi të mbron nga humbjet e një diskut por është i keq ne rast se ndryshohet një pjese e vogel e informacionit. Ne se një sektor ndryshon, është e nevojshme të lexosh të gjithe sektoret për të llogaritur paritetin, i cili duhet të rishkruhet. Ose mund të lexohet përbajtja e vjetër dhe pariteti i vjetër dhe të rillogaritet pariteti i ri prej tyre. Edhe me ketë optimizim, një ndryshim i vogel kerkon dy shkrime dhe dy lexime.

Si rrjedhoje e vellimit, punes se madhe ne diskun e paritetit, ai mund të kthehet ne problem. Për zgjidhjen e ketij problemi niveli 5 i RAID shpërndan bitet e paritetit ne të gjithe disqet, ne menyre round-robin, sic tregohet ne Fig. 5-19(f). Megjithatë ne rastin e deshtimit të njërit prej disqeve, llogaritja e biteve të humbura është një proces i veshtire.

CD-ROM-s

Ne vitet e fundit, disqet optike (si kundershtare të atyre magnetike) jane bere të pranishem ne treg. Ata kane dendesi shkrimi me të madhe se disqet e zakonshme magnetike. Disqet optike u zhvilluan si fillim, për të regjistruar programet televizive, por ata mund të përdoren si pajisje ruajtëse të kompjuterave. Për shkak të kapacitetit potencial memorizues, disqet optike jane bere subjekti i shume kerkimeve dhe kane kaluar një evolucion të shpejtë.

Gjenerata e pare e disqeve optike u shpik nga një grup inxhinieresh elektronik hollandez të Philips-it për të mbajtur filma. Ata kishin diametër 30cm dhe emer LaserVision, por nuk patën sukses, përvencse ne Japoni.

Ne 1980, Philips bashke me Sony, zhvilluan CD (Compact Disc), të cilët shpejt zevendesuan regjistrimet prej vinili 33 1/3-rpm. Detajet teknike të sakta për CD u publikuan ne Standartin Nderkombetar (IS 10149), i njojur jo-zyrtarisht si Red Book (libri i kuq), për shkak të ngjyres se kapakut të tij, (International Standard-standartet nderkombetare nxirren nga International Organization for Standardization, i cili është forma nderkombetare e grupeve kombetare të standardeve. Çdo standart ka një numer IS). Qellimi i publikimit të specifikimeve te diskut dhe pajisjes se tij ne një Standart Nderkombetar, është ti beje CD-të nga shtëpi të ndryshme diskografike të punojne ne pajisje të prodhuesve të ndryshem. Të gjithe CD-të kane diametër 120mm, dhe 1.2mm të trashe, me një vrime 15mm ne mes. CD audio ishte e para pajisje kujtese dixhitale e suksesshme ne tregun e gjere. Ato supozohen të durojne deri ne 100 vjet. Ju lutem kontrolloni ne 2080 si shkoi parashikimi.

Përgatitet një CD duke përdorur një laser infra-red për të djegur vrima me diametër 0.8-mikron ne një master disk mbuluar me qelq. Nga ky master, përgatitet një kallep, me gunga aty ku ishin vrimat e laserit. Ne ketë kallep, një përzierje polikarbonatesh të shkrire injektohet për të formuar një CD me të njëjtin formacion vrimash si ne masterin prej qelqi. Pastaj një shtrese pasqyruese shume e holle alumini, vendoset mbi polikarbonat, dhe mbulohet nga një shtrese mbrojtëse llaku, e ne fund etiketohet. Thellimet ne shtresen e polikarbonatit quhen gropeza(pits), dhe pjesa e padjegur midis gropezave quhet fushe (lands).

Gjatë punes, një diode lazer me fuqi të ulet ndricon gropezat dhe fushat me dritë infra-red, me gjatësi vale 0,78 mikron. Meqe lazeri ndodhet ne krahun e polikarbonatit, gropezat do ti duken gunga ne një sipërfaqe qe përndryshe do të ishte e sheshtë. Meqe gropezat kane një lartësi prej cerek gjatësie vale, drita e reflektuar nga gungat (gropezat nga ana tjetër e diskut) do të kene një shfazim prej gjysem vale, ne lidhje me dritën e reflektuar nga sipërfaqja përreth. Si rezultat i ketij shfazimi, dy rrezet e dritës interferojne ne menyre desktruktive dhe i kthejne fotodetektorit të pajisjes me pak dritë se ne rastin kur pasqyrimi ndodh ne një fushe. Kjo është menyra se si një pajisje tregon dallimin e një gropeze nga një fushe. Megjithese duket e natyreshme përdorimi i një gropeze për të regjistruar 0 dhe të një fushe për 1, është me e sigurt përdorimi i tranzicionit gropez/fushe apo fushe/gropez për simbolizuar 1 dhe mungesen e tranzicionit për 0. Kjo skeme përdoret edhe ne praktike.

Gropezat dhe fushat shkruhen ne një spirale të vetme të vazhdueshme qe fillon tek vrima dhe shkon deri ne një largesi 32mm nga bordura e diskut. Spiralja bën 22,188 rrotullime rrëth qendres se diskut (rrëth 600 rrotullime për 1 mm). Ne se do të c'mbeshtillej, spiralja do të ishte 5.6 km e gjatë. Spiralja ilustrohet ne Fig. 5-20.

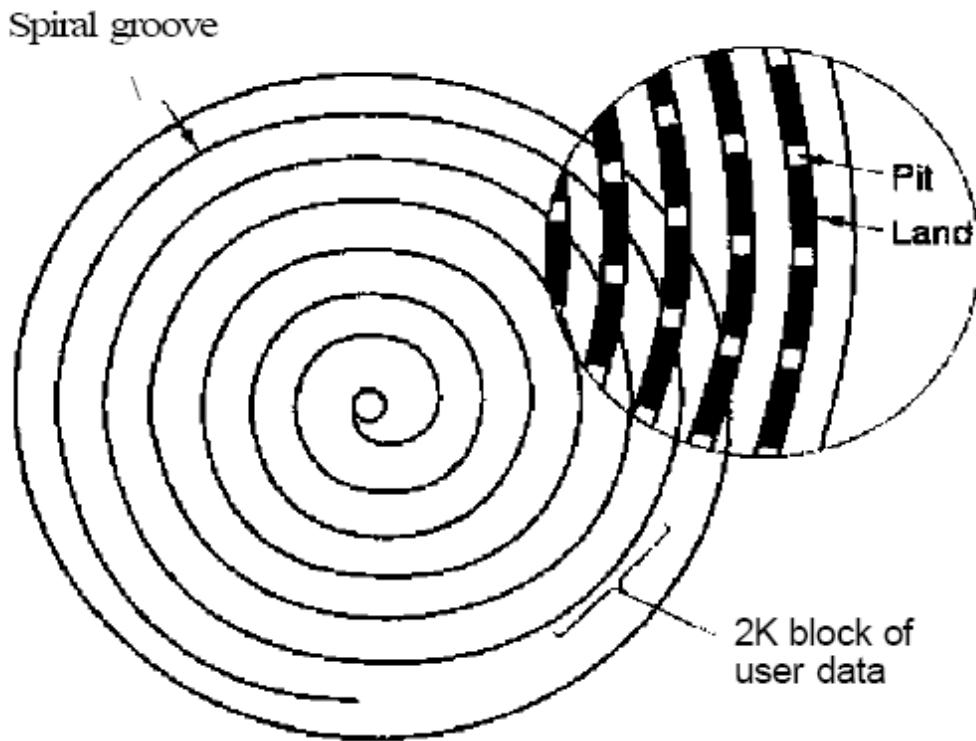


Figura 5-20 Struktura e regjistrimit të një CD (Compact Disc)

Për luajtur muzike me ritem kostant, është e nevojshme qe gropezat dhe fushat të kalojne përpara fotodiodes, me shpejtësi lineare konstante. Pra duhet qe shpejtësia e rrotullimit të CD të ulet vazhdimesht, kur koka lexuese e CD leviz nga pjeset me afer qendres drejt pjeseve me larg saj. Ne brendesi, shpejtësia rrotullimit është 530 rrot/min për të arritur shpejtësinë e deshiruar të kalimit të gropezave dhe fushave prej 120 cm/sek; tek gjurmët me të jashtëm rrotullimet ulen ne 200 rrot/min për ti dhene kokes lexuese të njëjtën shpejtësi lineare. Një pajisje me shpejtësi lineare konstante është ndryshe nga një disk magnetik, i cili punon me një shpejtësi kendore kostante, pavaresisht vendit se ku ndodhet koka lexuese. Gjithashtu shpejtësia prej 530 rrot/min është shume larg shpejtësise 3600 deri ne 7200 rrot/min qe arrijne shumica e disqeve magnetike.

Ne 1984, Philips dhe Sony kuptuan potencialin e përdorimit të CD për të ruajtur të dhenat e kompjuterave, keshtu botuan Librin e Verdhe (Yellow Book) duke përcaktuar standartin e atij qe sot quajme CD-ROM (Compact Disc – Read Only Memory). Për të përfituar nga tregu i madh i audio CD ne atë kohe, CD-ROM do të kishin të njëjtën madhesi si audio CD, do të ishin të pajtueshem me to për nga ana mekanike e optike, dhe do të prodhoheshin nga e njëjta makine qe injektonte polikarbonat për CD audio. Ky vendimi solli nevojen e motorreve të ngadaltë me shpejtësi të ndryshueshme. Gjithashtu kosto e prodhimit të një CD-ROM do të ishte poshtë 1 dollar.

Ajo cfare përcaktoi Libri i Verdhe ishte forma ruajtjes se të dhenave të kompjuterit. Gjithashtu u përmiresua mundesia e korigimit të gabimeve, një hap shume i rendesishem. Megjithese degjuesve të muzikes nuk u prishtë pune humbja e ndonjë bit-i here pas here, përdoruesit e kompjuterit e vene re, shume ketë gje. Forma kryesore e

ruajtjes se të dhenave ne CD-ROM është duke koduar çdo byte ne një simbol 14-bit. Sic pame me sipër, 14-bit Jane të mjaftueshem për të koduar sipas Hamming një byte 8-bitesh dhe 2 bite taprojne. Ne fakt përdoret një sitem kodimi me i fuqishem. C'kodimi 14-ne-8 behet nepërmjet hardware-it.

Një nivel me sipër, një grup 42 simbolesh të njëpasnjëshem formojne një imazh (frame) prej 588 bit. Secili frame mban 192 bit me të dhena (24 byte). Pjesa tjeter qe mbetur, 396 bite, përdoren për kontroll dhe korigjim gabimesh. Deri ketu kjo skeme është e njëjtë si për CD audio ashtu edhe për CD-ROM.

Libri i Verdhe shton edhe grupimin e 98 frame ne një sektor të CD-ROM, e treguar ne Fig.5-21. Çdo sektor i CD-ROM fillon me një hyrje (preamble) 16 byte, ku 12 të paret Jane 00FFFFFFFFFFFFFFFFFFF00 (ne forme hekza decimal), përfundon me një numrin e sektorit, sepse kerkimi i të dhenave ne spiralen e vetme të CD-ROM është me e veshtere se kerkimi ne gjurmët bashkeqendrore të disqeve magnetike. Për të kerkuar një sektor, software-i i pajisjes llogarit me përafersi vendin ku të kerkoje. Pasi vendoset aty fillon të kerkoje përskuencen e fillimit të një sektori. Byti i fundit i hyrje eshte byti mode.

Libri i Verdhe përcakton dy menyrat (mode). Menyra 1 përdor organizimin e Fig.5-21, me një sekuese hyrje prej 16 bit, 2048 byte të dhenash, dhe një kod korigjimi 288 byte (një nga kodet Reed-Solomon). Menyra 2 kombinon byte-et e ECC (Error Correction Code) me të dhenat, përfundon me një numri të datave apo korigjimin e gabimeve, si për shembull audio apo video. Vini re se përfundon se sigurat besueshmeri të lartë, përdoren tre skema përkorrigejimin e gabimeve; brenda një simboli, brenda një frame-i, brenda një sektori të CD-ROM.

SEC. 5.4

DISKS

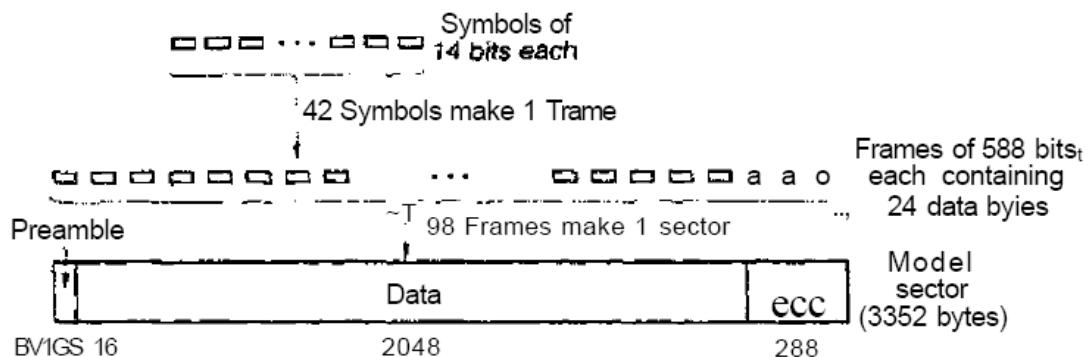


Figura 5-21 Organizimi llogjik i të dhenave ne një CD

Gabimet me një bit të vetëm korrigohen ne nivelin me të ulet, segmentë të shkurtër gabimesh korrigohen ne nivelin e frame-ve, dhe çdo gabim i mbetur pa kapur detektohet ne nivel sektoresh. Për të patur ketë besueshmeri sakrifikojmë rendimentin. Na duhen 98 frame prej 588 bit secili (7203 byte) përfundon me 2048 byte informacion, me një rendiment prej vetëm 28%.

Pajisjet CD-ROM me një shpejtësi punojne me 75 sektore/sek, qe do të thotë një shpejtësi të dhenash 153,600 byte/sek me menyren 1 dhe 175,200 byte/sek me menyren 2. Pajisjet me dy shpejtësi jane dyfish me të shpejta, dhe keshtu me rradhe deri ne shpejtësinë me të madhe. Keshtu një pajisje 40x mund lexoje të dhena me një shpejtësi 40×153600 byte/sek, duke supozuar se bus-i i nderfaqes se pajisjes dhe sistemi operativ mund të përballojne ketë shpejtësi. Një CD Audio standarte ka hapesire për 74 min muzike, i cili ne se përdoret me menyren 1, jep një kapacitet 681,983,000 byte. Kjo zakonisht jepet si ²⁰ 650 MB, sepse 1 MB është 2²⁰ byte (1,048,576 byte) dhe jo 1,000,000 byte.

Edhe një pajisje CD-ROM 32x (4,915,200 byte/sek) nuk krasohet me një disk magnetik SCSI-2 me shpejtësi 10 MB/sek, megjithese shume pajisje CD-ROM përdorin nderfaqe SCSI (ekzistojne edhe pajisje CD-ROM me nderfaqe IDE). Kur kujton se koha e kerkimit është disa qindra milisekonda, behet e qartë se pajisjet CD-ROM nuk janë ne të njëjtën kategorji rendimenti me disqet magnetike, pavaresisht kapacitetit të tyre të madh.

Ne 1986, kompania Philips pati përseri sukses me Librin Jeshil (Green Book), duke i shtuar grafike CD-ROM-it dhe aftësine për të mbajtur audio, video e të dhena ne të njëtin sektor, një veti e nevojshme për CD-ROM multimediale.

Ceshtja e fundit e CD-ROM është system file (sistemi i file-ve). Për të bere të mundur përdorimin e të njëjtët CD-ROM ne kompjutera të ndryshem, nevojitej një marreveshje mbi sistemin e file-ve të CD-ROM. Për të arritur ne ketë marreveshje, përfaqesues të shume kompanive u mblodhen ne ligenin Tohoe të High Sierras-as ne kufirin Kaliforni-Nevada dhe krijuan një system file qe u quajt High Sierra. Me pas kjo marreveshje evoloi ne Standartin Nderkombtar (IS 9660). Ai ka tre nivele. Niveli 1, përdor emra file-esh deri ne 8 karaktere, një prapashtese të mundshme prej 3 karakteresh (marreveshja e emrave të file-ve të MS-DOS). Emrat e file-ve mund të përmbajne vetëm shkronja të medha, numra, dhe simbolin “_” (quajtur: underscore). Direktoritë mund te përmbajne njëra-tjetren por deri ne një hierarki prej 8 shkallesh, dhe emrat e tyre nuk mund të kene prapashtesa. Niveli 1 i detyron të gjithe file të jene të vazhdueshem, gje qe nuk është problem për një disk qe do shkruhet vetëm një here. Çdo disk i shkruar sipas standartit IS 9660, Niveli 1, mund të lexohet ne MS-DOS, komjuterat Apple, ne kompjuterat UNIX, apo ne çdo kompjuter tjetër. Prodhuesit e CD-ROM e shikojne ketë veti si një plus i madh, nivelit 1.

IS 9660, niveli 2, lejon emra deri ne 32 karaktere dhe niveli 3 lejon file jo të vazhdueshem. Prapashtesa “Rock Ridge” (e quajtur keshtu cuditërish nga një film i Gene Wilder, Blazing Saddles) lejon emra shume të gjatë (për UNIX), UID, GID dhe link-e simbolike, por CD-ROM-et papërputhshem me nivelin 1 nuk do të lexohen nga të gjithe kompjuterat.

CD-ROM-et jane bere shume popullore për të nxjerre ne shitje lojra, filma, enciklopedi, atllase dhe materiale të çdo lloji. Shumica e software-ve tanë gjenden ne CD-ROM. Kombinimi i kapacitetit të tyre të madhe dhe kostos se ulet të prodhimit, i bën ata të përshtatshem për përdorime të shumta.

CD-Rercordable (CD-të shkrueshem)

Fillimisht, pajisja e nevojshme për prodhimin e një CD-ROM master (ose CD audio) ishte shume e shtrenjtë. Por si zakonisht ne industrine e kompjuterave, asgje nuk vazhdon se qeni e shtrenjtë për shume kohe. Ne mesin e viteve 1990, shkruesit e CD, jo me të medhenj se një pajisje e zakonshme periferike, ishin ne shitje ne shumicen e dyqaneve të kompjuterave. Keto pajisje akoma ishin të ndryshme nga disqet magnetike, sepse ne se CD-ROM-et shkruheshin, nuk mund të fshiheshin me. Sidoqoftë, ata gjetën përdorim si kujtesa për ruajtjen e kopjeve (back up) të hard disqeve të medhenj. Gjithashtu CD-ROM-i i lejoi individet apo edhe kompanitë e vogla të prodhonin CD-ROM e tyre, apo të bënin mastera qe do tu coheshin fabrikave të medha për shumefishimin. Keto pajisje njihen si CD-R (CD-Recordables).

Fizikisht, CD-R Jane disqe bosh, me diametër 120mm me polikarbonat, njësoj si të CD-ROM-et, përvecse CD-R kane një kanal 0.6mm për të drejtuar laserin gjatë shkrimit. Kanali ka një ndryshim sinusoidal prej 0.3mm me një frekuencë saktësisht 22.05 kHz, për të kontrolluar vazhdimesh shpejtësine e CD, dhe ne se është e nevojshme të ndryshohet ajo. CD-R duken njësoj si CD-ROM-et e zakonshem, përvecse ata kane ngjyre ari, ne vend të ngjyres se argjendtë. Ngjyra e artë vjen nga përdorimi i arit ne vend të aluminit si shtrese reflektuese. Ndryshe nga CD, qe kane gropeza (pit) ne to, aftësia reflektuese e ndryshme e gropezave (pits) dhe fushave (lands) duhet të simulohet ndryshe. Kjo behet duke shtuar një shtrese boje trasparente midis polikarbonatit dhe shtreses reflektuese prej ari, sic tregohet ne Fig. 5-22. Dy lloje bojerash përdoren, cyanina, e cila është jeshile, dhe phthalocyanina, e cila është portokalli ne të verdhe. Kimistët mund të diskutojne pafund se cila është me e mire. Keto bojera të ngjashme me ato të përdorura ne fotografitë, gje qe shpjegon faktin qe Eastman Kodak dhe Fuji Jane prodhuesit me të medhenj të CD-R bosh.

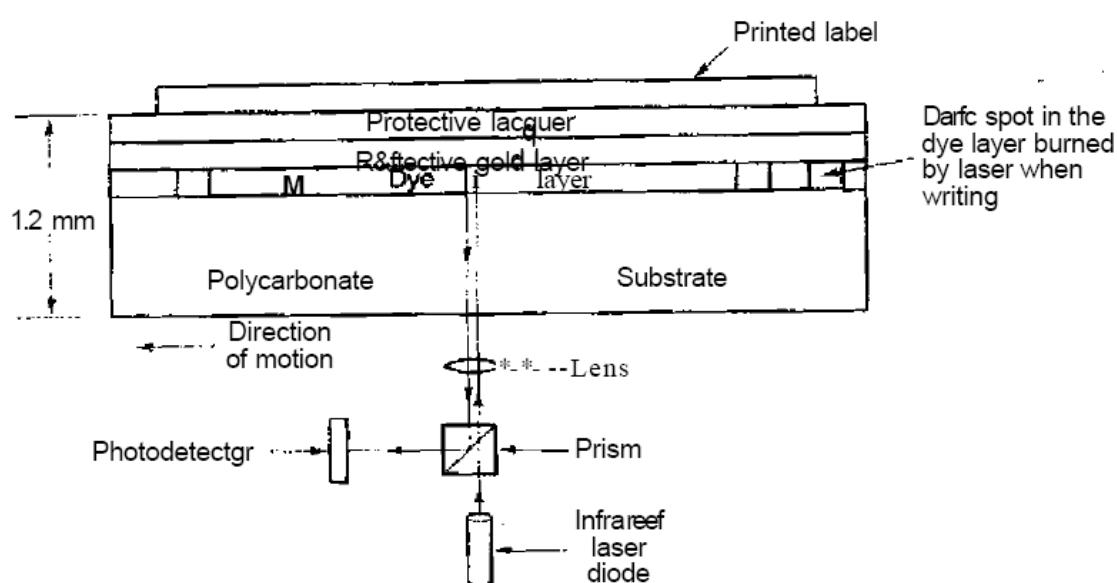


Fig. 5-22 Prerja tërthore e një disku CD-R dhe e lazerit (jo me të njëjtën shkalle). CD-ROM-et e argjentë kane struktura të njëjtë, përvèc shtreses se bojes dhe shtreses se arit, të përfaqesuar nga shtresa e argjendit me gropeza

Ne gjendjen e saj fillestare shtresa e bojes është trasparente dhe lejon dritën e laserit ti kaloje përmes, e të reflektohet mbi shtresen e arit. Për të shkruar, laseri e CD-R rregullohet të punoje me fuqi të madhe (8-16 mW). Kur rrezja godet një zone të bojes, ajo ngrohet duke prishur lidhjen kimike. Ky ndryshim ne strukturen molekulare krijon një njolle të erret. Kur lexohet (me 0.8 mW) fotodetektori shikon një ndryshim midis zonave të erreta, ku boja është goditur, dhe zonave trasparente ku boja është e paprekur. Ky ndryshim interpretohet si ndryshimi midis gropezave dhe fushave (pits dhe lands), edhe kur CD-R lexohet ne lezuesat e CD-ROM-it të zakonshem, ose edhe tek audio CD player. Asnjë lloj i ri CD nuk u shpik i pashoqeruar nga libri i tij me ngjyre, keshtu qe Libri Portokalli (Orange Book) u publikua ne 1989 për CD-R. Ky dokument përshkruan CD-R dhe gjithashtu edhe një forme të re të tij, **CD-ROM XA**, i cili i lejon CD-R të shkruhen disa here: disa sektore sot, disa të tjere nesër dhe disa muajin tjetër. Një grup sektoresh të njëpasnjëshem të shkruar njehersh quhet **gjurme e CD-ROM (CD-ROM track)**.

Një nga përdorimet e pare të CD-R ishte Kodak PhotoCD. Ne ketë sistem klienti sjell një film me fotografi dhe PhotoCD e tij të vjetër dhe merre të njëtin PhotoCD me fotografitë e reja, të shtuara tek ato të vjetrat. Informacioni i ri për tu hedhur merret nga skanimi i fotove ne film. Fotot hidhen ne PhotoCD, ne një gjurme tjetër CD-ROM-i (CD-ROM track). Shkrimi disa here i CD-R ishte i nevojshem sepse kur ky produkt u hodh ne treg, CD-R bosh ishin shume të shtrenjtë për të blere një të ri sa here mbaronte një film me foto.

Megjithatë, shkrimi disa here i CD-R krijon një problem të ri. Përpara se të dilte Libri Portokalli, të gjithe CD-ROM-et kishin vetëm një **VTOC (Volume Table Of Content – libri i tableles se përbajtjes)** ne fillim. Ajo skeme nuk punonte kur shkruhej disa here (multitrack). Zgjidhja per te shkruar ne Librin Portokalli ishte qe ti jepej çdo gjurme të CD-ROM-it, VTOC e vet. File-t e shkruara ne VTOC mund të përfshijnë disa ose të gjitha file-t nga gjurmet e meparshme. Mbasi një CD-R futet tek pajisja lexuese, sistemi operativ kerkon të gjithe gjurmet (tracks) e CD-ROM-it për të gjetur VTOC me të re, e cila i jep gjendjen aktuale të diskut. Duke përfshire vetëm disa, por jo të gjitha file-t ne VTOC me të re, është e mundur të krijohet iluzioni i fshirjes se file-ve. Gjurmet (tracks) mund të grupohen ne sesione, duke cuar ne CD-ROM-e **multisession (multisession)** audio CD player standart nuk mund ti lexojne CD me multisessione, sepse ata presin vetëm një VTOC ne fillim të diskut.

Çdo gjurme duhet të shkruhet me një veprim të vetëm pa ndaluar. Si rrjedhim, hard disku nga i cili do të vijne të dhenat duhet të jetë i shpejtë, për t’ia dorezuar ato pajisjes shkruajtëse ne kohen e duhur. Ne se file-i qe do të kopjohet është i shpërndare nepër hard disk, koha e kerkimit të tij mund të beje qe të dhenat (data stream) qe po shkruhen ne disk të shterojne. Ne se kjo ndodh, ju do përfundoni me një gje të re shkelqyese dhe të bukur (por deri diku të shtrenjtë) për të vene gotat, apo një frisbee 120mm ngjyre ari. Zakonisht

software-et e CD-R ofrojne mundesi te mbledhjes se të gjithe file-ave ne një imazh CD-ROM (image) të vazhdueshem 650 MB, por ky proces zakonisht dyfishon kohen e shkrimit, kerkon 650 MB të lira ne hard disk, dhe përseni nuk të mbron nga hard disku, vecanerisht ne caste “paniku” kur hard disku vendos të beje rikalibrimin termik, në qoftë se nxehet shume.

CD-R bejne të mundur qe individe apo edhe kompanitë të kopjojne me lehtësi CD-ROM-et (dhe audio CD), zakonisht duke shkelur ligjin mbi të drejtat e autori. Disa skema janë shpikur për të bere piraterine me të veshtire dhe për të bere me të veshtire leximin e CD-ROM-eve nga software të tjere, përvecse atij të prodhuesit. Një prej tyre është shkrimi i gjatësise se file-ve ne CD-ROM, si të ishin shume gigabyte, duke shmangur keshtu çdo përpjekje për kopjimin e CD ne hard disk. Gjatësia e vertet ndodhe ne software-in e prodhuesit, apo e fshehur (mundeshi e enkriptuar) ne një zone të pa parashikueshme të CD-ROM-it. Një skeme tjetër është përdorim me vetëdije i ECC gabim, ne sektore të caktuar, me shpresen qe software-i për kopjimin e CD do te “korigoje” gabimet. Ky software-i kontrollon vetë ECC, duke refuzuar të punoje ne se ata janë të rregullt. Gjithashtu është e mundur përdorimi i hapesirave jo standarte midis gjurmave (track) apo i “difikteve” të tjera fizike.

CD-Rewritables (CD-të rishkruajtshem)

Megjithese njerezit janë mesuar me gjera qe shkruehet vetëm një here, sic janë letra, filmat fotografik, ka një kerkese të madhe për CD-ROM të rishkruajtshem. Një teknologji e disponueshme është **CD-RW (CD-ReWritable)**, e cila përdor disqe me të njëjtën madhesi si CD-R. Megjithatë, ne vend të bojes cyanine apo phthalocyanine, CD-RW përdorin një aliazh argjenti, indiumi, antimoni dhe tellurumi si shtrese regjistruese. Ky aliazh ka dy gjendje të qendrueshme: kristaline dhe jokristaline (amorphous-amorfe), me veti të ndryshme pasqyruese.

Pajisjet për CD-RW kane lazer me tre fuqi të ndryshme. Me fuqine me të lartë, lazeri shkrin aliazhin, duke e kthyer nga gjendja kristaline shume pasqyruese, ne një gjendje amorfë jo pasqyruese, për të paraqitur një gropë(pit). Me fuqine e mesme, aliazhi shkrihet dhe kthehet ne gjendjen e tij natyrale, kristaline, për tu bere përseri fushe(land). Me fuqine me të ulet, “ndjehet” gjendja e materialit (për lexim), por nuk ka ndryshim shenje. Arsyjeja se përsë CD-RW nuk ka zevendesuar CD-R, është sepse CD-RW bosh janë me të shtrenjtë se CD-R bosh. Gjithashtu, për përdorime të tillë si kopjimi (buck up – rezerve) e hard diskut, fakti qe CD-R po të shkruehet nuk fshihen me, është një plus i madh.

DVD

Format kryesore të disqeve optike, CD/ CD-ROM, kane ekzistuar qe prej 1980. Teknologja është përmiresuar qe prej atëhere, keshtu qe disqet optike me kapacitet me të lartë janë ekonomikisht të përballueshem dhe ka një kerkese të madhe për to. Hollywood-i, do të enderronte të zevendesonte kasetat video analoge me disqet dixhitale, meqë disqet kane një cilesi me të lartë, janë me të lira për tu prodhuar, zgjasin me shume, kerkojne me pak hapesire nepër dollapet e videotekave dhe nuk kane nevoje të mbeshtilen. Kompanit

e konsumit elektronik po kerkojne për një produkt të ri, meqë kompanit e kompjuterave duan ti shtojne multimedia software-ve të tyre.

Ky kombinim i teknologjise dhe kerkeses se tregut nga tre industri shume të pasura e të fuqishme, coi ne krijimin e **DVD**, ne fillim e njojur si **Digital Video Disk (disk i videos dixhitale)**, por tani njihet zyrtarisht si **Digital Versatile Disk (disk dixhital i gjithanshem)**. DVD përdorin të njëtin parim si CD, një disk 120mm polikarbonati me gropeza dhe fusha, qe ndricohet nga një diode lazer dhe lexohet nga një fotodetektor. Të rejet tek DVD jane përdorimi i:

1. gropezave me të vogla (0.4 mikron ndryshe nga 0.8 mikron tek CD)
2. një spiraleje me të ngushtë (0.74 mikron ndermjet gjurmeve ne dallim nga 1.6 mikron të CD)
3. një laser të kuq (0.65 mikron ne krahasim me 0.78 mikron të CD)

Bashke, keto përmiresime rrisin kapacitetin shtatë here, ne 4.7 GB. Një pajisje DVD 1x punon me 1.4 MB/sek (ne dallim nga 150 KB/sek tek CD). Fatkeqesish ndryshim ne laser të kuq, si ai i përdorur tek supermarketët, do të coje ne nevojen e një lazeri të dytë, apo të ndonjë metode të sofistikuar optike, për të lexuar CD dhe CD-ROM ekzistues. Ketë nuk e bejne të mundur të gjitha pajisjet. Gjithashtu mund të mos jetë e mundur leximi i CD-R dhe CD-RW ne pajisjet e DVD.

A Jane të mjaftueshme 4.7 GB? Ndoshta. Duke përdorur ngjeshjen (compression) MPEG-2 (me standart IS 13346), një disk DVD 4.7GB, mund të mbaje 133 minuta me video me ekran të plotë (full-screen), me levizje (full-motion), ne cilesi (resolution 720x480) të lartë, gjithashtu edhe me fonogram (soundtrack) deri ne 8 gjuhe dhe 32 titra (subtitles). Rreth 92 përqind e të gjithe filmave qe ka bere Hollywood Jane me pak 133 minuta. Megjithatë, disa aplikacione si lojrar multimediale, apo informacioni i një pune të caktuar, mund të kerkajo me shume. Po ashtu edhe Hollywood-i do të kerkonte të fuste disa filma ne një disk, keshtu qe u përcaktuan katër forma:

1. Një ane të vetme, me një shtrese të vetme (4.7 GB)
2. Një ane të vetme, me dy shtresa (8.5 GB)
3. Dy ane, me nga një shtrese (9.4 GB)
4. Dy ane, me nga dy shtresa (17 GB)

Po përse kaq shume formate? Me pak fjale: politika. Philips dhe Sony donin disqe me një ane dhe me dy shtresa si version me kapacitet të lartë të DVD, por Toshiba dhe Time Warner donin disqet me dy ane, me një shtrese. Philips dhe Sony mendonin se njerezve nuk do tu pelqente të kthenin diskun nga ana tjetër. Ndersa Timer Wanver nuk besonte se mund të vendoseshin dy shtresa ne një disk. Kompromisi ishte : të gjithe kombinimet e mundshme, por tregu do të vendoste se cili do të mbijetonte.

Teknologja me dy shtresa vendos shtresen reflektuese mbi një shtrese gjysem-reflektuese (semireflektive). Ne varesi të fokusimit të lazerit, ai pasyrohet nga njëra apo tjera shtrese. Shtresa me e ulet ka nevoje për gropeza dhe fusha (pits and lands) pak me të medha, qe të lexohet me lehtë. Kapaciteti i saj është pak me i vogel se ai i shtreses se sipërme.

Disqet me dy ane behen duke mare dy disqe 0.6mm me 1 ane dhe duke i ngjitur ata bashke. Për ti bere të gjitha versionet të kene të njëtën trashesi, disku me një ane përbhet nga një disk 0.6mm i ngjitur me një baze (disk bosh), (ose ndoshta ne të ardhmen, ky disk s'do të jetë bosh po do ketë 133 minuta me reklama, me shpresen se njerezit do të behen kurioz mbi përbajtjen e anes tjetër). Struktura e diskut me dy ane, me dy shtresa është treguar ne Fig. 5-23.

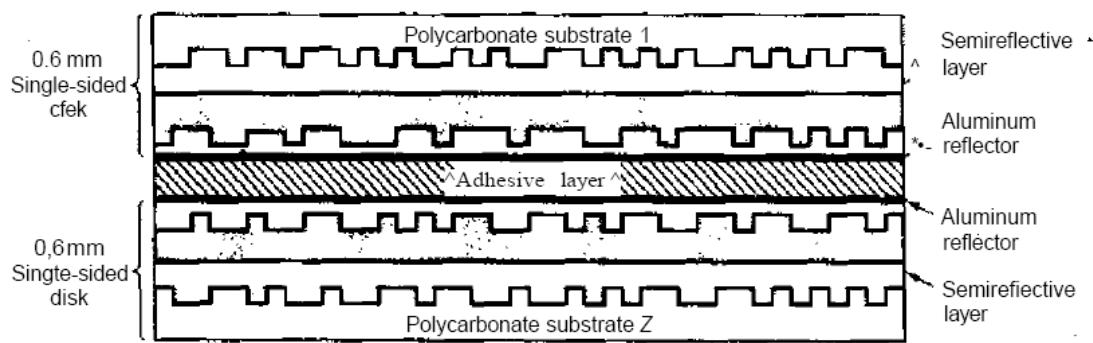


Figura 5-23 Një disk DVD me dy ane, me nga dy shtresa.

DVD u shpik nga një partneritet prej 10 kompanish elektronike, shtatë prej të cilave ishin japoneze, ne bashkepunim me studiot me të medha të Hollywood-it (disa prej të cilave janë prone e kompanive japoneze ne partneritet). Industritë e kompjuterave dhe e telekomunikacionit nuk u ftuan ne “piknik”, dhe si rezultat DVD-të u shpiken me qellim kryesor përdorimin për shitjen apo dhenien me qera të filmave. Për shembull, një tipar standart është kalimi ne kohe-reale (real-time) të “skenave të pista” (për ti lejuar prinderit ti bejne filmat e shikueshem edhe për femijet), audio me gjashtë kanale dhe suportimi i Pan-and-Scan. Vecoritë me të fundit i lejojne lexuesit e DVD (DVD player) të vendosin ne menyre dinamike ne se të presin pjesen e djathë apo të majtë të filmit (me rapport gjeresi:lartësi 3:2) për tu përshtatur me televizoret (me rapporte gjeresi:lartësi 4:3).

Një tjetër gje të cilën industria e kompjuterave nuk do ta kishte menduar ishte mospërputhja nderkombetare midis disqeve për Shtetet e Bashkuara dhe disqet për Europen, përseri standarte të ndryshme për kontinente të ndryshme. Hollwood-i kerkoi ketë “vecori” sepse filmat e rinj dalin ne fillim ne Shtetet e Bashkuara dhe me pas dergohen ne Europe. Idea ishte qe videotekat ne Europe të mos blinin videot ne Sh.B.A. me shpejt se sa ata qe do tu dergoheshin, duke ulur keshtu edhe shikimin e filmave ne kinematë Europiane. Ne se Hollwood-i do të kishte drejtuar industrine e kompjuterave, do të kishte cakuar floppy-disc 3.5 inch për Shtetet e Bashkuara dhe floppy-disc 9 cm Eruopen.

5.5.2 Formatimi i Diskut (Disc Formatting)

Hard disku përbehet nga një pirg me pjata prej alizhi alumini, ose qelqi, me diametër 5.25 ose 3,5 inch (1 inch=2.54 cm), (ose edhe me të vegjel ne komjuterat portative). Çdo pjatë ka një shtrese oksid metali, të magnetizueshem. Mbas prodhimit, nuk ka asnjë informacion ne disk. Përpara se disku të përdoret, çdo pjatë ka nevoje ti nenshtrohet një **formatimi të nivelit të ulet (low-level format)**, i bere nga software-i. Formatimi ka të beje me krijimin e gjurmeve (tracks), ku secila gjurme përmban një numer të caktuar sektoresh, me hapesira të vogla midis sektoreve. Përbajtja e një sektori tregohet ne Fig.5-24:

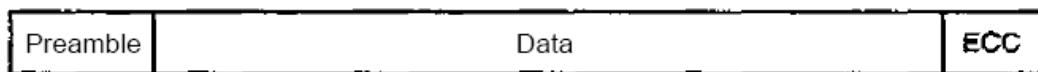


Figura 5-24 Sektori i një disku

Hyrja (preamble) fillon me një motiv të caktuar bit-esh qe i lejon hardware-it të njohe fillimin e një sektori. Gjithashtu ajo përmban numrin e cilindrit të sektorit si dhe disa informacione me shume. Madhesia e pjeses qe mban të dhenat, vendoset nga programi formatues i nivelit të ulet. Shumica e disqeve i kane sektoret nga 512-byte. Pjesa ECC (Error Correcting Code) përmban informacion shtese qe mund të përdoret për të zgjidhur gabimet ne lexim. Nuk është e cuditshme qe pjesa e ECC të ketë 16-byte. Për me tepër, të gjithe hard disqet kane disa sektore rezerve, për të zevendesuar sektoret me difekt prodhimi.

Pozicioni i sektorit 0 ne secilen gjurme (track) është shmangur nga pozicioni ne gjurmen paraardhese, kur behet formatimi i nivelit të ulet. Kjo zhvendosje, qe quhet **shtremberimi-cilindrit (cylinder-skew)**, behet për të përmiresuar efektshmerine. Ideja është, qe ti lejosh diskut të lexoje disa gjurme me një veprim të vetëm të vazhduar, pa humbur të dhena. Problemi mund të dallohet duke pare Fig.5-18(a). Supozoni se një kerkese ka nevoje për 18 sektore duke filluar nga sektori 0 ne gjurmen me të brendshme. Për të lexuar 16 sektoret e pare nevojitet një rrotullim i plotë i diskut, por duhet të bejme një kerkim për të levizur tek gjurma e jashtme dhe të marrim sektorin e 17. Gjatë kohes qe koka lexuese ka levizur tek gjurma tjeter, sektori 0 i saj e ka kaluar koken, keshtu qe nevojitet një rrotullim i plotë qe ky sektor të kaloje përseri poshtë. Ky problem zgjidhet duke zhvendosur sektoret, sic tregohet ne Fig.5-25.

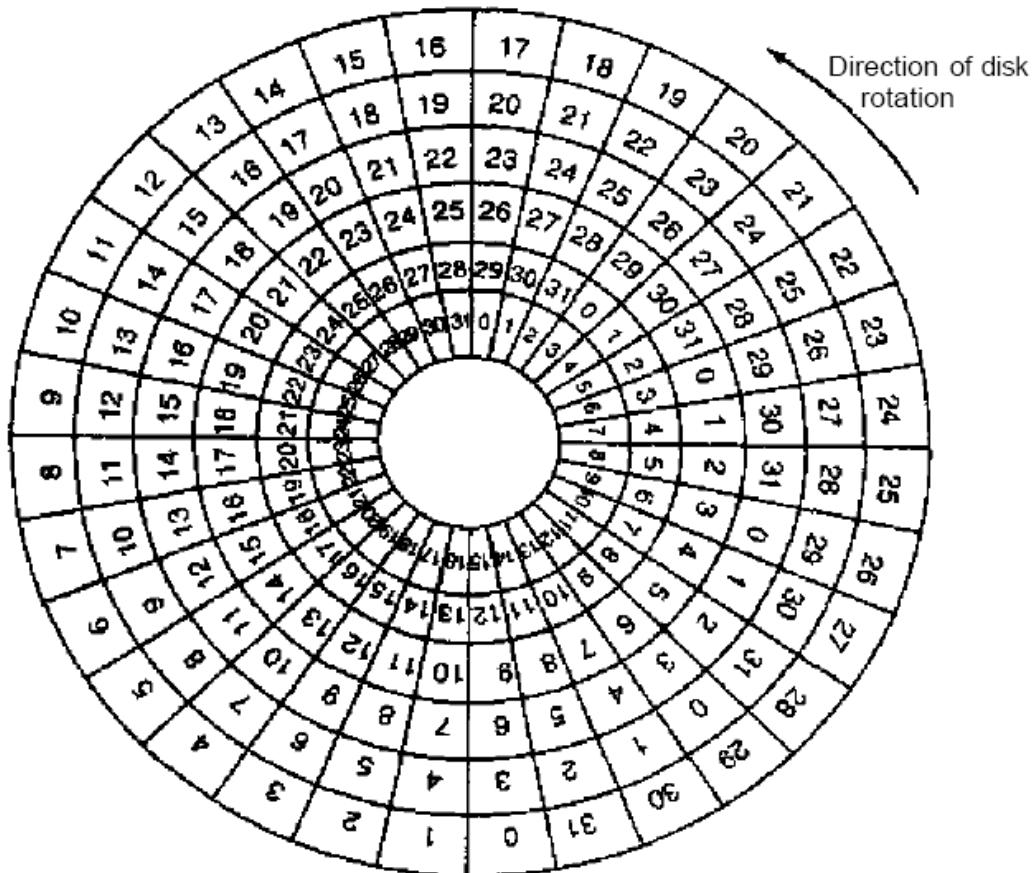


Figura 5-25 Një ilustrim i shtremberimit të cilindrit (cylinder skew)

Madhesia e shtremberimit të diskut varet nga gjeometria e pajisjes. Për shembull: një paisje me 10,000rrot/min rrrotullohet ne 6msek. Ne se një gjurme mban 300 sektore, një sektor i ri do të kaloje çdo 20 μ sek. Ne se koha e kerkimit gjurme-me-gjurme është 800 μ sek, do të kene kaluar 40 sektore gjatë kerkimit. Keshtu qe shtremberimi i cilindrit duhet të jetë 40 sektore ne vend të 3 sektoreve të treguar ne Fig.5-25. Duhet të përmendet se nderrimi i kokave lexuese kerkon një kohe të fundme, pra ka edhe një **shtremberimit-të-kokes (head-skew)** ashtu sic ka shtremberim të cilindrit, por shtremberimi-i-kokes nuk është shume i madh.

Si rezultat i formatimit të nivelit të ulet, kapaciteti i diskut zvogelohet, ne varesi të madhesise se hyrjes, hapesirave ndermjet sektoreve dhe ECC, por gjithashtu edhe i sektoreve rezerve. Shpesh kapaciteti pas formatimit është 20% me i ulet se kapaciteti i para formatimit. Sektoret rezerve nuk befne pjese ne kapacitetin pas formatimit. Të gjithe disqet e një lloji të caktuar kane të njëjtin kapacetit kur dergohen, pavaresisht numrit të sektoreve të prishur qe kane (ne se numri i sektoreve të prishur i tejkalon numrin e atyre rezerve, pajisja do të kthehet, e nuk do të shitet).

Ka një ngatërrim rreth kapacitetit të disqeve, sepse disa prodhues reklamojnë kapacitetin e diskut të pa formatuar. Kjo behet për ti bere të duken me të medha pajisjet e tyre, ndryshe nga cfare jane me të vertetë. Për shembull konsideroni një pajisje, kapaciteti i pa formatuar i të ciles është 20×10^9 byte. Kjo mund të shitet si një disk prej 20GB.

Megjithatë pas formatimit, vetëm $2^{34} \approx 17.2 \times 10^9$ byte mund të mbushen me të dhena. Akoma me shume, puna ngatërrohet për shkak se sistemi operativ do ta raportoje ketë kapacitet si 16GB, jo 17.2GB sepse ai e konsideron $1GB = 2^{30}$ (1,073,741,824)byte, dhe jo 10^9 (1,000,000,000) byte.

Gjerat nderlikohen me shume kur konsiderohet bota e komunikacionit, ku 1Gbps do të 10^9 thotë 1,000,000,000 bit/sek. Kjo ndodh sepse parashtesa *giga* ne të vertetë do të thotë 10^9 (një kilometër është 1000 metra , dhe jo 1024 metra). Vetëm me memorien dhe me madhesite e disqeve, kilo, mega, giga, tera janë sipas rradhes: $2^{10}, 2^{20}, 2^{30}, 2^{40}$.

Formatimi ndikon gjithashtu edhe ne efektshmeri. Ne se një disk me 10,000 rrot/min ka 300 sektore për gjurmë, me nga 512 byte për çdo sektor, do ti duhen 6msek për të lexuar 153,600 byte ne një gjurmë, duke dhene një shpejtësi prej 25,600 byte/sek ose 24,4 MB/sek. Është e pamundur qe të lexoje me shpejt se kaq, pavaresisht nga lloji i nderfaqes qe ti vendosesh, edhe po të jetë një nderfaqe SCSI me shpejtësi 80 MB/sek ose 160 MB/sek.

Për të lexuar vazhdimisht me ketë shpejtësi do të duhej një buffer i madh tek kontrolleri. Konsideroni për shembull, një kontroller me një buffer prej një sektori madhesi, qe ka marre komanden për të lexuar dy sektore të vijueshem. Mbasi lexohet sektori i pare nga disku, e pas llogaritjes se ECC, të dhenat duhet të trasferohen tek memoria kryesore. Gjatë kohes qe kryhet ky trasferim, sektori tjetër e kalon koken lexuese. Kur mbaron kopjimi ne memorien kryesore, kontrolluesi duhet të presi pothuajse për një periode rrotullimi, ne menyre qe sekotri i dytë të vije poshtë kokes lexuese përseni.

Problemi mund të eliminohet duke i numeruar sektoret ndryshe, kur të behet formatimi. Ne Fig. 5-26 (a) është treguar numerimi (duke injoruar shtremberimin e cilindrave). Ne Fig.5-26 (b) tregohet **single interleaving**, i cili i jep kontrolluesit pak pushim gjatë sektoreve të njëpasnjëshem, ne menyre të tille qe të kopjoje buffer-in ne memorien kryesore.

Ne se proçesi i kopjimit është shume i ngadaltë, është e nevojeshme **double interleaving**, i treguar ne Fig. 5-27(c). Ne se një kontroller ka një buffer prej vetëm një sektori, nuk ka rendesi ne se kopjimi do të kryhet nga kontrolluesi, nga CPU apo nga DMA, asaj do ti duhet një fare kohe. Për të shmangur nevojen e një numerimi të atille të sektoreve, kontrolluesi duhet të jetë ne gjendje qe të mbaje një gjurmë të tërë. Shume kontrollera modern mund ta bejne ketë gje.

Mbas formatimit të nivelit të ulet, disku është i ndare ne particion-one (pjese). Llogjikisht çdo particion është si një disk i ndare. Tek Pentium-i dhe ne shumicen e kompjuterave të tjere, sektori 0 përmban **master boot record**, (të dhenat kryesore të fillimit të punes), e cila mban disa reshta kod për fillimin e punes se kompjuterit plus tabelen e particioneve ne fund.

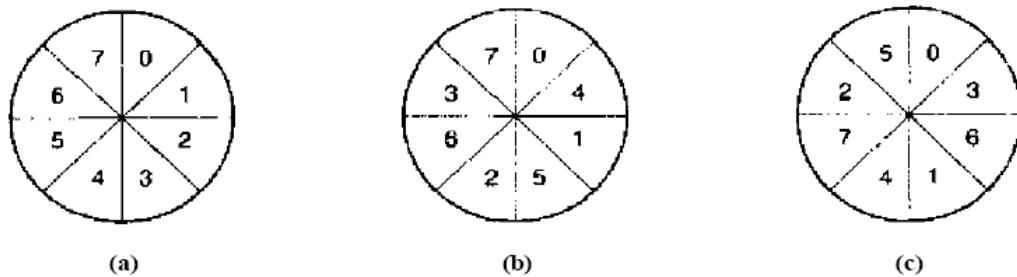


Figura 5-26 (a) Pa interleaving (b)Single interleaving (c) Double interleaving

Tabela e particioneve (partition table) jep sektorin e fillimit dhe madhesine e çdo particioni. Tek kompjuterat Pentium, tabela particioneve ka vend për katër particione. Ne se të gjitha jane për Windows, ata do të quhen C:, D:, E: e F: dhe do të trajtohen si pajisje me vetë. Ne se tre prej tyre jane për Windows e një është për UNIX, atëhere Windowsi do ti quaj C:, D:, E:. CD-ROM-i i pare do të quhet F:. Për të filluar punen kompjuteri (boot) nga leximi i njërit prej sektoreve, ai duhet të jetë i shenuar si aktive ne tabelen e particionve.

Hapi i fundit ne përgatitjen e një disku për përdorim, është kryerja e një **formatimi të nivelistës së lartë (high-level format)** të secilit particion (veçmas). Ky operacion vendos një bllok të fillimit të punes se kompjuterit (boot block), administrimin e hapesirave boshe (free list ose me bitmap), direktorin rrenjë (root directory) dhe një sistem bosh file-sh. Gjithashtu ai vendos një kod ne tabelen e particionve qe tregon cili sistem file-sh është përdorur ne atë particion. Kjo behet sepse sisteme operative të ndryshem mund të lexojne sisteme file-sh të ndryshem dhe të papajtueshem (për arsyen historike). Tani sistemi mund të filloje punen (boot-ed).

Kur ndizet kompjuteri, ekzekutohet ne fillim BIOS dhe pastaj lexohet master boot record. Ky program kontrollon se cili nga particionet është aktive. Pastaj ai lexon sektorin për fillimin (boot sector) dhe e ekzekuton atë. Sektori për fillim (boot sector) përmban një program të vogel i cili kerkon direktorine rrenjë e një programi të caktuar (ose të sistemit operativ ose të një ngarkuesi të sistemit të fillimit, me të madh; - bootstrap loader). Ai program ngarkohet ne memorie dhe ekzekutohet.

5.4.3 Algoritmet e planifikimit të krahut të diskut (Disk arm Scheduling Algorithms)

Ne ketë pjese do të shikojme disa ceshtje të përgjithshme rrëth pajisjeve të disqeve. Ne fillim, mendoni se sa do të jetë koha e nevojshme për leximin apo shkrimin e një bllokut. Koha e nevojshme përcaktohet nga tre faktore:

1. koha kerkimit (koha për të levizur krahun ne cilindrin e duhur)
2. vonesa e rrotullimit (koha qe sektori i duhur të rrotullohet poshtë kokes)
3. koha reale e transferimit të të dheneve.

Për shumicen e disqeve, koha kerkimit mbizotëron mbi dy kohet e tjera. Duke ulur kohen mesatare të kerkimit mund të përmiresohet ndjeshem efektshmeria.

Ne se pajisja pranon vetëm një kerkese dhe e përfundon atë, pra First-Come First-Served (FCFS), nuk mund të behet ndonjë gje e madhe për të përmiresuar kohen e kerkimit. Megjithatë mund të përdoret një strategji tjeter, kur disku është shume i ngarkuar. Është e mundur qe gjatë kohes qe krahu po punon për plotësimin e një kerkese, proçese të tjera të kene kerkesat e tyre për sherbim nga disku. Shume pajisje mbajne një tabele, e rradhitur sipas numrit të cilindrit, me të gjitha kerkesat e paplotësuara për çdo cilinder. Kerkesat e shumta për një cilinder mbahen ne lista zinxhir, të cilat e kane fillimin tek rrjeshtat e tabelles.

Me ketë lloj strukture të dhenash, mund të përmiresojme algoritmin e planifikimit first-come, first-served. Kosideroni një disk imagjinar me 40 cilindra. Vjen një kerkese për leximin e një bloku ne cilindrin 11. Gjatë kohes se kerkimit të cilindrit 11, vijne kerkesa të reja për cilindrat 1,36,16,34,9 dhe 12, me ketë rradhe. Ato futen ne tabelen e kerkesave të pambaruara (pending request), me një listë zinxhir (linked list) për çdo cilinder. Kerkesat tregohen ne Fig. 5-27:

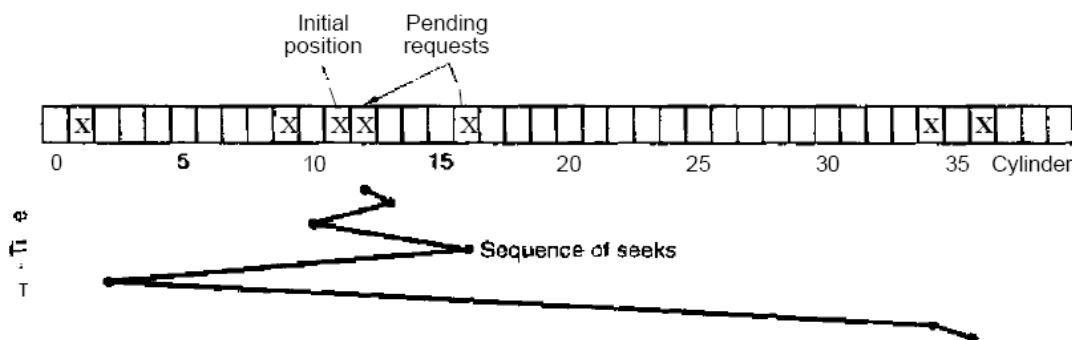


Figura 5-27 Algoritmi : i pari, kerkimi me i shkurtër (Shortest Seek First – SSF)

Kur kerkesa e pare (ajo për cilindrin 11) mbaron, pajisja mund të zgjedhe se cilen kerkesë të plotësoj me pare. Duke përdorur FCFS ajo do të shkonte tek cilindri 1, pastaj tek 36, e keshtu me rradhe. Ky algoritem do coje ne një zhvendosje prej 10,35,20 18,25, dhe 3 cilindrash respektivisht, për një total prej 111 cilindrash.

Ndryshe, pajisja mund ti përgjigjet gjithmone kerkeses me të afert të pares, për të minimizuar kohen e kerkimit. Me kerkesat e Fig.5-27, sekuenca e kerkesave do të jetë 12, 9, 16, 1,34 dhe 36, të treguara me vijen e thyer poshtë fig.5-27. Me ketë renditje, levizjet e krahet do të jene 1, 2, 7, 15, 33 dhe 2, me një total prej 61 cilindrash. Ky algoritem **Shortest Seek First (SSF = i pari, kerkimi me i shkurtër)** pothuajse e përgjysmon kohen e kerkimit ne krahasim me FCFS.

Fatkeqeshishit, SSF ka një problem. Supozoni se gjatë kohes se kryerjes se kerkesave të Fig. 5-27 do vijne vazhdimesht kerkesa të reja. Për shembull, mbasi krahu ka shkuar tek cilindri 16, vjen një kerkese për cilindrin 8, e cila do të ketë prioritet mbi kerkesen për cilindri 1. Ne se me pas vjen një kerkese për cilindrin 13, krahu do të shkoje tek cilindri 13, ne vend të cilindrit 1. Me një disk të ngarkuar, krahu do të tentoje të qendroje ne mes

të diskut shumicen e kohes. Keshtu kerkesat ne dy skajet do tu duhet të presin për një fluktuacion statistikor të kerkesave të reja, i cili të shkaktoje mungesen e kerkesave ne mes të diskut. Kerkesat larg qendres do të kene sherbim të varfer. Qellimet e kohes minimale dhe barazize se kohes se pritjes jane ne konflikt ne ketë rast.

Edhe ndertesat e larta duhet të përballohen me ketë problem. Problemi i planifikimit të levizjes se një ashensori ne një ndertëse të lartë, është i njëjtë me atë të planifikimit të levizjes se krahut të diskut. Kerkesat vijnë vazhdimesh duke therritur ashensorin ne një kat të caktuar (cilinder). Kompjuteri mund të mbaje rendin e kerkesave ne të cilën klientët kane shtypur butonin dhe ti sherbeje ata duke përdorur FCFS. Ai mund të përdore edhe SSR.

Megjithatë, shumica e ashensoreve përdorin një algoritem tjetër për ekuilibrimin e efektshmerise dhe drejtësise. Ata vazhdojnë të levizin ne të njëjtin drejtim derisa nuk ka me kerkesa ne atë drejtim. Pastaj ata nderrojnë drejtim. Ky algoritem, i njohur si ne botën e ashensoreve ashtu edhe ne atë te disqeve me emrin **algoritmi i ashensorit (elevator algorithm)**, ka nevoje qe software-i të mbaje një bit për drejtimin: lart ose poshtë. Kur mbaron një kerkesë, kontrollohet bit-i i drejtimit. Ne se është *lart* krahu apo kabina levizin tek kerkesa lart me e afert, e pambaruar. Ne se nuk ka me kerkesa të pambaruara ne ketë drejtim bit-i i drejtimit ndryshohet. Kur bit-i është *poshtë*, levizja behet ne drejtim të kerkeses me të afert poshtë, të pambaruar, ne se ka.

Figura 5-28 tregon zbatimin e algoritmit të ashensorit duke përdorur të njëjtat kerkesa si ne Fig.5-27, duke supozuar se te bit-i drejtimi ka qene fillimi i *lart*. Renditja ne të cilën u sherbehet kerkesave është 12, 16, 34, 36, 9 dhe 1. Kjo con ne keto levizje të krahut: 1, 4, 18, 2, 27 dhe 8; me një shume prej 60 cilindrash. Ne ketë rast algoritmi i ashensorit është pak me i mire se SSF, megjithese zakonish është me i keq. Një vecori e algoritmit të ashensorit është qe kufiri maksimal i numrit total të levizjeve, pavaresisht shpërndarjes se kerkesave, është sa dyfishi i numrit të cilindrave.

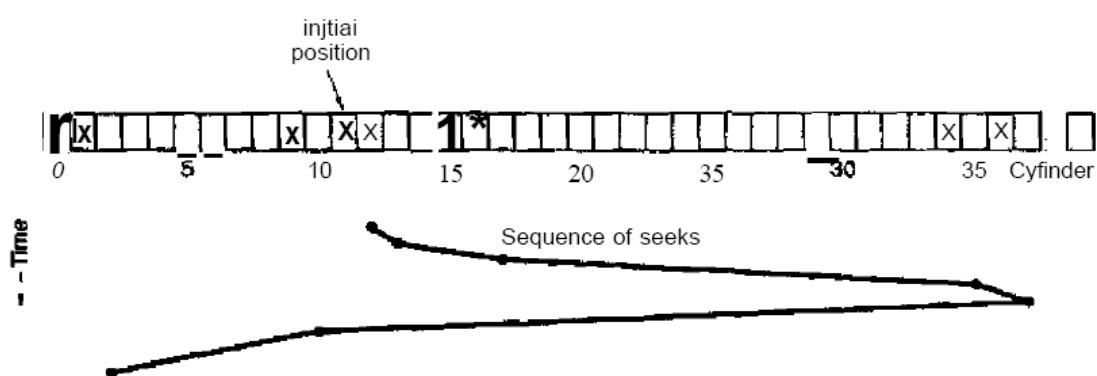


Figura 5-28 Algoritmi i ashensorit për menaxhimin e kerkesave të diskut

Një modifikim i ketij algoritmi, me një kohe të përgjigjes me pak të ndryshueshme, është të kerkosh ne atë drejtim. Kur cilidri me numrin me të madh dhe me një kerkesë të pambaruar, mbaron pune, krahu shkon ne cilindrin me numer me të vogel dhe me një kerkesë të pambaruar. Me pas vazhdo të leviz ne drejtimin rritës. Si rezultat, cilindrit me numrin me të vogel mendohet të jetë mbi cilindrin me numrin me të madh.

Disa kontrollues të diskut bejne të mundur qe software-i të kontrolloje numrin e sektorit të pranishem nen koken lexuese. Me një kontrollues të tille, është e mundur një organizim tjetër. Ne se ka dy ose me shume kerkesa të paplotësuara për të njëjtin cilinder, pajisja mund të sherbeje kerkesen, sektori i të ciles do të kaloje e para poshtë diskut. Vini re se, kur një cilinder ka disa gjurme, kerkesat e njëpasnjëshme mund të jene për gjurme të ndryshme, pa vonesa të tjera ne kohe. Kontrolleri mund të zgjedhe çdonjëren prej kokave menjehere, sepse zgjedhja e kokes lexuese nuk ka të beje as me levizje të krahut, as me vonesa ne rrotullim.

Ne se disku ka veti të tille qe koha e kerkimit është me e vogel se vonesa e rrotullimit, atëherë duhet të përdoret një strategji tjetër përmiresimi. Kerkesat e paplotësuara, duhet të rradhitën sipas numrit të sektorit. Sapo të afrohet sektori tjetër qe do lexohet, krahu duhet të levize tek gjurma e duhur për ta lexuar apo shkuajtur.

Ne disqet modern, vonesat e kerkimit dhe rrotullimit ndikojne kaq shume ne efekshmeri, saqe të lexosh një sektor të vetëm çdo here, nuk është aspak e frytshme. Për ketë arsy, shume kontrollues disqesh lexojne dhe mbajne për pak kohe (cache) gjithmone disa sektore, edhe kur nevojitet vetëm një. Zakonisht një kerkesa për leximin e një sektori, do të coje ne leximin e disa apo të të gjithe sektoreve të tjere të gjurmës, ne varesi të madhesise se kujteses cache të kontrolluesit. Disku i treguar ne Fig.5-17, ka ne cache prej 2MB ose 4MB, për shembull. Përdorimi i cache përcaktohet nga kontrolluesi. Me menyren me të thjeshtë, cache ndahet ne dy pjese, një për leximin dhe një për shkrimin. Ne se kerkesa pasuese mund të plotësohet nga kujtesa cache, të dhenat e kerkuara kthehen menjehere.

Duhet përmendur se cache-ja e kontrolluesit të diskut është plotësisht e pavarur nga cache-ja e sistemit operativ. Cache-ja e kontrolluesit zakonisht mban bлоqe të dhenash të cilat ne të vertetë nuk jane kerkuar për lexim, por ishte i përshtatshem leximi i tyre, sepse u ndodhi të kalonin poshtë kokes, si efekt anesor i leximit të një sektori. Ndersa, çdo cache tjetër qe mban sistemi operativ do përbaje bлоqe të cilat jane lexuar ne baze të një kerkesë dhe të cilat sistemi operativ mendon se do ti duhen përseni ne të ardhmen.

Kur një kontroller ka disa pajisje, ai duhet të mbaje nga një tabele kerkesash të paplotësuara (pending requests) për çdo pajisje. Kur ndonjë nga pajisjet është papune, duhet të krijohet një kerkim qe të levize krahun tek cilindri, ku do të lexohet me vone (duke supozuar se kontrolleri lejon kerkimet e mbivendosura). Kur një trasferim të dhenash mbaron, behet një kontroll për të pare ne se ndonjë nga pajisjet është ne cilindrin e duhur. Ne se ka një ose me shume, trasferimi tjetër mund të filloje nga ajo pajisje qe është vendosur ne cilindrin e duhur. Ne se asnë nga krahët nuk është vendosur ne vendin e duhur, driver-i duhet të urdheroje një kerkim të ri (seek) ne pajisjen qe sapo mbaroi trasferimin, dhe të prese interruptin tjetër për të pare se cili nga krahët ka arritur i pari.

Është e rendesishme të kuptohet se të gjithe algoritmat e mesipërm supozonin qe gjeometria e disqeve reale është e njëjtë me atë virtual. Ne se nuk jane të tille, planifikimi i kerkesave të disqeve nuk ka kuptim sepse sistemi operativ nuk mund të dalloje ne se cilindri 40 apo cilindri 39 është afer cilindrit 39. Nga ana tjetër, ne se kontrolluesi i diskut mund të pranoje kerkesa të shumta menjehersh, ai mund ti përdor vetë keto algoritme. Ne ketë rast algoritmat Jane të vlefshme, por të zbatuara ne një nivel me poshtë, tek kontrolluesi.

5.4.4 Trajtimi i Gabimeve (Error Handling)

Prodhuesit e disqeve po kapërcejne limitet e teknologjise duke rritur gjithnjë e me shume dendesine lineare të bite-ve. Një gjurmë ne mes të një diskut 5,25 inch ka një perimetër rrëth 300mm. Ne se gjurma mban 300 sektore prej 512 byte, dendesia lineare mund të jetë rrëth 5000 bit/mm, duke marre parasysh se një pjese humbet për fillimin e sektoreve (preable), ECC dhe hapesirat ndermjet sektoreve. Për të regjistruar 5000 bit/mm nevojitet një sipërfaqe shume uniforme dhe një mbulese shume e mire oksidi. Fatkeqeshish, nuk është e mundur prodhimi i disqeve me keto specifikime, pa difekte. Për sa kohe përmiresohet teknologjia e prodhimit ne atë pike qe lejon punen pa gabime me keto dendesi, projektuesit e disqeve do të përdorin dendesi edhe me të larta për të rritur kapacitetin. Kjo gje do të rifusi përseri, me shume mundesi difektet.

Difektet e prodhimit cojne ne sektore të prishur, të cilët janë sektore nga ku nuk mund të rilexohet një vlerë e sapo shkruar. Ne se difekti është shume i vogel, le të themi vetëm disa bit, është e mundur të përdoret ky sektori i prishur dhe ECC le të korigjoje çdo here gabimet. Ne se difekti është me i madh, gabimet nuk mund të fshihen.

Ka dy zgjidhje të përgjithshme për sa i përket bloqeve të prishur (bad block): të merret kontrolluesi me to ose të merret sistemi operativ me to. Ne zgjidhjen e pare, përparrë se diskut të niset për shitje nga fabrika, testohet dhe ne të shkruhet një listë me sektoret e prishur. Çdo sektor i prishur, zevendesohet nga një sektor rezerve.

Ka dy menyra për ta bere ketë. Ne Fig. 5-29(a), shohim vetëm gjurmë të një disk me 30 sektore dhe dy rezerve. Sektori 7 është me difekt. Ajo cfare mund të beje kontrolleri, është të shenoje një nga sektoret rezerve si sektori 7, sic tregohet ne fig.5-29(b). Menyra tjetër është qe të zhvendosesh të gjithe sektoret me një vend, sic tregohet ne Fig.5-29(c). Ne të dy rastet kontrolluesi duhet të njohe se cilët janë sektoret rezerve dhe ato të prishur. Ai mban shenim ketë informacion ne një tabelë të brendshme (një për çdo gjurmë) ose duke i rishkuar pjeset hyrese (preamble) të sektoreve, për ti dhene sektoreve të ndryshuar numrin e duhur. Ne se shkruhen pjese hyrese (preambles), metoda e figures 5-29(c), kerkon me shume pune, por pa dyshim qe efektshmeria do të jetë me e lartë. E gjithe gjurma do të mund të lexohet me një rrotullim.

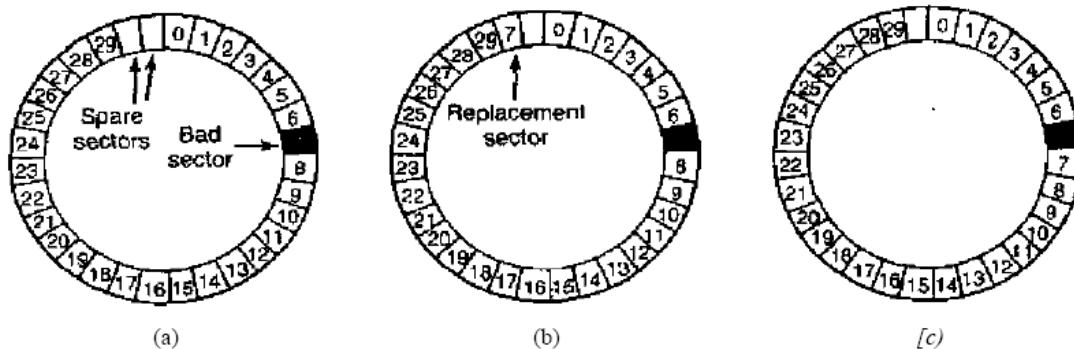


Figura 5-29 (a) Gjurma e një disku me një sektor të prishur. (b) Zevendesimi i sektorit të prishur me një rezerve. (c) Zhvendosja e të gjithe sektoreve për të anashkaluar një të prishur

Gabimet mund të lindin edhe gjatë punes, pasi është instaluar pajisja. Gjeja e pare qe behet kur kemi një gabim të cilin ECC nuk mundt ta rregulloje, është thjeshtë të provohet edhe një here leximi. Disa gabime jane të përkoheshme, do të thotë se shkaktohen nga grimca pluhuri poshtë kokes qe do të largohen ne proven e dytë. Ne se kontrolluesi dallon se po lexon një gabim të vazhdueshem ne një sektor të caktuar, ai mund të kaloje tek ata rezerve, përpara se sektori të prishet plotësisht. Ne ketë menyre, nuk humbasin të dhena dhe sistemi operativ e përdoruesi as nuk e vene re problemin. Zakonisht metoda e Fig. 5-29(b) duhet të përdoret kur sektoret e tjere kane të dhena. Përdorimi i metodes se Fig.5-29 do të kerkonte jo vetëm të rishkruaje pjeset hyrese (preamble), por edhe të kopjoje të dhenat.

Me pare u tha se ka dy menyre për të trajtuar gabimet: ti zgjidhe kontrolluesi ose ti zgjidhe sistemi operativ. Ne se kontrolluesi nuk ka mundesine qe të rregulloje sektoret ne menyre të pandjeshme, sic u tha me sipër, atëherë sistemi operativ duhet beje të njëjtën gje por me software. Ne fillim ai duhet të marre një listë me sektoret e prishur, ose duke i lexuar nga disku, ose duke e testuar vetë të gjithe diskun. Kur i ka gjetur sektoret e prishur, ai mund të ndertoje tabelen e rinumerimit. Ne se sistemi operativ do të përdore zgjidhjen e Fig.5-29(c), ai duhet të zhvendose me një sektor me sipër të dhenat nga sektori 7 ne sektorin 29.

Ne se merret sistemi operativ me rinumerimin e sektoreve, duhet sigurohet qe sektoret e prishur të mos kene ndonjë file dhe gjithashtu qe ata të mos gjenden asnjeherë ne listën ose hartën e hapesirave boshe (free list ose bitmap). Një zgjidhje e ketij problemi është krijimi i një file-i i cili përbehet nga të gjithe sektoret e prishur. Ne se ky file nuk vendoset ne sistemin e file-ve, përdoruesi nuk ka mundesine qe ta lexoje gabimisht (ose me keq akoma ta fshije atë, për të liruar vend)

Megjithatë ka edhe një problem tjetër: back up (kopjet rezerve). Ne se diskut i behet ndonjë kopje rezerve file për file, është e rendesishem qe programi kopjues të mos mundohet të kopjoje biloqet e prishur të file-ve. Për të parandaluar ketë, sistemi operativ duhet të fsheh file-in qe përbehet nga biloqet e prishur kaq mire, saqe edhe një program për back up mos ta gjeje atë. Ne se diskut i behet sektor për sektor do të jetë e veshtire, mos e pamundur, parandalimin e gabimeve të leximit gjatë kopjimit. Shpresat e vetme është qe programi kopjuese të ketë aq “zgjuarsi: sa të dorezohet pas 10 deshtimeve dhe të vazhdoje me sektorin tjetër.

Sektoret e prishur nuk Jane burimi i vetëm i gabimeve. Gabime kerkimi ndodhin për shkak të problemeve mekanike tek krahu. Kontrolluesi ndjek pozicionin e krahut. Për të realizuar një kerkim, ai i leshon një seri pulsesh motorrit të krahut, një puls për cilinder për të levizur krahan mbi një cilider të ri. Kur krahu mberrin aty ku duhet, kontrolluesi lexon numrin e cilindrit nga pjesa hyrese (preamble) e sektorit me të afer. Ne se krahu është ne vendin e gabuar, gjenerohet një gabim kerkimi (seek error).

Shumica e kontrolluesve të disqeve rregullojnë gabimet ne kerkim automatikisht, por shumica e kontrolluesve të floppy-it (duke përfshire edhe Pentium-in) thjeshtë vendosin një bit gabimi dhe ia lene driver-it të merret me të. Driver-i zgjidh ketë problem duke dhene një komande rikalibrimi, për të levizur krahan sa me ne periferi të diskut dhe të

rivendose cilindrin qe kontrolleri mendon të jetë 0. Zakonisht kjo zgjidh problemin. Ne se nuk e zgjidh, pajisja duhet të rregullohet.

Sic kemi pare, kontrolleri është me të vertetë një kompjuter i vogel i specializuar, me software-in e tij, variabla, buffer-a dhe ndonjë here me gabimet e tij të programimit (bug). Mund të ndodh qe një seri ngjarjesh të pazakonta, sic është ardhja e një interrupti ne një pajisje, njëkohesht me komanden e rikalibrimit ne një tjetër, të shkaktojne probleme (shfaqjen e bug-eve) dhe të sjelle një cikel të pafund të kontrolluesit ose humbjen e asaj cfare po bënte me përpara. Projektuesit e kontrolluesave zakonisht parashikojne rastin me të keq dhe i pajisin ata me një kunjë (pin) ne chip i cili kur nevojitet e detyron kontrolluesin të harroje cfaredo gjeje qe po bënte dhe të rifilloje (reset) nga e para. Ne se çdo gje tjetër deshton mund të vendos një bit të caktuar ne 1, për të kerkuar ketë sinjal dhe reset-uar kontrollerin. Ne se kjo nuk punon, e vetmja gje qe mund të beje driver-i është të shfaqe një mesazh gabimi dhe të dorezohet.

Rikalibri i një disku, përvec një zhurme të cuditshme, nuk është shqetësuese. Megjithatë ka një situatë ku rikalibri është një problem i madh: sistem me detyrime të kohes-reale (real-time constrain). Kur po luhet një video nga hard disku, ose po shkruhen ne CD-ROM file nga hard disku, është e nevojshme qe bitet të vijne nga hard disku me një shpejtësi uniforme. Ne keto kushte rikalibri fut hapesira ne rrjedhen e bite-ve (bit stream), e cila është e pa pranueshme. Pajisje speciale, të quajtura disqe AV (Audio Video disc), të cilat nuk rikalbrohen kurre, përdoren për keto aplikacione.

5.4.5 Ruajtje e Qendrueshme (Stable Storage)

Pame qe disqet nganjehere kane gabime. Sektore të mire mund të prishen papritmas. E gjithe pajisja mund të prishet pa parashikim. RAID-et mbrojne nga prishja e disa sektoreve ose edhe deshtimi i një pajisje të tërë. Megjithatë ata nuk mbrojne nga gabimet qe ne shkrim të të dhenave. Gjithashtu nuk mbrojne nga deshtimet gjatë shkrimit, të cilët prishin të dhenat origjinale pa i zevendesuar ato me të tjera.

Për disa zbatime, është themelore qe të dhenat të mos humbin ose të prishen kurre, edhe ne se ka gabime të diskut ose CPU-s. Një disk ideal duhet të punoje gjithe kohes pa gabime. Fatkeqesisht kjo nuk mund të arrihet. Ajo cfare mund të arrihet është një disk, nensistem, i cili ka ketë veti: kur komandohet një shkrim, disku ose shkruan pa gabime të dhenat, ose nuk bën asgje, duke i lene të dhenat ekzistuese të paprekura. Një sistem i tillë quhet **stable storage (ruajtje e qendrueshtme)** dhe zbatohet me ane të software-it (Lampson dhe Sturgis, 1979). Poshtë po përshkruajme një variant të idese origjinale.

Përpara se të përshkruajme algoritmin, është e rendesishme të kemi një model të quartë të gabimeve të mundshme. Modeli supozon se kur shkruhet një bllok (një ose me shume sektore), shkrimimi është ne rregull ose i gabuar dhe ky gabim mund të detektohet nga një lexic pasues i cili kontrollon vlerat e fushave të ECC. Ne parim një zbulim i sigurt i gabimeve nuk është i mundur kurre. Për shembull me një 16 byte ECC, të cilat sherbejnë

⁴⁰⁹⁶

për kontrollin e 512 byte të sektorit, ka 2^{144} vlera të mundshme të të dhenave dhe vetëm

2¹⁴⁴ vlera të mundshme të ECC. Keshtu ne se ndodh një ngatërrese gjatë një shkrimi por ECC është ne rregull, ka biliona kombinime të mundshme të gabuara qe cojne ne të njëjtën ECC. Ne se ndodh njëri prej tyre, gabimi nuk do të zbulohet. Probabiliteti qe një e

dhene rastesore të ketë ECC e duhur për mos tu zbuluar është rrëth 2^{-k} , e cila është aq e vogel sa mund ta quajme zero, por ne të vertetë nuk është zero.

Modeli gjithashtu merr parasysh faktin se një sektor i shkruar mire, mund të prishet vetveti dhe të behet i palexueshem. Megjithatë, supozohet se një ngjarje e tille është kaq e rralle, saqe probabiliteti i prishjes se të njëjtë sektor ne një pajisje tjetër (të pavarur) gjatë një intervali të arsyeshem kohe (per shembull një ditë) është aq e vogel sa të injorohet.

Modeli gjithashtu supozon se edhe CPU deshton. Ne ketë rast ai thjesht ndalon. Çdo shkrim i diskut ne castin e deshtimit, gjithashtu nderpritet, qe con ne të dhena të gabuara ne një sektor dhe ECC të gabuar, i cili mund të zbulohet me vone. Me gjithe keto kushte, stable storage mund të behet 100% i besueshem ne kuptimin qe kur shkruhet ose shkruhet pa gabime, ose lihen të dhenat e vjetra të paprekura. Sigurisht, nuk mbron nga katastrofat natyrore, si për shembull renia e një termeti dhe kompjuteri qe bie nga 100m lartësi ne një liqen me llave. Është e veshtire të mbrohesh ne keto rrethana me ane të software-it.

Stable storage përdor një cift disqesh identike, ku blloqet e njëjta ne disqeve të ndryshem formojne bashke një bllok pa gabime. Ne mungese të gabimeve blloqet korresponduese ne të dy pajisjet jane të njëjta. Njëri prej tyre mund të lexohet për të arritur ne të njëjtin rezultat. Për të arritur ketë qellim, jane përcaktuar tre veprimet e meposhtme.

1. Shkrim i qendrueshem (stable writes). Një shkrim i qendrueshem ka të beje me shkrimin e bllokut ne diskun 1 të parin, pastaj ne leximin e tij për verifikim. Ne se nuk është shkruar mire, shkrimi dhe leximi përsëriten derisa mos ketë gabime, ose deri ne **n** here. Ne se pas **n** deshtimeve rradhazi, blloku vendoset ne një sektor rezerve, dhe veprimi përsëritet derisa të ketë sukses, pavaresisht se sa sektor rezerve duhet të provohen. Mbas shkrimit ne pajisjen 1 u krye me sukses, do të shkruhet blloku i njëjtë ne pajisjen 2, dhe do të rilexohet, disa here, derisa edhe ketu shkrimi të ndodhe pa gabime. Ne mungese të deshtimeve të CPU-se, kur mbaron një shkrim i qendrueshem, blloku është shkruajtur pa gabime ne të dy pajisjet dhe është verifikuar tek të dyja.

2. Leximi i qendrueshem (stable read). Një lexim i qendrueshem lexon ne fillim bllokun nga pajisja 1. Ne se kjo ka një ECC jo te saktë, lexohet përseri, deri ne **n** here. Ne se të gjitha leximet jepin një ECC gabim, blloku përkatës do të lexohet tek pajisja 2. Duke ditur se një shkrim i qendrueshem krijon dy kopje të rregullta, dhe se sipas supozimit tone probabiliteti qe një bllok të prishet tek të dy pajisjet është i neglizhueshem, një lexim i qendrueshem është gjithnjë i sukseshem.

3. Rimarja nga deshtimet (Crash recovery). Mbas një deshtimi, një program rimarje (recovery) shqyrton të dy disqet duke krahasuar blloket e njëjta. Ne se një cift blloqesh jane pa gabime dhe të njëjtë, nuk behet asgje. Ne se njëri prej tyre ka një gabim ne ECC, blloku me gabim rishkruhet me të dhenat e bllokut tjetër pa gabime. Ne se një cift blloqesh nuk kane gabime, por jane të ndryshem, blloku ne pajisjen 1 shkruhet ne bllokun e pajisjes 2.

Ne mungese të deshtimeve të CPU, kjo skeme punon gjithmone sepse shkrimi i qendrueshem githmone shkruan dy kopje të vlefshme për një bllok dhe u supozua se gabime të vetvetishme nuk ndodhin anjehere njëkohesisht ne të dy kopjet e tij. Po ne se ndodh një deshtim i CPU gjatë një shkrimi të qendrueshem? Kjo varet nga casti se kur ndodh ky deshtim. Ka pese mundesi, sic tregohet ne Fig. 5-30:

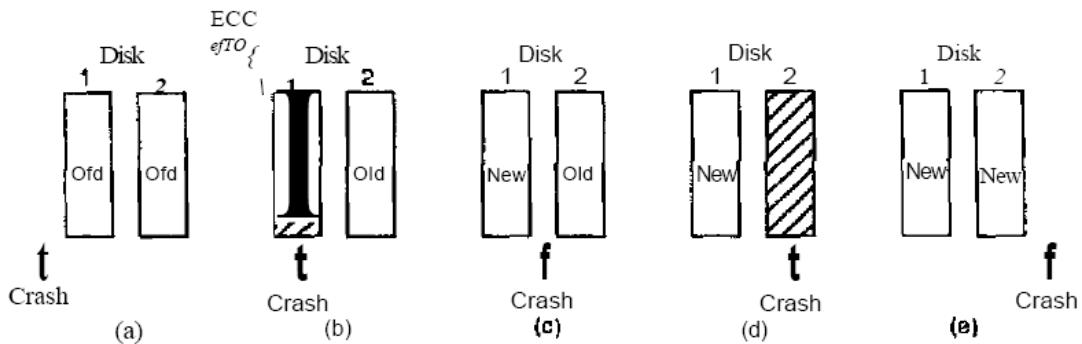


Figura 5-30 Analiza e ndikimit të deshtimeve ne shkrimin e qendrueshem

Ne Fig. 5-30 (a) deshtimi i CPU ndodh përpara se të shkruhet ndonjë nga kopjet e bllokut. Gjatë rrimarjes nga deshtimi, asgje nuk do të ndryshoje dhe të dhenat e vjetra do të ekzistojne akoma, gje qe lejohet.

Ne Fig.5-30(b) deshtimi i CPU ndodh gjatë shkrimit të pajisjes 1, duke shkateruar përbajtjen e bllokut të saj. Megjithatë programi rrimarjes do të detektoje gabimin dhe do të kopjoje bllokun e pajisjes 2 ne bllokun e prishur të pajisjes 1. Keshtu efekti i deshtimit zhduket dhe rrimeret plotësisht gjendja e meparshme.

Ne Fig.5-30(c) deshtimi i CPU ndodh pasi është shkruar pajisja 1 por përpara shkrimit të pajisjes 2. Është arritur pika e moskthimit: programi i rrimarjes do kopjoje bllokun nga pajisja 1 ne pajisjen 2. Shkrimi përfundon pa gabime.

Fig.5-30(d) është ngjashme me Fig.5-30(b). Gjatë rrimarjes, blloku pa gabime mbishkruhet ne atë të prishurin. Përseri, vlera e të dy blloqueve do të jetë ajo e bllokut të ri.

Se fundmi, ne Fig.5-30(e) programi i rrimarjes shikon se të dy blloqet jane të njëjtë, keshtu qe nuk ka nevoje për ndryshime dhe shkrimi përfundon me sukses.

Ka disa përmiresime të mundshme të kesaj skeme. Si fillim, krasimi i të gjithe cifteve të blloqueve gjatë rrimarjes është e mundur, por kerkon shume kohe. Një përmiresim i madh është të ruash numrin e gjurmës se bllokut i cili po shkruhej, keshtu qe do të nevojitet të kontrollohet vetëm një bllok gjatë rrimarjes. Disa kompjutera kane një memorie të vogel jo të zhdukshme (**nonvolatile RAM**) e cila është një memorie CMOS speciale e ushqyer nga një bateri litiumi. Keto bateri zgjasin vite, mundesisht gjatë gjithe jetës se kompjuterit. Ndryshe nga memoria kryesore, e cila humbet mbas një deshtimi, nonvolatile RAM nuk humbet pas një deshtimi. Ora e ditës mbahet ketu (dhe rritet nga një qark special). Kjo shpjegon faktin qe kompjuteri di gjithmone oren edhe po ta heqesh nga priza.

Supozoni se disa byte të nonvolatile RAM jane ne dispozicion të sistemit operativ. Shkrimit i qendrueshem mund të vendosi numrin e bllokut qe do të rifreskoje ne një nonvolatile RAM përparrë se të filloje shkrimin. Pas shkrimit të suksesshem, numri i bllokut tek nonvolatile RAM mbishkruhet me një numer jo të vlefshem, per shembull,. - 1. Ne keto kushte, mbas një deshtimi, programi i rrimarjes mund të kontrolloje nonvolatile RAM për të pare, ne se po kryhej një shkrim gjatë deshtimit. Ne se po, cili ishte blloku qe po shkruhej kur ndodhi deshtimi. Me pas dy kopjet e bllokut mund të kontrollohen për saktësimin dhe përputhjen e tyre.

Ne se nuk ka nonvolatile RAM, ai mund të sajohet sic vijon. Ne fillim të një shkrimi të qendrueshem, një bllok fiks të pajisjes 1 shkruhet me numrin e bllokut qe do të shkruhet. Ky bllok rilexohet për ta verifikuar. Pasi të shkruhet pa gabime, i njëjtë bllok shkruhet ne pajisjen 2 dhe verifikohet. Pas një shkrimi të qendrueshem, të dy keto blloqe mbishkruhen me një numer të pavlefshem. Edhe ketu, mbas një deshtimi është e mundur të përcaktohet lehtësisht ne se po kryhej një shkrim i qendrueshem ose jo. Kjo teknike kerkon me shume veprime me diskun, për të shkruar një bllok pa gabime, keshtu qe duhet të përdoret sa me rralle.

Edhe një pike tjetër qe ia vlen ta përmendesh. Ne supozuam se vetëm një prishje e vetëvetishme të një blloku ndodh gjatë ditës. Ne se kalojne disa ditë, mund të prishet edhe kopja e tij. Pra, një here ne ditë duhet të behet një kontroll i të dy disqeve për të rregulluar demtimet e mundeshme. Ne ketë menyre, çdo mengjes të dy disqet janë gjithmone identik. Edhe ne se prishen të dy blloqet e një cifti, prishen gjatë një periudhe disa ditore, të gjithe gabimet rregullohen.

5.5 CLOCKS

Clocks (gjithashtu te quajtur **timers “kohezues”**) jane te rendesishme ne punen e nje multi-programuesi per nje sere arsyesh. Ata mirembajne oren e sakte dhe ndalojne nje proces te monopolizoje CPU-ne, perc te tjerave. Clock-u i softwaret mund te marri dhe formen e nje driver te pajisjes, megjithese clock eshte nje bllok pajisjesh, si disku, jo nje karakter pajisjesh, si mausi. Ne seksionet e meposhteme ne do ta analizojme clock ne fillim si nje clock hardware dhe pastaj si clock software.

5.6 CHARACTER-ORIENTED TERMINALS

Çdo kompjuter ka të paktën një tastier dhe një ekran (monitor ose ekran të sheshtë) qe përdoren për të komunikuar me të. Megjithatë, tastiera dhe ekranit ne një kompjuter personal jane paisje teknikisht të ndara, të cilat punojne se bashku. Ne mainframe, ka shume përdoruesa të ndryshem ne distance të cilët përdorin paisje me një tastier dhe një ekran të ngjitur ne të. Keto paisje historikishte jane quajtur **terminale**. Ne do të përdorim ketë term edhe kur të diskutojme per kompjuterat personal.

Terminalet i kemi ne shume forma. Tre nga format qe i shohim me shpesh ne praktik jane:

1. Terminale të jashtëm me nderfaqe seriale RS-232 të përdorur ne mainframe.
2. Ekran të kompjuterave personal me GUI.
3. Terminale rrjeti.

Ne seksionin pasardhes do të përshkruajme secilin nga keta tipe me rradhe.

5.6.1 RS-232 Terminal Hardware

Terminalet RS-232 jane paisje hardware të cilat përbajne tastieren dhe ekranin, të cilat komunikojne me një nderfaqe seriale, një bit ne një cast kohe (shiko Fig.5-34). Keto terminale përdori konektor me 9 ose 25 pin, nga të cilët një pin përdoret për të transmetuar të dhenat, një pin është për të marre të dhenat dhe një tjeter është toka. Pinet e tjera Jane për të kontrolluar funksionet e ndryshme, shumica e tyre nuk përdoren. Linjat, ne të cilat karakteret dergojn një bit ne një cast kohe quhen **linja seriale**. Të gjithe modemat përdorin ketë nderfaqe. Ne UNIX linjat seriale kane emra si /dev/tty 1 dhe /dev/tty 2. Ne Windows ato kane emrat COM 1 DHE COM 2.

Për të derguar një karakter ne një terminal RS-232 ose modem nepërmjet një linje seriale, kompjuterit i duhet të transmetoj një bit ne një cast kohe, me një bit ne fillim si prefix i ndjekur nga një ose dy bit ndalimi qe caktone kufijt e karakterit. Një bit pariteti mund të vendoset duke paraprire bitin e ndalimit, gjithsesi kjo kerkohet vetëm ne komunikimet ne sistemet mainframe.

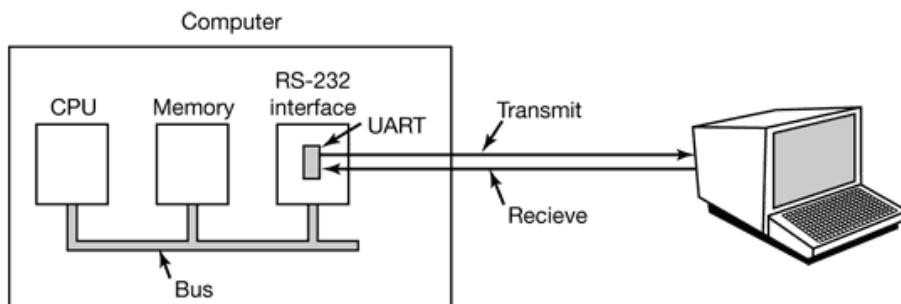


Figure 5-34. An RS-232 terminal communicates with a computer over a communication line, one bit at a time.

Terminalet RS-232 jane akoma shume të përdorshme ne botën e mainframe-eve të cilat befne të mundur komunikimin me user-at ne distance duke përdorur modem-at ose një linjë telefonike. Keto gjejne përdorim ne agjensit e fluturimit, ne sistemet bankare dhe ne industri të ndryshme. Edhe kur ato zevendesohen nga kompjutera personal, PC zakonisht e emulojn terminalin e vjetër RS-232 qe të shmangin ndryshimet ne softwaret e mainframe-it.

Keto terminale përdoren për të dominuar botën e minikompjuterave. Një arritje e madhe e software-ve qe përdoren ne sistemet e tilla, bazohen ne keto terminale. Për shembull të gjithe sistemet e UNIX i suportojne keto paisje.

Gjithesesi, akoma me e rendesishme, shume sisteme UNIX koherent të japid mundesin e krijimit të një dritareje, e cila përmban linja me numra të një teksti. Shume programuesa punojne ne text mode ne dritare të tillë si ne ndonjë kompjuetër personal, ose ne workstation. Keto dritare zakonisht stimulojnë disa terminale RS-232 dhe keshtu qe ato kane një përdorim sa me të gjere të software-it baze të shkruajtur për ketë terminal.

Tastiera dhe software i terminalit për ketë emulim terminali është njësoj si te terminali real. Meqene se emulatoret e terminaleve jane ne përdorim të gjere, software është akoma I rendesishem, keshtu qe do të përshkruajme ne dy seksionet e meposhtme.

RS-232 është character oriented. Me ketë kuptojme qe ekrani ose dritarja tregojne një numer të linjave të një teksti, secila ne një mase maksimale. Një përmase normale është 25 linja me 80 karaktere secila. Ndonë se disa karaktere jane ndonjhere mbështetës, keto terminale jane tekste ne parim.

Meqene se kompjuteri dhe terminali punojne me të gjithe karakteret duke komunikuar ne menyre seriale me një bit ne një moment kohe, chipet Jane zhvilluar ne menyre të tille qe të bejne konvertimin nga karakter ne serial dhe anasjelltas. Ato quhen **UART**-s (Universal Asynchronous Receiver Transmitters). Ato Jane të lidhura me kompjuterin nepërmjet nderfaqes RS-232 me busin të treguar si ne Fig. 5-34. Ne shume kompjutera, një ose shume porta seriale Jane të ndertuara brenda bordit prind.

Për të treguar një karakter, driverat e terminalit e shkruajne karakterin ne kartën e nderfaqes ku me pas bufferohet dhe nxirret jashtë linjës seriale me nga një bit ne një cast kohe nga UART. Për shembull, për një modem analog qe operon me 56,000 bps, do ti duhen 179 µsec për të derguar një karakter. Si rezultat i ketij rate të vogel transmetimi, driveri zakonisht nxjerr një karakter ne kartën RS-232 dhe bllok, duke pritur interruptin e gjeneruar nga nderfaqja, kur karakteri është transmetuar dhe UART është gati për të pritur një karakter tjeter. UART mund të pres dhe të dergoj karaktere njëkohesisht. Një interrupt gjenerohet kur merret një karakter dhe zakonisht një numer i vogel I karaktereve ne hyrje mund të bufferohet. Driveri I terminalit duhet të kontrolloj një regjistër kur merret një interrupt qe të përcaktoj shkakun e interruptit. Disa karta të nderfaqes kane një CPU dhe një memorie dhe mund të përballojne shume linja, duke marre peshen e paisjeve I/O nga CPU qendrore. Terminalet RS-232 mund të nendahen ne tre kategori. Me të thjeshtët Jane terminalet hardcopy. Karakteret e shtypur ne tastier transmetohen direkt ne kompjuter. Karakteret e derguar nga kompjuteri printohen ne letër.

Terminalet CRT punojne ne të njëjtën menyre, vetëm ne vend të letres kane një ekran. Keto zakonishtë quhen “glass ttys” sepse nga ana funksionale Jane njësoj si hardcopy ttys. (Termi “tty” është shkurtim I Teletype, një kompani e cila ishte pionere ne bisnesin e terminaleve kompjuterik: “tty” tregon çdo terminal).

Terminalet CRT Inteligjentë jane ne fakt kompjutera të specializuar ne miniatur. Ato kane një CPU dhe një memorie, dhe përbajne software, zakonisht ne ROM. Duke I pare nga ana e sistemit operativ, ndryshimi kryesor midis glass tty dhe një terminali Inteligjent është se ky I fundit I kalon disa sekuenca. Për shembull, duke derguar karakteret ESC ne ASCII, duke I pasuar me karaktere të ndryshme, mund të ndodh qe kurzori të leviz ne çdo pozicion të ekranit. Terminalet Inteligjent Jane të vetmet qe përdoren ne sistemet mainframe dhe mund të emullohen nga sisteme të ndryshme operative. Me softwaret e tyre do të merremi me poshtë.

5.6.2 Input software

Ekrani dhe tastiera Jane paisje të pavarura nga njëra tjetra dhe prandaj do ti trajtojme ne menyre të ndare. Ato nuk Jane tamam të pavarura nga njëra tjetra pasi kur shruajme një fjale ne tastier ajo na shfaqet ne ekran.

Puna kryesore e driver-it të tastieres është qe të mbledh inputet nga tastiera dhe ti dergoj ato të programet user kur të lexohen nga terminali. Dy filozofi të mundshme mund ti përshtaten driver-it. E para, qe puna e driverit është thjesht të marri inputet dhe ti dergoj ne një nivel me të lartë pa I përpunuar. Një program qe lexon terminalin merr një sekuenç të papërpunuar ne kodin ASCII.

Kjo filozofi është shume e përshtatshme për nevojat e ekraneve të sofistikuar si *emacs*, e cila e lejon përdoruesin të lidhet me një aksion arbitrar me çdo karakter ose me sekuenç karakteresh. Në qoftë se përdoruesi shtyp dste ne vend të date, dhe me pas e korrigjon gabimin duke shtypur tre here backspace dhe ate, programi I përdoruesit do të jape të gjithe kodet 11 ASCII si me poshtë:

dste ← ← ← ate CR

Jo të gjithe programet duan kaq shume detaje. Zakonisht ato duan inputin e saktë dhe jo sekuençen se si është proceduar. Kjo verejte na con ne filozofin e dytë: driveri e përballon të gjithe montimin intraline, dhe vetem sa dergon linjat korrekte programeve ne përdorim. Filozofia e pare është character oriented; ndersa e dyta është line oriented. I referohen **raw mode** dhe **cooked mode**, respektivisht. Standartet POSIX përdorin canonical mode për të përshkruar line-oriented. **Noncanonical mode** është ekuivalente me raw mode, megjithese shume detaje të sjelljes se terminaleve mund të ndryshohen.

Sistemet kompatibile POSIX jepin librari funksionesh të ndryshme të cilat suportojne selektimin dhe ndryshimin e shume aspekteve të konfigurimit të terminalit.

Puna e pare e driverit të tastieres është të mbledhi karakteret. Në qoftë se çdo shtypje e celsit shkakton interrupt, driveri mund ta marre karakterin gjatë interruptit. Në qoftë se interruptet kthehen ne mesazhe nga nivel i ulet i software-it, është e mundur qe të

vendosim karakteret e reja ne mesazh. Ato vendosen ne një buffer të vogel të memories dhe mesazhi përdoret për ti thene driverit qe dicka erdhi.

Në qoftë se terminali është ne canonical (cooked) mode, karakteret duhet të ruhen derisa një linjë e tëre të akumulohet, sepse përdoruesi me pas mund të vendos të fshij një pjese të tij. Edhe ne se terminali është ne raw mode, programi mund të mos ketë kerkuar akoma input prandaj duhet qe karakteret të bufferohen.

Dy rruget e bufferimit të karaktereve jane të thjeshta. Ne të paren, driveri përmban një grup me buffera, ku secili buffer ka 10 karaktere. Ngjitur mbas çdo terminali jepet edhe struktura e të dhenave, e cila përmban midis të tjera një pointer ne grupin e bufferave për inputet të mbledhura nga terminali. Sa me shume karaktere të shtypen, aq me shume buffera do të kerkohen, të cilat do të bashkohen me grupin e bufferave. Kur karakteret të kalojnë për te programi ne përdorim, bufferat do të kthehen përseri të grupei I bufferave.

Rruga e dytë është qe bufferimi të behet direkt nga terminali I strukture se të dhenave pa gene nevoja e një grupei bufferash. Meqene se për përdoruesin është bere e zakonshme qe përvèc disa komandave të cilat kerkojne kohen e vetë edhe keto, të shkruaj dhe disa linja të tjera përbri tyre, dhe për të qene I sigurtë driveri alocon rreth 200 karaktere për terminal. Ne një sistem large-scale timesharing me 100 terminale, të alokosh 20K për çdo here është një makth I vertet, prandaj një hapesire prej 5K do të ishte mese e mjaftueshme. Nga ana tjetër, një buffer I dedikuar për çdo terminal e bën driverin me të thjeshtë dhe prandaj preferohet kjo menyre ne kompjuterat personal me një tastier të vetme. Fig.5-35 tregon ndryshimin midis ketyre dy menyrave.

Megjithese tastiera dhe ekranii jane dy paisje të ndara nga njëra tjetra, shume përdoruesa jane mesuar të shikojne se ajo c'ka shkruajne ne tastier të shfaqet ne ekran. Disa terminale (të vjetër) shfaqin automatikisht atë cfare shtypet ne tastier, gje qe limiton fleksibilitetin e editoreve të sofistikuar dhe programeve të ndryshme. Fatmiresisht me terminalet e sotëm asgje nuk shfaqet automatikisht kur shtypet një celes. Është e gjitha ne dore të software-it ne kompjuter për të shfaqur atë qe do. Ky proces quhet **echoing**.

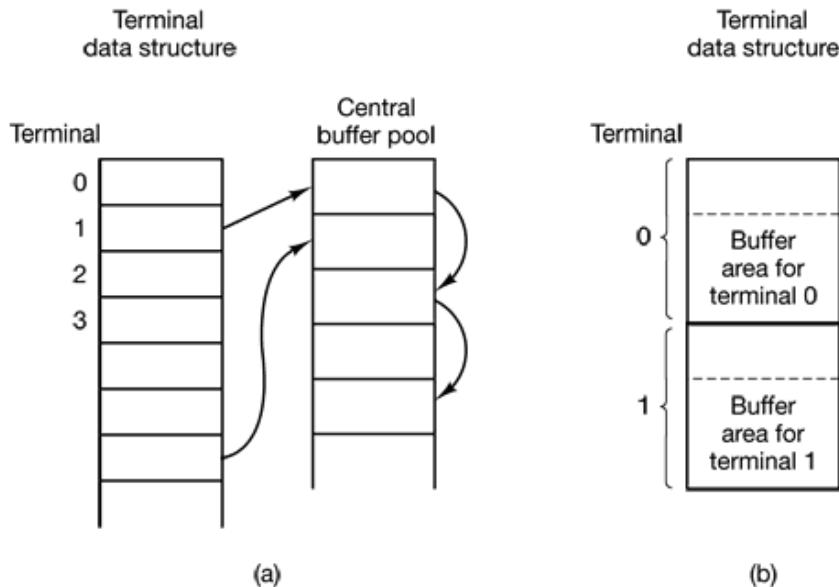


Figure 5-35. (a) Central buffer pool. (b) Dedicated buffer for each terminal.

Echoing është e komplikuar ne faktin qe programi mund të jetë duke shkruajtur ne ekran kur përdoruesi të jetë duke shtypur. Deri pak kohesh, driverat e tastieres i duhej të gjenin vetë se ku të vendosnin inputet e reja pa i mbishkruajtur nga programi output.

Echoing komplikohet gjithashtu kur me shume se 80 karaktere duhet të tregohen ne ekran me një linjë me 80 karaktere. Duke u varur nga aplikacionet, të hedhurit ne linjën tjetër mund të jetë e përshtatshme. Disa drivera nuk i njojin karakteret e tjera mbas karakterit të 80-të.

Një tjetër problem është edhe tab handling. Është pune e driverit të vendos se ku është vendosur kursori, duke mbledhur ne llogarin e tij outputet nga programi dhe outputet nga echoing dhe të vendosi numrin e duhur të hapesires për tu pasqyruar.

Tani vijme ne problemin e ekuivalences se paisjeve. Llogjikisht ne fund të një linje të një teksti, dikush kerkon një transport mbapsht, për të levizur kursorin mbapsht per shembull për të rreshti I pare dhe një linefeed për të shkuar te rreshti tjetër. Është pune e driverit për të konvertuar cfare do lloj gjeje qe vjen ne format standart të përdorur nga sitemi operativ.

Ne se forma standarte është vetëm për të ruajtur një linefeed, atëhere mbetja kthehet ne një linefeed. Në qoftë se formati I brendshem duhet qe ti ruaj të dyja, atëhere driveri duhet të gjeneroj një linefeed ku do të kete një mbetje dhe një mbetje qe do të mbaj linefeed. Terminali mund ti kerkoj të dyja, linefeed dhe mbetjen për ti pasqyruar qe të marri update-in e ekranit ashtu sic duhet. Meqene se një kompjuter I madh mund të ketë terminale të ndryshme të lidhur me të, është pune e driver-it të tastieres qe të konvertoj mbetjet e ndryshme dhe kombinimet linefeed ne standartet e brendshme dhe të caktoj qe çdo pasqyrim të kryhet ne menyre të drejtë.

Kur operohet me canonical mode, një numer I karaktereve ne hyrje kane një kuptim special. Fig.5-36 tregon të gjitha kuptimet speciale të kerkuar nga POSIX. Default-et Jane karaktere kontrolli të cilat nuk duhet të krijojne konflikt me textet ose me kodet të përdorura nga programi, por të gjitha, përvèc dy të fundit mund të ndryshohen nën kontrollin e programit.

Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Figure 5-36. Characters that are handled specially in canonical mode.

Karakteret ERASE lejojne përdoruesin qe të nxjerr të gjithe karakteret sic shkruhen. Është zakonisht backspace (CTRL-H). Ajo nuk i shtohet rradhes se karaktereve por prape arrin të nxjerr nga rradha karakterin paraardhes. Ajo duhet të pasqyrohet si një sekuenc me tre karaktere, backspace, space dhe backspace ne menyre qe të levizi karakterin paraardhes nga ekran. Ne se karakteri paraardhes ishte një tab, fshirja e tij varet se si është proceduar për shtypjen e tij. Në qoftë se vetë tab është ruajtur ne një rradhe input, ai mund të levize dhe e gjithe linja mund të dale përseri. Ajo nuk mund të fshij një mbetje dhe të kthehet mbrapa për shembull, ne linjën paraardhese.

Kur përdoruesi shikon një gabim ne fillim të një rreshti qe po shkruhet, ngadonjehere është me e përshtatshme ta fshij të gjithe rreshtin dhe ta filloj nga e para. Karakteri KILL fshin një linje komplet. Shume sisteme e zhdukin linjën e fshire nga ekran por disa e pasqyrojne atë, plus mbetjet dhe linefeed sepse disa përdoruesa kane deshire të shikojne linjën e vjetër. Dhe me ERASE zakonisht nuk është e mundur të kthehet me mbrapa sesa linja aktuale. Kur një bllok karakteresh vritet, për driverin nuk mund të jetë problem qe ta kthej bufferin ne grup, ne qofte se ai po përdoret.

Shpesh here karakteret ERASE ose KILL mund të futen si të dhena normale. Karakteret LNEXT sherbejne si karaktere **escape**. Ne UNIX CTRL-V është default. Si një shembull,

sistemet e vjetra zakonisht përdorin shenjën @ për komanden KILL, por interneti e përdor ketë shenje për t'ju referuar një adresë si *linda@cs.washington.edu*. Kush është mesuar të përdor shenjat e vjetra përshebull @ ne vend të KILL, për t'ju referuar një adresë interneti do ti duhet qe karakterin @ ta nxjerr nga komanda CTRL-V @. CTRL-V mund të futet vet nga komanda CTRL-V CTRL-V. Karakteret LNEXT nuk futen ne rradhen e karaktereve.

Për ti lejuar përdoruesit qe të ndalojne një pamje të figures, kodet e kontrollit e ngrijne ekranin dhe me pas rinisin me vone. Ne UNIX keto jane TOP, (CTRL-S) dhe START, (CTRL-Q) respektivisht. Ato nuk ruhen, por ripërdoren për të futur dhe pastruar një flamur ne terminalin e struktires se të dhenave. Për çdo përpjekje të outputit, flamuri është i kontrolluar. Në qoftë se është i caktuar nuk ndodh asnë output.

Zakonisht është e nevojshme qe të vrasesh një program qe po ekzekutohet. Karekteret INTR (DEL) dhe QUIT (CTRL-\) përdoren për ketë qellim. Ne UNIX, DEL i dergo një signal SIGT te gjithe proceseve të startuara nga terminali. Implementimi I DEL mund të jetë me pasoja. Pjesa me e veshtire është marrja e informacionit nga driveri për te pjesa e sistemit qe përmban sinjalet, të cilat mbas gjithe kesaj nuk kane kerkuar për këtë informacion.

Një karakter tjetër special është EOF (CTRL-D), I cili ne UNIX shkakton një kerkes për lexim për terminalin I cili është I gatshem për bufferin, edhe pse bufferi mund të jetë bosh. Shkrimi I CTRL-D ne fillim të një linje shkakton qe programi të marri ne lexim prej 0 byte I cili interpretohet si end-of-file

5.6.3 Output Software

Outputi është me I thjeshtë se inputi. Ne pjesen me të madhe, kompjuteri dergon karaktere për te terminali dhe ato shfaqen atje. Zakonisht, një bllok karakteresh, për shembull një linjë shkruhet ne terminal ne një thirrje të sistemit. Metoda qe përdoret me shpesh ne terminalet RS-232 është ajo me buffera output të shoqeruar ne çdo terminal. Bufferat mund të vijne nga i njëti grup si buffera input, ose mund të jene të dedikuar. Kur një program shkruan ne terminal, outputi është I pari qe kopjohet ne buffer. Ne menyre të ngjashme dhe outputi I pasqyrimit kopjohet ne buffer. Pasi të gjithe outputet kopjohen ne buffer, karakteri I pare është output dhe driveri e mbaron punen e tij. Kur vjen interrupti, karakteri tjetër është output, dhe keshtu me rradh.

Editoret e ekranit dhe shume programe të tjere të sofistikuar kane nevoje të jene të gatshem për të update-uar skenen ne shume menyra komplekse duke zevendesuar një linjë ne mes të ekranit. Për të bere ketë, shume terminale kane një seri komandash për të levizur kursorin, për të futur dhe për të fshire karakteret, etj. Keto komanda zakonisht quhen **escape sequence**. Ne kohet e zhvillimit të terminaleve RS-232, kishte shume tipe terminalesh secili me sekuençen escape të tij. Për rrjedhoje, ishte e veshtire për të shkruajtur software qe të punonin lehtësisht ne keto terminale.

Një zgjidhje e cila u prezantua ne Berkeley UNIX, ishte një terminal data base i quajtur **termcap**. Kjo paket software përbante një numer të aksioneve baze, si për shembull levizjen e kurzorit ne (rradhe, rreshta). Për të levizur kurzorin ne një pozicion cfare, një editor do të përdor një escape sequence e cila me pas konvertohej ne escape sequence aktuale për terminalin ne të cilin do të shkruhej. Ne ketë menyre editori punonte ne çdo terminal.

Industria pa se ishte e nevojshme standartizimi i escape sequunce, keshtu qe u zhvillua një standart ANSI. Disa nga keto vlera jane të treguara ne Fig. 5-37.

Konsideroni se si kjo escape sequence mund të përdoret nga një editor texti. Supozoni se përdoruesi përdor një komand ku I thotë editorit të fshij të gjithe linjën 3 dhe të mbylli hapesireni midis linjës 2 dhe 4. Editori do ti dergonte ketë escape sequence nepërmjet linjës seriale terminalit:

ESC [3;1 H ESC [0 K ESC [1 M

Kjo komand leviz kurzorin ne filim të linjës 3, fshin të gjithe linjën dhe me pas fshin linjën boshe qe sa po u krijua, duke sjell qe të gjitha të nisin nga linja 5, dhe cfare ishte linja 4 kthehet ne linjën 3, cfare ishte linja 5 behet linja 4 dhe keshtu me rradh. Escape sequence mund të përdoren për të shtuar texte ne mes të ekranit. Fjalet mund të shtohen ose mund të hiqen ne menyra të ngjashme.

Escape sequence	Meaning
ESC [n A	Move up n lines
ESC [n B	Move down n lines
SSC [n C	Move right n spaces
ESC [n D	Move left n spaces
ESC [m ; n H	Move cursor to (m, n)
ESC [s J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [s K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [n L	Insert n lines at cursor
ESC [n M	Delete n lines at cursor
ESC [n P	Delete n chars at cursor
ESC [n @	Insert n chars at cursor
ESC [n m	Enable rendition n (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Figure 5-37. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and n , m , and s are optional numeric parameters.

5.7 Nderfaqja Grafike e Përdoruesit

PC-të mund të përdorin nderfaqe të bazuara mbi karakteret (germat). Ne fakt, prej vitesh MS-DOS-i, i cili është i bazuar mbi karakteret, ka dominuar skenen. Ne ditët e sotme shume PC përdorin **GUI – Graphical User Interface**.

GUI u shpik nga Douglas Engelbart dhe nga grupi i tij kerkimor ne Institutin e Kerkimeve të Stanford. Ne atë kohe u kopjua dhe nga kerkuesit ne Xerox PARC. Një ditë, Steve Jobs, bashke-themeluesi i APPLE, po shetiste ne PARC dhe pa një GUI ne një Computer Xerox. Kjo i dha atij idene për një kompjuter të ri, i cili u be me vone Apple Lisa. Lisa ishte shume i shtrenjtë dhe një deshtim komercial, por pasardhesi i tij Macintosh-i, ishte një sukses gjigand. Macintosh ishte inspirimi për Microsoft Windows dhe sistemet e tjera qe bazoheshin të GUI (nderfaqja grafike e përdoruesit).

GUI – ka 4 elementë esenciale, të paraqitura nga karakteret WIMP (Windows, Icon, Menu, Pointing device). Windows (dritaret), Jane blloqe drejtkendore të ekranit qe përdoren për ekzekutimin e programit. Ikonat Jane simbole të vogla, mbi të cilat mund të klikohet duke shkaktuar një veprim. Menu-të Jane lista me veprime nga të cilat mund të zgjidhet njëra nga to. Pointing Device (pajisja shenjuese), është mouse, trackball (sfera e gjurmave), ose pajisjet e tjera hardware qe përdoren për të zhvendosur krusorin nepër ekran, për të selektuar items.

5.7.1 Tastiera e Personal Computer, Mouse, dhe Display Hardware

PC moderne sot kane gjithmone një tastiere dhe një bit-oriented memory mapped display. Keto komponentë jane një pjese integrale e vet kompjuterit. Tastiera dhe ekrani jane plotësisht të ndara, ku secila ka driverin e saj vetjak.

Tastiera mund të nderfaqesohet nepërmjet një porte seriale, një portë paralele, ose një portë USB. Ne çdo veprim kyc qe kryet, CPU-ja nderpritet (interrupted) dhe driveri i tastieres extracton karakterin e shtypur duke lexuar një portë I/O. Çdo gje tjetër ndodh ne software, kryesisht ne driverin e tastieres.

Ne një pentium, tastiera përbën një mikroproçesor të futur (embedded) i cili komunikon nepërmjet një porte seriale të specializuar me një controller chip qe ndodhet ne parentboard. Një interrupt gjenerohet kur shtypet një tast, dhe gjithashtu liriohet ky tast. Vec kesaj, ajo cfare hardware i tastieres ofron është numri i tastit (key number), dhe jo kodi ASCII. Kur shtypet A-ja, kodi i tastit (30) vendoset ne një regjistër I/O. Është ne doren e driverit të përcaktohet ne se: është kapitale (e madhe), (e vogel), CTRL-A, ALT-A, CRTL-ALT-A, apo ndonjë kombinim i tille. Meqene se driveri mund të tregojë se cili tast është shtypur, por jo cili është leshuar (shembulli Shift), ai ka informacion të mjaftueshem për të kryer detyren.

Për shembull, Sekuena e tasteve: DEPRESS SHIFT, DEPRESS A, RELEASE A, RELEASE SHIFT

Tregon një germe kapitale A.

Gjithashtu DEPRESS SHIFT, DEPRESS A, RELEASE SHIFT, RELEASE A

Tregon germe kapitale A.

Edhe pse kjo nderfaqe e tastieres vendos gjithe ngarkesen tek software, ajo është shume flexible. Për shembull, programet e përdoruesit mund të kene interes të dine ne se një shifer është shtypur nga rreshti i sipërm i tastieres apo nga numer pad ne të djathë të tastieres. Ne princip driveri mund ta ofroje ketë informacion.

Shume PC kane një mouse, ose shpesh here një trackball, i cili është një mouse i shtrire me kurriz. Lloji me i zakonshem i mouse-it ka një sfere gome brenda tij, qe del përparrë nepërmjet një vrime ne fund dhe rrrotullohet ndersa mouse-i leviz mbi një sipërfaqe të ashpër. Ndersa sfera rrrotullohet, ajo ferkohet me rollerat e ferkimit qe jane vendosur ne boshtet ortogonale. Levizja lindje-përendim shkakton boshtin paralel me aksin Y të rrrotullohet, levizja veri-jug shkakton boshtin paralel me aksin X të rrrotullohet. Çdo zhvendojs e mouse-it ose shtypje, ose lirmi i butonave sa do minimale të jetë, bën qe kompjuterit ti cohet gjithmone një mesazh. Distanca minimale e levizjes është 0.1 mm (ajo mund të përcaktohet tek software). Disa njerez e quajne ketë njësi - **mickey**. Mouse-t mund të kene 1, 2 ose 3 butona ne varesi të vleresimit qe bejne disenjatoret ne se përdoruesit Jane të aftë të mbajne dot gjurmet e 1 apo me shume butonave.

Mesazhi qe shkon të kompjuteri ka 3 items, ΔX , ΔY , butonat. Item-i i pare është ndryshimi i pozicionit X qe nga mesazhi i fundit. Me pas, vjen ndryshimi i Y qe nga mesazhi i fundit. Se fundmi përfshihet statusi/gjendja e butonit. Formati i mesazhit varet nga sistemi dhe nga numri i butonave qe ka mouse-i. Zakonisht, ai merr 3 byte.

Vini re qe mouse-i tregon vetëm ndryshimin e pozicionit dhe jo pozicionin absolut të tij. Ne qoftë se mouse-i ngritet sipër dhe ulet poshtë me kujdes pa e levizur sferen, ne ketë rast kompjuterit nuk do ti cohet mesazh.

Disa GUI dallojne single click nga dopjo click të një mouse-i. Ne qoftëse dy click Jane ‘shume afer’ ne hapesire (mickeys) dhe ne kohe (milisekonda), atëherë sinjalizohet një dopjo click. Maksimumi i ‘shume afer’ është ne varesi të software-it, me të dyja parametrat qe mund ti caktoje vetë përdoruesi.

Le të kthehmi të Display Hardware; Pajisjet e shfaqjes ne ekran (Display Devices), mund të ndahen ne dy kategori. Pajisjet me **Vector Graphics (Grafike Vektoriale)** të cilat mund të pranojnë dhe të nxjerrin jashtë comanda si për shembull, vizatimi i pikave, i vijave, i figurave gjeometrike dhe i teksteve. Ne kontrast, pajisjet me **Raster Graphics** e tregojnë zonen e daljes, si një rrjetë me pika të quajtura **pixels**, secili nga to ka një vlere të shkalles gri (gray scale value), ose një ngjyre. Ne fillimet e kompjuterave, pajisjet me vector graphics ishin të zakonshme, por tanj ploterat Jane të vetmet pajisje me vector graphics. Çdo gje tjeter përdor raster graphics, shpesh e quajtur **bitmap graphics**.

Display me raster graphics Jane të implementuara nga një pajisje e quajtur **graphic adapter**. Një graphic adapter përmban një memorie speciale të quajtur **video RAM**, e cila formon pjesen e hapesires se adresave të kompjuterit dhe adresohet nga CPU-ja ne të njëjtën menyre si pjesa tjeter e memories (shih Fig. 5-38).

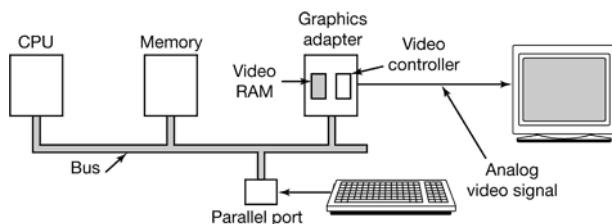


Figure 5-38 Ne display me memomy mapped, driver shkruan ne menyre direktë tek video RAMi i displayt

Pamja e ekranit ruhet ketu, ose ne menyren karakter (**character mode**) ose ne menyren bit (**bit mode**). Ne menyren karakter (**character mode**), çdo byte (ose 2 byte) i video RAM-it përmban një karakter për të shfaqur ne ekran. Ne menyre bitmap (**bitmap mode**), çdo pixel ne ekran riprezantohet me vetë ne video RAM-in, me 1bit/pixel për llojin bardh e zi të ekranit, dhe 24bit/pixel ose me shume për ekrane me cilesi ngjyrash me të lartë. Gjithashtu **video controller** është një chip qe bën pjesë ne graphic adapter. Ky chip nxjerr karakteret ose bitet jashtë video RAM dhe gjeneron një video sinjal i cili përdoret për të drejtar (drive) monitorin. Një monitor gjeneron një tufe elektronesh e cila skanon horizontalisht ekrinin, duke lyer (paint) me linja (vija) ne të. Zakonisht një ekran ka një 480 – 1024 linja (vija) nga sipër deri poshtë, dhe me 640 – 1200 pixel për linjë. Sinjali i Video Controllerit modulon tufen e elektroneve, duke përcaktuar ne se një pixel i dhene është i ndritshem apo i erret. Monitoret me ngjyra kane 3 tufa (beam), për të kuqen (Red), për të gjelbertën (Green), për blune(Blu); (RGB), të cilat modulohen ne menyre të pavarur. Ekranet e sheshtë (Flat) përdorin pixelat ne 3 ngjyra, por menyra se si punojne keto ekrane shkon përtëj qellimit të ketij libri.

Video Controllerat kane 2 menyra: Character Mode (qe përdoret për tekset e zakonshme), dhe Bit Mode (qe përdoret për çdo gje tjeter). Ne character mode, kontrolleri mund të përshtas çdo karakter ne një kuti prej 9 pixel të gjere dhe 14 pixel të lartë (duke përfshire dhe hapesirat midis karaktereve) dhe të ketë 25 linja me 80 karaktere. Ekranit i duhet

atëhere të ketë 350 linja skanimi me 720 pixel për secilen linjë. Secila nga keto frame rizivatohet 60 – 100 here ne sekond, ne menyre qe të shmangen dridhjet.

Për të shfaqur textin ne ekran, video controlleri mund të tërheqi/kapi (fetch) 80 karakteret e para nga video RAM-i, dhe të gjeneroje 14 linjat vijuese dhe keshtu vazhdon. Ne menyre alternative, ai mund të tërheqi/kapi çdo karakter, një here për çdo linjë skanimi për të eleminuar nevojen e të paturit një buffer ne controller. 9 për 14 bit pattern për karakteret mbahet ne ROM, të cilin e përdor video controlleri. (RAM mund të përdoret gjithashtu për support/ndihmuar fontet). ROM-i adresohet me 12 bit adresa, 8 bits nga kodi i karakterit dhe 4 bits për të specifikuar linjën e skanimit. 8 bitet ne çdo byte të ROM kontrollojne 8 pixels; pixeli i 9-të qe ndodhet midis karaktereve është gjithmone blank (bosh). Keshtu $14 \times 80 = 1120$ referencia memorjeje për video RAM nevojiten për linjën e tekshit ne një ekran. I njëjtë numer referencash behen për gjeneratorin e karaktereve ROM.

Ne Fig. 5-39(a) ne shikojme një porcion (pjese) të video RAM-it për një ekran i cili punon ne character mode. Çdo karakter ne ekranin e Fig. 5-39 (b) ze 2 bytes ne RAM. Karakteri i rendit të poshtëm (low order) është kodi ASCII për shfaqjen e atij karakteri. Karakteri i rendit të lartë (high order) është një byte atributi, i cili përdoret për të specifikuar ngjyrën, kthimin mbrapa të videos, blinking, etj... Një ekran me (25 me 80) karaktere i nevojitet 4000 bytes të video RAM ne ketë mode.

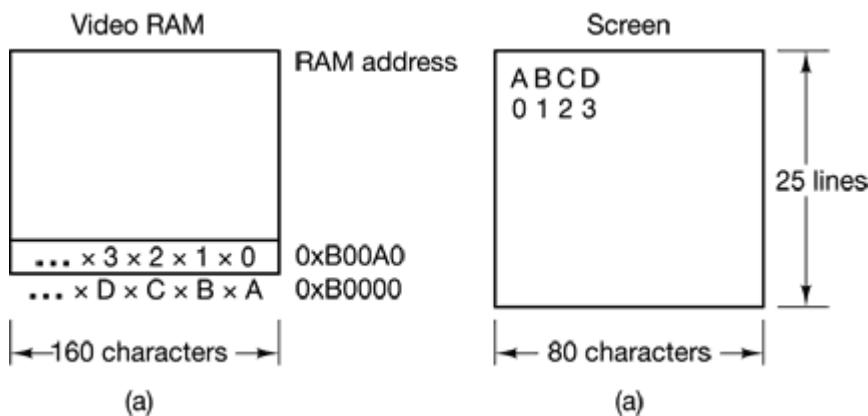


Figura 5-39. (a) Një imazh i video RAM-it për një ekran monokrom të thjeshtë ne character mode. (b) Ekrani korrespondues. ×-et jane bytet e attributeve.

Punimi ne bitmap mode përdor të njëjtin princip, përvèc se çdo pixel ne ekran kontrollohet individualisht dhe paraqitet individualisht nepërmjet 1 ose disa bits ne video RAM. Ne konfigurimin me të thjeshtë për një monitor monokrom, çdo pixel i ekranit ka një bit korrespondues ne video RAM. Ne ekstremin tjetër, çdo pixel i ekranit paraqitet nepërmjet një numri 24 bit ne video RAM, me 8 bit për secilen nga intensitetet Red, Green, Blue. Raprezantimi i RGB-se përdoret sepse e kuqja, e gjelberta dhe bluja janë ngjyra primare shtese, nga të cilat mund të përftohet çdo lloj ngjyre vetëm duke mbledhur intensitetet e ndryshme të ketyre ngjyrave.

Madhesite e ekianeve mund të varojne, nga 640×480 (VGA), ne 800×600 (SVGA), 1024×768 (XGA), 1280×1024 , dhe 1600×1200 . Përvec 1280×1024 , të gjithe të tjerat jane rapport 4:3, ky report përshtatet me reportin e NTSC television dhe jep pixel katrore. 1280×1024 duhej të kishte qene 1280×960 , por joshja e 1024 ishte shume e madhe për ti rezistuar, edhe pse ajo shtremberon lehtësisht pixelat dhe bën me të veshtire shkallezimet ne madhesite e tjera. Si per shembull, një ekran me ngjyra 768×1024 me 24 bits/pixel kerkon 2.25 MB RAM thjesht sa mban imazhin. Në qoftë se full screen rifreskohet (refresh) 75 here/sec, video RAM-i duhej të jetë i aftë të dergoje te dhena ne menyre të vazhdueshme ne 169 MB/sec.

Për të shmangur menaxhimin e imazheve të ekianeve kaq të medha, disa sisteme janë të afta të balancojne rezolucionin e ngjyres kundrejt madhesise se imazhit. Ne skeme me të thjeshtë, çdo pixel paraqitet nepërmjet një numri 8 bit. Kjo vlerë është një indeks ne një tabele me 256 entries, ku secili entry mban një vlerë 24 bit të R,G, B. Kjo tabele quhet **color palette** dhe shpesh ruhet ne hardware, ajo lejon qe ekranit të mbaje një numer arbitrar ngjyrash 256 ne çdo cast. Duke ndryshuar entry 7 ne paletën e ngjyrave, kjo sjell ndryshimin e ngjyres të gjithe pixelave ne imazh me një vlerë 7. Duke përdorur një paletë ngjyrash me 8 bit pakesohet sasia e hapesires se ngjyrave qe nevojitet për të ruajtur imazhin e ekranit nga 3 byte/pixel ne 1 byte/pixel. Cmimi qe paguhet është vrazhdesia ne rezolucionin e ngjyrave. Skema e kompresimit GIF punon me një palete ngjyrash të tille. Është gjithashtu e mundur të përdoret një palete ngjyrash me 16 bit/pixel. Ne ketë rast paleta e ngjyrave përmban 65,536 entries keshtu qe mund të përdoren njehersh 65,536. Por kursimi ne hapesire është me i paket meqene se çdo pixel tanë kerkon 2 bytes ne video RAM. Gjithashtu, në qoftë se paleta e ngjyrave mbahet ne hardware (për të shmangur një kerkim (lookup) ne çdo pixel), ai duhet të jetë dedikuar për ruajtjen paletes. Gjithashtu është e mundur qe të menaxhohen ngjyrat 16 biteshe, duke ruajtur vlerat e RGB si tre numra 5 biteshe, me 1 bit të lene pas (mbrapa) (ose ti jepet të gjelbres 6 bits, meqë syri është me i ndjeshem ndaj jeshiles sesa të kuqes dhe bluse). Ky sistem është i njëjtë me atë me 24 bit color, përvet se me me pak hije për çdo ngjyre të mundshme.

5.7.2 Software i hyrjes (Input Software)

Pasi tastiera e mori karakterin, ajo duhet të filloje ta proçesoje (përpunoje) atë. Meqë tastiera dergon key numbers, sesa kodet e karaktereve qe përdoren nga aplikacionet, driveri duhet të konvertoje kodet duke përdorur tabelen. Jo të gjitha pajisjet ‘kompatibel’ e IBM-se përdorin key number standart, keshtu qe në qoftë se driveri kerkon të supportoje keto makina, ai duhet të kete map tastiera të ndryshme me tabela të ndryshme. Një arritje e thjeshtë kompilimi i një tabele e cila mapon kodet të ofruara nga tastiera dhe kodet ASCII ne driverin e keyboard, por kjo nuk kenaqë përdoruesit e gjuheve të ndryshme nga anglishtja. Tastierat janë të arranzhuara ne menyre të ndryshme ne shtete të ndryshme, dhe bashkesia e karaktereve ASCII nuk është adeguate as për pjesen me të madhe të njerezve ne hemisferen e perendimit, ku folesit e Spanjishtes, Portugalishtes dhe Frengjishtes kerkonin, karakteret me shenjat theksit qe nuk përdoren ne Anglisht. Për t'ju përgjigjur nevojes për fleksibilitet ne layout-s e tastieres ne gjuhet e ndryshme, shume sisteme operative ofrojnë një **keymap** ose **code page**, e cila ben te

mundur zgjedhjen e mapping midis keyboard codes dhe codes qe i dergohen aplikacionit, dhe ne rast kur sistemi është i boot-uar tashme ose boot-oet me vone.

5.7.3 Programet e Daljes për Dritaret (Windows)

Softwari i daljes për GUI-t është një teme shume e madhe. Shume libra 1500 faqesh jane shkruar për nderfaqet grafike të dritare vetem nga (Windows GUI). Ne ketë seksion do të shikohen disa koncepte. Për ta bere diskutimin disi me konkret, ne do të përshkruajme Win32 API, e cila suportohet nga të gjitha versionet 32 bitesh të windowsit. Software i daljes për GUI-t ka shume detaje të ndryshme.

Itemi baze i ekranit është një zone drejtkendore e quajtur **dritare (window)**. Pozicioni dhe madhesia e dritares jane të përcaktuara ne menyre unike duke dhene kordinatat (ne pixel) e qosheve të kunderta. Një dritare mund të mbaje një **title bar**, një **menu bar**, a **tool bar** dhe një **horizontal scroll bar** (shirit horizontal të levizshem). Një dritare tipike tregohet ne Fig. 5-40. Vini re qe sistemi kordinativ Windowsit, e vendos origjinin ne skajin e majtë të sipërm dhe qe shtohet me y duke zbritur poshtë, e cila është e ndryshme nga kordinatat Karteziane qe përdoren ne matematike.

Kur krijohet një dritare, parametrat specifikojne ne se dritarja mund të levizet apo ti ndryshohen përmasat, apo të scroll (rreshqase/leviz duke têrhequr Thumb ne scroll bar) nga useri. Dritarja kryesore qe prodhohet nga pjesa me e madhe e programeve mund të levizet, ti ndryshohet përmasat apo scroll, dhe kjo gje ka shume pasoja për menyren se si progamet e Windows-it jane shkruar.

Ne vecanti, programet duhet të jene te informuara ne lidhje me ndryshimet e përmasave të dritareve të tyre dhe duhet të jene të përgatitura të ri-vizatojne përbajtjen e dritares ne çdo cast të kohes, edhe ne momentin me pak të pritshem.

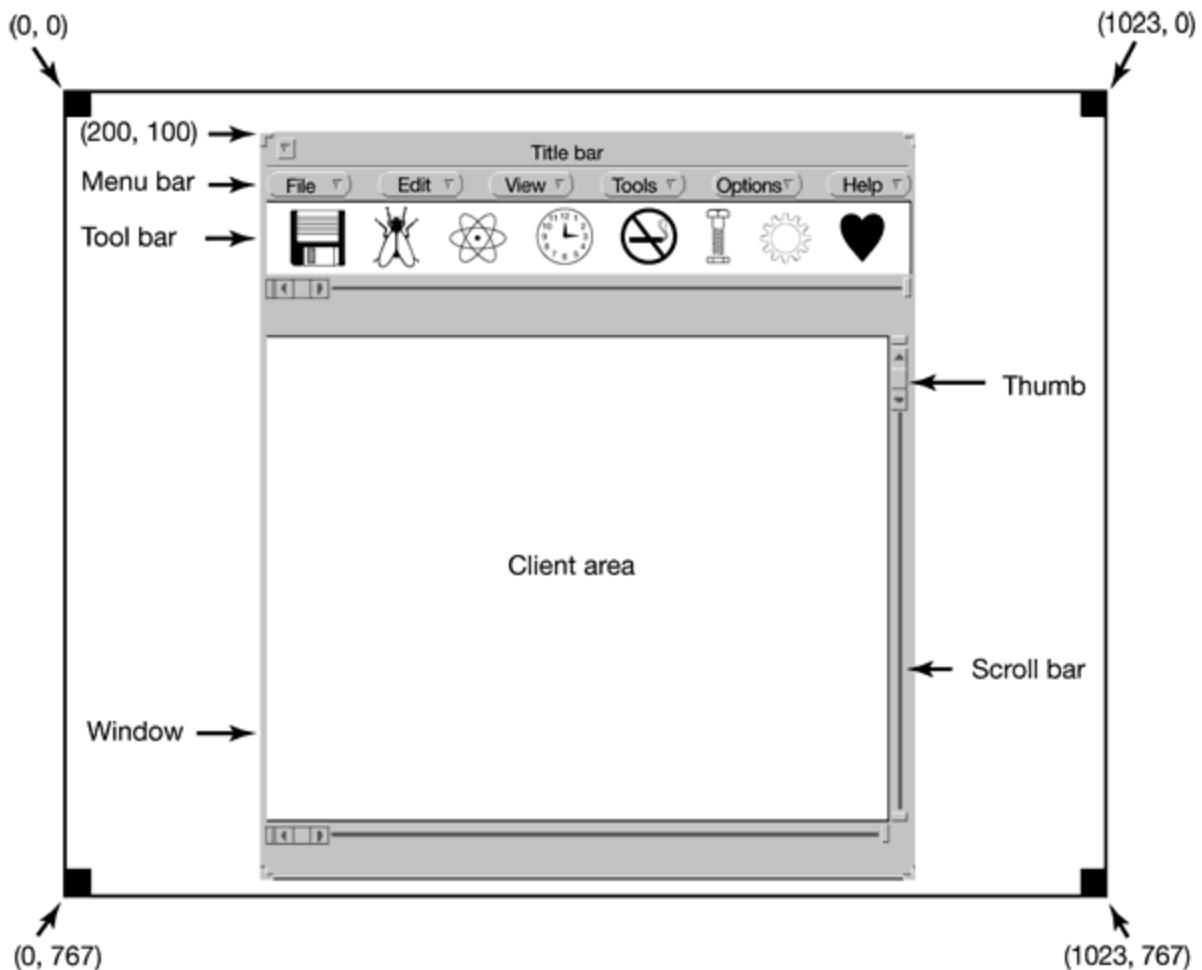


Figure 5-40. Një shembull dritarjeje (Window)

Si rrjedhoje, programet e Windows jane të orientuara drejt mesazheve (message oriented). Veprimet e përdoruesit qe përfshijnë tastierën dhe mousin, kapen nga dritaret dhe konvertohen ne mesazhe drejt programeve qe zotëron dritaren e adresuar. Çdo program ka një radhe (queue) ne të cilën ndodhen mesazhet qe i dergohen dritares. Loop-i kryesor i programit konsiston ne nxjerrjen jashtë plotësisht të mesazhit të ardhshem dhe ne procesimin e tij duke thirrur një procedure të brendeshme për atë tip mesazhi. Ne disa raste, vetë Windows mund të therrasin ne menyre direkte keto procedura, duke anashkaluar radhen e mesazheve (message queue). Ky model është shume i ndryshem nga modeli UNIX i kodit procedurial i cili bën thirrjet e sistemit për të ndervepruar me sistemin operativ.

Për ta bere me të qartë ketë model progamimi, do të shohim shembullin e Fig. 5-41. Këtu ne shikojme skeletin e një progami kryesor për Windows. Nuk është i plotë dhe nuk bën kapje gabimesh, por ai tregon mjaft detaje për qellimet tonë. Ai fillon duke përfshire një file header, *windows.h*, qe përmban disa macro, lloje datash, konstante, prototipe funksioneve dhe informacione të tjera të nevojshme për progamat Windows.

```
#include <windows.h>
```

```

int WINAPI WinMain (HINSTANCE h, HINSTANCE hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass; /* class object për ketë dritare */
    MSG msg;           /* mesazhet ardhesh ruhen ketu */
    HWND hwnd;         /* handle (pointer) to the window object */

    /* Initialize wndclass */
    wndclass.lpfnWndProc = WndProc; /* tregon cila procedure të therritet */
    wndclass.lpszClassName = "Program name"; /* Tekst për title bar */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* ngarko ikonen e programit */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW); /* ngarko kurSORin e mouse */

    RegisterClass(&wndclass); /* tregon Windows përrreth wndclass */
    hwnd = CreateWindow ( ... ); /* alokon storage për dritaren */
    ShowWindow(hwnd, iCmdShow); /* shfaq dritaren ne ekran */
    UpdateWindow(hwnd); /* thuaji dritares të vizatoje vetveten */

    while (GetMessage (&msg, NULL, 0, 0)) { /* merr mesazhin nga rradha queue */
        TranslateMessage (&msg); /* përkthe mesazhin */
        DispatchMessage (&msg); /* dergo mesazhin procedures se duhur */
    }
    return(msg.wParam);
}

long CALLBACK WndProc (HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* Deklarimet vihen ketu. */

    switch (message) {
        case WM_CREATE: ...; return ...; /* krijo dritaren */
        case WM_PAINT: ...; return ...; /* ri-vizato përbajtjen e dritares */
        case WM_DESTROY: ...; return ...; /* shkaterro dritaren */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam)); /* default */
}

```

Figure 5-41. Skeleti i program kryesor Windows.

Programi kryesor fillon me një deklarim duke dhene emrin e tij dhe parametrat. WINAPI macro është një instruksion për kompilatorin qe të përdori një parametër të caktuar duke kaluar convention dhe nuk do të na shqetësoje me. Parametri i pare, *h*, është një instance handle dhe përdoret për ta identifikuar programin me pjesen tjeter të sistemit. Win32 është një object oriented, qe do të thotë qe sistemi përmban objekte (për shembull

programet, files dhe dritaret) të cilat kane një gjendje dhe një kod të asiocuar të quajtur **metoda** të cilat punojne ne atë gjendje. Objektet referohen duke përdorur handles dhe ne ketë rast, *h*-ja identifikon programin. Parametri i dytë shfaqet vetëm për arsyet e backward compatibility. Por nuk përdoret me tani. Parametri i tretë, *szCmd* është një string qe mbaron me zero e cila përmban command line qe nisi programin, edhe në qoftë se nuk niset nga command line. Parametri i katërt, *iCmdShow*, tregon se cfare pjese të ekranit do të okupoje (zeri) dritarja, të gjithe ekranin apo një pjese të tij apo asnjë pjese të tij (vetëm taskbar).

Ky deklarim ilustron një convention qe ka përdorur Microsoft-i, të quajtur Hungarian Notation. Emri është një loje fjalesh ne Polish Notation, sistemi postfix u shpik për riprezantimin e formulave algjebrike duke mos përdorur përparesine ose kllapat. Hungarian notation u shpik nga programuesi Hungarez i Microsoftit, Charles Simonyi, dhe përdorte disa nga karakteret e para të një identifikatori për të specifikuar tipin. Germat e lejuara dhe tipet përfshinin c (character), w (word, tanë ka kuptimin e një integer 16 bit pa shenjë), i (integer 32 bit me shenjë), s (string), sz (string qe përfundon ne fund me byte zero), p (pointer), fn (function), dhe h (handle). Keshtu *szCmd* është një string qe mbaron me zero dhe *iCmdShow* është një integer. Shume programues besojne qe enkodimi i tipit te emrit të variablit ne ketë menyre nuk ka shume vlore dhe bën kodin e Windows të vesh tire për tu lexuar. Kjo gje nuk shfaqet ne UNIX.

Çdo dritare duhet të ketë të asociuar një class object e cila përcakton vetit e saj. Ne Fig. 5-41, ajo class object është *wndclass*. Një objekt i tipit WNDCLASS ka 10 fusha, 4 nga të cilat jane të inicializuar ne Fig. 5-41. Ne një program aktual, 6 fushat e tjera duhen të inicializohet me se miri. Fusha me e rendesishme esthe *lfnWndProc*, e cila është një pointer i gjatë (long 32 bit) i funksionit qe menaxhon (handles) mesazhet qe i drejtohen kesaj dritareje. Fushat e tjera qe inicializohen ketu tregojne cili emer dhe cila ikon do përdoret ne title bar, dhe cili simbol do përdoret për kurzorin e mousit.

Mbasi inicializohet *wndclass*, therriet *RegisterClass* për t'ju kaluar Windows. Ne vecanti pas kesaj thirrjeje, Windows e di cilen procedure të therrase kur ndodhin ngjarje të ndryshme qe nuk shkojne tek radha e mesazheve. Thirrja tjeter, *CreateWindow*, alokon memorjen për strukturen e të dhenave të dritares dhe kthen një handle për ta referencuar atë me vone. Programi bën 2 thirrje të tjera rresht, për ta vedosur konturin e dritares ne ekran, dhe se fundemi ta mbushi atë plotësisht.

Ne ketë pike ne vijme të loopi kryesor i programit, i cili konsiston ne marrjen e mesazheve, duke bere disa transferime (përkthime) të caktuara, dhe duke ia kaluar mbrapsh Windows ne menyre qe Windows të therrase *WndProc* për ta proçesar atë. Për t'ju përgjigjur pyetjes ne se mund të behej me i thjesht ky menakanizem; po mund të behej, por keshtu është bere kohe me pare dhe për shkak të historise ne kemi ngelur me ketë.

Ne vijim të programit është procedura **WndProc**, e cila menaxhon mesazhe të ndryshme të cilat mund ti dergohen dritares. Përdorimi i *CALLBACK* ketu, si përdorimi i WINAPI sipër, specifikon sekuecen thirrese për të përdorur parametrat. Parametri i pare është menaxhimi (handle) i dritares qe përdoret.

Parametri i dytë është tipi i mesazhit. Parametri i tretë dhe i katërt mund të përdoren për të ofruar informacione shtese ne raste nevoje.

Tipet e mesazheve *WM_CREATE* dhe *WM_DESTROY* dergohen ne fillim dhe ne fund të programit, respektivisht. Ato i jepin programit mundesine, për shembull i alokimit të memories për strukturat e të dhenave dhe të kthimit të tyre.

Tipi i tretë i mesazhit, *WM_PAINT*, është një instrukcion për programin, për të plotësuar (mbushur) dritaren. Ai nuk thirret vetëm kur dritarja është ne vizatimin e pare, por shpesht edhe gjatë ekzekutimit të programit. Ne dallim me sistemet e bazuar mbi tekstet, ne Windows një program nuk mund pretendohet qe ajo cfare vizatohet ne ekran, të qendroje aty derisa dikush ta heqi atë. Dritaret e tjera mund të tërhiqen, menu-të mund të hapen, tooltips mund të shfaqen. Windows i tregon një programi menyren si ai të ri-vizatoje një dritaren, duke derguar një mesazh *WM_PANIT*. Gjithashtu ofrohet informacion se cila pjese e dritares është mbishkruar, ne rast se është me lehtë për të ri-gjeneruar atë pjese të dritares ne vend qe të ri-vizatohet e gjitha.

Ka dy menyra qe Windowsi mund ti thotë një programi ne menyre qe ai të kryeje dicka. Menyra e pare është të postoje një mesazh ne radhen e tij të mesazheve (message queue). Kjo metode përdoret për inputet e tastieres, për inputet e mouse dhe timerat qe kane skaduar. Menyra tjetër, është dergimi i një mesazhi dritares, duke involvuar Windowsin të therrasi direkt *WndProc*. Kjo metode përdoret për të gjitha ngjarjet e tjera. Meqe Windows lajmerohet kur një mesazh është plotësisht i procesuar, ai mund të shmangi një thirrje të re derisa thirrja e meparshme të mbaroje. Ne ketë menyre kushtet e gares menjanojen.

Ka edhe tipe të tjera mesazhesh. Për të menjanuar sjellje të gabuara duhet qe të vije një mesazh i papritur, gjeja me e mire është të therrijet *DefWindowProc* ne fund të *WndProc* për të lene handlerin të kujdeset për rastet e tjera.

Përbledhurazi, një program i Windows zakonisht krijon një ose disa dritare me një class object për seclin nga to. Me çdo program është asiocuar një radhe mesazhesh (mes queue) dhe një bashkesi mes procedurash menaxhimi (handler). Se fundmi, sjellja e programit drejtohet nga eventet qe vijnë, të cilët procesohen nga procedurat e menaxhimit (handler). Ky është një model shume i ndryshem i botës sesa pamja proceduriale qe merr UNIX.

Vizatimi i tanishem i ekranit menaxhohet nga një paketë me qindra procedura të cilat janë lidhur se bashku për të formuar **GDI (Graphics Device Interface)**. Ajo mund të menaxhoje tekstu dhe çdo lloj grafike dhe është e disenjuar qe të jetë e pavatur nga platforma dhe device. Përpara se një program të vizatoje brenda një dritare, ai ka nevoje të siguroje një **device Context** e cila është një struktura të dhenash te brendeshme qe përmban vetitë e dritares, si fonti i tanishem, ngjyra e tekstit, ngjyra e background, etj. Shume thirrje të GDI-s përdorin device context edhe për vizatim, ose përmarrjen/vendojsen e vetive.

Menyra të ndryshme ekzistojne për të siguruar ekzistencën e një device context. Një shembull i thjeshtë i sigurimit dhe i përdorimit është:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

Statement i pare merr një handle për një device content, *hdc*. E dyta përdor device context për të shkruar një rresht texti ne ekran, duke specifikuar kordinatat (*x,y*) ku fillon stringa, një pointer për vetë stringen dhe gjatësinë e saj. Thirrja e tretë liron device

context për të indikuar qe programi po vizaton ne ato momente. Vini re qe *hdc* është përdorur ne një menyre analoge me përshkruesin e file-ve të UNIX. Gjithashtu vini re qe *ReleaseDC* përmban informacion të tepërt (përdorimi i *hdc* ne menyre unike specifikon një dritare). Përdorimi i informacionit të tepërt qe nuk ka vlore akutale është e zakonshme ne Windows.

Një tjetër gje interesante është qe kur *hdc*-ja sigurohet ne ketë menyre, programi mund të shkruaje vetëm ne zonen e dritares të klientit dhe jo ne title bar, apo ne pjeset e tjera. Ne brendesi, ne strukturen e të dhenave të device context, mbahet një zone e levizhme. Çdo vizatim jashtë kesaj zone injorohet. Ka një tjetër menyre për sigurimin e device context, *GetWindowDC*, e cila vendos zonen e levizhme ne të gjithe dritaren. Thirrjet e tjera kufizojne zonen e levizeshme ne menyra të ndryshme. Një karakteristikë tjetër e Windows është patja e thirrjeve të shumfishta të cilat bejne të njëjtën gje.

Një trajtim i plotë i GDI është jashtë vendit ketu. Për lexuesit e interesuar, referencat e cituar me sipër ofrojne informacion shtese. Megjithatë, disa fjale përreth GDI ia vlen qe ti përmendim për të treguar rendesine e tij. GDI ka procedura thirrjeje të shumta për të marre dhe liruar device context, për të përfthuar informacion përreth device context, për marrjen dhe vendosjen e atributeve të device context (për shembull ngjyra e background), manipulimi i objekteve GDI, si për shembull penat, fontet, furcat, ku secila ka atributet e saj. Se fundmi, sigurisht ka një numer të madh thirrjesh GDI për të vizatuar ne ekran.

Procedurat e vizatimit bien ne 4 kategori: vizatimi i vizave dhe vijave të lakuara, vizatimi i zonave të mbushura, menazhimi i bitmaps, shfaqja e tekstit. Ne pame një shembull të vizatimit të tekstit me sipër, keshtu le te hedhim një shikim të shpejtë.

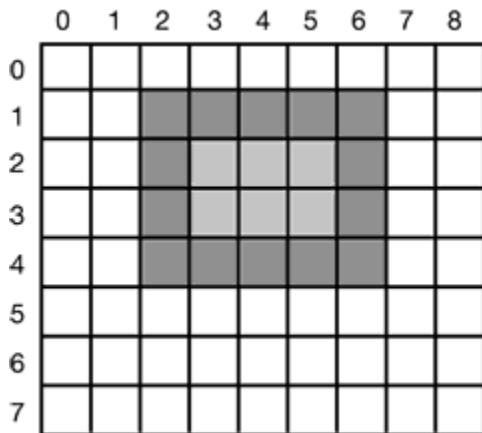
Thirrja:

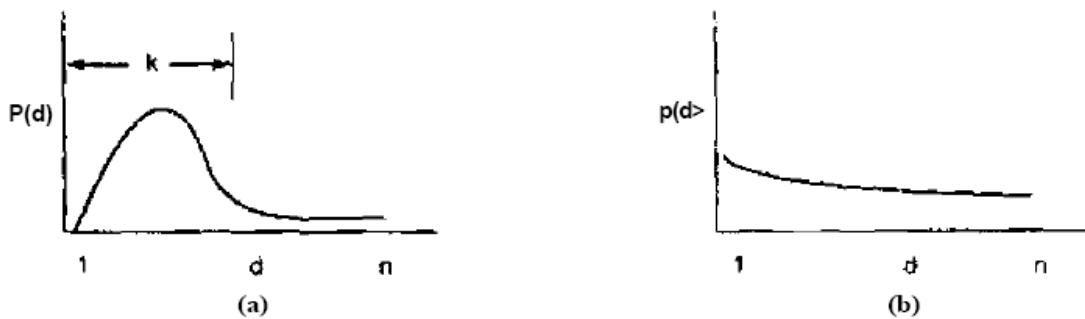
`Rectangle(hdc, xleft, ytop, xright, ybottom);`

Vizaton një drejkendesh të mbushur skajet e të cilit jane (*xleft*, *ytop*) dhe (*xright*, *ybottom*). Për shembull

`Rectangle(hdc, 2, 1, 6, 4);`

Do të vizatoje një drejkendesh të treguar ne Fig. 5-42. Trashesia e vijes, ngjyra dhe mbushja e ngjyres merren nga device context. Thirrjet e tjera të GDI janë të ngjashme ne shije.



Figure**5-42.****Figure 4-26.** Probability density functions for two hypothetical distance strings.**Bitmap**

Procedural GDI Jane shembuj të **vector graphics**. Ato përdoren për të vendosur tekste dhe figura gjeometrike ne ekran. Keto grafiqe mund të shkallezohen lehtësisht ne ekran me të medhenj ose me të vegjel (numri i pixels ne ekran është i njëjtë). Gjithashtu ato Jane relativisht të pavarura nga pajisjet.

Një koleksion thirrjesht të procedurave GDI-se mund të asemblohet ne një file të vetëm qe mund të përshkruaje një vizatim kompleks. Një file i tille është quajtur Windows **metafile**, dhe është gjeresisht i përdorur për të transmetuar vizatime nga një program i Windows ne një program tjeter. Keto lloj file kane prapashtesen **.wmf**.

Shume programe të Windows lejojnë përdoruesin të kopjoje një pjese të një pikture dhe ta vendosi atë ne clipboard e Windows-it. Përdoruesi mund të shkoje ne një program tjeter dhe të ngjisi përbajtjen e clipboardit ne një document tjeter. Një menyre për ta bere ketë është qe programi i pare ta paraqesi pikturen si një windows metafile dhe ta vendosi atë ne clipboard ne formatin **.wmf**. Gjithashtu ekzistojne metoda të tjera.

Jo të gjitha imazhet qe kompjuterat manipulojnë mund të gjenerohen duke përdorur vector graphics. Për shembull fotot dhe videot nuk i përdorin vector graphics (grafiken vektoriale). Keto items skanohen duke mbivendosur një rrjetë mbi imazhin. Mesatarja e vlerave të ngjyrave të kuqe, jeshile dhe blu ne çdo kuadrat të kesaj rrjete kampionohen dhe ruhen si vlere e një pixeli. Një file i tille quhet bitmap. Windows ofron lehtësira të shumta për ti manipuluar bitmaps.

Një tjeter përdorim i bitmap është për tekste. Një menyre për të paraqitur një karakter, një stil font është si një bitmap i vogel. Shtimi i tekstit ne ekran behet një ceshtje thjesht levizjeje të bitmaps.

Një menyre e përgjithshme e përdorimit të bitmap është nepërmjet një procedure të quajtur **bitblt**. Ajo therrihet si me poshtë:

Bitblt (dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);

Ne formen e tij me të thjeshtë, ajo kopjon një bitmap nga një drejtkendesh ne një dritare, të një drejtkendeshi ne një dritare tjeter (ose ne të njëjtën). Tre parametrat e pare specifikojnë dritaren e destinacionit dhe pozicionin. Me pas vijne gjeresia dhe gjatësia. Me pas është dritarja burim dhe pozicioni. Vereni qe çdo dritare ka sistemin e vet të koordinatave, me (0, 0) ne skajin e sipërm-majtas të dritares. Parametri i fundit do të përshkruhet si me poshtë:

Efekti i BitBlt (hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);

Është treguar ne Fig. 5-43. Vini re me kujdes qe zona e plotë 5×7 e shkronjës A është kopjuar duke përfshire dhe ngjyren e sfondit.

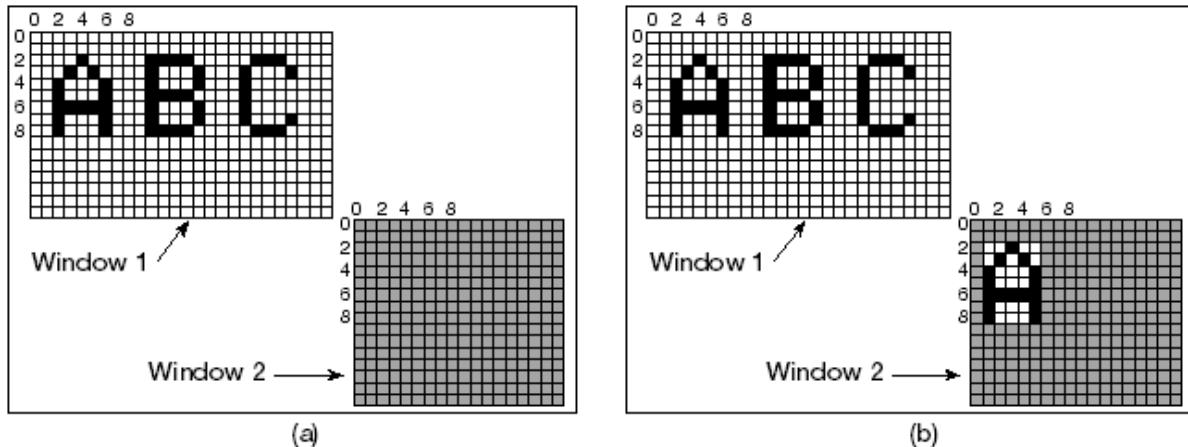


Figura 5-43. Kopjimi i bitmaps duke përdorur BitBlt. (a) Me Pare. (b) Me pas.

BitBl mund të beje me shume sesa thjesht kopjimin e bitmaps. Parametri i fundit jep mundesine per të kryer veprime Boolean, për të kombinuar bitmapin burim me atë destinacion. Për shembull, burimi mund të jetë Ored, për tu shkrire me pas ne destinacionin. Gjithashtu mund të jetë EKSKLUZIVE ORed ne të, e cila ruan karakteristikat si të burimit ashtu dhe të destinacionit.

Një nder problemet e bitmap është qe ato nuk mund të shkallezohen. Një karakter qe ndodhet ne një kuti 8×12 , ne një display me 640×480 do të dukej e arsyeshme. Megjithatë, ne qoftë se ky bitmap do të kopjohet ne një faqe të printuar me 1200 dots/inch, e cila është $10200 \text{ bit} \times 13200 \text{ bit}$, gjeresia e karakterit (8 pixel) do të jetë $8/1200 \text{ inch}$ ose 0.17 mm e gjere. Vec kesaj, kopjimi midis pajisjeve me veti të ndryshme ngjyrash ose midis monochrome dhe ngjyrave nuk funksionon mire.

Për ketë arsyen, Windows mbështet informacionin ne strukturen e të dhenave te ashtuquajtur **DIB (Device Independent Bitmap)**. File-at qe përdorin ketë format përdorin prapashtesen *.bmp*. Keto file-s kane file header dhe information headers dhe një tabele ngjyrash përpapara pixelave. Ky informacion e bën me të lehtë levizjen e bitmap midis pajisjeve të ndryshme.

Fontet

Ne versionet paraardhese të Windows 3.1, karakteret ishin të paraqitur si bitmaps dhe kopjoheshin ne ekran ose printer duke përdorur *BitBlt*. Problemi ne ketë menyre ishte, sic e pame dhe pak me lart, bitmapi qe ka kuptim për ekranin është shume i vogel për printerin. Gjithashtu një bitmap i ndryshem nevojitet për çdo karakter, ne çdo madhesi të tij. Me fiale të tjera, ne qoftë se do të jepet bitmapi për A-ne me madhesi 10 point, nuk ka asnje menyre për ta llogaritur atë për madhesine 12 point. Sepse çdo karakter i çdo lloj fonti, për madhesite qe variojne nga 4point deri ne 120point, ka të nevojshme një numer të madh bitmaps. I gjithe sistemi ishte i pavolitshem për teksten.

Zgjidhja ishte futja ne përdorim i TrueType Fonts të cilat nuk jane bitmap, por thjesht konture të shkronjave. Çdo karakter TrueType është i përcaktuar nga një sekuese pikash

rreth perimetrit të tij. Të gjitha pikat Jane relative me origjinen (0, 0). Duke përdorur këtë sistem, është e thjeshtë ti shkallezosh karakteret. Ajo cfare duhet të behet është të shumezohet çdo koordinat me të njëjtin faktor shkallezimi. Ne ketë menyre një TrueType karakter mund të shkallezohet ne çdo madhesi, gjithashtu madje edhe ne madhesi thyesore. Pasi të jetë arritur madhesia e duhur, pikat mund të bashkohen duke përdorur metoden e njojur si “ndiq-pikat” e mesuar ne kopesht. Pasi konturi të jetë kompletuar karakteri mund të mbushet. Një shembull i disa karaktereve të shkallezuar ne 3 madhesi të ndryshme është treguar ne Fig. 5-44.

20 pt: abcdefgh

53 pt: abcdefgh

81 pt: abcdefgh

Figura 5-44. Disa shembuj të kontureve të karaktereve ne përmasa të ndryshme.

Pasi shkronja e mbushur të jetë e gatshme ne formen matematikore, mund të rasterizohet, qe do të thotë të konvertohet ne një bitmap ne cfare dolloj rezolucioni të deshiruar. Duke e shkallezuar si fillim, dhe me pas duke rasterizuar, ne mund të jemi të sigurt qe karakteret e paraqitura ne ekran dhe ato qe do të shfaqen ne printer do të jene aq të përaferta sa të jetë e mundur, duke ndryshuar vetëm ne gabimin e kuantizimit. Për ta përmiresuar cilesine akoma me shume, është e mundur të përfshihet (embed) hints ne secilen shkronjë duke treguar menyren se si do behet rasterizimi. Për shembull, të dyja (serifs) shkallezimet ne krye të shkronjës T duhet të jene identike, dicka qe mund të mos jetë e vertetë si pasoje e gabimit gjatë rrumbullakimit (round off).

5.8 TERMINALET E RRJETIT

Terminalet e rrjetit perdoren për të lidhur një përdorues kompjuteri ne distance ne nje rrjet, si ne LAN ashtu dhe ne WAN. Jane dy filozofi të ndryshme sesi terminalet e rrjetit duhet të punojne. Ne njëren ane, terminali duhet të ketë një fuqi te madhe llogaritjeje dhe memorje me qellim qe të kemi protokolle komplekse për të kompresuar sasi të medha të dhenash të derguara ne rrjet, (një protokoll është një marreveshje per nje bashkesi

kerkesash dhe përgjigjesh, qe një dergues dhe marres te mund te komunikojne ne rrjet ose nderfaqe të tjera.). Ne anen tjetër terminali duhet të jetë jashtëzakonisht i thjeshtë dhe ne menyre qe ta bej ate shume të lire. Ne vazhdimin e dy seksioneve ne do të diskutojmënga një shembull per secilen filozofin. Se pari ne do të meremi me sistemin window X të sofistikuar. Me pas ne do të shikojme pak terminalin SLIM.

5.8.1 Sistemi Window X

Terminalet Inteligjent të përfunduar është një terminal qe përmban një mikroproçesor aq të fuqishem sa një kompjuter kyesor, me memorje megabytes, një tastjere dhe një mouse. Një terminal i ketij tipi është terminali X, i cili vepron ne sistemin window X, (shpesh e quajme me një fjale X). Zhvilluar ne M.I.T, si pjese e projektit Athine. Një terminal X është një kompjuter qe egzekuton software X dhe i cili bashkevepron me programet duke funksionuar ne një kompjuter të largët.

Programi brenda terminalit X qe mbledh të dhena nga tastjera ose mouse-i, dhe pranon komandat nga një kompjuter i largët është quajtur server X. Ai duhet të mbaj trajektorën ne të cilën window-si është aktualisht selektuar (ku shenjuesi i mouse-it është), keshtu qe ai njeh cili klient dergon ndonjë fjale të re ne të. Ai komunikon ne rrjet me klient X duke funksionuar ne disa host-e të largëta. Ai dergon atyre te dhena nepermjet mausit dhe tastjeres dhe pranon shfaqen e komandave nga ata.

Mund të duket pa kuptim të kesh një server X brenda një terminali dhe klient ne host-in ne largesi, por puna e serverit X është të shfaqi bitet, keshtu qe ai ka kuptim të jetë afer përdoruesit nga pikepamja e programit, sepse eshte klienti qe i thote serverit cfare te bej, si per shembull shfaqen e tekstit dhe figurave gjeometrike. Një marreveshje midis klientit dhe server-it është treguar ne fig.5-45.

Ai është gjithashtu i mundeshem të funksionoj ne sistemin window X ne maje te tij ose ONIX ose sisteme operative të tjera. Ne fakt, shume sisteme UNIX veprojnë ne sistemin window X si standartet e sistemit windowing, madje dhe ne kompjutera te vecante per te aksesuar një kompjuter ne largesi nepermjet internetit. Cfare ne të vertetë sistemi window X përcakton është protokolli ndermjet klientit X dhe serverit X, të treguar ne fig.5-15. Nuk ka rendesi ne se klienti dhe server-i jane ne të njëjtën makine, të ndara nga 100 metra ne LAN, ose jane mijera kilometra larg dhe të lidhura nga interneti. Protokolli dhe sistemi operative është identik ne të gjitha rastet.

X është pikerisht një sistem windowing. Ai nuk është një GUI kompletë. Të besh një GUI të kompletuar, shtresat e tjera të software Jane ne veprim ne maje të tij. Një shtrese është Xlib. e cila është një bashkesi librarish qe vepron për aksesimin dhe funksionalitetin e X. Keto procedura formojnë bazat e sistemit window X dhe cfare ne do të kryejme me poshtë, por ata Jane shume primitiv për aksesimin direkt per me te shumten e perdoruesve. Për shembull, çdo klik e mouse-it raportohet i ndare, keshtu qe duke llogaritur qe dy klikime të mouse-it realisht formojnë një klik të dyfishtë duhet të merret me Xlib.

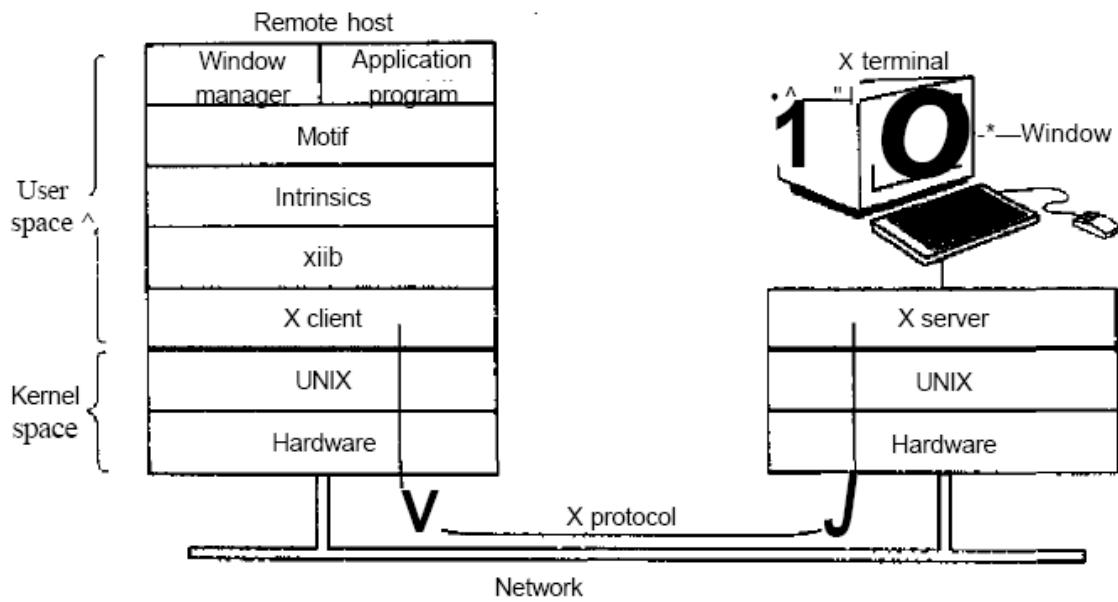


Figura 5-45.

per tab ere programimin me X me te thjeshte, një kuti të dhenash qe konsiston ne *Intrinsics* është shfaqur si pjese e X. Kjo shtrese menaxhon butonat, scroll bars dhe elementet e tjere GUI të quajtura *widgets*. Të besh një nderfaqe të vertetë GUI, me një pamje uniforme, nevojitet edhe një shtrese tjetër. Me e populluara ose me shume e përdorshme është shtresa qe quhet *Motif*. Aplikacionet e shumta përdorin Motif.

Gjithashtu duke i kushtuar vemendje qe administrimi window-s nuk është pjese e vet X. Vendimi pér evitimin e tij është totalisht i qellimshem. Ne vend të, një procesi te ndare client X, i qujatur window menager, i cili kontollon krijimin, fshirjen dhe levizjet e window ne ekran. Per menaxhimin e window-s, ai dergon komanda tek server-i X duke i treguara se cfare duhet të bej. Ai shpesh vepron ne të njëjtën makine si klient X, por ne teori mund të veproje kudo.

Ky modular i dizenuar konsiston ne disa shtresa te ndryshme dhe ne shume programe, kjo e bën X tepër te menaxhueshem dhe fleksibel. Ai ka qene drejtuar ne versionet e shumta të UNIX duke përfshire Solaris, BSD, AIX, Linux dhe të tjera duke bere të mundur qe zhvillimi i aplikimeve te kete një nderfaqe standarte pune per shume platforma. Ai ka qene gjithashtu drejtuar ne sisteme te tjera operimi. Ne kontrast, ne Windows, sistemet windowing dhe GUI jane nderthurur se bashku ne GDI dhe të lokalizuara ne kernel (berthame ne qender), e cila i bën ato me të veshtira të shfrytëzohen. Pér shembull, windows 98 GUI është akoma krejtësisht 16 bits, ku me shume se një dekade me pare procesoret Intel ishin 32 bits.

Tani le të hedhim një shikim të shkurtër ne X duke e para nga niveli Xlib. Kur një program X fillon, ai hap lidhjen tek njëri ose me shume servera X le ti quajme ato stacione pune (workstacion), madje ata mund të jene ne të njëjtën makine si programet e

veta të X. X konsideron qe kjo lidhje eshte e sigurt, i besueshem, ne kuptimin qe humbja dhe dublikimi i mesazheve jane kapur nga software i rrjetit dhe nuk duhen të shqetësohet për gabimet e komunikimit. Zakonisht TCP/IP përdoret ndermjet client dhe server.

Katër tipe mesazhesh kaojne nepermjet kesaj lidhje.

1. Nxjerrje komandash nga programi tek workstation.
2. Përgjigje nga workstacion tek pyetjet e programit.
3. Tastjera, mouse dhe ngjarje të tjera njoftohen.
4. Mesazhet EiT.

Komandat e shumta jane derguar nga programi tek workstation si një rruge kalimi e mesazheve. Nuk kemi kthim përgjigje. Shkaku për ketë është qe kur proçeset klient dhe server jane ne makina të ndryshme, mund të doje një period kohe të konsiderueshme për komandat të arrijne serverin dhe të kryhen. Duke bllokuar programin aplikativ gjatë kesaj kohe mund ta uli ndjeshem atë pa qene e nevojshme. Nga ana tjetër kur programi kerkon informacion nga worksation ai thjesht duhet të presi deri sa përgjigjja të kthehet. Si windows, X është tepër i goditur. Ngjarjet rrjedhin nga workstation tek programi, zakonisht, ne përgjigje të disa veprimeve njerezore, si shtypja e tastjeres, levizjet e mouse-it, ose një window –us duke qene i pa mbuluar. Çdo mesazh është 32 bytes, ku bytet e pare jepin tipin e ngjarjes dhe 31 bytes e tjere kane informacionin.

Koncepti i një celesi ne X është një burim. Një burim është një strukture të dhenash qe mbajne një informacion të caktuar. Programet aplikative krijojnë burime ne workstation. Burimet mund të jene ne përdorim të përbashket përgjatë proçeseve të shumta ne worksation. Burimet priren të jene të shkurtra dhe nuk mbijetojnë reboot-ve të workstation. Burimet tipike përfshire windows, fonts, ngjyra paletesh, pixmaps dhe kontekset grafike. Të fundit jane përdorur të shoqerojne vetitë ne windows dhe jane të njëjtë ne koncept të vendosin kontekstet ne windows.

Një strukture jo e plotë e një programi X është treguar ne fig 5-46. Ajo fillon duke përfshire disa kerkesa ne fillim dhe me pas disa deklarime variablesh. Ai me pas lidhet me specifikimet e server-it X si parametra të **XopenDisplay**. Me pas alokohen burimet window dhe rregjistrohen ne window. Ne praktike, disa inicializime duhet të ndodhin ketu. Pas kesaj i tregon menaxherit window qe ekziston një window i ri, keshtu qe menaxheri window mund ta menaxhoje atë.

```

#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                      /* server identifier */
    Window win;                        /* window identifier */
    GCgc;                             /* graphic context identifier */
    XEvent event;                     /* storage for one event */
    int running = 1;

    disp = XOpenDisplay("display ..name"); /* connect to the X server */
    win = XCreateSimpleWindow(disp, ...); /* allocate memory for new window */
    XSetStandardProperties(disp,...);   /* announces window to window mgr */
    gc = XCreateGC(disp, win, 0, 0);   /* create graphic context */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);           /* display window; send Expose event */

    while (running) {
        XNextEvent(disp, &event);    /* get next event */
        switch (event.type) {
            case Expose: ...; break; /* repaint window */
            case ButtonPress: ...; break; /* process mouse click */
            case Keypress: ...; break; /* process keyboard input */
        }
    }

    XFreeGC(disp, gc);               /* release graphic context */
    XDestroyWindow(disp, win);       /* deallocate window's memory space */
    XCloseDisplay(disp);            /* tear down network connection */
}

```

Figure 5-46

Thirrja e **Xcreate GC** krijon një kontekst grafik ne të cilin vetit e window Jane rregjistruar. Ne një program me të plotë ato mund të incializohen ketu. Hapi tjetër, thirrja e **XselectInputU** i tregon server-it X, cili program është përgatitur të mbahet. Ne ketë rast është me interes ne klikimet e mouse-it, ne shtypjen e tastave ne tastjere, dhe window duke qene i pa mbuluar ose i pa siguruar. Ne praktike, një program real duhet të jetë me interes ne veprime të tjera si goditja. Dhe se fundmi, thirrja e **XmapRaised** vendoz window-sin e ri ne ekran si window me i fuqishem. Ne ketë pike window-si behet i dukshem ne ekran.

Laku kryesor konsiston ne paraqitje dhe është llogjikisht shume me i thjesht se laku korrespondues ne windows. Paraqitja e pare ketu jep një ngjarje dhe i dyti jep goditjen e fundit ne tipin e veprimeve për shqyrtim. Kur disa ngjarje tregojne qe programi ka mbaruar, running vendoset ne 0 dhe laku përfundon. Përpëra egzekutimit, programi liron kontekstin grafik, window, dhe lidhjen. Eshte me interes te permendim, qe jo çdo njëri pelqen një GUI. Shume programues përfersojne nderfaqen e orjentuar të zakonshme e tipit të diskutuar ne seksionin 5.6.2.

5.8.2 Terminali i rrjetit SLIM

Gjatë viteve, modeli informatike kryesor ka varuar ndermjet centralizimit dhe decentralizimit informatik. Kompjuterat e pare, si EINAC, ishin ne fakt kompjutra personal, megjithese ishte një i madh, sepse vetëm një person mund ta përdori ne të njëjtën kohe. Me pas erdhi sistemi timesharing, ne të cilin shume përdorues të larget ne terminale të thjeshta ndanin një kompjuter të madh qendror. Me pas erdhen PC para, ne të cilin përdoruesit kishin përseri kompjutrat e tyre personale.

Ndersa decentralizimi i modelit PC kishte avancuar, gjithashtu kishte disa disavantazhe qe jane për tu pare seriozisht. Me sa duket problemi me i madh është qe çdo PC ka një hard disk të madh dhe një software kompleks qe duhet mirmbjatur. Për shembull, kur del një shkarkim i ri ne sistemin operativ, një pune e madhe duhet bere per perfomimin e upgrade ne çdo makine me vetë. Ne te shumten e korporatave, kostua e punes qendron ne mirmbjtjen e ketyre lloj hardware dhe softwaresh. Për përdoruesit shtëpiak, puna është teknikisht e lire, por pak njerez jane të zotë ta bejne atë me korrektsi. Me sistemet e centralizuara, vetëm një ose pak makina duhet të behen update dhe keto makina kane një staf ekspert qe bejne punen.

Një problem i afert është qe përdoruesit duhet të bejne backup-e rregullisht të file-ve të tyre të sistemit qe kane një sasi të madhe të dhenash deri ne gigabyte, por vetëm pak prej tyre e bejne ketë. Kur dicka e keqe ndodh, është një problem shume i madh për ti siguruar ato file. Me sistem të centralizuar, backup – et behen çdo natë automatikisht ne një shirit automatik.

Një tjetër avantazh është qe përdorimi i përbashket i burimeve është me i lehtë me sisteme të centralizuar. Një sistem me 64 përdorues të larget, secili me 64 MB RAM do të ketë ketë me te shumten e ketij RAM, ne shumicen e kohes pa vene ne funksionim. Me një sistem të centralizuar me 4 GB RAM, nuk ndodh kurre qe disa përdorues përkohesisht kane nevoje me shume RAM, por nuk mund ta marin atë sepse ndodhet ne një PC të një personi tjetër. I njëjtë argument qendron dhe për disqet e memorjes dhe burimet e tjera.

Është me sa duket një konkluzion i drejtë të themi qe përdoruesit e shumtë kerkojne informatike tërheqese me performance të lartë, por nuk duan realisht të administrojnë një kompjuter. Kjo ka lejuar kerkuesit, të ri-egzaminojne timesharing duke përdorur terminalet dumb qe takojne terminalet modern. X ishte një hap ne ketë drejtim, por një X server është akoma një sistem kompleks me një software të madh megabytes qe duhet të upgraded here pas here. Grali i shenjtë duhet të jetë një performance e lartë ne të cilin makinat e përdoruesit nuk kane software për shqetësim. Me poshtë ne do të përshkruajme një sistem, i zhvilluar nga zbuluesit ne Sun Microsystems dhe Stanford University.

Sistemi quhet SLIM, i cili ngrihet nga Stateless Low – level Interface Machine. Idea është bazuar ne timesharing e centralizuar të zakonshem sic tregohet ne figuren 5 – 47.

Makinat e klientit jane dumb 1280 x 1024 bitmap, me një tastjere dhe mouse, por jo software të instaluar nga përdorues. Ata jane ne ndikimin e Inteligences se vjetër te terminalet character – oriented, qe kane pak kujdesje të interpretojne kodeve qe kane shpetuar, por jo software të tjere. Si tek terminalet character – oriented të vjetër, funksioni i vetëm i përdoruesit është fikja dhe ndezja. Terminalet e ketij tipi qe kane kapacitet procedimi të paket jane quajtur thin clients.

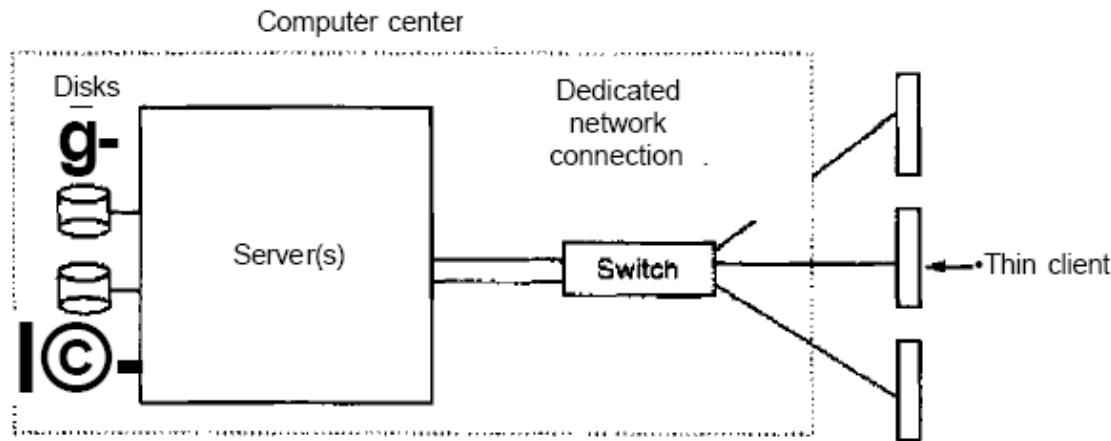


Figure 5-47.

Modeli me i thjeshtë qe ka server ship bitmaps ne rrjet te klientët SLIM, 60 here ne sekond nuk punon. Nevojitet rreth 2 Gbps gjeresi brezi të rrjetit, i cili është shume për rrjetin e tanishem të manovrohet. Modeli tjetër i thjeshtuar duke regjistruar imazhe ne ekran me përshtatës frame brenda terminalit dhe duke rifreskuar 60 here ne sekond ne vend të caktuar është me premtuese. Ne vecanti ne qoftë se serveri qendror ruan një kopje të çdo frame të terminalit dhe dergon vetëm update sipas nevojes, kerkosat e gjeresise se brezit jane shume modeste. Kjo është se si punon një SLIM thin client.

Ndryshe nga protokolli X, i cili ka qindra mesazhe komplekse për menaxhimin e windows-it, figurave me vizatim gjeometrik, dhe shfaqja e teksteve ne shume madhesi, protokolli SLIM ka vetëm pese mesazhe shfaqjeje, të radhitura ne figuren Fig.5-48 (ka gjithashtu një numer të vogel mesazhesh kontrolli jo të radhitura). I pari, SET, zevendetson një drejtkendesh bufferin e frame-it me pikxzela të rinj. Çdo pikxel i zevendesuar përmban 3 byte ne mesazh për të specifikuar vleren e plotë të ngjyrave (24-bit). Teorikisht, ky mesazh është i mjaftueshem për t'a kryer ketë detyre, të tjerat Jane vetëm për optimizim.

Message	Meaning
SET	Update a rectangle with new pixels
FtLL	Fill a rectangle with one pixel value^
BITMAP	Expand a bitmap to fill a rectangle
COPY	Copy a rectangle from one part of the frame buffer to another
CSCS	Convert a rectangle from television color (YUV) to RGB

Figure 5-48. Messes ust-d in the SLIM protocol from the server *la* the terminals.

Mesazhi FILL mbush një trekendesh plotësishët ne ekran me një vlerë të vetme pikxeli. Përdoret Tor për të mbushur një background të njëtrajtshëm. Mesazhi BITMAP mbush një katërkendesh të plotë duke përsëritur një motiv të përfshire ne mesazh bitmap. Kjo komande është e përdorshme për të mbushur background-et me motive qe kane disa ndertime dhe nuk jane një ngjyre e njëtrajtëshme.

Mesazhi COPY udhezon terminalin për të kopjuar një katërkendesh brenda një buffer frame tek një pjese tjeter e buffer frame-it. Është shume e dobishme për të rrotulluar ekranin dhe levizur dritaren. Se fundmi, mesazhi CSCS konverton sistemin e ngjyrave YUV të përdorur ne televizionet e SH.B.A-se (NTSC) qe vendosin sistemin RGB të përdorur nga monitoret e kompjuterit, fillimisht përdoret kur një frame video e papunuar ka kaluar tek një terminal ne sistemet YUV dhe duhet qe të konvertohet ne RGB, qe të shfaqet. Duke diskutuar për ketë nga ana algoritmike është e thjeshtë por harxon kohe, pra është me mire të shkarkosh punen ne terminale. Në qoftë se terminalet nuk do të përdoren për të pare videon, ky mesazh dhe funksionalitetet e tij mund të mos jene të nevojshme.

Idea e përgjithshme e klientëve "thin" Ne prototipin switcheve 100-MBPS Fast Ethernet është përdorur ne të dyja segmentimet server-to-switch dhe switch-to-terminals. Nga fuqia, një rrjet me gigabit duhet të përdoret midis serverit dhe switch sepse ai segment është lokal ne dhomen e kompjuterit qendror. Madhesia e pare ndeshet me karakteret qe përseriten ne ekran. Çdo karakter i shtypur dergohet ne server, i cili përpunon se cili piksel do të update-et për të vendosur karakterin ne vendin e duhur ne ekran, madhesine dhe ngjyren. Madhesite tregojne qe ajo merr 0.5 msec për karakterin, për t'u shfaur ne ekran. Ndryshe, ne një workstation lokal koha e përseritjes është 30 msec përgjatë buferimit të kernelit.

Pjesa tjeter e testeve maste performancen me përdorues qe po ekzekutojne programe aplikative, interaktive moderne si Adobe Photoshop, Adobe Frameworker dhe netscape. Nga vrojtimi doli qe gjysma e komandave të përdorueseve kerkonin update-m me të vogel se 10 000 piksel, e cila është 30000 byte e pakompresuar. Ne 100 Mbps ajo merr 2.4 msec për të nxjerre 10000 piksela për ne telë, për një kohe totale 5.1 msec (por varet nga kushtet rrethanore). Derisa koha e reagimit të qenieve njerezore është rrëth 100 msec, disa update duken si të castit. Ndoshta update-et e medha ishin pothuajse të castit. Gjithashtu, kur përdoret kompresimi, me tepër se 85% e update-eve Jane nen 30000 byte. Eksperimentet ishin përseritur ne një rrjet 10 Mbps, një rrjet 1-Mbps dhe një rrjet 128-Kbps. Tek 10 Mbps sistemi ishte virtualisht i castit dhe 1 Mbps ishte akoma mire. Ne 128 Kbps ajo ishte shume i ngadaltë për t'u përdorur. Lidhjet deri ne 1 Mbps Jane duke u bere shume shpejt realitet duke përdorur rrjetin kabllor TV dhe ADSL, duket qe kjo teknologji mund të aplikohet ne përdorueses shtëpiake ashtu si dhe ne përdorues biznesi.

5.9. Menaxhimi i fuqise

Qellimi i pare i pare i kompjuterave elektronike, ENIAC ka 18000 tuba vajisjeje dhe konsumon 140000 wat fuqi. Si rezultat, kjo ngre lart faturen elektrike. Pas zbulimit të transistorit përdorimi i fuqise ra dramatikisht dhe industria e kompjuterave humbi interesin ne kerkimet e fuqise.

Sado ne ditët tona menaxhimi i fuqise është kthyer ne projektor për disa arsyje dhe sistemi operativ është duke luajtur një rol ketu.

Le të fillojme me desktop PCs. Një desktop PC shpesh 1200 wat fuqi (85% është eficiente, kurse 15% humbet për ngrohje). Në qoftë se 100 milion nga keto makina jane hapur një heresh ne të gjithe botën, se bashku ato përdorin 20.000 Megawat energji elektrike. Ky është përfundimi total i 20 uzinave me fuqi nukleare.

Në qoftë se kerkimet e fuqise mund të lihen përgjysem, ne mund të shpetojme 10 uzina nukleare (ose ne krahasim të mbetjeve te karburanteve të uzinave) është një fitore e madhe dhe një kursim i vlefshem.

Vendet e tjera ku fuqia është një përdorim i madh, është ne fuqine e baterise të kompjuterave, përfshire notebooks, laptop-ët, palmtops dhe wedpods midis të tjereve.

Thelbi i problemit është qe bateritë nuk mund të karikohen per te zgjatur shume përdorimi i tyre, e shumta disa ore. Megjithë kerkimet masive te perkrahura nga kompanite e prodhimit të baterive, kompanitë e kompjuterave dhe kompanitë elektronike konsumatore, zhvillimi është i ngrire.

Ne një industri e cila eshte mesuar te permiresoje performancen çdo 18 muaj, mospatja e një progresi duket sikur po thyhen ligjet e fizikes, por kjo është një situata e tanishme.

Si rrjedhoje, te prodhosh kompjutera qe përdorin me pak energji, sjell per pasoje dhe zgjatjen e përdorimit te baterise, qe eshte dhe qellimi i axhendave te te gjithe prodhuesve. Sistemi operativ luan një rol kryesor ketu, qe ne do ta shohim me poshtë.

Jane dy perpjekje të përgjithshme për të zvogeluari konsumimin e energjise.

E para është qe sistemi operativ te fiki disa pajisje (me se shumti pajisje I/O), sepse kur një pajisje eshte e fikur përdor pak ose aspak energji.

E dyta është qe programi aplikativ te përdori me pak energji, per te zvogeluari kohen e përdorimit te baterise, por me shume mundesi degradon cilesia e eksperiencave se përdoruese.

Ne do ti shikojme te dyja keto perpjekje me radhe, por fillimisht ne do të flasim pak rreth ndertimit te hardware-t, duke respektuar përdorimin e fuqise.

Hardware issues

Bateri Jane dy llojesh: me një përdorim dhe te rikarikueshme.

Baterite me një përdorim (me shume AAA, AA dhe D qelizat) mund të përdoren për pajisje te vogla (handhandle), por nuk ka energji të mjaftueshme për laptop të fuqishem me ekran të ndritshem.

Baterite e rikarikueshme, ndryshe nga te parat, mund të kenë energji të mjaftueshme për një laptop për disa ore.

Bateritë me nikeli dhe kodium kane dominuar ketu, por ato u hapen rruge baterive hibride me metal nikeli, qe zgjasin me shume dhe nuk e ndotin ambientin kur jane te shkarkuara.

Bateritë prej hekuri dhe litiumi jane akoma me mire dhe mund të karikohen pa qene plotesisht te thata, por kapaciteti i tyre është rreptësisht i limituar.

Per shitesit e kompjuterave, zgjidhja ne lidhje me baterite u gjet qe, CPU-ja, Memoria dhe pajisjet I/O te kene disa gjendje: on, sleeping, heberating dhe off.

Per te perdorur pajisjen, ajo duhet te jete ne gjendjen; on. Kur pajisja nuk nevojitet per nje kohe te shkurter ajo mund te kaloje ne gjendjen; sleep, e cila sjell nje konsumim me te vogel te energjise. Kur pritet qe pajisja te mos perdoret per nje kohe me te gjate ajo kalon ne gjendjen; hibernated, e cila sjell nje kursim me te madh te energjise. Por problemi ketu eshte qe duhet me shume energji per ta nxjerre pajisjen nga gjendja, hibernated, se nga gjendja sleeping. Perfundimisht kur pajisja eshte e fikur ajo nuk harxhon energji. Duhet te permendin qe te gjitha pajisjet i kane te kater gjendjet, por eshte detyra e sistemit operativ te menaxhoje gjendjet e tranzicionit, ne nje moment te caktuar.

Disa kompjutera kane 2 ose 3 butona te fuqise. Nje nga keto butona mund ta kaloje te gjithe kompjuterin ne gjendje sleep, nga e cila mund te dali menjehere duke shtypur nje karakter ose duke levizur mouse. Tjetra e ve ne gjendjen Hibernation, nga e cila duhet nje kohe me e gjate per te dale. Ne te dyja keto raste, ato nuk bejne gje tjeter vetem se i dergojne nje sinjal sistemit operativ, i cili e pushon menjehere software-in. Ne disa shtete, pajisjet elektrike, te detyruara me ligj, duhet te kene nje celes mekanik, e cila e fik menjehere automatikisht pajisjen, per arsyte sigurie.

Menaxhimi i fuqise sjell një sere pyetjesh me te cilat sistemi operativ duhet te perballat. Ato jane si me poshte; cila pajisje duhet te kontrollohet? Jane ato ne gjendje On/Off apo jane ne gjendje te ndermjetme? Sa energji eshte kursyer duke e mbajtur ne gjendjet e uleta? A harxhohet energji kur sistemi ristartohet? Kursehet energji kur kalohet ne nje gjendje me te ulet? Sa kohe duket per tu kthyer ne gjendjen maksimale?

Sigurisht përgjigjet e ketyre pyetje ndryshojne nga pajisja ne pajisje dhe sistemi operativ duhet te kete parasysh nje sere mundeshish.

Kerkues te ndryshem, kane egzaminuar laptopet për të pare kur iken korenti dhe kane arritur ne konkluzionet qe tregohen ne figuren 5.49. Ata thjesht konfirmuan qe pajisjet fikeshin sipas kesaj radhe, ne fillim ekranit, hard disk dhe CPU-ja. Por keto parametra nuk mund te pergjithesohen, sepse marka te ndryshme kompjuterash, kane kerkesa te ndryshme per matjen e energjise. Por nga tabela me poshte duket qarte qe, ekranit, hard diskut dhe CPU-ja jane objektet kryesore per kursimin e energjise.

Device	Li et al. (1994)	Lorch and Smith (1998)
Display	68%	39%
CPU	12%	18%
Hard disk	20%	12%
Modem		6%
Sound		2%
Memory	0.5%	1%
Other		22%

Operating system issues (Dalja ne qarkullim e sistemit operativ):

Sistemi Operativ luan një rol kryesor ne menaxhimin e energjise. Ai kontrollon te gjitha pajisjet, ai mund te vendosi ke pajisje duhet te fik dhe kur duhet ta bez ate. Kur fik nje pajisje dhe ajo duhet pas pak kohe te perdoret, atehere mund te kemi nje vonese jo te kendshme kur duhet ta ristartojme ate, nga ana tjeter neqoftese koha per fikjen e nje pajisje eshte e gjate, atehere nuk kemi kursyer energji. Hilja ketu qendron qe te gjenden algoritmet e duhura, qe sistemi operativ te vendosi, cilen pajisje duhet te fiki dhe kur.

Një përdorues mund të pranoje qe pas 30 sekondash pa përdorim të kompjuterit ai të marre 2 sekonda për tu përgjigjur një shtypje të butonit të tastieres. Një përdorues tjetër mund të mos ta pranoje kete zgjidhje. Ne mungese të audio input-it kompjuteri nuk mund ti vecoje keta përdorues.

Ekrani

Le të shikojme tani shpenzuesit me te medhenj te energjise për të pare se cfare mund të bezme. Konsumuesi me i madhe i energjise eshte ekranı.

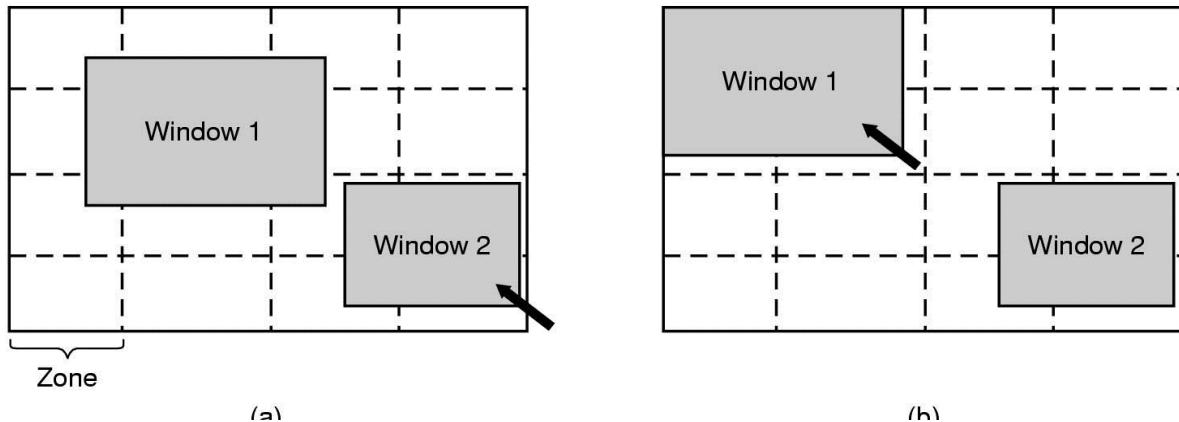
Për të patur një imazh të ndritshem ekranit duhet të riktheje dhe të japi energjine.

Disa sisteme operativ orvaten të shpetojne energjine nga mbyllja e paraqitjes kur nuk ka aktivizim per disa minuta.

Shpesh përdoruesi duhet të zgjidhe se cfare intervali është mbyllja, duke pushuar tregimin midis boshlekut të ekranit dhe përdorimit të baterise mbrapsht. Fikja e ekranit eshte gjendja sleep, sepse ai mund te kthehet pothuaj menjehere ne gjendjen e meparshme, apo nje nga karakteret te jete shtypur ose te levizi pajisja qe pointon.

Një përmiresim i mundshem u tregua nga Flinn & Satyanarayanan. Ata sygjeronin qe ekranit te konsistonte ne nje numer zonash, te cilat mund te ndizeshin ose fikeshin ne menyre te pamvarur. Ne figuren 5.50 (a), jane paraqitur 16 zona te cilat jane ndare per tu dalluar me vija me te trasha. Kur kursori është ne dritaren 2, sic tregohet edhe ne figure vetëm cereku i zones eshte i ndricuar, tjetra eshte ne erresire, si pasoje kursethet $\frac{3}{4}$ e energjise qe harxon ekranit. Kur përdoruesi leviz kursorin ne dritaren 1, zona e dritares 2 mund të jetë erresuar dhe mbrapa saj mund të jene rindezur. Megjithate, dritarja 1 shtrihet ne 9 zona dhe nevojitet me shume energji.

Në qoftë se menaxheri i windows, e kupton se cfare eshte duke ndodhur ai mund ta zhvendosi zonen 2, ne menyre qe te shtrihet vetem ne kater zona, me nje lloj prerje te shpejte te zones, sic tregohet ne figuren 5.50 (b). Për të arritur ketë ulje prej 9/10 të fuqise se plotë ne 4/16 të fuqise totale, menaxheri i windows duhet te kuptoje si te menaxhoje energjine ose është i aftë të pranoje instruksionet nga disa pjese të tjera të sistemit. Me shume e sofistikuar mund të jetë aftësia e ndricimit të pjesshem të dritares qe nuk është tëresisht ne përdorim.



HDD

Një tjetër konsumator potencial i energjisë është edhe hard disku. Ai kerkon shume energji per tu rrotulluar (spin) me shpejtesi te larte, edhe ne qofte se nuk ka akses. Disa nga kompjuterat, vecanerisht laptopet, arrijne te shuajne rrotullimet e diskut pas disa minutash aktivitet, ne qofte se ai nevojitet prape, ai vihet ne gjendje pune serish. Fatkeqesisht ndalimi i diskut është ne gjendjen hibernating dhe jo ne gjendjen sleeping, kjo do te thote qe per riaktivizimi i tij do te duhen disa minuta, e cila do te shkaktoje me shume vonesa per perdoruesin.

Per me teper, restartimi i diskut shkakton harxhim ekstra te energjisë si pasoje, ka një periode rrotullimi karakteristike, T_d , ne te shumten e rasteve me shpejtesi maksimale 5 rrotullime ne 5 sec. Supozojme qe disku tjetër te aksesohet ne kohen t. Në qoftë se $t < T_{th}$, shpenzohet me pak energji nese disku bahet ne pune, se sa ta fikim dhe ta ndezim prape me vone. Në qoftë se $t > T_{th}$, atehere kursethet me shume energji duke e fikur diskun dhe rindezur me vone. Ne praktike shume sisteme jane konservatore dhe e fikin diskun pas disa munutash qendrimi ne getesi (joaktiv).

Menyre tjetër për të kursyer energji, eshte pajisja e RAM-it me një cache te konsiderueshme. Nese nevojitet blloku qe ndodhet ne cache, nuk eshte nevoja te restartohet disku i cili eshte ne gjendjen IDL, per te lexuar ne te. E njejtë eshte dhe kur duam te shkrumë ne disk, ne qofte se veprimin e kemi ruajtur ne cache, nuk eshte e nevojshme te restartojme diskun per te shkruar ne te. Disku do te qendroje i fikur deri ne rastin kur kemi nje miss cache.

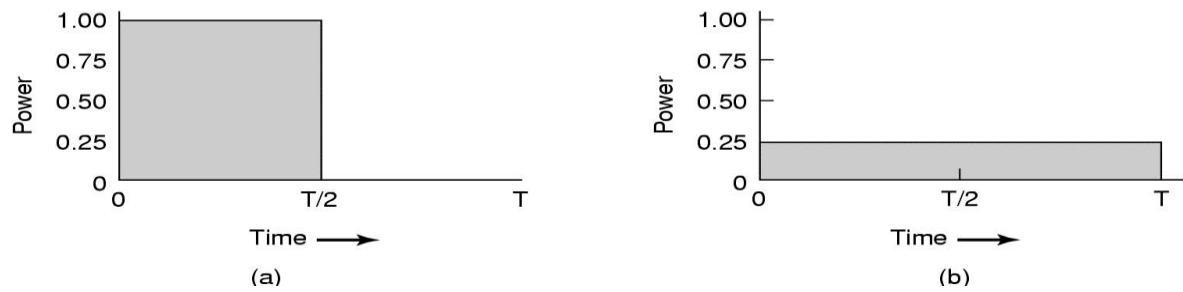
Menyre tjeter per te menjanuar startimet e panevojshme te diskut, eshte qe sistemi operativ te informoje programet qe jane duke u ekzekutuar rrëth gjendjes se diskut, duke i derguar atij mesazhe ose sinjale. Per shembull, një proçesor word, mund te kerkoje te shkrueje një file, duke edituar diskun here pas here ne pak minuta. Ne qofte se proçesori i word-it do ta dije qe disku eshte ne gjendje te fikur per momentin, ai do ta menjanonte veprimin e shkrimit, duke e shtyre ate per momentin qe disku eshte serish ne pune.

CPU

CPU mund te menaxhohet qe te kurseje energji, gjithashtu.

CPU e laptopit mund t \ddot{e} kalohet ne gjendjen sleep, duke e cuar konsumimin e energjisë ne zero. E vjetmja gje qe ajo mund te beje pas kesaj gjendje, eshte te zgjohet kur vjen nje interrupt. Por edhe kur CPU-ja eshte idle ose ne pritje per I/O, ajo kalon ne gjendje sleep. Ne shume kompjutera, ka nje lidhje ndermjet tensionit te CPU-se, ciklit te clock dhe perdorimit te energjisë. Tensioni i CPU-se shpesh mundet te reduktohet ne software, e cila sjell kursim te energjisë, por gjithashtu dhe shkurtimin e ciklit te clokut. Derisa kursimi i energjisë eshte ne perpjestim me tensionin ne kattror, duke zvogeluar tensionin pergysem, e ben CPU-ne edhe gjysem here me te shpejte, por kursen dhe $\frac{1}{4}$ e energjisë. Kjo veti mund te perdoret tek programet me deadline te mire percaktuar, te tilla si multimedia viewers, e cila duhet te dekompressoje dhe te shfaqi nje imazh çdo 40 msec, por kalon ne gjendjen idle kur duhet ta bej me shpejt. Supozojme se nje CPU perdor x xhaul energji derisa eshte duke vepruar me fuqi te plote me 40 msec dhe perdor $\frac{1}{4}$ e xhaul energji, per te vepruar me gjysmen e shpejtesise. Ne qofte se nje multimedia viewer mund te dekompressoje dhe te shfaqi nje imazh per 20 msec, sistemi operativ mund te veproje me fuqi te plote per 20 msec dhe pastaj te fiket per 20 msec e mbetur, pra ne total perdor $\frac{1}{2}$ e xhaul energji. Alternativisht, ai mund te veproje me gjysmen e fuqise dhe te arrije deadline-n, por ne te vertete perdor vetem $\frac{1}{4}$ xhaul energji. Nje krahasim i veprimit me shpejtesi te plote dhe fuqi te plote per disa intervale, dhe ai i veprimit me gjysmen e shpejtesise dhe $\frac{1}{4}$ e fuqise per dy, eshte treguar ne Fig. 5-51. ne te dyja rastet eshte kryer pune, por ne Fig 5-5 K(b) eshte konsumuar vetem gjysma e energjisë.

Ne menyre te ngjashme, ne qofte se nje perdorues shtyp nje karakter ne sekonde, por nevojitet qe karakteri i marre te procesohet ne 100 msec, eshte me mire per sistemin operativ te vendosi periudha me te gjata pauze dhe te uli shpejtesine e procesorit, gati 10 fish. Pra, duke punuar ngadale kursen me shume energji sesa duke punuar shpejt.



Memorja

Vetem dy opsiione te mundshme egzistojne per te kursyer energji ne memorie. E para, cache mund te zbrazet dhe pastaj te fiket. Por, ajo gjithmonë mund te ringarkohet nga memorja kryesore, pa patur humbje te informacionit. Ringarkimi duhet bere ne menyre dinamike dhe te shpejte, pra fikja e cache-s eshte pothuaj kalim ne gjendje sleep.

Nje opzion tjeter me drastik eshte qe te shkruhet e tere permajtja ne memorien kryesore te diskut, dhe pastaj te fiket eshte vete memorja kryesore. Kjo alternative eshte gjendja hibernation, sepse ne menyre virtuale e gjithe fuqia mund ti hiqet memorjes dhe pastaj te harxhohet nje kohe e konsiderueshme per ringarkim, vecanerisht ne qofte se edhe disku eshte i fikur, gjithashtu. Kur memorja eshte cut off, edhe CPU-ja ne gjendje qetesie duhet te fiket gjithashtu ose te egzekutohet jashte ROM-it. Ne qofte se CPU-ja eshte e fikur, interrupti qe e zgjon ate, duhet te shkaktoje nje kercim ne kodin e memorjes ne Rom, ne menyre qe memorja te mund te ringarkohet perpara se te perdoret. Megjithë keto qe permendem, te nderpresesh memorjen per periudha te gjata kohe, mund te jete me leverdi ne qofte se ristartimi ne pak sekonda mund te konsiderohet me i deshirueshem sesa rebooting, i sistemit operativ nga disku, i cili zakonisht merr nje ose disa minuta kohe.

Wireless Communication

Me rritjen e numrit te kompjuterave portativ, shume prej tyre perdorinin lidhjen wireless me boten jashte. Radio transmetuesi dhe marresi jane shpesh harxhuesit me te medhenj te energjise. Ne vecanti, radio marresi eshte gjithmone i ndezur, ne menyre qe te degjoje per e-mailet qe vijne dhe bateria mund te thahet me shpejt. Ne anen tjeter, ne qofte se radio fiket me vone, le te themi pas nje minute ne gjendje pauze, mesazhet hyrese mund te humbasin, e cila eshte e padeshirueshme.

Nje zgjidhje efekte e ketij problemi u dha nga Kravets dhe Krishan. Thelbi i zgjidhjes se tyre ishte qe kompjuterat portative te komunikojne me stacione baze fikse, qe kane disk dhe memorje te madhe dhe te pakufizuar ne fuqi. Ajo cfare ata propozuan ishte qe kompjuterat portative te dergojne nje mesazh ne stacionin baze, kur ai eshte gati duke e fikur radion. Qe ne ate moment, bufferi i stacionit baze e con mesazhin qe mori ne disk. Kur kompjuteri portativ e fik perseri radion, ai njofton stacionin baze. Ne ate pike, çdo mesazh i akumular mund ti dergohet atij.

Mesazhet dalese qe jane gjeneruar ndersa radio eshte e fikur Jane ne bufferin e kompjuterit portativ. Ne qofte se bufferi rrezikon te mbushet plot, radio fiket dhe ja le radhen per te transmetuar stacionit baze.

Kur duhet te fiket radio? Nje mundesi eshte qe ti lihet perdoruesit ose programit aplikativ per ta vendosur. Mundesi tjeter eshte qe te fiket pas disa sekondave ne gjendje pauze. Kur duhet ndezur perseri? Ate e vendos perdoruesi ose programi, ose mund te ndizet periodikisht per te kontrolluar trafikun e brendshem dhe per transmetuar ndonje mesazh qe eshte duke pritur ne radhe. Sigurisht, ajo mund te ndizet kur output buffer eshte i mbushur plot. Ka dhe plot mundesi te tjera.

Thermal management

Menaxhimi termik, eshte pak me i ndryshem, por prape lidhet me clirimin e energjise. CPU-te moderne nxehen jashte zakonisht shume punojne me shpejtesi te madhe. Desktopet e makinave, kane normalisht nje ventilator te brendshem per te nxjerre ajrin e

nxehte jashe kases. Derisa reduktimi i konsumimit te fuqise nuk ka nje ndikim te madhe te desktopet e makinave, ventilatori eshte zakonisht i ndezur gjate gjithe kohes.

Ne laptopet situata eshte ndryshe. Sistemi operativ duhet te monitoroj temperaturen ne menyre konstante. Kur temperatura eshte afer maksimumit te lejueshem, sistemi operativ ben nje zgjedhje. Ai mund te ndezi ventilatorin, i cili ben zhurme dhe harxhon fuqi. Ne menyre alternative, ai mund te reduktoje konsumimin e fuqise duke reduktuar ndricimin nga mbrapa te ekranit, duke ulur aktivitetin e CPU-se, duke qene me agresiv per te ulur shpejtesine e rrotullimit te diskut dhe te tjera.

Futja e disa inputeve nga perdoruesi mund te cilesohen si guide. Per shembull, nje perdorues mund ta percaktoje qe me perpara qe zhurma e ventilatorit eshte e pakendshme, ne menyre qe sistemi operativ te mund te reduktoje konsumimin e fuqise.

Menaxhimi i baterise

Ne kohet e vjetra një bateri furnizonte me rryme derisa te shkarkohej, kohe ne te cilen ajo ndalonte punen. Tani jo me. Laptopet perdonin bateri inteligjente, te cilat mund te komunikojne me sistemin operativ. Pervec kerkesave, ato mund te raportojne dhe per gjera si maksimumi i tensionit, tensoni aktual, maksimumi i karikimit, karikimi aktual, maksimumi i shkalles se shkrakimit te baterise, shkalla aktuale e shkarkimit, etj. Shume kompjutera laptop kane programe qe punojne me ane te query (pyetjeve) dhe shfaqin ne ekran te gjithe keto parametra. Baterite inteligjente mundeni gjithashtu te instruktohen per te ndryshuar parametra te ndryshem nen kontrollin e sistemit operativ.

Disa laptopë kane bateri te shumefishta. Kur sistemi operativ zbulon se nje bateri eshte afer fundit, ai duhet te percaktoje se cila do te jete bateria tjeter qe do te vazhdoje te funksionoje, pa shkaktuar ndonje dem gjate tranzicionit. Kur bateria e fundit eshte ne perfundim te saj, eshte detyre e sistemit operativ te lajmeroje perdoruesin dhe pastaj te bej nje dalje te rregullt, per shembull, te siguroje qe file system nuk eshte demtuar.

Driver Interface

Sistemi Windows, ka nje mekanizesh te zhvilluar per te kryer menaxhimin e fuqise, te qujatu **ACPI** (Advanced Configuration and Power Interface). Sistemi operativ mund te d drejtoje komanda sipas perkatesise, duke pyetur pajisjet per te raportuar per kapacitetin dhe gjendjen aktuale te tyre. Ky tipar eshte vecanerisht i rendesishem kur kombinohet me fik e ndiz (plug and play), sepse menjehere pas reboot-imit, sistemi operativ nuk e di cilat pajisje jane prezente, je me te dije vetite e tyre per te menaxhuar konsumimin e fuqise dhe te energjise.

Ai mundet gjithashtu ti dergoje komanda driver-it per ta instruktuar per te ulu nivelin e fuqise (kjo bazuar dhe ne mundesite qe ai ka patur fillimisht0. ne vecanti, kur nje pajisje si tastiera ose mouse vihet ne funksionim pas nje periudhe pauze, kjo eshte nje sinjal per sistemin qe te kthehet mbrapsht, ne aktivitet normal).

Degraded Operation

Me siper ne tame se si sistemi operativ mund te kurseje energji nga perdorimi i pajisjeve te ndryshme. Por ka dhe nje mundesi tjeter, ti thuash programit te perdori me pak energji, edhe pse kjo do te thote me pak pervoje per perdoruesin. Vecanerisht, ky informacion nevojitet kur bateria eshte poshte pragut te lejueshem te karikimit. Pastaj i duhet programit te vendosi ndermjet degradimit te performances duke e mbajtur baterine ne pune ose te ruaje performance dhe te rrezikoje ikjen e energjise.

Nje nga pyetjet qe lind ketu eshte si mundet nje program te degradoje performance per te kursyer energji? Kjo pyetje u studiuva nga Flinn dhe Satyanarayanan (1999). Ata solllen kater shembuj si degradimi i performances mund te kurseje energji. Ne do ti shikojme shkurtimisht.

Ne kete studim, informacioni i paraqitet perdoruesit ne forma te ndryshme. Kur nuk kemi degradim, na eshte paraqitur informacioni me i mire i mundshem. Kur degradimi eshte present, besueshmeria e infomacionit qe eshte prezent per perdoruesin eshte me e keqja qe mund te egzistonte. Ne do shohim disa shembuj, shkurtimisht.

Ne menyre qe te masi perdorimin e energjise, Flin dhe Satyanarayanan shpiku nje software tool te quajtur PowerScope. Ajo cfare ajo ben, eshte qe te siguroje nje profil te programit te perdorimit te energjise. Per ta perdorur ate, kompjuterit duhet ti jete montuar nje pajisje e jashtme per fuqine, nepermjet nje multimeter digital sofiwzTccontroWed. Duke perdorur multimetrin, software mund te lexoje numrin e miliamper qe futen nga pajisja e jashtme e fuqise, dhe ne kete menyre mund te percaktoje ne moment fuqine qe harxhon kompjuteri. Ajo cfare PowerScope ben, eshte qe periodikisht program counter dhe perdoruesi i fuqise i shkruan keto te dhena ne nje file. Pasi programi ka perfunduar punen, file perdoret per tu analizuar per te ditur energjine qe eshte perdorur nga çdo procedure. Kjo forme matje e ka bazen tek observimi. Matjet per kursimin e energjise ne hardware, jane perdorur dhe formojne bazen per matjet per degradimin e performances.

Programi i pare qe jane kryes matjet ishte video player. Ne menyre qe mos te kemi degradim, ai luan 30 imazhe/sec me rezolucion te plete dhe me ngjyra. Nje menyre degradimi eshte qe te braktisim shfaqjen e informacionit me ngjyra dhe te shfaqi vidiion ne bardhe e zi. Nje tjeter forme degradimi eshte te redukojme shkallen e imazhit, e cila do te coje ne dridhje dhe filmi do te kete nje cilesi jo te mire. Nje tjeter forme degradimi eshte te redukojme pixel ne te gjitha drejtimet, kjo con ne uljen e rezolucionit dhe e zvogelon madhesine e ekranit. Masat e ketij lloji con ne kursimin e 30% te energjise.

Programi i dyte ishte fjalimet ne publik. Si shembull ketu mund te marrim valen qe formon mikrofoni. Kjo vale mund te analizohet dhe nga nje kompjuter laptop ose te cohet nepermjet nje linku te radios per tu transmetuar ne nje kompjuter fiks. Duke kryer kete veprim kursejme energji nga CPU-ja, por harxhojme energji per radion. Degradimi eshte kur transmetojme nje fjalim te shkurter dhe perdorim nje model akustik te thjeshte. Fitimi ketu eshte 35%.

Shembulli tjeter eshte nje map viewer, i cili e transmeton harten nepermjet nje linku te radios. Degradimi qendron ne prerjen e harten ne dimensione me te vogla ose i thote serverit ne distance qe te lere jashtje paraqitjes rruget e vogla, meqe kjo kerkon dhe me pak bit per tu transmetuar. Fitimi ketu eshte perseri rreth 35%.

Eksperimenti i katert ishte transmetimi i imazheve JPEG ne nje web browser. Standartet JPEG lejojne algoritma te ndryshem, marrjen e nje cilesie imazhi me te mire, por duke

abuzuar me madhesine, ketu fitojme rrerh 9%. Nga te gjitha sa thame, pamje se duke pranuar degradimin e cilesise, perdoruesi mund te perdori me gjate nje bateri.

KAPITULLI I GJASHTË

File-at Sistem

Të gjithe aplikacionet e kompjuterave kane nevoje të memorizojne dhe të rinxjerin informacion. Derisa një proces të jetë ne ekzekutim, ai mund të memorizoje një sasi të limituar informacioni, ne hapesiren e vet te adreses. Gjithesesi kapaciteti i memorizimit eshte i kufizuar nga madhesia e hapesires se adreses virtuale. Për disa aplikacione kjo hapesire është e mjaftueshme, por për të tjere është shume shume pak.

Një problem i dytë ne mbajtjen e informacionit ne hapesiren e adreses se nje prosesi eshte kur se kur ky proces mbaron, informacioni humbet. Per shume aplikacione (per shembull, database) ky informacion duhet te ruhet per disa javë, muaj apo dhe per gjithmone. Humbja e tij kur procesi mbaron eshte e papranueshme. Gjithashtu, nuk duhet te zhduket kur nje deshtim ne kompjuter vret procesin.

Një problem i tretë qe shtrohet, është qe zakonisht nevojitet për procese të shumefishta, të aksesojne informacion ne te njëjtën kohe. Ne se kemi një telefon të lidhur ne linjë dhe të regjistruar drejtpërdrejt ne hapesiren e adresave të një procesi të vetëm, atëhere vetëm ai proces mund të aksesoje atë informacion. Menyra për ta zgjidhur ketë problem, është qe të mund të bejme vetë ketë informacion të paaksesueshem nga çdo proces tjetër.

Ketu ne kemi tre kerkesa thelbesore për te ruajtur informacionin per nje periudhe afatgjate :

1. Mund të jetë e mundur të memorizojme një sasi të madhe informacioni.
2. Informacioni duhet të survejoje mbarimin e procesit i cili po e perdon.
3. Proseset e shumefisht duhet të jene të aftë të aksesojne informacionin ne menyre konkurese.

Zakonisht menyra e zgjidhjes se problemit është, memorizimi i informacionit ne ndonjë disk ose ne ndonjë pajisje tjetër të jashteme, ne njesi qe quhen **File**. Proseset mund të lexojne dhe shkruajne një informacion të ri, ne se është nevoja. Infomacionet e memorizuara ne file, duhen të jene kembegules, qe të mos ndikohen nga krijimi dhe mbarimi i nje procesi të caktuar. Një file duhet të zhduket vetëm ne se, vet përdoruesi i tij do ta fshij atë.

File-t menaxhohen nga vet sistemi operativ. Se si ata të jane strukturuar, emeruar, aksesuar, përdorur, mbrojtur dhe implementuar nga pjesa me e madhe, jane ceshtjet kryesore ne diznjimin e nje sistemi operativ. Ne per gjithese ajo pjese e sistemit operativ e cila merret me filet, eshte quajtur file system dhe eshte objekt i ketij kapitulli.

Nga kendveshtrimi i perdoruesit, aspekti kryqesor i nje file systemi eshte se si ai u shfaqet atyre, qe do te thote ku konsiston file, si emerohet dhe ruhet, cfare operacionesh lejohen te kryhen ne file, etj.. Detajet kur bitmap ose nje liste eshte duke ruajtur te dhena

ne nje hapesire te lire dhe ne sa sektore eshte i ndare nje bllok logjik, jane me pak interes per ne, pavaresisht se ato jane me nje rendesi te madhe per projektuesin e nje file system. Për ketë arsy, ne kemi strukturuar kapitullin ne nenkapituj. Dy te paret trajtojne respektivisht, ndertimi i nderfaqes se perdonuesi me file-t dhe direktorite. Pastaj jepet ne menyre me te detajuar se si jane implementuar filet sistem. Ne fund, ne japim disa shembuj te nje file sistem te vertete.

6.1 FILES

Ne faqet ne vazhdim do ti shohim filet nga ana e përdoruesit, kjo është ajo se si ato do të përdoren dhe se cfare vetish të brendeshme ata kane.

6.1.1 File Naming (emertimi i tyre)

Filet jane një abstraksion i makines (kompjuterit). Ato sigurojne menyren e ruajtjes se informacionit ne hard disk, i cili mund te lexohet dhe me pas. Kjo mund të behet ne një menyre qe përdoruesi të mos të merret me detaje të tjera për menyren e ruajtjes se informacionit dhe si realisht punon edhe disku.

Shpesh karakteristika me e rendesishme e çdo abstraksi mekanik, është menyra e menaxhimit dhe emertimit te nje objekti, keshu qe ne do të fillojme ekzaminimin e file-ve të sistemit me emerimin e nje file. Kur një proces krijon një file, i cakton atij file një emer. Kur procesi mbaron, file vazhdon të ekzistoj dhe mund të aksesohet nga një tjetër proces qe lind dhe e përdor atë file me të njëjtin emer të meparshem.

Disa nga rregullat strikte të emertimit të file-ve, ndryshojne nga sistemi ne sistem, por te gjithe sistemet operative te tanishme, lejojne nje string nga nje deri ne tete karaktere te emerojne nje file. Për shembull keta emra si andrea, bruce, and cathy, mund të jene emra të një file cfaredo. Shpesh shifra ose karaktere te vecanta jane te lejueshme, keshtu dhe emrat si 2, urgent!, dhe të Fig.2-14, jane te vlefshme. Shumica e file system mund të suportojne emra deri ne 255 karaktere.

Disa file system arrijne te bejne ndryshimin midis karaktereve me germe te madhe dhe germe te vogel, kurse të tjere jo. Gabimet e UNIX ne kategorine e pare me gabimet e MS-DOS ne kategorine e dytë. Një sistem UNIX mund të ketë të treja llojet e ndryshme të file-ve si: maria, Maria and MARIA, kurse tek MS-DOS, te gjitha keto emra i referohen te njejtit file.

Një ane tjetër e file-ve të sistemit do ta themi tani: Windows 95 dhe Windows 98 të dy sistemet e përdorin file të MS-DOS, dhe si rrjedhoje trashegon shume nga vetit e tij, per shembull, si jane ndertuar emrat e fileve. Ne vazhdim mund të themi se Windows NT dhe Windows 2000 përbajne file të MS-DOS, gjithashtu ato trashegojn veti te tij. Megjithate, mund te themi qe dy sistemet e fundit kane nje file te veten, te lindur, file NTFS, qe ka veti te ndryshme (per shembull, emri i file eshte unik). Ne ketë kapitull kur i referohemi, file sistem të Windows-it, kuptojme file të MS-DOS-it, të cilët jane të vetmit file të sistemit të suportuar nga të gjitha versionet e Windows-it. Ne do të flasim rreth file te lindur të sistemit të Windows 2000 ne kapitullin 11.

Shume sisteme operative suportojnë dy pjesë të emrit të file-it, me të dyja pjeset të ndara ne kohe, si ne programin c.

Për pjeset ne vazhdim kjo periudhe do të quhet file extension (zgjatja e file-ve), dhe zakonisht tregon dicka rreth file-it. Ne MS-DOS, për shembull, emrat e file-ve jane të gjatë 1 deri ne 8 karaktere, plus një zgjatje optimale 1 deri ne 3 karaktere. Ne UNIX, madhesia e zgjatjes se file-it, nuk është fiks, varet nga përdoruesi, ai e cakton atë dhe një file nuk mund të ketë 2 ose me shume zgjatje ose prapashtesa, sic ndodh ne prog.c .Z, ku .Z zakonisht perdoret për të treguar qe file (prog.c) është komprimuar duke përdorur algoritmin e komprimimit Ziv-Lempel .

<u>Prapashtesa</u>	<u>Kuptimi</u>
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figura 6-1. Disa nga prapashtesat tipike të file-ve.

Ne disa sisteme operative përfshire dhe UNIX prapashtesat e file-ve jane thjesht konveksione dhe nuk jane të detyrueshme nga sistemi operativ, por filet mund të ruhen edhe me prapashtesa të tjera por qe ne fund të fundit nuk mund të njihen nga sistemi. Një file i emertur si file.txt (text) mund të ketë disa lloje të tjere, por ky emertim është me shume si për të kujtar përbajtjen e file-it. Nga ana tjetër një kompilator i gjuhes C, insiston qe file qe ai duhet të kompiloje duhet të jetë një file.c absolutisht dhe nuk mund të kompiloje file.* të tjere.

Konveksione të tilla janë vencanërisht shume të përdorshme, kur disa programe mund të kene ndryshime nga ato të file-it. Kompilatori i gjuhes C, për shembull mund të ketë një listë të file-ve që mund të kompiloje dhe linkoje, disa nga file-at e C-se me disa file te assemblerit. Prapashtesat behen esenciale për kompilatorin, per te kuptuar cilat file janë të C, cilat assembler dhe cilat të llojeve të tjere.

Ne ndryshim, Windowsi është një program i vetedijshme per prapashtesat, të cilat i japin file-ve një kuptim. Përdoruesit mund të ruajne prapashtesat edhe si ato të sistemit dhe mund te percaktojne per secilen, se cili program i zoteron ato. Kur një përdorues klikon dy here mbi emrin e file-it, programi shenjon ne prapashtesen e programit qe do te egzekutohet, bashke me file-n.

6.1.2 File Structure (Struktura ose ndertimi i një file)

File-t është e mundur të strukturohen ne menyrat me të mundeshme janë të ilustruara ne figuren 6-2. File-i ne figuren 6-1(a) është një sekuese e pastrukturuar bitesh. Ne fakt, sistemi operativ nuk njeh as nuk kujdeset se cfare përban file. Gjithcka cfare ai shikon ose me saktë njeh, jane thjesht një mori e madhe bitesh. Cfaredo kuptimi qe mund ti japim file-ve, duhet të jetë ne nivelin e përdoruesit për një kuptim me të mire psikologjik. Të dy sistemet opërotive si UNIX dhe Windows –si kane të njëjtë konceptimet.

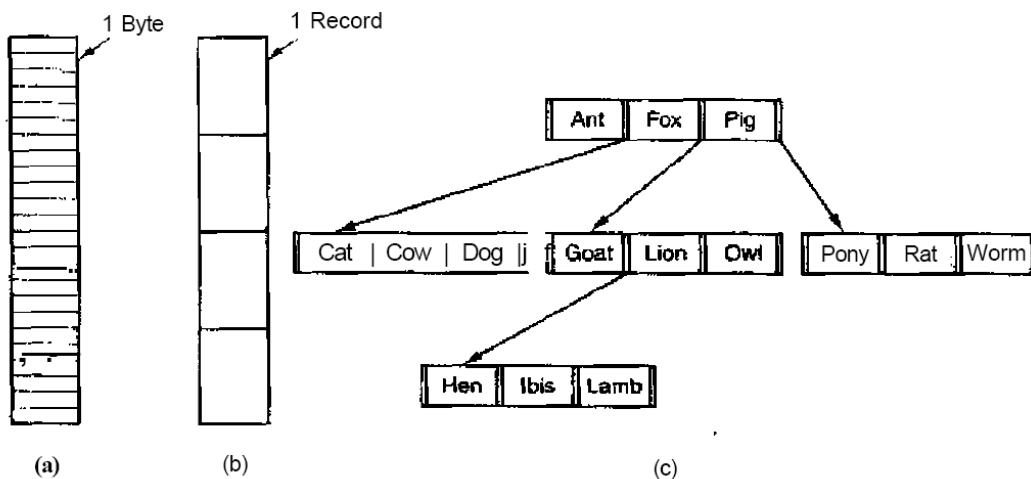


Figura 6-2. Të tre llojet e file-ve. (a) Sekuese bitesh. (b) Sekuese regjistrimesh. (c) Pema structure.

Të kesh kujdesjen e sistemit operativ mbi file, nuk është asgje tjetër vecse, një sekuese e caktuar bitesh qe lejon një fleksibilitet të madh. Përdoruesit e programeve mund të vendosin cfaredo qe deshirojne ne file-t e tyre dhe ti emertojne ato ne çdo menyre të deshiruar. Sistemi operativ nuk të ndihmon ne ketë gje, por gjithashtu as nuk ta ndalon. Për përdoruesit qe deshirojne të manipulojne dicka, pjesa ne vazhdim është e rendesishme.

Hapi i pare i strukturimit tregohet ne fig. 6-2(b). Ne ketë model, një file trajtohet si një sekuence rekordesh me gjatësi fikse. Qendra kryesore e ideve për fillimin e një sekuence rekordesh , është ideja qe operacioni i leximit kthen një rekord dhe operacioni i shkrimit e mbishkruan një rekord.

Shume (mainframe), sisteme operative të bazohen ne file sistem, me file te n, dhe keto file të ndertuara nga rekorde me nga 80 karaktere, qe ne fakt jane imazhe te kartave. Keto sisteme suportojne edhe file të ndertuara nga rekorde me 132 karaktere, qe jane të destinuar për printerin. Programet lexojne ne hyrje, daljen e 80 karaktereve dhe e shkruan ne dalje me 132 karaktere, edhe pse 52 te fundit mund te jene hapesira boshe. Pra, sistemi funksionon ne kete menyre.

Tipi i tretë, tregohet e figuren.6-2(c). Ne ketë organizem, një file është i ndare ne një peme rekoresh, jo domosdoshmerishtë të gjithe të të njëjtës gjatësi, disa prej tyre përbajne një fushe çeles ne një pozicion te fiksuar të rekordit. Pema është e renditur ne baze të fushave çeles, ne menyre qe të ketë një kerkim të shpejtë të një çelësi ne vecanti.

Funksioni krysor ketu, nuk është ai qe të kemi rekordin tjetër pasardhes, por qe është dëshume e mundeshme, por funksioni është qe të kemi rekordin me një çeles të veçantë. Për file-in zoo të figures 62(C), mund ti drejtohem i sistemit për të patur një record, celsi i te cilit per shembull është pony, pa u preukupuar për vendodhjen e saktë të tij ne file. Gjithashtu, një rekord i ri mund të shtohet ne file, nga vetë sistemi operativ, dhe nuk eshte perdoruesi ai qe vendos se ku do te qendroje. Ky tip file është shume i ndryshem nga ai i biteve të pastrukturuar ne UNIX dhe ne Windows, por është shume i përdorur ne kompjuterat mainframe.

6.1.3 File Types (Tipat e File-ve)

Shume sisteme operative suportojne tipe te ndryshme file-ash. UNIX dhe Window, për shembull, kane file dhe direktori të rregullta, UNIX ka gjithashtu karaktere dhe bllokues special për file-t. Filet e rregullt Jane të vetmit qe përbajne informacion për përdoruesit. Të gjithe file-t e figures 6-2 Jane të rregullt. Direktoritë Jane file sistem për mirembajtjen e struktura të fileve sistem. Ne do të studojme direktoritë e ndermjetme. Karateret speciale të file-ve Jane të realizuar për input/output dhe përdoren tek modeli serial I/O devices ashtu si dhe terminalt, printerat dhe rrjeti. Bllokuesit special të file-ve përdoren tek modelet e disqeve.

Filet e rregullt Jane përgjithesisht ASCII ose binare. Filet ASCII Jane të ndertuar nga rreshta karakteresh text. Ne disa sisteme secili rresht mbaron me një karakter kthimi. Ne të tjere përdoret një rresht i ri. Disa sisteme i përdorin të dyja llojet. Rrreshtat nuk duhen të jene të gjithe ne të njejten gjatësi. Avantazhi i madh i file-ve ASCII është qe mund të jene të dukshem dhe të printueshem dhe mund të modifikohen me çdo lloj editori text. Gjithashtu, mund të sherbejne si input i ndonjë programi.

Filet binare ose numerike, qe nuk Jane file me karaktere ASCII, zakonisht kane një struktura të brendeshme te dukshme për programet qe i përdorin. Për shembull, ne figuren 6-3(a), shohim një file binare të thjeshtë, të ekzekutohet nga një version i UNIX. Edhe pse file është një sekunce bitesh, sistemi operativ e ekzekuton atë file vetëm ne se

jane të njohur nga sistemi si file të ndonjë programi. Kjo behet ne 5 seksione: koka, teksi, data, rilokalizimi i briteve dhe simboli tabelar.

Koka fillon me një të ashtu quajtur numer magjik, i cili e njeh atë si një file të ekzekutueshem për të ndaluar aksidente ne ekzekutim të një file qe nuk është i formatit të duhur. Pastaj kemi dimensionet e ndryshme të pjeseve të file, adresa e marre ne fillim, ne të cilën fillon ekzekutimi i tij dhe pastaj disa nga flamujt e gjendjeve. Keta ngarkohen ne memorje duke përdorur transferimin bit për bit. Symboli tabelar është përdorur për testimin.

Shembulli jone i dytë i një file binar është një arkive, po nga UNIX. Konsiston ne një përbledhje e procedurave te librarise (moduleve) të kompiluara, por jo të linkuara. Secila prej tyre është e përfaqesuar nga nga koka e cila përmban emrin e tij, krijimin e të dhenave, kodin e sigurise dhe madhesine.

Çdo sistem operativ duhet të jetë ne gjendje të njohe të paktën një lloj file. Por disa sistemi njojin shume lloje. Sistemet e vjetra si TOPS-20, shkuan shume larg ne ekzaminim e kohes se krijimit kur çdo file fillon ekzekutimin. Ne se nuk do të ishte keshtu, rikompilohet edhe njëhere koka e filet. Ne UNIX programi është i ndertuar ne shell execute. Prapashtesat e fileve jane të detyrueshme për tu shkruar, keshtu qe sistemi operativ mund të interpretoje se cili program eshte perfshuar dhe nga cili burim vjen.

Duke patur file pa gabime si ato të problemave me lartë, edhe sikur përdoruesi të mos të dije asgje nga projektimi i sistemit operativ, nuk gabon. Duke e konsideruar si një shembull, sistemi ne të cilin një program krijon një file si output qe ka si prapashtese *.dat

Ne se një përdorues shkruan një program qe mund të lexoje a.c, e kthen atë ne indentitet standart dhe shkruan filen e transformuar si output, output-i do të jetë po prape *.dat. Ne se përdorusi do ta kompiloje atë, sistemi do ta refuzoje sepse file është me prapashtesen e gabuar. Për ta kopjuar nga file.dat ne file.c, do të trajtohet si e pasaktë nga sistemi për të mos bere realitet gabimet logjike të përdoruesve.

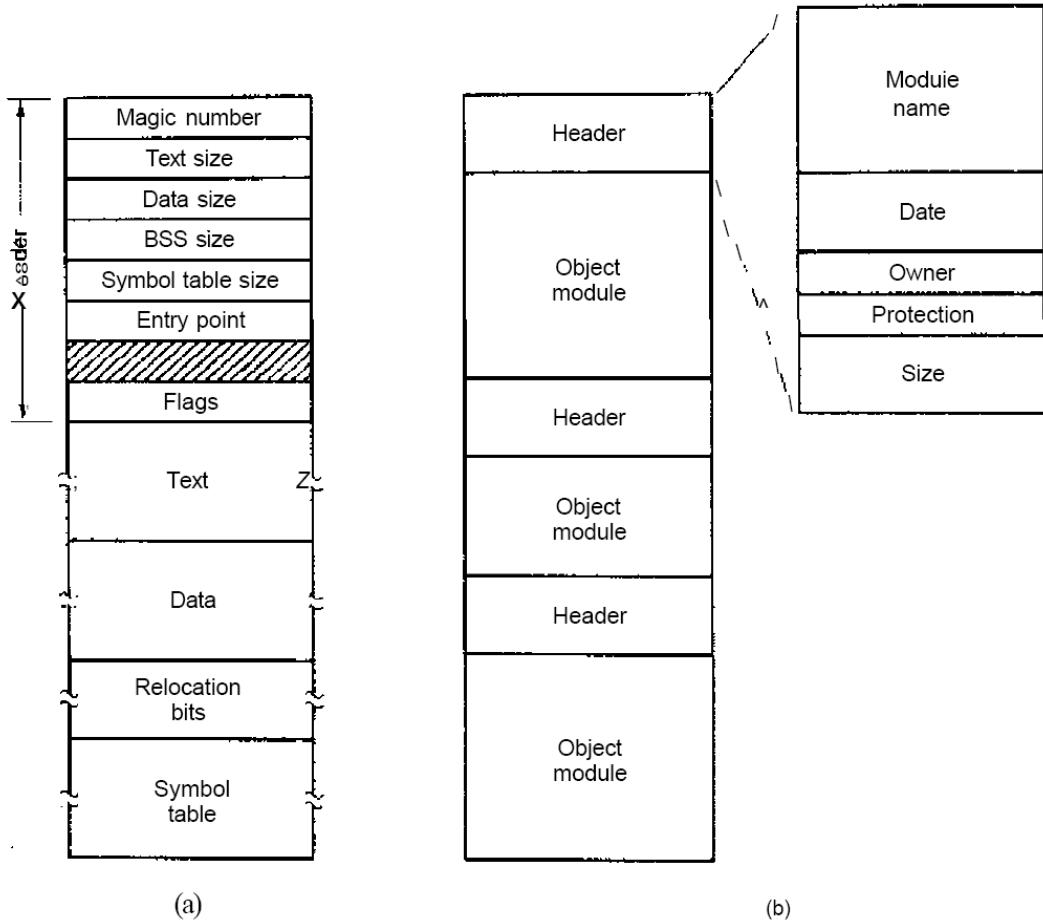


Figura 6-3. (a) File i ekzekutueshem. (b) File arkive.

6.1.4 File Access (aksesimi i File-ve)

Sistemet e para operative të siguronin vetem një lloj aksesimi të file-ve: akses sekuencial. Ne keto sisteme, një proces mund të lexoje të gjitha bitet ose rekordet të një file të rregullt, duke filluar nga koka, por nuk mund ti anash-kalonte pa i lexuar me rregull. Filet sekuencial mund të riorganizohen, keshtu qe ata mund të lexohen sa here qe të jetë e nevojshme. Filet sekuenciale, jane të nevojshem kur kemi memorizim ne disket, sesa ne disk.

Disku kur është i vene ne pune për memorizimin e file-ve, është e mundur të lexohen bitet ose rekordet të një file jashte rregullit ose të aksesosh registrat me ane të çelësit, sesa nga pozicioni.

Bitet ose file-at ne te cilin rekordet mund të lexohen sado i rregullt qoftë file, jane të quajtur të aksesueshem rastesisht. Jane të kërkueshem nga shume aplikacione. Aksesimi rastesor është thelbesor për shume aplikacione, për shembull, sistemet me baza të

dhenash. Për shembull, sikur ne një aeroport të marre ne telefon një klient për të prenotuar një udhetim të caktuar, baza e të dhenave duhet ta kerkonte e te gjeje vetëm prenotimet për ketë udhetim të caktuar, pa u dashur të shikoje me mijera regjistrime të tjera qe nuk kane lidhje me udhetimin ne fjale.

Dy Jane metodat e përdorura për të përcaktuar se nga duhet të filloje leximi. I pari, ne çdo veprim leximi, kthen pozicionin e filet për të filluar leximin. I dyti, një veprim i vecantë është parashikuar për të fiksuar pozicionin e saktë. Pas kerkimit, file mund të lexohet ne menyre sekuenciale duke filluar nga pozicioni përkatës. Ne disa sisteme operative te mainframeve të vjetër, file-t Jane të klasifikuar nga sekuencialiteti ose nga aksesimi rastesor, qe nga momenti qe Jane krijuar. Kjo gje e lejon sistemin të përdore teknika të ndryshme për memorizimin e seciles klase të file-ve. Sistemet operative moderne nuk e bejne ketë klasifikim, të gjithe file-t e tyre Jane automatikisht me aksesim rastesor.

6.1.5 File Attributes (atributet e file-ve)

Çdo file ka një emer dhe të dhenat vetjake të tij. Gjithashtu, të gjithe sistemet operative Jane të familjarizuar me çdo informacion të file-ve, data dhe ora kur është krijuar file dhe përmasat e tij ne kujtese. Ne do ti quajme keto informacione ekstra të file, me një emer të vetëm, atributet e tij. Lista e atributeve varion dhe ndryshon nga sistemi ne sistem. Tabela e figures 6-4, na tregon disa nga keto ndryshime. Asnjë sistem operativ nuk i përmban të gjitha ato, por secila prej tyre është e pranishme ne një sistem operativ.

Kater atributet e para kane lidhje me mbrojtjen të file-t, dhe tregojne se sa i aksesueshem është ky file. Të gjitha tipet e regjimeve Jane të mundeshme, disa prej të cileve do ti studjme me vone. Ne disa sisteme operative, përdoruesit duhet të ketë një fjale kalim për aksesimin e file-it, ne ketë rast, fjale-kalimi duhet të jetë një nga atributet.

Flamujt e gjendjeve Jane bite qe kontrollojne fusha të vecanta të karakteristikave. Filet e fshehur, për shembull, nuk Jane të dukshem nga përdoruesit. Flamujt e gjendjeve Jane si një arkiv të cilët ruajne gjurmë të një ekzekutimi të mundshem të file-t ne backup. Programi i backup-it, pastron dhe sistemi operativ e kthen ate, pavaresisht se kur eshte ndryshuar file. Ne ketë menyre programi i backup mund te na tregoje se cili file ka nevoje per backing up. Flamujt e përkohshem të një file, lejon qe file te shenohet per fshirje automatike kur procesi qe ke ka krijuar ka mbaruar. Gjatësia e rekordit, pozicioni i celesit, gjatësia e celesit të fushave Jane të pranishme vetëm ne filet, rekordet e te cilave mund të kerkohen nepermjet një celesi. Keta të fundit jepin informacionin e duhur për të gjetur celesat.

Ne te shumten e hereve ngelen gjurmet kur eshte krijuar file, kur eshte aksesuar per here te fundit dhe kur eshte modifikuar. Keto Jane të nevojshme për qellime të ndryshme. Për shembull, një file burim, i cili është modifikuar pas krijimit të filet objekt korrespondues, duhet të rikompilohet përseri.

Dimensionet aktuale tregojne qe sesa i madh është file aktualisht. Disa sisteme operative ne mainframe të vjetër, kerkonin qe madhesia maksimale e filet te specifikohej kur file ishte duke u krijuar, ne menyre qe te lejonte sitemin operativ, te krijonte qe ne fillim hapesiren e duhur ne memorje. Sistemet operative te workstation dhe PC Jane mjaftë inteligjente për ta bere ketë gje vet, pa pasur nevojen e një funksioni të tille.

Atributi	Kuptimi
Siguria	Kush mund të aksesoje file dhe ne c'menyre
Fjale-kalimi	Fjale-kalim i nevojshem për të aksesuar file-n
Krijuesi i celesit	Celesi i personit i cili e ka krijuar file-n
Pronari	Pronari aktual
Read-only flag	0 për lexim/shkrim; 1 vetëm për lexim
Hidden flag	0 për normal; 1 për mos ta shfaqur ne list
System flag	0 për file normal; 1 për file sistem
Archive flag	0 është ekzekutuar backup; 1 duhet të ekzekutohet backup
ASCII/binary flag	0 për file ASCII; 1 për file binare
Random access flag	0 vetëm për aksesim sekuencial; 1 për aksesim rastesor
Tëmporary flag	0 për normal; 1 për file të fshire kur mbaron proçesi
Lock flags	0 për të pa-kycurit; jo-zero për të kycurit
Record length	Numri i biteve ne një rekord
Key position	Celesi i brendshem i secilit rekord
Key length	Numri i biteve ne fushen e celesit
Creation time	Data dhe ora e krijimit të file-t
Time of last access	Data dhe ora e aksesimit të fundit të file
Time of last change	Data dhe ora e modifikimit të fundit të file
Current size	Numri aktual i biteve ne file
Maximum size	Numri maksimal i biteve ne file

Figura 6-4. Disa atribute të mundeshme të një file të caktuar.

6.1.6 File Operations (Operacionet e Fileve)

Filet ekzistojne për të memorizuar informacion dhe qe mund të rikuperohet me vone. Sisteme të ndryshme përdorin operacione të ndryshme për të memorizuar dhe rikuperuar me vone informacionin. Tani do të japim shembuj të sistemeve qe therrasin si proçes rikuperimin e një file :

1. Krijimi. File krijohet pa asnë të dhene. Qellimi i therritjes se tij është të lajmeroje qe file është ne ardhje dhe duhet të memorizohen disa nga atributet kryesore të tij.
2. Fshirja. Kur file nuk nevezitet me, ai duhet të fshihet qe të liroje hapesiren e memorjes. Egziston një thirrje sistem për ketë qellim.
3. Hapja. Përpara se ta përdorim një file, është e domosdoshme qe një proçes ta hapi atë. Qellimi i hapjes përpresa se ta përdorim është qe sistemi të memorizoje disa nga atributet e tij dhe të përcaktoje se ku ndodhet fillimi i tij ne memorje, dhe për ta therritur atë me vone për ndonjë veprim të mundeshem.
4. Mbyllja. Kur të gjitha aksesimet e file-it kane mbaruar, atributet e tij dhe adresa ne disk nuk jane me të nevojshme si informacione, sepse file duhet të mbylljet për të liruar memorjen ne tabele. Shume sisteme e realizojne kete duke imponuar një numer te caktuar file-sh qe mund te hapen ne një proces. Një disk është i shkruar ne blloqe, dhe mbyllja e një file, sjell dhe heqjen e te dhenave qe jane shkruar ne bllokun e fundit, edhe pse blloku mund të mos jetë i mbushur plot.
5. Leximi. Të dhenat lexohet nga file-i. Natyrisht, bitet vijne nga pozicione të sakta aktuale. Proçesi qe e therret filen duhet të përcaktoje sasine e të dhenave dhe gjithashtu mund ti vendosi ato ne një buffer.
6. Shkrimi. Të dhenat jane të shkruara ne file, zakonisht ne pozicionin aktual. Ne se pozicioni aktual është ne fundin e file-it, madhesia e file-it rritet. Ne se pozicioni aktual është ne qender të file-it, të dhenat egzistuese, mbi-shkruhen dhe humbasin per gjithmone.
7. Append. Ky proçes thirrje është një forme e të shkruajturit e limituar. Mundet vetëm të shtosh të dhena ne fund të file-it. Sistemet qe të ofrojnë një sasi thirrjesh të vogel, zakonisht nuk e përdorin Append, por shume sisteme ofrojnë menyre te ndryshme per ta realizuar ate, dhe keta sisteme ndonjehere kane Append.
8. Kerkimi. Për të aksesuar file të rastesishem, është një metode e detyrueshme për të specifikuar se nga do të meren keto të dhena. Një metode e zakonshem është një therje sistem, e cila kerkon qe pointuesi i file-t te specifikoje vendin ne file. Pas ketij veprimi i kerkimit është i kompletuar, dhe të dhenat mund të lexohen ose të shkruhen tek ai pozicion.
9. Marrja e atributeve. Proçeset shpesh kane nevoje për sa me pak atribut për të bere punen e tyre. Për shembull, programi qe bën UNIX është gjeresisht i përdorur për të menaxhuar projekte të zhvillimit të programeve, duke u bazuar ne shume burime file-ash. Per te realizuar kete, duhet te shikoje atributet, emrin dhe kohen e modifikimit.

10. Vendosja e atributave. Disa prej atributave jane të përdoruesit dhe mund të ndryshohen vetëm ne se file është krijuar. Kjo thirrje e sistemit qe bën të mundur modulet e sigurise të të dhenave të file-it është një shembull shume i qartë. Pjesa me e madhe e flamujve hyjne ne ketë kategori.
11. Ri-emertimi. Shpesh ndodh qe përdoruesi duhet të ndryshoje emrin e një file qe është krijuar me pare. Kjo thirrje e sistemit qe e bën të mundur kete, nuk është gjithmon e nevojshme sepse zakonisht mund të jetë i kopjuar ne një file të ri me emer të ri dhe file i vjetër është fshire.

6.1.7 Një shembull programi i cili përdor thirrjet sistem.

Ne kete seksion, do te shqyrtejme, një program te thjeshtë ne UNIX, qe kopjon një file nga ai burim ne një file destinacion. Kjo është e shprehur ne figuren 6-5 . Programi ka funksionet minimale dhe shume gabime, por jep një ide te qarte sesi thirrjet sistem lidhen me file qe Jane ne pune. Programi është një “copy-file”, qe mund edhe të therritet edhe nga linja e komandave si për shembull:

copyfile abc xyz emri i filet burim, emri i filet destinacion.

Për ta kopjuar file abc tek file xyz, ne se ekziston ky file ai do të mbishkruhet, ne rast të kundert do të krijohet.

“#include” i katërt qe përdoret afer pjeses se lartë të programit, realizon një numer të madh përcaktimesh dhe prototipe të funksioneve të ndryshme qe duhet te perfshihen ne program. Reshti tjetër është një prototip për një funksion kryesor, qe eshte kerkuar nga ANSI C, por nuk është e rendesishme për qellimet tonë.

“#define” i pare është përcaktimi i një macro qe përcakton stringat BUF_SIZE si një macro qe bën pjesë ne numrin 4096. Programi të lejon të lexosh dhe të shkruash ne bloqe prej 4096 bitesh. Do të shfrytëzoje emrat e konstanteve.

“#define” i dytë mund të aksesoje file-n e hyrjes. Programi kryesor i thirrur ka dy argumenta , argc dhe argv. Keto Jane të dhene nga sistemi operativ kur programi thirritet. I pari tregon se sa stringa ka ne rreshtin e komandave qe ka thirrur programi, përfshire ketu edhe emrin e programit. Kurse i dyti është një matrice pointer-ash nepër argumenta të cilët mund të kene keto vlera:

argv[0] = "copyfile"

argv[1] = "emri i filet burim"

argv[2] = "emri i filet destinacion"

Kemi deklaruar gjithesej pese variabla, dy të paret in_fd dhe out_fd mbajne të shtypur përshkrueshit të file-ve kur një file hapet. Rd_count dhe wt_count janë bite të kthyera nga leximi e shkrimi i thirrjeve të sistemit. Buffer-I i fundit është përdorur për të mbajtur të dhenat e marra për shkrim.

```
#include <sys/types.h> /* include necessary header files */
#include <fcntl.h>
```

```

#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]); /* ANSI prototype */
#define BUF_SIZE 4096 /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700 /* protection bits for output file */
int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];
    if(argc != 3) exit(1); /* syntax error if argc is not 3 */
    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY); /* open the source file */
    if(in_fd < 0) exit(2); /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
    if (out_fd < 0) exit(3); /* if it cannot be created, exit */
    /* Copy loop */
    while(TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if(rd_count <= 0) break; /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
    }
    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0) /* no error on last read */
        exit(0);
    else
        exit(5); /* error on last read */
}

```

Figura 6-5. Një program i thjeshtë për të kopjuar një file.

Argc kerkon argumentat dhe shikon ne se është 3 apo jo. Ne se jo atëhere ajo ekziston ne gjendje kodi baras me 1. Cdo gjendje te kodit ndryshe nga 0, nenkupton qe ka ndodhur një gabim. Kur ne kerkojme të hapim file-in burim dhe krijojme filen destinacion, atëhere ne se file burim është plotësisht i hapur, sistemi i bashkangjit një numer të vogel të plotë tek in_fd, për të specifikuar filen. Nenthirrjet mund ta inkludojne ketë numer të plotë keshtu qe sistemi e di cili eshte file qe ai kerkon. Ne menyre të ngjashme, ne se file destinacion është plotësisht i krijuar, atëhere out_fd merr një vlerë për ta specifikuar atë. Argumenti i dytë për të krijuar, vendos modulin e sigurise. Ne se si ata të hapur ose të krijuar deshtojne, atëhere file konrrespondues vendoset ne 1 dhe programi mbaron me një gabim ne kod.

Tani kemi ciklin e kopjimit. Ai fillon me leximin ne 4 KB të të dhenave të buffer-it. Thirret procedura lexo, e cila invokon thirrjen lexo të sistemit. Paramatri i pare indetifikon filen, i dyti e con ne buffer, dhe i treti tregon sa bite ka për të lexuar. Vlera qe

i bashkangjitet rd_count jep numrin e biteve të lexuara aktualisht. Normalisht, do të jetë 4096 e pritur, ne se jane të vendosur të gjithe bitet ne file. Kur kemi fundin e filet do të jetë 0 ne se rd_count është gjithmone 0 ose negative, atëherë kopjimi nuk mund të vazhdoje keshtu qe kemi një break ne ekzekutim.

Thirja për shkrim output-et nga buffer-i për ne filen destinacion, bën qe parametri i pare të përcaktoje filen, i dyti ta coj ne buffer dhe i treti të tregoje se sa ka shkruar dhe lexuar paralelisht. Veme re qe numerimi i biteve është numri aktual i biteve të lexuara, jo BUF_SIZE. Kjo është e rendesishme sepse leximi i fundit nuk do të ktheje 4096, por do të shtohet me 4KB.

Kur file ne hyrje është përdorur nga proçesi, atëherë thirja e pare parashikon fundin 0 tek rd_count, i cili do ta beje programin të dale nga cikli. Ne ketë rast dy filet Jane të mbyllur dhe programi do të mbyllët normalisht.

6.1.8 Memory-Mapped Files

Sisteme të ndryshme kane ofruar një menyre për të hartëzuar filet ne hapesiren e adresave të një proçesi ne ekzekutim.

Nga ana konceptuale mund të imagjinojme pranine e dy thirrjeve të ndryshme të sistemit operativ, harteziur dhe i joharteziuar. I pari jep një emr të filet dhe një adresë virtuale qe bën qe sistemi operativ, hartezon një file ne hapesiren e adresave virtuale.

Për shembull, Supozojme se një file F qe ka gjatësine prej 64KB, është i hartëzuar ne adresa virtuale duke filluar nga 512K. Keshtu qe çdo instruksion makine qe lexon përbajtjen e biteve ne 512K, vine 0 bit tek file-i. Ne të njëjtën menyre një adresë e shkruar nga 512K + 21000 bite 21000 modifikon byte 21000 te file. Kur proçesi mbaron, file-i i modifikuar, lihet ne disk si të ishte i ndryshuar nga një kordinim kerkim-shkrim nga thirrjet e sistemit.

Cfare ndodh ne te vertete eshte se tabelat e brendeshme te sistemit jane ndryshuar për ta bere filen të behet suport për të mbajtur memorjen 512K të lokalizuar ne 576K. Keshtu një lezim prej 512K, shkaton një page fault, duke e cuar ne faqen 0 te filet. Po ashtu qe një lexim nga 512K + 1100 shkakton një page fault, duke sjelle ne faqe permbajtjen e asaj adresë, pas se ciles shkrimi ne memorie mund te ndodhi. Ne qofte se kjo faqe eshte menjanuar nga algoritmat e zevendesimit te faqeve, ajo eshte shkruar perseri, ne vendin e duhur ne file. Kur proçesi mbaron, të gjitha faqet e hartëzuara, faqete modifikuara, u kthehen fileve te tyre. Hartëzimi i fileve, funksionon me mire ne një sistem qe suporton segmentimin.

Ne një sistem të tille çdo file mund të hartëzohet ne segmentin e vet, ne menyre qe bitet e file-it të jene edhe bitet e segmentit.

Figura 6-6(a), na tregon një proçes qe ka dy segmente me tekst dhe te dhena. Supozojme se ky proçes i kopimit të file-it si dhe programi i meparshem, hartëzon me pare filen burim ne një segment, me pas krijohet një boshlik segmentar dhe hartëzimi i file-it destinacion vendoset aty.

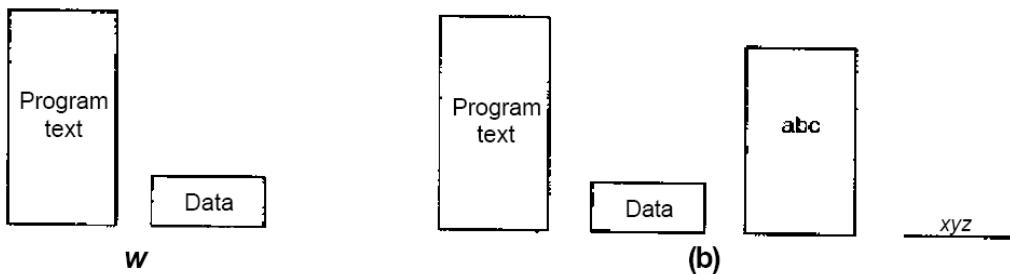


Figure 6-6. (a) Një proçess segmentar përparr Hartëzimit të file-ve ne hapesiren e adresave. (b) Proçesi pas Hartëzimit ekziston një file, brenda një segmenti dhe krijon një segment të ri.

Proçesi kopjon segmentin burim ne segmentin destinacion duke përdorur një cikel të zakonshem kopjimi. As thirrjet lexim dhe shkrim të sistemit nuk jane të nevojshme, dhe kur gjithçka eshte kryer, ai mund të ekzekutoje thirrjen c'hartëzimin të sistemit të levize filen nga hapesira e adresave e pastaj të dali nga programi. File output *jrvz* nuk do të ekzistoje deri sa të krijohet ne menyre të drejtpërdrejtë.

Nepërmjet hartëzimit të file-ve, eleminojme nevojen e I/O dhe kjo mund ta beje programin me të lehtë. Ka disa probleme vetjake:

1. është e veshtire për sistemin të njohe gjatësine e saktë të file-it output *jrvz*.
2. mund të na tregoj numrin maksimal të faqeve të shkruara, por ne asnjë menyre të njohe se sa bite jane shkruar ne të.
3. supozojme se programi përdor faqen 0, por pas ekzekutimit edhe bitet do të jene 0.
4. File hartëzohet nga një proçes por hapet per tu lexuar nga një proçese tjetër. Ne qofte se procesi i pare ka modifikuar një faqe, ky ndryshim nuk do te reflektobet ne file qe eshe ne disk, deri sa faqja te jete hequr. Sistemi duhet te kete shume kujdes per tu siguruar qe dy proceset nuk e shohin versionin kontrakditor te file.
5. file mund të jetë me i gjatë se segmenti ose me i gjatë se hapesira e adresave virtuale. E vetjma menyre e zgjidhjes është qe sistemi te hartezoje një pjese te file, sesa te gjithe file.

6.2 DIREKTORIT

Për të ruajtur ose mbajtur pjese nga skedare, sistemet e skedareve kane direktori ose foldera (dosje), të cilat ne shume sisteme janë vet skedaret. Ne ketë paragraf do të diskutojme për direktoritë, organizimin e tyre, karakteristikat dhe operacionet e ndryshme qe mund të ekzekutohen ne to.

6.2.1 Sistemet me një nivel direktorie

Forma me e thjeshtë e sistemeve të direktorive është ai me një direktori qe përbën të gjithe skedaret. Nganjëherë quhet edhe direktoria rrrenjë, por meqenese është e vetmja

direktori, emri nuk ka shume rendesi. Ne kompjuterat e hershem personal ky system ishte i zakonshem sepse përgjithesisht kishte vetëm një përdorues. Megjithatë cuditërisht superkompjuteri i pare ne botë CDC 6600 kishte një direktori të vetme për gjithe skedaret edhe pse përdorej nga shume përdorues ne të njëjtën kohe. Kjo gje behej pa dyshim për të mbajtur dizajnin e software-it sa me të thjeshtë.

Një shembull i sistemit me një direktori është dhene ne figure 6-7. Ketu direktoria përmban 4 skedare. "Zotërueshit" e skedareve jane treguar ne figure, jo emrat e skedareve (sepse na interesojne ato qe i përbajne, gje për atë qe do flasim me poshtë). Avantazhi i kesaj skeme qendron tek thjeshtësia e saj dhe tek aftësia për të lokalizuar shpejt skedaret, pasi ne fund të fundit ekziston vetëm një vend për të kerkuar.

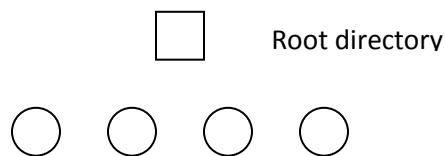


Figura 6-7. Sistem me një nivel direktorie qe përban 4 skedare qe zotërohen nga njerez të ndryshem A , B

Problemi ne të pasurin vetëm një direktori ne një sistem me shume përdorues, është se përdorues të ndryshem mund të përdorin aksidentalish të njëjtin emer për skedaret e tyre. Për shembull ne qoftë se përdoruesi A krijon një skedar të quajtur *mailbox* dhe me pas përdoruesi B krijon gjithashtu një skedar me të njëtin emer *mailbox*, skedari i B-se do të mbishkruhet mbi atë të A-se. Për pasoje kjo skeme nuk përdoret me ne sistemet me shume përdorues, por mund të përdoret për një sistem të vogel për shembull ne një sistem ne një makine qe është dizenuar për të ruajtur profilet për një numer të vogel drivers.

6.2.2 Sistemet me dy nivele direktorish

Për të shmangur konfliktet të shkaktuara nga përdorues të ndryshem qe zgjedhin të njëtin emer për skedaret e tyre, hapi i dytë është ti japim çdo përdoruesi një direktori private. Ne ketë menyre emrat e zgjedhur prej një përdoruesi nuk interferojne me emrat e zgjedhur nga një përdorues tjeter dhe nuk ka problem, në qoftë se i njëjti emer mund të shfaqet ne me shume se një direktori. Ky dizajn con tek sistemi i paraqitur ne figuren 6.8. Ky dizajn përdoret me shume ne kompjuterat me shume përdorues ose ne një rrjet të thjeshtë kompjuterash personal qe punojne me një server të përbashket ne një rrjet lokal.

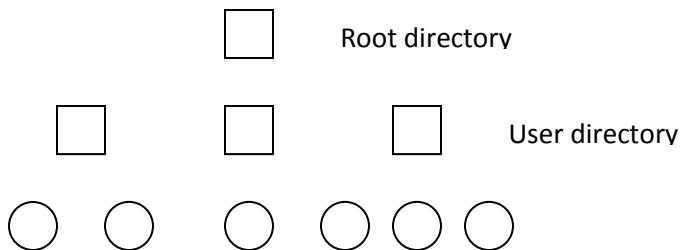


Figura 6-8. Një sistem direktorie me dy nivele. Shkronjat tregojne zotëruesit e direktorive dhe skedareve.

Ajo cfare nenkuptohet ne ketë dizajn eshte qe, kur përdoruesi përpinqet të hape një skedar sistemi e di cili përdorues është, ne menyre qe të dije se ne cilen direktori të kerkonte. Për pasoje një procedure hyrese është e nevojshme, ne të cilën përdoruesi specifikon emrin identifikues ose hyres, dicka qe nuk kerkohet ne sistemet me një nivel direktorie.

Kur ky sistem zbatohet ne menyren e tij me themelore, përdoruesit kane të drejtë hyrje vetëm ne skedaret e direktorive të veta. Megjithatë një shtese e vogel është qe i lejon përdoruesit të hapin skedaret e përdoruesve të tjere duke dhene disa të dhena për skedaret e të cilit po hap. Keshtu për shembull:

Hap (" x ")

Mund të jetë komanda ose thirrja për të hapur skedarin x të direktorise se përdoruesit dhe

Hap (" nancy / x")

Mund të jetë thirrja për të hapur direktorine e një përdoruesi tjetër, Nancy.

6.2.3. Sisteme me shume direktori (hierarkike)

Hierarkia me dy nivele shpreh konfliktet e emrave ndermjet përdoruesve, por nuk është e kenaqshme për përdoruesit me një numer shume të madh skedaresh. Edhe kompjuterat personal për një përdorues të vetëm nuk është i përshtatshem. Shpesh përdorues të ndryshem ndjejne nevojen apo kane deshire ti grupojne skedaret e tyre sipas një logjike. Për shembull, një profesor deshiron të ketë një koleksion skedaresh qe të gjitha bashke të formojne një liber qe po shkruan për një kurs të vecantë, një koleksion të dytë skedaresh qe përbajne programet e studentëve të pranuar për një kurs tjetër, një grup të tretë skedaresh qe përbajne kodet e një sistemi të avancuar shkrues-kompilues qe ai po nderton, një grup të katërt skedaresh qe përban projekt-propozime, si dhe një skedar tjetër për postën elektonike, oraret e takimeve, letrat qe po shkruan, lojra e keshtu me radhe. Ne një fare menyre është e nevojshme qe keto skedare të grupohen ne një menyre të thjeshtë, të zgjedhur nga vetë përdoruesi. Për ketë vjen ne ndihme një hierarki e përgjithshme (një peme direktorish). Me ketë lloj përafrimi çdo përdorues mund të ketë aq direktori sa ka nevoje ai, ne menyre qe skedaret të grupohen ne menyrë sa me të natyrshme. Ky lloj përafrimi është dhene ne figuren 6-9. Ketu direktoritë A, B dhe C, nen

direktorine rrenjë i përkasin secila një përdoruesi të ndryshem, dy prej të cileve kane krijuar nendirektori për projektet të ndryshme qe ato po punojne.

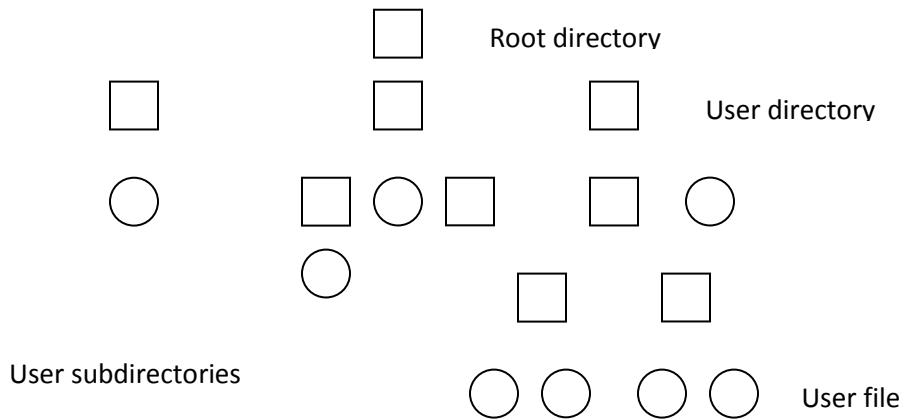


Figura 6-9. Një sistem direktorish hierarkike

Mundesia qe kane përdoruesit për të krijuar një numer arbitrar nendirektorish ofron një mjet strukture të fuqishem për përdoruesit, ne menyre qe të organizojne punen e tyre. Për ketë arsy, afersisht të gjithe sistemet modern të skedareve jane të organizuara ne ketë menyre.

6.2.4 Emrat e path-it

Kur sistemi i skedareve është organizuar si një peme direktorish, ne një fare menyre është e nevojshme të specifikohen emrat e skedareve. Zakonisht dy lloj metodash përdoren. Ne metoden e pare çdo skedari i është dhene një **emer path-i**, i quajtur **path absolut** qe përmban path-in nga direktoria rrenjë deri ne skedar. Si për shembull, path-i `/usr/ast/mailbox` nenkupton qe direktoria rrenjë përmban një nendirektori `usr`, e cila përmban një tjetër nendirektori `ast`, e cila nga ana e saj përmban skedarin `mailbox`.

Path-et absolutë të emrit fillojne gjithmone nga direktoria rrenjë dhe jane unike. Ne UNIX komponentët e trajektores jane të ndara me /. Ne Windows ndaresi është \. Ne MULTICS ndaresi është >. Domethene, i njëjtë path absolut i emrit mund të shkruhej ne keto tre sisteme keshtu:

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox
MULTICS	>usr>ast>mailbox

Nuk ka rendesi se cfare karakteri përdor, mjafton, ne qoftë se karakteri i pare i path-it të emrit është ndaresi atëhere kemi të bejme me path absolutë.

Lloji tjetër është **path-i relative i emrit**. Ky përdoret ne lidhje me konceptin e **working directory** (direktoria e punes), e quajtur gjithashtu **direktoria aktuale**. Një përdorues

mund të përcaktoje një direktori si working directory aktuale ne të cilen emrat e path-it qe nuk fillojne nga directoria rrenjë merren relativisht ne workind directory.

Për shembull, ne qoftë se working directory është */usr/ast* atëherë skedari path-i absolutë i të cilit është */usr/ast/mailbox* mund ti referohemi thjesht si *mailbox*. Me fjale të tjera komanda ne UNIX

```
cp/usr/ast/mailbox/usr/ast/mailbox.bak
```

dhe komanda

```
cp mailbox mailbox.bak
```

bejne ekzaktësisht të njëjtën gje, ne qoftë se working directory është */usr/ast*. Forma relative shpesh është me i përshtatshem por bën të njëjtën gje si forma absolute.

Disa programe kane nevoje qe të kene hyrje ne disa skedare specifike pa marre parasysh se cfare working directory është. Ne ketë rast duhet përdorur gjithmone path-i absolutë i emrit. Për shembull, një spelling checker mund të ketë nevoje qe të lexohet */usr/lib/dictionary* për të punuar me të. Duhet të përdoret path-i absolutë i emrit ne ketë rast, sepse nuk e di se cila working directory mund të jetë kur të thirret komanda. Path-i absolut i emrit funksionon gjithmone pavaresisht se cila working directory është.

Sigurisht ne qoftë se spelling checker ka nevoje për një numer të madh skedaresh nga */usr/lib*, një mundesi tjetër mund të ishte qe ne fillim të ndryshoje direktorine ku po punon (working directory), ne */usr/lib* dhe pastaj të përdore thjesht *dictionary*, si parametrin e pare për ta hapur. Për thjeshtësi, ndryshimi i working directory qe sigurisht dihet se ku ndodhet ne pemen e direktorive, mund të përdoret path-i relative.

Çdo proces ka working directory e vet keshtu qe, kur një proces ndryshon working directory dhe me vone kthehet përseni, asnë nga proceset e tjera nuk preken apo demtohen, dhe asnë gjurmë e ndryshimeve të bera nuk mbetet ne sistemin e skedarit. Ne ketë menyre është gjithmone e sigurt ndryshimi i working directory për një proces, kurdohere qe është e përshtatshme. Nga ana tjetër, ne qoftë se *library procedure* ndryshon working directory e vet dhe nuk kthehet ne gjendje e vet fillestare kur mbaron pune, pjesa tjetër e programit mund të mos punoje meqenese supozimi se ku mund të ndodhet tani mund të jetë i pavlefshem. Për ketë arsy, procedurat library rralle ndryshojne working directory dhe kur ndodh e ndryshojne përseni përparrë se të kthehen. Shumica e sistemeve operative qe mbështetin sistemet e direktorive hierarkike kane dy hyrje të vecanta ne çdo direktori ".." dhe ".." të quajtura zakonisht "dot" dhe "dotdot". Dot i referohet direktorise aktuale; dotdot i referohet burimit të tij. Për të pare se si përdoren, shikojme pemen e skedareve të UNIX ne figuren 6-10. Një proces caktuar */usr/ast* working directory e vet.

Mund të përdoret, për të shkuar tek pema. Për shembull mund të kopjohet skedari */usr/lib/dictionary* ne vetë direktorine e vet duke përdorur komanden;

```
cp ..lib/dictionary.
```

Path-i pare instrukton sistemin të shkoje lart (tek direktoria *usr*), pastaj të shkoje poshtë ne direktorine *lib* për të gjetur skedarin *dictionary*.

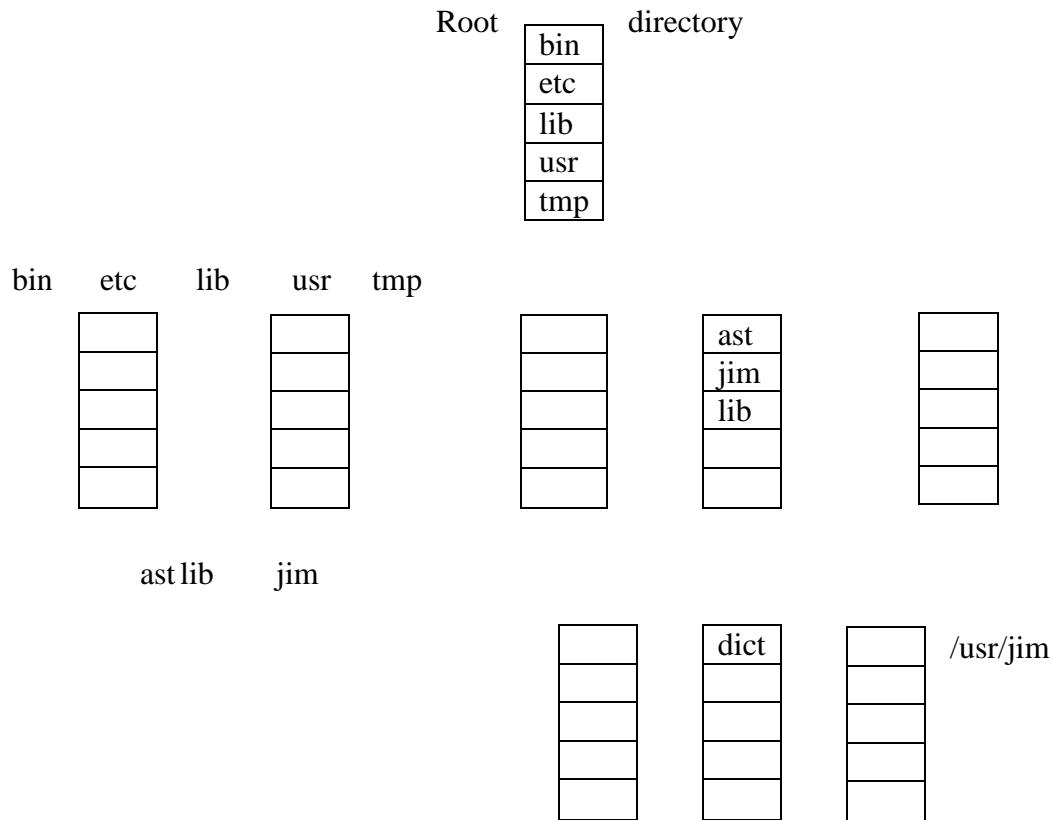


Figura 6-10. Një peme direktorish ne UNIX

Argumenti i dytë (dot) emeron direktorine aktuale. Kur komanda *cp* kap një emer direktorie (duke përfshire edhe dot) si argumenti i tij i fundit, kopjon gjithe skedarin e gjendur atje. Sigurisht një menyre normale e të kopjuarit do të ishte duke shtypur;

Cp/usr/lib/dictionary.

Ketu përdorimi i dot (pikes) i jep mundesi përdoruesit të mos shkruaje *dictionary* për të dytën here. Edhe pse të shkruarit

cp/usr/lib/dictionary dictionary

punon njëloj sikurse edhe

cp/usr/lib/dictionary/usr ast/dictionary

Të gjitha keto bejne ekzaktësish të njëjtën gje.

6.2.5 Operacionet e direktorive

Thirrjet e sistemit të lejuar për të menaxhuar direktoritë shfaq me shume variacon nga sistemi ne sistem se sa sistemi i thirrjeve për skedaret. Për të dhene një ilustrim se cfare jane dhe punes qe bejne po japim një shembull (marre nga UNIX)

1. Create (Krijo). Një direktori është krijuar. Ajo është boshe, përvèc dot (pikes) dhe dotdot (pike pike) të cilat vendosen aty automatikisht nga sistemi (ose ne disa raste nga *mkdir* program).
2. Delete (Fshi). Një direktori është fshire. Vetëm një direktori bosh mund të fshihet. Një direktori qe përmban vetëm dot dhe dotdot konsiderohet boshe, pasi keto nuk mund të fshihen zakonisht.
3. Opendir. Direktoria mund të lexohet. Për shembull, për të listuar gjithe skedaret e një direktorie, një program listues hap direktorine qe të lexoje emrat e gjithe skedareve qe ajo përmban. Qe një direktori të lexohet ajo duhet me pare të hapet, analoge me hapjen dhe leximin e një skedari.
4. Closedir. Kur një direktori është lexuar ajo duhet të mbyllt ne menyre qe të liroje hapesiren e tableles se brendshme.
5. Readdir. Kjo thirrje kthen hyrjen e ardhshme ne një direktori të hapur. Formalisht ishte e mundur qe direktoritë të lexoheshin duke përdorur sistemin e zakonshem të leximit, por ky përafrim detyronte programuesin të dinte dhe të merrej me strukuren e brendshme të direktorive. Ndersa readdir kthen një hyrje ne formen standarte, pavaresisht se cila strukture e mundshme e direktorive po përdoret.
6. Rename(riemero). Ne shume referime direktoritë Jane njëlloj si skedaret dhe mund të riemerojen me të njëjtën menyre sikurse dhe skedaret.
7. Link (lidh). Linking (nderlidhja) është një teknike qe lejon, qe një skedar të shfaqet ne me shume se një direktori. Ky sistem thirres specifikon një skedar ekzistues dhe një path emri dhe krijon një lidhje (link) nga skedari ekzistues tek emri i specifikuar nga path-i. Ne ketë menyre i njëjtë skedar mund të shfaqet ne shume direktori. Një link sipas kesaj menyre qe shton numeruesit ne nyjet e skedareve (nyjet e skedareve sherbejne për të ruajtur gjurmet e hyrjeve të shumta të direktorive qe përbajne skedarin), zakonisht quhet lidhje e veshtire **hard link**.
8. Unlink (prish lidhjen). Një hyrje e direktorise është hequr. Ne qoftë se skedari qe iu prish lidhja ndodhet vetëm ne një direktori (rasti i zakonshem), atëherë ai është hequr fare nga sistemi i skedareve. Ne qoftë se ndodhet ne shume direktori, vetëm pathi-i i specifikuar i emrit është hequr. Të tjerat mbetën. Ne UNIX sistemi i thirrjeve për të fshire skedaret (i përmendur me lart) është ne fakt unlink.

Lista e dhene me lart jep disa nga thirrjet me të rendesishme, por ka edhe disa të tjera gjithashtu të rendesishme si për shembull, menaxhimi i informacionit të proteksionit shoqeruar me një direktori.

6.3 IMPLEMENTIMI I FILE SYSTEM

Tani është koha për të kaluar nga kendveshtrimi i përdoruesit, mbi file system, ne kendveshtrimin e implementuesit. Përdoruesit jane të interesuar të dine si emerohen file-at, çfare veprimesh i lejohen atyre, si do të jene pemet e direktive, dhe çeshtje të tjera të nderfaqimit. Implementuesit jane të interesuar si ruhen file-t dhe direktoritë, si behet menaxhimi i hapesires ne disk, dhe si të bejne qe çdo gje të funksionoje me eficence dhe qendrueshmeri. Me poshtë do të analizohen disa nga problemet e mesipërme për të pare kush jane kerkesat, problemet dhe të metat.

FILE SYSTEM LAYOUT (sistemimi i file-system)

File system-et ruhen ne disk. Shumica e disqeve mund të ndahen ne një ose me shume particione, me file-system të pavarura nga një particion ne tjetrin. Sektori 0 i diskut quhet **MBR** (Master Boot Record) dhe përdoret për të boot-uar kompjuterin. Fundi i **MBR**-s përmban tabelen partacion (**partition table**). Kjo tabele jep adresat e fillimit dhe të fundit të çdo particioni. Një nga particonet ne tabele shenohet si particioni aktiv. Kur kompjuteri boot-oitet, BIOS-i lexon dhe ekzekuton MBR-n. Gjeja e pare qe programi MBR bën, është të lokalizoje particionin aktiv, të lexoje ne bllokun e tij të pare, të quajtur **boot block**, dhe ta ekzekutoje atë. Programi ne boot block ngarkon SO, qe ndodhet ne atë particion. Për të qene uniforme, çdo particion fillon me një boot block, edhe pse ai mund të mos përbaje një sistem operativ boot-imi. Por duhet ta ketë një të tille ne një të ardhme, prandaj rezervimi i një boot-block është një ide e mire, gjithsesi.

Përveç startimit me një boot block, sistemimi i particonit të një diskut varion nga një file-system ne tjetrin. Zakonisht file-system-et do të përbajne një nga objektet e meposhtme të treguara ne fig. 6.11.

I pari është një **superblock**. Ai përmban të gjithe parametrat kyç mbi file system dhe lexohet ne memorje kur kompjuteri boot-oitet, ose kur file system përdoret për here të pare. Ne superblock, informacioni tipik qe përfshihet është një numer magjik qe identifikon tipin e file system, numrin e blloqeve ne file system dhe informacione të tjera kyçe administrative.

Me pas mund të ketë informacion mbi blloqet e lira ne file system, për shembull ne formen e një bitmap-i ose një liste pointer-ash. Ky informacion mund të ndiqet nga nyje, rreshta me strukturuara datash, një për file, qe tregojne çdo gje për file-in. Mbas kesaj vjen direktoria root, qe përmban majen e pemes se file system-it. Si përfundim, pjesa e mbetur e diskut përmban të gjithe direktoritë dhe file-t e tjera.

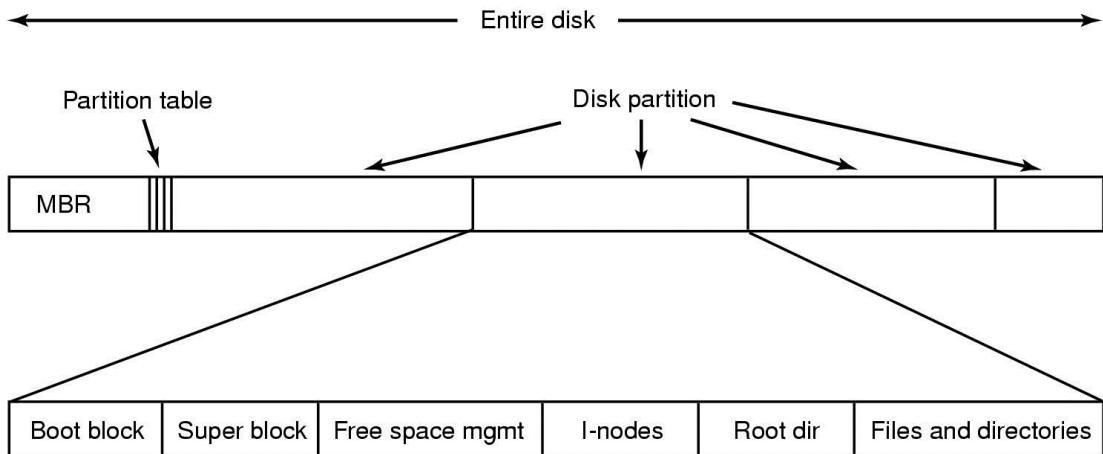


Fig. 6.11 Një file system i mundshem

6.3.2 IMPLEMENTIMI I FILE-AVE

Çeshtja me e rendesishme ne implementimin e fileve është të dish për çdo bllok disku, kujt file i përket. Ne SO të ndryshme përdoren metoda të ndryshme. Ketu do të analizojme disa prej tyre.

ALOKIMI I NJËPASNJËSHEM (CONTIGUOUS ALLOCATION)

Skema me e thjeshtë e alokimit, është të ruash çdo file si një ekzekutim i njëpasnjësheim i blloqeve të diskut. Keshtu, ne një disk me blloqe 1KB, një file 50KB do të vendoset ne 50 blloqe të njëpasnjësheim. Me blloqe 2 KB, ai do të alokohet ne 25 blloqe të njëpasnjësheim.

Ne fig. 6.12(a), tregohet një shembull i alokimit të njëpasnjësheim, ku 40 blloqet e para tregohen duke u nisur me bllokun 0 ne të majtë. Fillimi, disku ishte bosh. Me pas një file A, me gjatësi 4 blloqe shkruhet ne disk duke filluar nga blloku 0. Menjëhere pas mbarimit të ketij file shkruhet file B, file prej 6 blloqesh. Vihet re qe çdo file fillon ne fillim të një blloku të ri, keshtu në qoftë se A ishte ishte me të vertetë 3 blloqe, humbet pak hapesire ne fund të bllokut të fundit. Ne figure, tregohen 7 file, secili fillon ne fillim e bllokut pas fundit të bllokut para tij. Hjejzimi është bere për të bere me të qartë idene e ndarjes se file-ve.

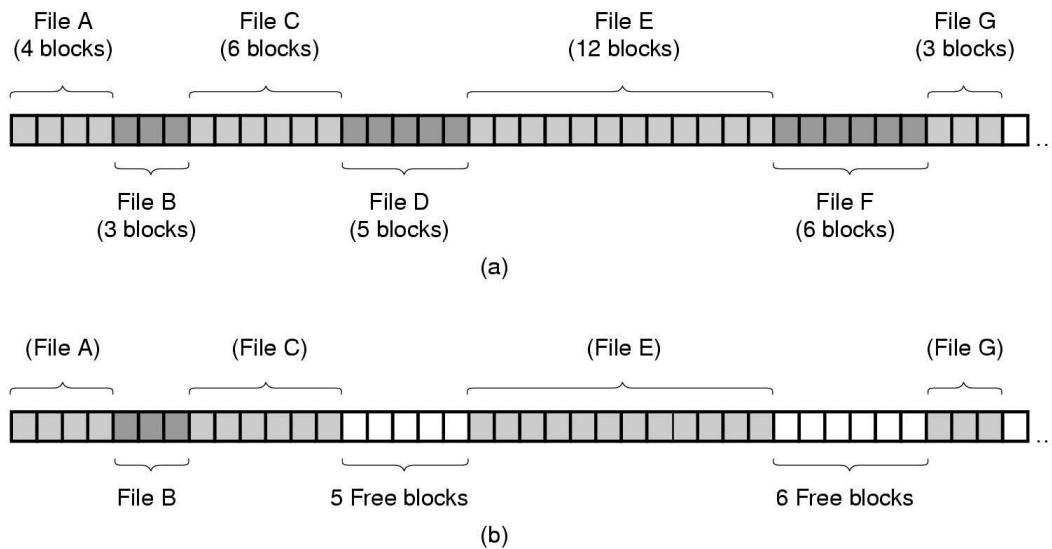


Fig. 6.12 (a) Alokimi i njëpasnjëshem i hapesires se diskut nga 7 file. (b) Gjendja e diskut pasi file-t D e F jane fshire.

Alokimi i njëpasnjëshem ne disk ka 2 avantazhe kryesore.

Është me i thjeshtë implementimi, sepse duke ditur ku jane blloqet e një file nuk ka me nevoje të mbahen mend 2 numra: adresen ne disk të bllokut të pare dhe numrin e blloqeve ne file. Duke dhene numrin e bllokut të pare, numri i çdo blloku tjetër mund të behet duke bere një mbledhje shume të thjeshtë.

Performanca e të lexuarit është shume e mire, sepse një file i tërë mund të lexohet me një operim të vetëm. Vetëm një kerkim nevojitet (ne bllokun e pare). Pas kesaj nuk nevojiten me kerkime dhe te dhenat futen me të gjithe gjeresine e brezit të ardhur nga disku.

Prandaj thuhet se alokimi i njëpasnjëshem është i thjeshtë për tu implementuar dhe ka një performance të shkelqyer.

Fatkeqesisht, alokimi i njëpasnjëshem ka një të metë të madhe: me kalimin e kohes disku fragmentohet. Për të kuptuar ketë analizoni fig 6.12(b). Ketu dy file-t D dhe F jane fshire. Kur një file fshihet, blloku i tij lirohet duke krijuar blloqe të lira për ekzekutim ne disk. Disku nuk është kompakt për zhdukur keto vrima, po të ndodhët kjo atëherë të gjithe blloqet pas vimes do të kopjoheshin (potencialisht miliona blloqe). Si rezultat, disku përbehet nga file dhe vrima, si të tregohen ne figure.

Fillimisht, fragmentimi nuk përbën një problem për sa kohe qe një file i ri shkruhet ne fund të diskut, duke ndjekur atë para tij. Megjithatë, disku do të mbushet dhe do të jetë e nevojshme të behet disku kompakt, gje qe është shume e shtrenjtë, ose të ripërdoret hapesira bosh e vrimave.

Ripërdorimi i hapesires bosh kerkon krijimin e një liste të vrimave. Megjithatë kur një file i ri krijohet, është e nevojshme të dihet madhesia e tij përfundimtare, për të zgjedhur keshtu një vrime me madhesine e duhur pér ta vendosur.

Imagjinoni pasojat përfundimtare të një dizenjimi të tille. Përdoruesi do të filloje të përdori një word processor ose një text editor pér të shkruar një dokument. Gjeja e pare qe pyet programi është: sa byte do të jetë dokumenti përfundimtar. Duhet t'i përgjigjesh kesaj pyetje qe të vazhdoje programi, në qoftë se nemri i dhene është shume i vogel, programi do të përfundoje para kohe, sepse vrima është plot dhe nuk ka ku të vendoset pjesa tjeter e file-t. Në qoftë se përdoruesi jep një numer shume të madh si madhesi përfundimtare, pér shembull 100MB, editori nuk do të jetë ne gjendje të gjeje një vrime kaq të madhe dhe lajmeron qe file nuk mund të krijohet. Përdoruesi mund ta riprovoje duke dhene një madhesi të re me të vogel, pér shembull 50 MB ose me shume derisa të gjendet një madhesi e përshtatshme. Megjithatë, akoma kjo skeme nuk është e kenaqshme.

Megjithatë ka një situatë, ku alokimi i njëpasnjëshem është i mundshem dhe mjaft i përdorur: ne CD-ROM-et. Ketu të gjithe madhesite e file-ve njihen ne avance dhe nuk do të ndryshojne kurre gjatë përdorimit të here pas hershem të CD-ROM file system. Me vone do të shqyrtohen CD-ROM file system-et.

Alokimi i njëpasnjëshem u përdor fillimisht ne disqe magnetike, kjo pér shkak të lehtësise ne operim dhe performancës se lartë (atëherë nuk kishte shume rendesi qe një kompjuter të ishte i thjeshtë pér një përdorues çfaredo, friendliness). Me pas kjo ide u hodh poshtë pér shkak të nevojes qe lindi pér të njohur ne avance madhesine e file-t. Por me shpikjen e CD-ROM-eve, DVD-ve dhe mediave të tjera optike qe shkruhen vetëm një here, ideja e alokimit të njëpasnjëshem pati përseri sukses. Pér ketë është e rendesishme të studiohen sistemet dhe idetë e vjetra, sepse ishin konceptualisht të lehta dhe të pastra, dhe ato mund të aplikohen ne sisteme ne të ardhmen, ne forma nga me të ndryshmet.

ALOKIMI ME LINKED LIST

Menyra e dytë e ruajtjes se file-ve është të ruash çdo file si linked list blloqesh të diskut, si ne fig 6.13. fjalë e pare e çdo blloku përdoret si një pointer pér bllokun pasardhes. Pjesa tjeter e bllokut përdoret pér datat.

Ndryshe nga alokimi i njëpasnjëshem, çdo bllok i diskut mund të përdoret ne ketë metode. Nuk humbet aspak hapesire bosh nga fragmentimi i diskut (përveç fragmentimit të brendshem ne bllokun e fundit). Gjithashtu, është e mjaftueshme pér hyrjen ne direktori, të ruhet ne disk adresa e bllokut të pare. Pastaj duke u nisur qe aty mund të gjesh pjesen tjeter.

Nga ana tjeter, edhe pse leximi sekuencial i file-ve është i drejtpërdrejtë, aksesimi random është shume i ngadaltë. Pér të shkuar ne bllokun n , SO duhet ta filloje leximin nga fillimi, domethene të lexoje $n-1$ blloqe para tij, nga nje per cdo rast. Është e qartë se të gjitha keto lexime janë shume të ngadalshme.

Gjithashtu, sasia e te dhenave të ruajtura ne një bllok nuk jane me fuqi e 2-it, sepse pointeri përdor disa byte për vetë. Nuk është për tu shqetësuar, por fakti i të pasurit një madhesi të veçantë është me pak eficentë, sepse shume programe lexojne dhe shkruajne ne blloqe, madhesia e të cilave është fuqi e 2-it. Pak nga bytet e para të çdo blloku i ze pointeri i bllokut pasardhes. Leximet e të gjithe madhesise se bllokut kerkojne informacionin përkatës dhe të grumbulluar nga 2 blloqet e disqeve, qe gjenerojne tituj nga kopjimi.

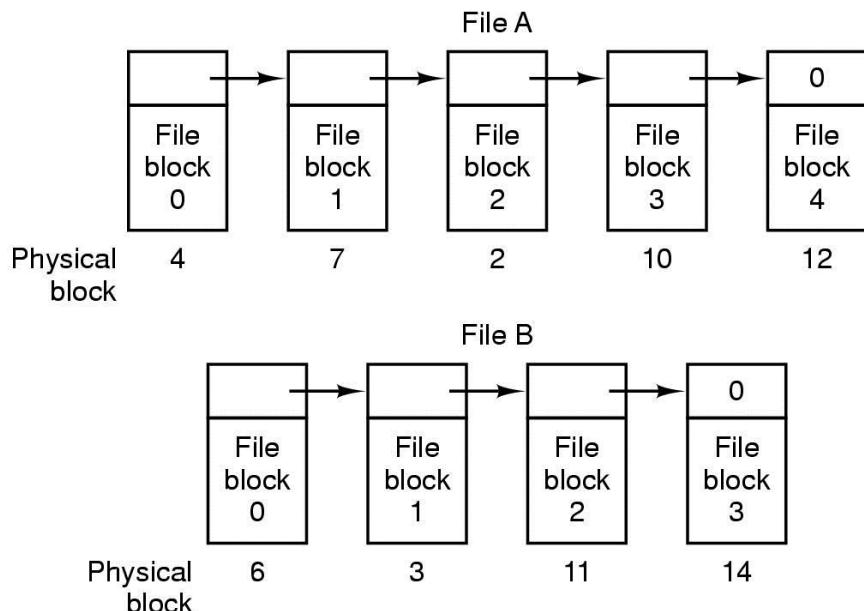


Fig. 6.13 Ruajtja e file-ve si një linked list e blloqeve ne disk

LINKED LIST E ALOKIMIT DUKE PËRDORUR NJË TABELE NE MEMORJE (Linked List Allocation Using a Table in Memory)

Dy disavantazhet e linked list të alokimit mund të eleminohen duke vendosur pointer word nga çdo bllok ne një tabelë ne memorje. Fig. 6.14, tregon si është e ndertuar tabela për rastin e fig. 6.13. ne të dy figurat. Kemi dy file. File A përdor blloqet 4, 7, 2, 10, 12 sipas kesaj rradhe dhe file B përdor blloqet 6, 3, 11, 14 sipas kesaj rradhe. Duke përdorur tabelen ne fig. 6.14, ne mund të fillojme me bllokun 4 dhe të ndjekim zinxhirin deri ne fund. E njëjtë gje mund të behet duke filluar nga blloku 6. Të dy zinxhiret shenohen me një shenjë të veçantë (për shembull -1), qe nuk është një numer blloku i vlefshem. Një tabelë e tille ne memorien kryesore njihet si **FAT** (File Action Table).

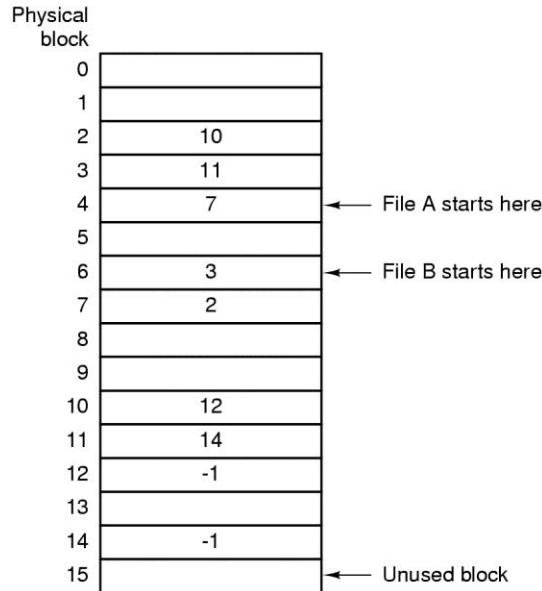


Fig. 6.14 (a) Linked list e alokimit, qe përdor një file alocation table.

Duke përdorur ketë organizim, i gjithe blloku është i vlefshem për te dhenat. Për më tepër, aksesimi i rastit është me i lehtë. Megjithatë duhet të ndiqet çdo hallke e zinxhirit për të gjetur një offset të dhene brenda një file, zinxhiri është i terti ne memorie, keshtu ai mund të ndiqet hallke pas hallke, pa bere ndonjë reference adresimi. Si metoda me përpala, për hyrjen ne direktori është e mjaftueshme të ruhet një integer i vetëm (numri i bllokut fillesttar) për të qene ne gjendje për të lokalizuar të gjith blloqet e tjera, pavaresisht sa i madh është file.

Disavantazhi i pare i kesaj metode është se e gjithe etiketa duhet të jetë ne memorje gjatë gjithe kohes se punimit. Me një disk 20 GB me blloqe 1KB, tabela i duhen 20 milion hyrje (entry), një për çdo 20 milion blloqet. Çdo entry duhet të jetë minimumi 3 byte. Për shpejtësine e kerkimit duhet të jetë 4 byte. Prandaj tabela do të zeje 60 ose 80MB të memories kryesore gjatë gjithe kohes, kjo varet nga fakti në qoftë se sistemi është i optimizuar ne hapesire ose ne kohe. Për rrjedhoje, tabela mund të vendoset ne memorje të faqueshme, por përseri do të zëj një pjese të madhe të memorjes virtuale dhe të diskut, gjithashtu do të gjeneroje një trafik të madh faqesh.

NYJET-I (I- nodes)

Metoda e fundit mbi njohjen e çdo blloku të file- ve përkatës, ka të beje me shoqerimin e çdo file me një strukture te dhenash të quajtur **i-node** (**index-node**), e cila liston cilesite dhe adresat e blloqeve të file-ve. Një shembull i thjeshtë paraqitet ne fig 6.15. Duke njojur i-node mund të gjenden të gjithe blloqet e file-ve. Avantazhi i madh i kesaj metode, mbi file-t e lidhura duke përdorur një tabelë brenda ne memorje është se mjafton

të jetë i-node ne memorje kur file përkatës është i hapur. Në qoftë se çdo i-node, ze n -byte dhe mund të hapan njëheresh k file, memoria totale e zene nga i-node-t për file-t e hapura do të jetë kn byte. Vetëm kaq hapesire duhet të rezervohet ne avance. Kjo hapesire është shume e vogel e kahasuar me hapesiren e zene nga tabela e file-ve të përshkruar ne metoden e meparshme. Arsyja është e thjeshtë. Tabela për listën e lidhur të të gjithe blloqeve është proporcionale me madhesine e diskut. Në qoftë se disku ka n -blloqe, tabeles i duhen n entry. Sa me shume rritet disku, aq me shum rritet dhe tabela, ne menyre lineare. Ne kontrast me i-node-t qe kerkojne ne memorje një hapesire, qe të jetë proporcionale me numrin maksimal të file-ve qe mund të hapan njëheresh. Nuk varen nga madhesia e diskut, qe mund të jetë 1GB, 10GB, 100GB.

Një problem me i-node-t është se çfare ndodh kur ka hapesire vetëm pér një numer fiks adresash dhe file i ka përmasat me të medha se ky limit?! Një zgjidhje do të ishte të ruhej adresa e fundit e diskut jo pér bllok te dhenash, por pér adresen e një blloku ku ndodhen adresa blloqesh të tjere, të treguara ne fig.6.15. Një zgjidhje me e avancuar do të ishte qe ne një bllok të ruheshin dy apo me shume adresa blloqesh, të mbushur me adresa blloqesh të tjeter. I-node-t do të analizohen me mire ne UNIX.

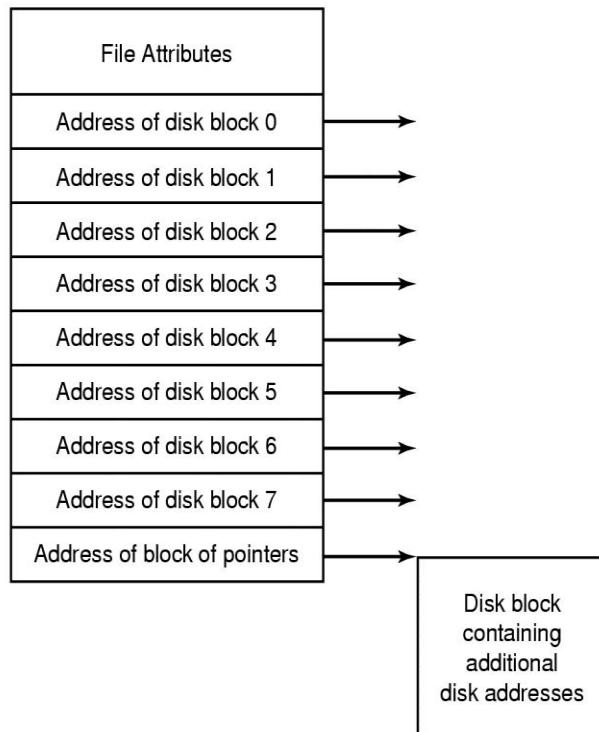


Fig 6.15. Një shembull i një i-node

6.3.3 IMPLEMENTIMI I DIREKTORIVE

Para se një file të lexohet ai duhet të hapet. Kur një file hapet, SO përdor path name të futur nga përdoruesi për të lokalizuar direktorine. Direktoria siguron informacionin e nevojshem për të gjetur blloqet. Ne varesi të sistemit, ky informacion mund të jetë adresa ne disk të të gjith file-it (alokimi i njëpasnjëshem), numri i bllokut të pare (2 skemat e linked list), ose numrin e **i-node**-ve.

Ne të gjitha rastet, funksioni kryesor i direktorise është të fusi emrin e file-it ne ASCII, tek informacioni i nevojshem për të gjetur te dhenat.

Një çeshtje e rendesishme është, se ku do të ruhen veçoritë e blloqueve të file-ve. Çdo file system përmban cilesite e veta, si për shembull, pronarin e çdo file, kohen e krijimit dhe të gjitha keto duhet të ruhen diku. Një mundesi do të ishte të ruheshin direkt ne direktori. Shumica e sistemeve bejne keshtu. Kjo menyre tregohet ne fig. 6.16(a). Ne ketë dizenjim të thjeshtë, direktoria përmban një listë me hyrje (entry) me madhesi fikse, një për çdo file, përmban emrin e filet (me gjatësi fikse), një strukturë të veçorative të file-it dhe një, apo me shume adresa ne disk, për të treguar ku vendosen blloqet ne disk.

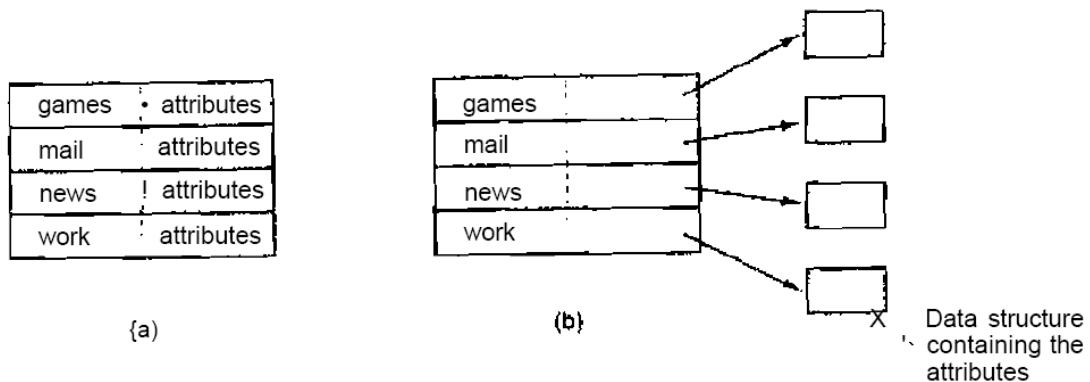


Fig. 6.16(a) Një direktori e thjeshtë qe përmban entry me madhesi fikse, adresat ne disk dhe cilesite e direktorise.

(b) Një direktori ku çdo entry i referohet një i-node.

Për sistemet qe përdorin **i-node**, një mundesi tjetër do të ishte ruajtja e cilesive ne **i-node**. Ne ketë rast, hyrja ne direktori do të jetë me e shkurtër, vetëm një emer file dhe një numer i-node. Kjo tregohet ne fig. 6.16(b). Siç do shihet me vone kjo metode ka avantazhet e veta mbi vendosjen e ketyre të dhenave ne direktori. Dy stukturat e mesipërme i përkasin përkatësisht MS/DOS Windows dhe UNIX.

Siç thame me lartë file-t kane emra me gjatësi fikse të shkurtër, file-t e MS/DOS e kane emrin baze 1-8 karaktere dhe një shtese 1-3 karaktere. Ne UNIX versioni 7, emrat e fileve ishin 1-4 karaktere duke përfshire dhe shtesat. SO moderne suportojne emra me të gjatë file-sh. Si mund të implementohet kjo?

Menyra e e thjeshtë është të vendoset një limit i gjatësise se emrit të file-t, tipikisht 255 karaktere, dhe me pas, përdoret një nga dizenjimet ne fig.6.16, me 255 karaktere të rezervuara për çdo file. Kjo struktura është e thjeshtë, por humbet një sasi të madhe hapesire të direktorise, me qene se pak file kane emra me gjatësi kaq të madhe. Për asje eficience është me e preferueshme një struktura tjetër.

Një alternative do të ishte të mos përdorej ideja, qe të gjithe entry-t ne direktori të kene madhesi të njëjtë. Me ketë metode, çdo entry ne direktori do të ketë një pjese fiksë, qe fillon nga gjatësia e entry-it, me pas ndiqet nga te dhenat me një format të caktuar, zakonisht përfshihet dhe pronari, data e krijimit, informacioni mbrojtës dhe veçori të tjera. Ky titull me një gjatësi fiksë ndiqet nga emri aktual i file-t, sa do i gjatë qe ai të jetë, siç tregohet ne fig 6.17(a) ne formatin big-endian (për shembull SPARC). Ne ketë shembull kemi tre file, *project-budget*, *personnel*, *foo*. Çdo emer file përfundon me një karakter të veçantë (zakonisht 0), qe paraqitet ne figure nga një kuti me një kryq brenda.

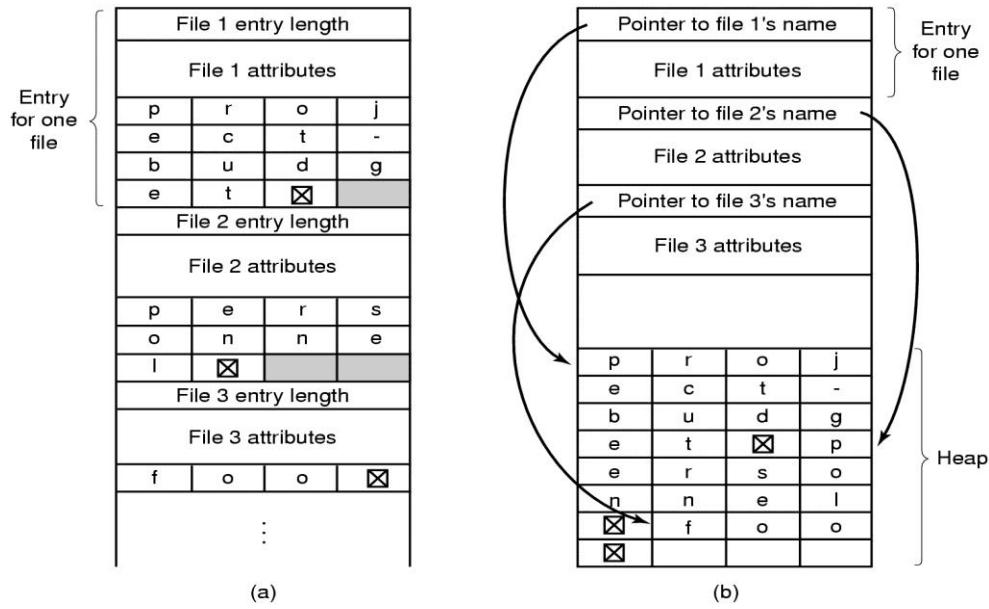


Fig. 6.17 Dy menyra për të trajtuar emrat e gjatë të file-ve ne një direktori. (a) ne rradhe (b) ne stive

Një disavantazh është, qe kur një file fshihet, gjenerohet ne direktori një gap me madhesi qe varion dhe file tjetër qe mund të futet aty mund të mos përshtatet, (si ne rastin e fileve të njëpasnjëshem, vetëm se ne ketë rast kompaktimi i direktorise është me i mundshem, sepse ndodhet komplet ne memorie).

Një problem tjetër është, qe një direktori e vetme mund të përfshije shume faqe, keshtu mund të ndodhi ndonjë gabim gjatë leximit të faqeve.

Një menyre tjetër për të trajtuar variablin e gjatësise se emrit, është duke i bere të gjithe entry-t me gjatësi fiksë dhe mbajtja e të gjithe emrave të file-ve të bashkuara ne një stive

ne fund të direktorise, siç tregohet ne fig.6.17(b). Avantazhi i kesaj metode është se kur një file do të fshihet nga direktoria, file tjetër qe do të futet do të përshtatët gjithmone me hapesiren e lene nga file-i fshire.

Gjithashtu duhet qe stiva të menaxhohet mire, sepse mund të ndodhin gabime gjatë leximit. Një fitim i vogel ketu është, se nuk është me e nevojshme qe emri i file-t të filloje ne kufinjtë e fjales, keshtu qe nuk nevojiten karaktere të veçanta mbushes pas emrit të file-t si ne fig 6.17(b) dhe qe Jane të pranishme ne fig 6.17(a).

Ne të gjithe dizenjimet, për të gjetur një file, direktoritë kerkohen linearisht nga fillimi deri ne fund.

Për direktori shume të gjata, kerkimi linear mund të jetë i ngadaltë. Një menyre për të rritur shpejtësinë, është nepërmjet hash table, të përdorur ne çdo direktori. Shenojme me n madhesine e tabelës. Për të futur emrit e një file, emri do të shtrihet nga $0 - n-1$, për shembull duke e pjestuar atë me n përftohet mbetja. Ne menyre alternative, fjalet duke përfshire dhe emrin e file-t mund të mblidhen dhe kjo sasi të pjesohet me n .

Ne ketë metode entry e tabelës qe i korrespondon hash kodit analizohet. Në qoftë se ai nuk është përdorur, ne file entry vendoset pointeri. File entry-it ndjekin tabelen hush. Në qoftë se ai është përdorur, ndertohet një listë e lidhur, qe vendoset ne krye të tabelës dhe mbledh të gjithe entry-t me të njëjtën vlerë të hush-it.

Kerkimi i një file ndjek të njëjtën procedure. Futet emri i file-it për të gjetur tabelen hash. Kontrollen të gjithe titujt e hallkave të zinxhirit për të pare ne se gjendet emri i file-it. Në qoftë se emri nuk ndodhet ne zinxhir do të thotë qe ai nuk ndodhet as ne direktori.

Përdorimi i një hash table, ka avantazhin e një kerkimi me të shpejtë, por ka dhe disavantazhin e një administrimi me kompleks. Ky model është një kandidat i mire, për direktoritë qe përbajne qindra file.

Një menyre komplet ndrysht për të përshpejtuar kerkimin ne direktori shume të medha është kerkimi ne cache. Përpara se të filloje kerkimi ne një direktorie, shikohet në qoftë se kjo direktori ndodhet ne cache. Në qoftë se po, lokalizimi behet shume shpejt. Sigurisht qe kerkimi ne cache është i vlefshem për një numer të vogel file-sh.

6.3.4 SHARED FILES

Kur shume përdorues punojne ne një grup, ndodh shpesh qe ata të përdorin file të share-aura. Për ketë është me e leverdisshem, qe një një shared file të shfaqet vazhdimesht ne direktori të ndryshme qe i përkasin përdoruesve të ndryshem. Ne fig 6.18 paraqitet file sistemi i fig 6.9 me një nga file e C-se prezente dhe ne direktorine e B-s po ashtu. Lidhja midis direktorise se B-s dhe një shared file quhet një **link**. File-i vet njihet si **Direct Acyclic Graph** ose **DAG**, dhe nuk quhet me peme.

Sharimi i fileve është i leverdisshem, por edhe ai ka disa probleme. Keshtu, në qoftë se direktoritë përbajne adresa ne disk, atëherë duhet bere dhe një kopje ketyre adresave ne direktorine e B-s kur behet linkimi i file-it. Në qoftë se B ose C i bashkangjiten ne menyre të njëpasnjëshme file-it, blloqet e reja do të listohen vetëm ne direktorine e

përdoruesit qe ka bere bashkangjitjen. Ne ketë rast nuk vlen ideja e share-imit, sepse ndryshimet e bera nuk jane të dukshme për përdoruesit e tjere.

Ka dy menyra për të zgjidhur ketë problem:

Ne menyren e pare, blloqet nuk listohen ne direktori, por ne një strukture të vogel te dhenash qe shoqeron file-in. Kjo menyre përdoret ne UNIX (ku struktura e vogel e te dhenave është i-node).

Ne menyren e dytë, B-ja lidhet me një nga file-t e C-se duke krijuar një file të ri ne sistem, të tipit LINK, dhe e fut ketë file ne direktorine e B-se. File i ri përbën path name të filet me të cilin ai është linkuar. Kur B-ja lexon nga linked file, SO kupton qe file qe po lexohet është i tipit LINK, me pas kerkon emrin e file-it dhe e lexon atë file. Kjo metode quhet **symbolic linking**.

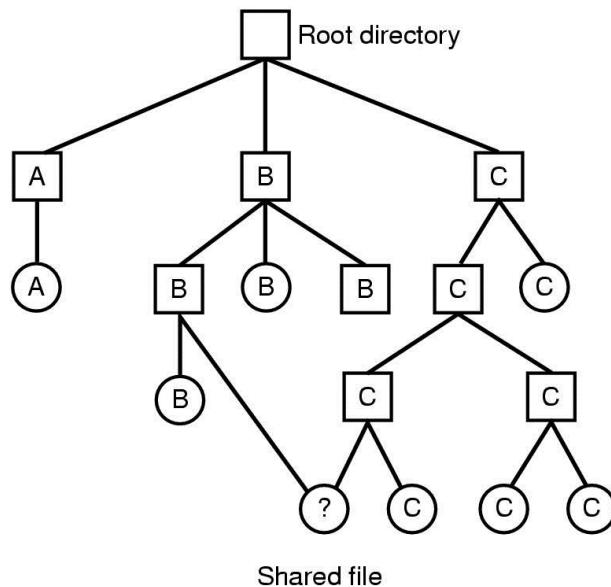


Fig 6.18 File sistemi i kombinuar me një shared file.

Çdo njëra nga keto metoda ka të metat e veta. Ne metoden e pare, ne momentin qe B-ja lidhet me shared file-in, i-node regjistron si pronar të file-it C-n. Krijimi i një linku nuk e ndryshon pronesine(fig 6.19), por rrit numrin e linkeve ne i-node-t, keshtu SO e di sa entry ne direktori pointojne ne file.

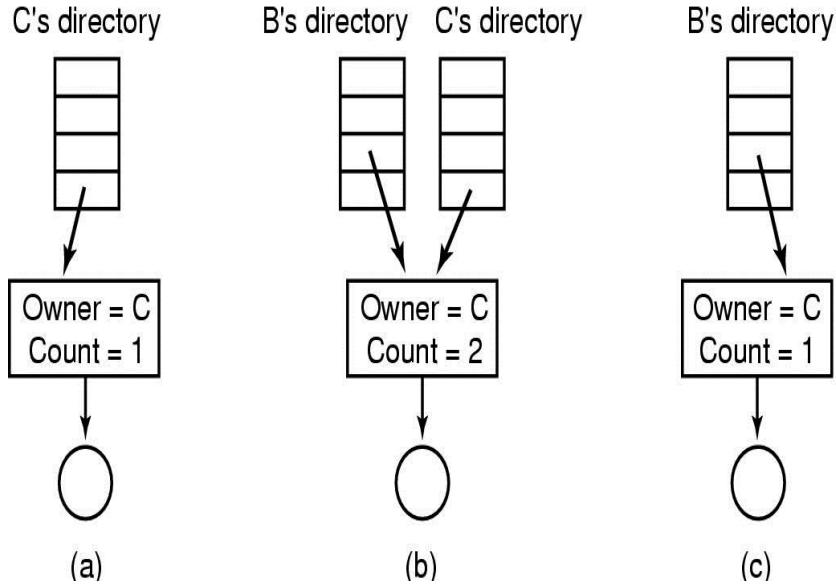


Fig. 6.19 (a) Situata para linkimit. (b) pasi krijohet linkimi (c) pas zhvendosjes se file-it nga pronari origjinal.

Në qoftë se C-ja kerkon të fshije file-in atëhere lind një problem. Në qoftë se ai e fshin file-in dhe fshin i-nodet, B-ja do të kete një entry ne direktori qe do të pointoje ne një i-node të pavlefshem. Në qoftë se i-node rishenohet tek një file tjetër i B-s, linku do të pointoje ne një file të gabuar. Sistemi mund ta kuptoje nga numri i i-node-it në qoftë se file është akoma ne përdorim, por nuk është e mundur të gjenden të gjithe entry-it e direktorise për file-in, me qellimin për ti fshire ato.

Pointerat e direktorive nuk mund të ruhen ne i-node sepse aty mund të ketë një numer të limituar direktorish.

E vetmja gje për të bere është fshirja e entry-t të direktorise se C-se, por e le i-node të paprekur, të setuar me 1, siç tregonet ne fig 6.19(c). Ne tani kemi një situatë ku B-ja eshtë i vetmi qe ka një directory entry ne C. Në qoftë se sistemi bën numerime, atëherë C-ja do të vazhdoje të jetë strehe për file-in, derisa B të vendosi ta fshije atë, vetëm në qoftë se ne një çast të caktuar numri shkon ne 0 dhe file fshihet.

Me symbolic links, ky problem nuk lind sepse vetëm përdoruesi e ka pointerin ne i-node. Përdoruesit qe kane linkuar ne file, vetëm kane path name-in, jo pointerat tek i-node. Kur pronari e fshin file-in, ai shkatërritet. Përpjekjet për të përdorur ketë file nepërmjet një symbolic link do të deshtojne kur sistemi nuk do të jetë ne gjendje të lokalizoje file-in. Fshirja e një symbolic link nuk ndikon fare tek file.

Problemi me symbolic links është se ata kerkojne një extra titull. File përmban path name qe duhet të lexohet, me pas ky path analizohet dhe ndiqet çdo komponent i tij ne menyre të njëpasnjëshme, derisa të arrihet tek i-node. Të gjitha keto veprime mund të kerkojne një numer të madh aksesimesh. Për me tepër për çdo symbolic link, nevojitet një extra i-node, siç është ne rastin e një extra blloku ku ruhet path-i, ku edhe pse emri i pathit është i shkurtër ai duhet te ruhet ne një i-node për një optimizim me të mire. Symbolic links kane avantazhin e të lidhurit (link) të file-ve kudo ne botë, vetëm duke siguruar adresen e network-ut të kompjuterit ku ndodhet file si dhe path-in e tij ne ketë kompjuter.

Symbolic links ose linke të një tipi tjetër kane edhe një problem tjetër. Kur lejohet një link, filet mund të kene dy ose me shume path-e. Programet, qe fillojne ne një direktori të caktuar dhe gjaje të gjithe filet ne atë direktori dhe nendifektivitet e saj, lokalizojne një linked file shume here. Për shembull, një program qe hedh të gjitha filet ne një direktori dhe nendifektivitet e tij ne një disk, themi se ai mund të beje shume kopje të linked fileve. Për me tepër, në qoftë se disku lexohet ne një makine tjetër, pa programin e kopjimit, linked file do të kopjohet dy here ne disk, ne vend qe të linkohet.

6. 3. 5 MENAXHIMI I HAPESIRES NE DISK (Disk Space Management)

File-t zakonisht ruhen ne disk, keshtu menaxhimi i hapesires ne disk është një problem shqetësues për dizenjatoret e sistemit. Mund të përdoren dy strategji ne ruajtjen e një file prej n-byte: alokohen n-byte të njëpasnjëshme ne disk, ose file ndahet ne një numer (jo e domosdoshme) blloqesh të njëpasnjëshme. I njëjtë problem paraqitet ne menaxhimin e sistemeve të memories ndermjet segmentimit të pastër dhe faqosjes.

Siq kemi pare, ruajtja e një file ne një sekuence të njëpasnjëshme byte-sh, ndeshet me problemin e zmadhimit të file-it, dhe për pasoje file duhet të vendoset ne disk. I njëjtë problem haset me segmentimin ne memorie, përveç rastit të zhvendosjes se një segmenti ne memorie, qe është një operacion relativisht i shpejtë krahasuar me zhvendosjen e një file nga një pozicion i diskut ne një tjetër. Për ketë arsy, gati të gjithe file systemet i copetojnë filet e tyre ne ne blloqe me madhesi fikse qe nuk duhen të jene të për brinjët (të afert).

Madhesia e Bllokut (Block size)

Kur u vendos qe ruajtja e fileve të behej ne blloqe me madhesi fikse, lindi pyetja se sa i madh duhet të ishte blloku. Duke ditur menyren e organizimit të diskut, kandida me të mire për njësi alokimi do të ishin: sektoret, track-u, cilindri (por të gjitha keto jane të varura nga pajisja, gje qe përbën një minus) ne sistemin me faqe, faqja do të ishte zgjidhja me e mire.

Pasja e një njësie alokimi të madhe, siq është për shembull cilindri, do të thotë qe një file edhe 1byte të jetë, do të zeje të gjithe cilindrin. Studimet kane treguar qe madhesia mesatare ne UNIX është 1KB, keshtu alokimi i një blloku 32 KB për çdo file do të shkaktonte humbjen e 31/32 ose 97% të hapesires totale ne disk.

Nga ana tjetër, përdorimi i një njësie alokimi të vogel do të beje qe çdo file të përbehet nga shume blloqe. Leximi i tyre do të kerkonte kohe dhe vonesa rrrotullimi, keshtu qe leximi i një file të përbere nga disa blloqe do të ishte i ngadaltë.

Si shembull po marrim një disk me 131.072 byte për track, me një kohe rrrotullimi 8.33 msec, dhe një kohe mesatare kerkimi 10msec. Keshtu koha ne msec për të lexuar një file do të ishte shuma e kohes se kerkimit, me vone se ne rrrotullimit, me kohet e transferimit.

Grafiku solid ne fig. 6.20, shpreh shpejtësine e te dhenave për një disk të tille, ne varesi të madhesise se bllokut. Për të llogaritur eficencen e hapesires, duhet bere përcaktimi i mesatares se madhesise se file-it. Për thjeshtësi, le të pranojme si një vlere mesatare të filet 2KB.

Dy grafiqet mund të shpjegohen si me poshtë. Koha e aksesimit të një blloku dominohet nga koha e kerkimit dhe nga vonesa e rrotullimit, keshtu koha për aksesimin e një blloku do të jetë 14 msec, sa me shume te dhena tërhiqen, me mire është. Shpejtësia e transferimit të te dhenave rritet me rritjen e bllokut. Me blloqe të vogla, fuqi të 2-it, nuk humbet hapesire ne blloqe. Megjithatë, me file 2KB dhe blloqe 4KB ka humbje hapesire ne bllok. Ne fakt, pak file jane shumefisha të madhesise se bllokut, keshtu gjithmone humbet pak hapesire ne bllokun e fundit të një file.

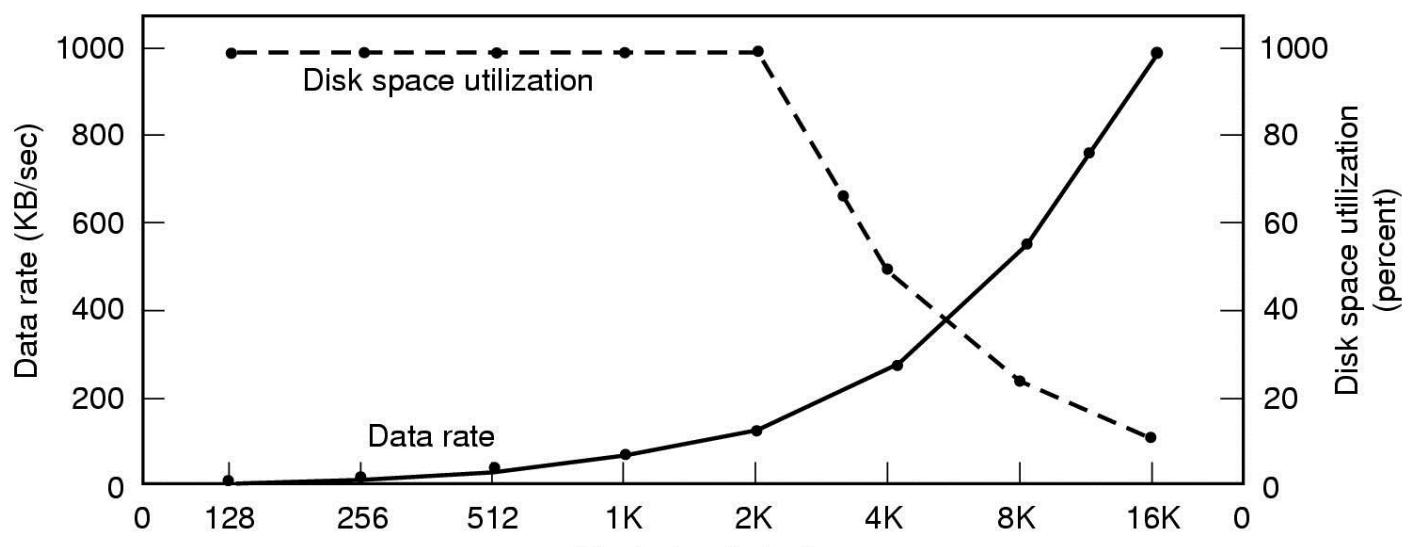


Fig 6.20. Grafiku solid (ne të majtë) paraqet shpejtësine e datave të një diskut. Grafiku me pikat (ne të djathtë) jep eficencen e hapesires ne disk. Të gjithe filet pranohen 2 KB.

Grafiqet tregojne se përfomanca dhe shfrytëzimi i diskut jane ne konflikt. Blloqet e vogla Jane të mire për shfrytëzimin e mire të hapesires ne disk, por të keqja për performancen. Prandaj nevojitet një kompromis mbi madhesine e blloqeve. Për keto te dhena, blloqe 4KB do të ishin një zgjidhje e mire, por shumica e SO e kane bere ketë zgjedhje para shume kohesh kur parametrat e kompjuterit ishin ndryshe. Për UNIX, me të përdorurat Jane blloqet 1KB. Për MS/DOS përdoren blloqe me madhesi fuqi të 2-it, nga 512byte – 32KB, por kjo madhesi përcaktohet dhe nga madhesia e diskut (numri maksimal i blloqeve ne një particion të diskut është 2^{16} , qe bën të kemi blloqe të medha ne disqe të medha).

U be një eksperiment nga Voges, për të pare sa ndryshonte Windows NT nga UNTX ne përdorimin e file-ve. Ai vuri re se përdorimi i file-ve ne NT ishte me kompleks dhe shkruajti:

Kur ne shkruajme disa karaktere ne një notepad text editor, ruajtja e tyre do të gjeneronte 26 thirrje sistem, duke përfshire dhe 3 përpjekje të deshtuara hapje, 1 file overwrite (të mbishkruar) dhe 4 sekuencia hapje dhe mbyllje shtese.

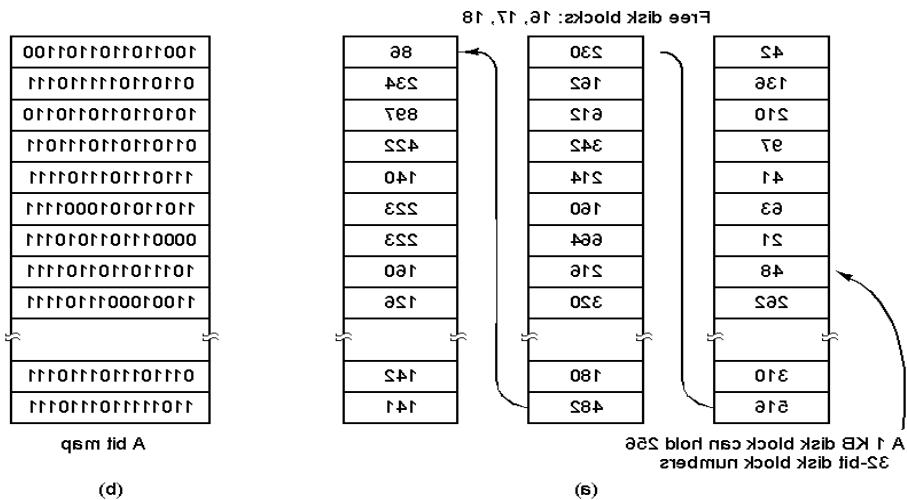
Ai vezhgoi, file me madhesi mesatare (nga pesha e perdomit), të lexuara si file 1 KB, file të shkruara si 2.3KB dhe file të lexuara dhe të shkruara si 4.2 KB. Duke pasur parasysh qe instituti Cornell kishte kompjutera shkencore te nje shkalle me te larte se te tjeter dhe diferenca ne teknikat e matjes, rezultatet konsistojne ne nje madhesi mesatare te file-t rreth 2 KB.

Analizimi i blloqueve të lira (Keeping Track of Free Blocks)

Një çeshtje tjetër e rendesishme pas çeshtjes se madhesise se bllokut, është si të analizohet hapesira e një blloku. Për ketë përdoren 2 metoda, të treguara ne fig. 6.21.

1. E para konsiston ne përdorimin e një linked list të blloqueve ne disk, ku çdo bllok mban sa me shume numra blloqesh të jetë e mundur. Çdo bllok ne listë do të përbaje një numer prej 255 blloqesh, në qoftë se bloku është 1 KB dhe ka numer blloku 32-bit (një ndarje nevojitet për pointerin ne bllokun tjetër).

Një disk 16GB ka nevoje për një listë të lire me max 16,794 blloqe, për të mbajtur 2^{24} numra blloqesh ne disk. Për të mbajtur listën e lire përdoren blloqe boshe.



2. Teknika tjetër e menaxhimit të hapesires është bitmap. Një disk me n blloqe kerkon një bitmap me n bite. Blloqet e lira paraqiten me 1 ne hartën e briteve, blloqet e alokuara me 0 (ose e anasjellta). Një disk 16-GB ka 2^{24} blloqe prej 1 KB, dhe prandaj duhen 2^{24} bite për hartën (map), qe kerkon 2048 blloqe. Nuk është përtu çuditur fakti, qe bitmap kerkon me pak hapesire, përderisa kjo metode përdor 1 bit për bllok, ndersa ne metoden linked list nevojiteshin 32-bit. Vetëm kur disku është pothuajse bosh (për shembull ka shume pak blloqe të lira) atëherë do të jetë metoda linked list ajo qe do të kerkoje me pak blloqe se metoda bitmap. Nga ana tjetër, në qoftë se ka blloqe të lira, disa prej tyre mund të merren borxh për të mbajtur free listën pa pasur humbje ne kapacitetin e diskut.

Kur përdoret metoda me free list, është e nevojshme të ruhen ne memorie vetëm një bllok pointerash. Kur krijohet një file, blloqet qe duhen përta ruajtur merren nga blloqet e pointerave. Kur file ekzekutohet, lexohet një bllok i ri pointerash nga disku. Ne menyre të ngjashme, kur fshihet një file, blloqet e tij mbeten të lira dhe i shtohen blloqeve të pointerave ne memorien kryesore. Kur ky bllok mbushet, ai shkruhet ne disk.

Ne situata të tilla, disk I/O jane të pavlefshme. Marrim ne konsiderate situatën e fig. 6.22(a), ku blloku i pointerave ne memorien kryesore ka vend vetëm përt dy entry të tjera. Në qoftë se një file prej 3 blloqesh lirohet, ndodh një overflow ne bllokun e pointerit dhe duhet të shkruhet ne disk, gje qe çon ne situatën e figures 6.22(b). Në qoftë se file prej 3 blloqesh shkruhet, i gjithe blloku duhet të lexohet përsëri, duke na kthyer tek fig.6.22(a). Në qoftë se file prej 3 blloqesh qe u shkruajt është një file temporal, kur ai lirohet, një tjetër shkrim ne disk nevojitet, përtë shkruar komplet bllokun e pointerave mbrapsht ne disk. Shkurt, kur blloku i pointerave është pothuajse bosh, shkrimi i një serie file-sh temporele me jetëgjatësi të shkurtër mund të shkaktoje shume disk I/O.

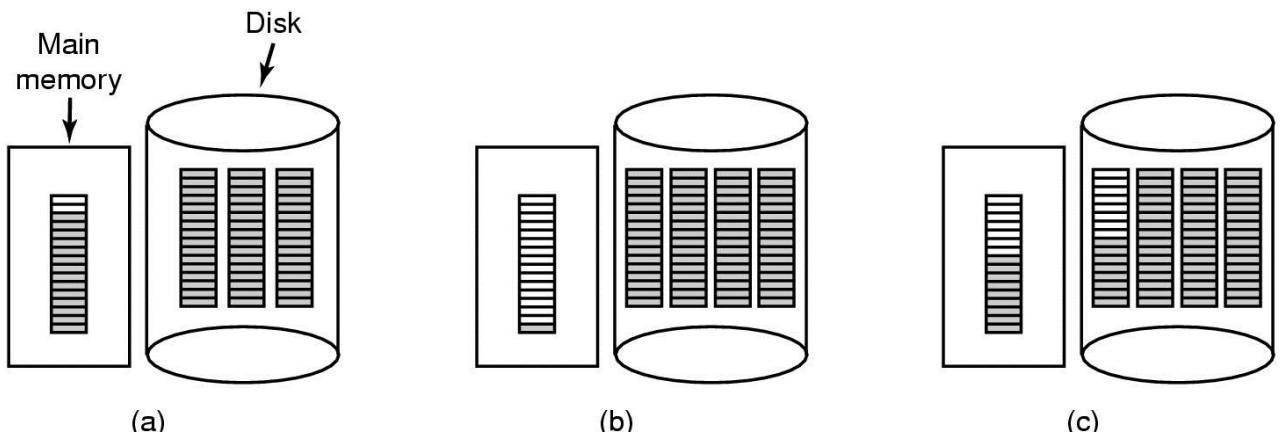


Fig 6.22. (a) Një bllok pointerash plot përt të liruar blloqet ne memorien kryesore dhe tre blloqe të pointerit ne disk. (b) Rezultati i lirimt të një file me tre blloqe. (c) Një strategji alternativa përt të trajtuar tre blloqet e lira, entry-it e vijezuara paraqesin pointerat ne blloqet e liruara.

Një menyre për të evituar shumicen e disk I/O, është ndarja e të gjithe bllokut të pointerave. Qe do të thotë se kur blloqet lirohen, nuk kalohet me nga fig. 6.22(a) ne 6.22(b), por kalohet nga 6.22(a) ne 6.22(c), për të bere ketë lidhje. Ne ketë menyre, sisteme mund të trajtoje serine e fileve tëmportale pa nevojen e disk I/O. Në qoftë se blloku ne memorie mbushet, ai shkruhet ne disk dhe blloku gjysem i mbushur lexohet. Ideja, ketu është qe shumica e blloqeve të pointerave ne disk, të mbahen të mbushura(për të minimizuar përdorimin e diskut) dhe mbajtja e atyre gjysem të mbushur ne memorie, keshtu ai mund trajtoje si krijimin e file-ve dhe fshirjen e tyre pa disk I/O ne freelistën.

Me një bitmap, është e mundur të mbahet vetëm një bllok ne memorie, qe shkon ne disk vetëm kur mbushet ose boshatiset. Një përfitim i kesaj metode është: berja e të gjithe alokimit nga një bllok i vetëm i bitmap-it, blloqet ne disk do të mbyllen se bashku, duke minimizuar punen e diskut. Meqenese bitmap është një strukture te dhenash me madhesi fikse, ne rastin kur kerneli është i faqosur, bitmapi mund të vendoset ne memorien virtuale dhe ze faqe prej saj, të faqosura sipas nevojes.

Kuotat e Diskut (disc quotas)

Për të ndaluar njerezit të ‘keqtrajtojnë’ hapesiren e diskut, SO multiuser zakonisht sigurojne një mekanizem për të mbrojtur hapesiren e diskut (kuotat e diskut). Ideja ne ketë rast është qe sistemi administrator t’i lejoje për përdorim çdo useri pjese filesh dhe blloqesh, duke u siguruar ne ketë menyre qe çdo përdorues mos të kapecuje kuotat e tij. Një mekanizem i tille pëershkruehet me poshtë.

Kur një user hap një file, atributet dhe adresat ne disk alokohen dhe vendosen ne një tabele të file-it të hapur ne memorien kryesore. Ndermjet attributeve (karakteristikave), është një entry qe tregon se kush është pronari. Zmadhimi i madhesise se çdo file, i ngarkohet kuotave të diskut.

Një tabele e dytë paraqet të gjitha kuotat për çdo përdorues me një file të hapur, edhe kur file mund të jetë hapur nga dikush tjetër. Tabela tregohet ne fig. 6.23, kur të githe filet jane të mbyllur, të dhenat e filet shkruhen ne file-in e kuotave.

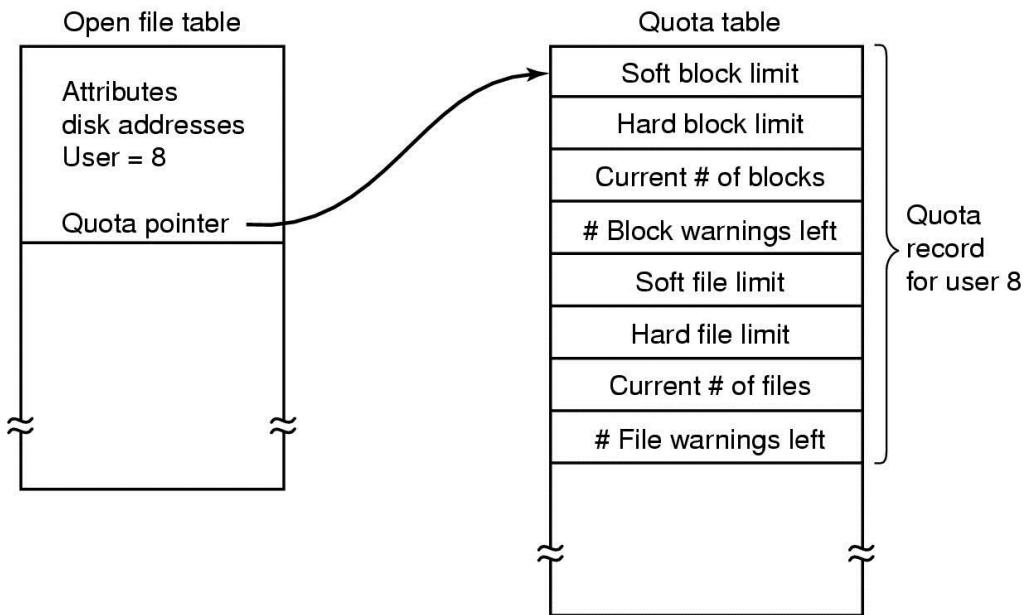


Fig. 6.23. Kuotat analizohen ne një baze për user-at ne një tabele kuotash.

Kur behet një entry e re ne tabelen e file-ve të hapur, ne të dhenat e kuotave të pronarit futet një pointer për të lehtësuar gjetjen e limeteve të veçantë.

Sa here qe një file-i i shtohet një bllok, numri total i bloqeve qe i jane ngarkuar pronarit inkrementohet, dhe i behet një kontroll kunder limitëve hardware-ik ose software-ik. Limitet soft mund të kapërcehen, por ato hard, jo! Përpjekjet për t’iu bashkangjitur një file kur është arritur limiti hard sjell gjenerimin e një error! Kontrolle analoge, gjithashtu ekzistojne edhe për numrin e file-ve.

Kur një user përpinqet të hyje, sistemi kontrollon filen e kuotave, për të pare në qoftë se useri e ka kaluar limitin soft të vendosur për numrin e fileve ose për numrinx e bloqeve. Në qoftë se ky limit është shkelur, shfaqet një paralajmerim, dhe numri i paralajmerimeve të mbetur zvogelohet me 1. Në qoftë se numri i paralajmerimeve shkon ne 0, do të thotë qe përdoruesi e ka injoruar parlajmerimin disa here, dhe nuk është lejuar të hyje. Lejimi për të hyre serish, do të diskutohet nga sistemi administrator.

Kjo metode ka karakteristikën qe usera-t mund ta kalojne limitin soft, gjatë një sesioni. Kjo s’mund të ndodhi me limitet hard.

6.3.6 Besueshmeria e File system

Shkatërrimi i një file system është ndonjëhere, një shkatërim me i madh se shkatërrimi i një kompjuteri. Do të ishte e merzitshme, në qoftë se një kompjuter shkatërrrohet nga zjarri, nga një shkendije elektrike, ose nga disa kafe të derdhura mbi tastjere, dhe një gje e tille do të kushtontë disa lek, por të gjitha keto mund të zevendesohen me lehtësi. PC jo të shtrenjta mund dhe të zevendesohen duke shkuar tek një tregetar.

Në qoftë se një file i një kompjuteri humbet ne menyre të parikthyeshme, përshkak të një defekti hardware-ik ose software-ik, rikthimi i të gjithe informacionit ne gjendjen e meparshme do të ishte shume e veshtire, do të kerkonte me shume kohe dhe ndonjëhere e pamundur. Për disa njerez humbja e disa programeve, dokumenteve, databasa-ve, planeve të marketingut, etj do të ishte me pasoja katastrofike. File system-i nuk mund të siguroje mbrojtjen fizike të sistemit nga shkatërrimi, por mund të siguroje mbrojtjen e sistemit. Ne ketë pjese do të analizohen disa nga menyrat përmirësojshmë e sigurtë të file system-it.

Floppy disket jane zakonisht përfekte për ketë pune, por kjo vetëm kur dalin nga fabrika, sepse shpesh ato prodhojnë blloqe të keqia, gjatë përdorimit. Edhe HDD, zakonisht kane blloqe të keqia duke filluar qe nga startimi. Do të ishte shume e shtrenjtë në qoftë se ato do të prodhoheshin pa asnje defekt. Siç pame ne kapitullin 5, blloket e keqia mund të trajtohen nga një kontroller, i cili i zevendeson sektoret e keqinj me pjese të rezervuara, pikerisht për ketë pune. Përveç kesaj teknike, ka dhe menyra të tjera të besueshme qe do të trajtohen me poshtë.

Backup-et

Shumica e njerezve mendojne se berja e backup-eve është diçka e pavlefshme, vetëm kur ne menyre të papritur disku i tyre ‘vdes’. Kompanitë ia dine vleren te dhenave të tyre dhe për ketë i bejne backup atyre çdo ditë, zakonisht ne një tape. Tape-t moderne sot variojnë nga disa dhjetra GB deri ne qindra GB, dhe çmimi i tyre përcaktohet \$/GB. Por berja e backup-eve nuk është aq e padobishme sa duket.

Backup-et përdoren për të trajtuar një nga problemet e meposhtme:

Ripërtëritja nga një shkatërrim
Ripërtëritja nga një gabim njerezor.

E para mbulon rastet kur kompjuteri ndodh të shkatërritet nga një zjarr, shkendije elektrike, ose një katastrofe natyrore. Ne praktike, keto gjera nuk ndodhin shume shpesh, prandaj shume njerez nuk i kushtojne rendesi backup-it. Të tille njerez, nuk kane as sigurimin nga zjarri ne shtëpitë e tyre, për të njëjtën arsyé.

Arsyeja e dytë, është qe përdoruesit, aksidentalisht, ndodh të fshijne një file, qe me vone mund t'i nevojitet përseri. Ky problem ndodh shume shpesh, prandaj Windows përdor një direktori të veçantë të quajtur **recycle bin**, qe ruan filet e fshira, dhe nga ku përdoruesi mund t'i marri për t'i përdorur kur t'i nevojiten. Backup-et ne një tape mund të rikthejne file qe jane fshire para një kohe të gjatë.

Berja e një backup-i kerkon shume kohe dhe shume hapesire, prandaj është e rendesishme qe ky proces të behet me eficience dhe ne menyre të leverdissime. Keto probleme ngrejne ketë çeshtje: se pari, a duhet bere backup të gjithe file system-it apo vetëm një pjese të tij?!

Shumica e instalacioneve dhe te programeve te ekzekutueshme (ne kod binary) ruhen ne pjese te limituara te pemes se file system-it. Nuk eshtë e nevojshme ti behet backup fileve, qe mund t'i behet instalimi nga CD-ROM-i i prodhimit. Gjithashtu, shumica e SO, kane direktori për file temporale, të cilat përdoren edhe keto për qellimin e backup-it. Ne UNIX, të gjithe file-t e veçantë (të pajisjeve I/O), ruhen ne direktorine /dew. Ne ketë direktori backup-i, jo vetëm qe nuk eshtë i nevojshem, por do të ishte dhe i rrezikshem, sepse backup-programi do të priste pafundesisht qe ky proces të përfundoje. Përfundimisht, eshtë e pelqyeshme t'i behet backup vetëm direktorive të veçanta dhe çdo gjeje brenda tyre, përvèç komplet file system-it.

Se dyti, eshtë e kotë t'i besh backup fileve qe nuk kane ndryshuar nga backup-i fundit, kjo na çon ne idene e inkrementimit të ‘depozitave’. Me e thjeshta për keto ‘depozita’ eshtë t'i behet ne menyre periodike një backup i plotë (javore ose mujore), dhe t'i behet një backup ditor atyre file-ve qe kane ndryshuar nga backup-i i fundit. Akoma me mire eshtë t'i behet backup, vetëm atyre fileve, qe kane ndryshuar. Kjo skeme minimizon kohen e backup-it, por e bën ripërtëritjen me komplekse.

Kjo sepse si fillim behet ripërtëritje e backup-it të plotë, i cili me pas ndiqet nga ‘depozitat’ inkrementuese të rradhitura ne kah të kundert. Për një ripërtëritje me të thjeshtë përdoren metoda me të sofistikuara, ketu përfshihen dhe skemat me ‘depozitat’ inkrementuese.

Se treti, sasia e madhe e te dhenave seleksionohet ne grupe, dhe do të ishte e pelqyeshme të behet me pare kompresimi i te dhenave, e me pas të shkruhen ne tape. Por, ne shume algoritma kompresimi, mund të ndodhi qe një boshllék i vetëm ne tape-in e backup-it, ta beje tape-in të pa lexueshem. Prandaj duhet të mendohet me kujdes ideja e kompresimit të backup-stream-it.

Se katërti, eshtë e veshtire t'i besh një backup një file system-i aktiv. Në qoftë se file ose direktori shtohen, fshihen ose modifikohen gjatë procesit të backup-it, backup-i përfundimtar nuk eshtë i qendrueshem. Meqenese procesi backup eshtë një proces qe kerkon kohe, do të ishte e nevojshme, qe sistemi të mbahej jashtë përdorimit për një natë, por një gje e tille eshtë e papranueshme. Për ketë arsy, algoritmat jane ndertuar ne atë menyre, qe të bejne ‘një fotografi’ të çastit të strukturave kritike të te dhenave të file system-it, dhe me pas të kerkojne ndryshime të metejshme të file-ve dhe direktorive duke i kopjuar ato, ne vend qe t'i update-oje ato. Ne ketë metode, ne momentin e kopjimit të gjendjes se çastit të sistemit, sistemi eshtë i ngrire.

Se pesti dhe se fundmi, berja e backup-eve sjell shume probleme jotekejke brenda një organizimi. Sistemi me i mire i sigurimit, mund të behet i pavlefshem, në qoftë se sistemi administrator i mban të gjithe file-et backup ne zyren e tij dhe e le atë të hapur për të zbritur ne sallë për të marre disa printime. Ajo, qe mund të bënte një spiun do të ishte t'i hidhte të gjithe të dhenat ne një disk dhe të zbristë poshtë me delikatese. Ne ketë rast: Mirupafshim sigurim! Gjithashtu një backup ditor nuk do të funksiononte ne rastin e një zjarri qe po djeg kompjuterin, ose qe po djeg disqet ku ai eshtë ruajtur. Prandaj disqet duhet të mbahen ne vende ku jane me pak të rrezikuara, por edhe kjo ka anet e veta

negative. Me poshtë do të trajtohen vetëm çeshtje teknologjike, qe kane të bejne me backup-in e sistemit.

Përdoren dy strategji për të bere një backup të një diskut ne një tape: një **backup fizik** ose një **backup logjik**. Një backup fizik starton ne adresen 0 të diskut, shkruan të gjithe blloqet e diskut ne tape-in output sipas rrades, dhe ndalon kur është kopjuar blloku i fundit. Një program i tille është i thjeshtë, gje qe nuk mund të thuhet për programet e tjera të përdorshme.

Është e rendesishme të behen disa komente për backup-in fizik.

Nga një ane nuk ka vlerë t'i besh backup blloqeve të papërdorura. Keshtu, në qoftë se një program backup-i ka akses ne struktura te dhenash të vendosura ne blloqe të lira, ai mund të menjanoje backup-in e blloqeve të papërdorura. Megjithatë, kapërcimi i blloqeve të papërdorura kerkon shkrimin e numrit të çdo blloku para çdo blloku (ose ekuivalentin), meqenese nuk është gjithmone i vertetë fakti qe blloku *k* ne tape të jetë blloku *k* ne disk.

Një shqetësim tjetër është backup-i i blloqeve të keqia. Në qoftë se të gjithe blloqet e keqia riadresohen nga kontollerri i diskut dhe i fshihen SO (siç u pëershkrua ne kapitullin 5.4.4), backup-i fizik do të funksiononte shume mire. Por në qoftë se ato jane të dukshme për SO dhe mbahen ne ‘bishtin e blloqeve të keqia’ ose ne bitmap-e, është shume e rendesishme, qe programi i backup-it të marri te drejtën për të hyre ne ketë informacion dhe të shbangi backup-in e tyre, për të parandaluar erroret e pafundme të leximit të diskut gjatë ketij procesi.

Avantazhet kryesore të backup-it fizik jane: thjeshtësia dhe shpejtësia e madhe (mund të arrije shpejtësine e diskut).

Disavantazhet kryesore të backup-it fizik jane: paaftësia për të kapërcyer direktoritë e zgjedhura, për të bere inkrementimin e ‘depozitave’ dhe ruajtja e file-ve individuale sipas kerkeses.

Një backup logjik fillon ne një ose me shume direktori specifike dhe grumbullon ne menyre rekursive të gjithe file-t dhe direktoritë e gjetura qe kane pesuar ndryshime nga një database i meparshem (për shembull, një backup për një ‘depozitë’ inkrementuese, ose një sistem instalimi për një backup të plotë). Për ketë ne një backup logjik, tape i ‘depozitave’ ka një një seri direktorish dhe filesh të identikuara me kujdes, qe e bejne me të thjeshtë ripërtëritjen e një file të vetëm ose direktoritë, sipas kerkeses.

Meqenese backup-i logjik është forma me e përdorur, le të analizojme ne detaje një algoritem të zakonshem duke përdorur shembullin e fig 6.24. Shumica e sistemeve UNIX e përdorin ketë algoritem. Ne figure shikohet një peme filesh me direktori (katroret) dhe file (rrathet). Objektet e hijezuara jane modifikuar qe nga baza e të dhenave dhe prandaj duhet të behet backup-i. Objektet e hijezuara nuk kane nevoje t'i behet backup-i.

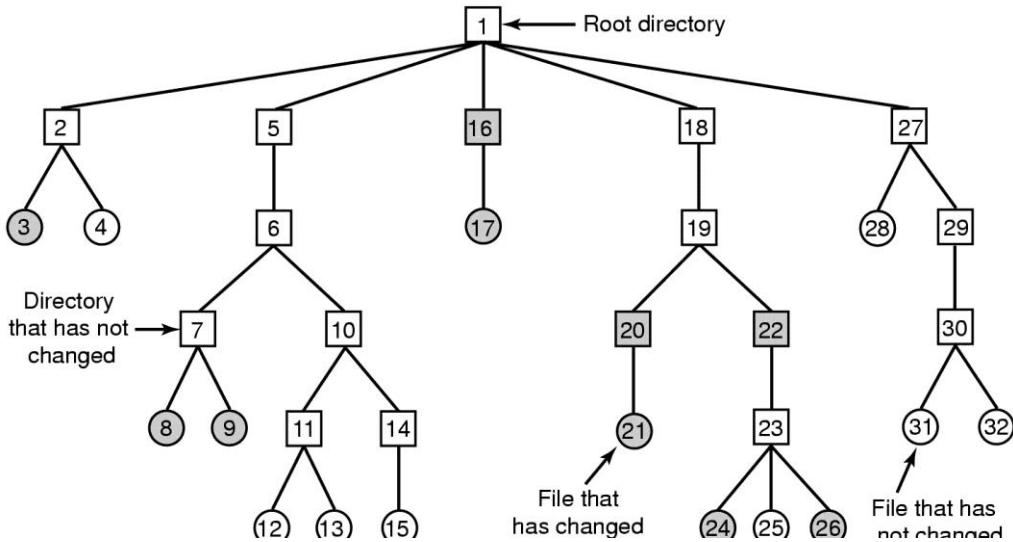


Fig. 6.24. Një file system qe do t'i behet backup-i. Katroret jane direktori kurse rrathet jane file. Objektet e hijezuara jane te dhenat e modifikuar nga backup-i i fundit. Çdo direktori dhe file etiketohet me numrin e i-node-it përkatës.

Ky algoritem, i bën backup të gjithe direktorive (madje dhe atyre të modifikuara) qe kane të njëjtin rrugekalim me një file ose direktori të modifikuar për dy arsyet:

Se pari, për të bere të mundur ripërtëritjen e file-ve dhe direktorive ne një file system të ri ne një kompjuter tjetër. Ne ketë menyre backup-i mund të përdoret për të transportuar një file system të tere midis kompjuterave.

Arsyeja e dytë e berjes backup direktorive të pamodifikuara është, për realizimin e një ripërtëritje inkrementuese të një file të vetëm (zakonisht është ripërtëritje nga një gabim njerezor). Supozojme se një backup i plotë është bere mbasditen e se djeles dhe një backup tjetër mbasditën e se henes. Të martën, direktoria `/usr/jhs/pmj/nr3` fshihet, me të gjithe direktoritë dhe file-t qe mbulonte. Të merkuren ne mengjes, përdoruesi do të doje të ripërtërije file-in `/usr/jhs/pro/nr3/plans/summary`. Por nuk është e mundur qe t'i behet ripërtëritje vetëm file-it `summary`, sepse nuk ka vend ku ai të vendoset.

Duhet, qe ne fillim, t'i behet ripërtëritje direktorive *numer 3* dhe *plans*. Për të marre informacion mbi pronarin e tyre, kohen, etj, keto direktori duhet të jene ne tape-in e backup-it, edhe pse ato mund të mos kene ndryshuar nga backup-i i fundit i bere.

Algoritmi i backup-it mban një indeks të bitmap-it, me disa bite për një i-node. Ky algoritem operon ne katër faza.

Faza 1 fillon ne direktorine e startimit (për shembull ne root) dhe ekzaminon të gjithe hyrjet ne të. Për çdo file të modifikuar, i-node përkatës është i shkruar ne bitmap. Çdo direktorie i vendoset një i-node ne bitmap, është apo jo modifikuar ai dhe me pas behet inspektimi ne menyre rekursive. Ne fund të fazes se pare, të githe filet e direktoritë e modifikuara kane të shkruar ne bitmap-in e tyre i-node-in përkatës, siç tregohet ne fig 6.25(a), (pjesa e hijezuar).

Ne fazen 2, po ne menyre rekursive analizohet pema për të dalluar file-et e direktoritë e pamodifikuara, të cilave u hiqen i-node-t nga bitmap-i. Kjo faze e le direktorine siç

tregohet ne fig. 6.25(b). Vihet re se direktoritë 10, 11, 12, 27, 29, 30, jane tani të pashnuara, sepse ato nuk përbajne ndonjë gje qe të jetë modifikuar. Atyre nuk do t'i behet backup. Kurse direktorive 5, 6 do t'i behet backup edhe pse ato vetë nuk jane modifikuar, kjo sepse ato do të duhen për të ripërtërire ndryshimet ne një makine tjeter. Prandaj 1 dhe 2 Jane të kombinuara ne një drejtim të vetëm të pemes.

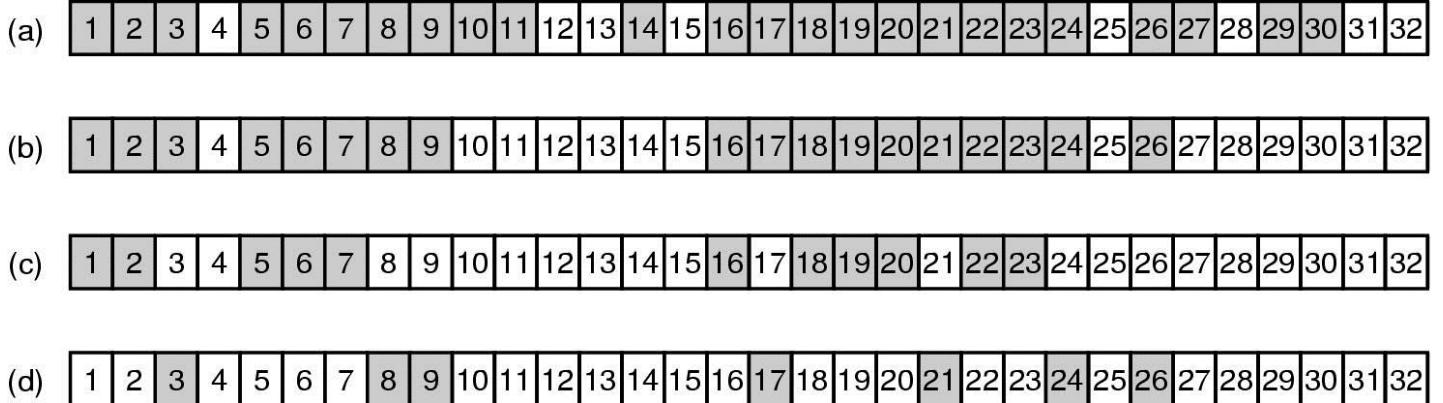


Fig. 6.25. Bitmap-et e përdorura nga algoritmi i backup-it logjik.

Ne ketë pike, njihen direktoritë dhe file-et, të cilave do t'i behet backup. Jane ato të shnuara ne fig. 6.25(b).

Faza 3 konsiston ne skanimin e të gjithe i-node-ve sipas rradhes numerike dhe ne berjen backup të të gjithe direktorive të shnuara për backup. Keto tregohen ne fig 6.25(c). Çdo direktorie i vendoset një prefiks nga karakteristikat e direktorise (pronari, koha ,etj), ne ketë menyre ato mund të ripërtërihen.

Përfundimisht, ne fazën 4, file-eve të treguar ne fig.6.25(d), i behet gjithashtu backup. Përseri atyre i vendoset një prefiks nga karakteristikat e tyre. Kjo përfundon komplet backup-in.

Edhe pse ripërtëritja e file system-it nga tape-et e backup-it behet ne menyre të drejtepërdrejtë, ka disa çeshtje problematike. Meqenese lista e blloqueve të lira nuk është një file, asaj nuk i behet backup dhe keshtu ajo duhet të rindertohet pas fshirjes, pasi behet ripërtëritja. Një gje e tille është e mundur të behet mqs bashkesia e blloqueve të lira është komplementi i bashkesise se file-ve qe ruhen ne të gjithe file-et e kombinuara.

Një tjetër çeshtje është linkimi (lidhja). Në qoftë se një file është i lidhur me dy ose me shume direktori, është e rendesishme qe file të ripërtërihet vetëm një here, dhe po keshtu të bejne dhe direktoritë e lidhura me të.

Një tjetër çeshtje është fakti qe shumica e file-ve ne UNIX mund të përbajne vrima. Është e lejueshme, qe të hapet një file, të shkruhen disa bit ne të, dhe të largohesh një fare distance nga offseti i file-it dhe të shkruhen disa bite të tjera. Blloqet ndermjet nuk jane pjese të file-it dhe nuk i duhet bere backup, e akoma me tepër ripërtëritje. File-t e Core-s

kane një vrime të madhe mes segmentit të datave dhe stack-ut. Në qoftë se kjo nuk trajtohet ashtu siç duhet, çdo file i Core-s i ruajtur, do ta mbushi ketë hapesire me zero me të njëjtën madhesi të adreses virtuale të hapesires (për shembull 2^{32} dhe akoma me keq 2^{64} byte).

File speciale të quajtura **pipe**, nuk i duhet bere backup, pavaresisht se ne cilen direktori ndodhen (ato nuk duhet të kufizohen ne direktorine `/dev`).

Qendrueshmeria e File System-it

Besueshmeria e file system-it është e lidhur ngushtë dhe me qendrueshmerine e file system-it. Shume file system-e lexojnë blloqe, i modifikojne ato dhe i shkruajne ato me vone. Në qoftë se sistemi shkatérrohet përpara se të gjithe file-ve të modifikuara t'i behet backup, file sistemi futet ne një gjendje të paqendrueshme. Ky problem është veçanerisht shume kritik, në qoftë se disa nga blloqet qe nuk jane shkruar Jane blloqe i-node, blloqe direktorish, ose blloqe qe përbajne një free list.

Për të menjanuar problemin e paqendrueshmerise, shumica e kompjuterave, kane një program qe kontrollon qendrueshmerine e file system-it. Për shembull, UNIX ka *fsck*, kurse Windows ka *scandisk*. Programi mund të ekzekutohet ne çdo moment qe boot-ohet sistemi, veçanerisht pas një shkatërrimi. Përshkrimi me poshtë tregon si funksionon *fsck*. *scandisk* është diçka ndryshe sepse ai operon ne një file system ndryshe, por kane të njëtin princip përdorimi dhe riparimi të file system. Çdo kontrollues i file system, verifikon çdo file system (çdo particion disku) ne menyre të pavarur nga të tjeret.

Mund të behen dy tipe kontrollesh të qendrueshmerise: të **blloqeve** dhe të **file-ve**.

Për të bere kontrollin e qendrueshmerise se një blloku, programi nderton dy tabela, çdo njëra përmban një numerues për çdo bllok, qe inicializohet me 0.

Numeruesi ne tabelen e pare numeron se sa here, përdoret një bllok ne një file; numeruesi ne tabelen tjetër rregjstron se sa here është përdorur një bllok ne free listën (ose ne një bitmap me blloqe të lire).

Programi me pas lexon të gjithe i-node-et. Duke filluar nga një i-node, mund të ndertohet një listë të të gjithe numrave të blloqeve të përdorura ne një file të caktuar. Nderkohe qe lexohet një numer blloku, numeruesi i tij ne tabelen e pare inkrementohet. Me pas programi analizon free listën ose bitmap-in, për të gjetur të gjithe blloqet qe nuk jane ne përdorim. Çdo prezence e një blloku ne free list, bën qe numeruesi ne tabelen e dytë të inkrementohet.

Në qoftë se file system është i qendrueshem, çdo bllok do të ketë një 1, ne një nga tabelat, siç tregohet ne fig. 6.26 (a). Pas një shkatërrimit të file system, tabelat do të duken si ne fig. 6.26(b), ku blloku 2 nuk është prezantë ne asnjë nga tabelat. Ai do të raportohet si një bllok i humbur (**missing block**). Edhe pse blloqet e humbura nuk shkaktojne ndonjë demtim të madh, ato zene shume hapesire, duke zvogeluar ne ketë menyre kapacitetin e

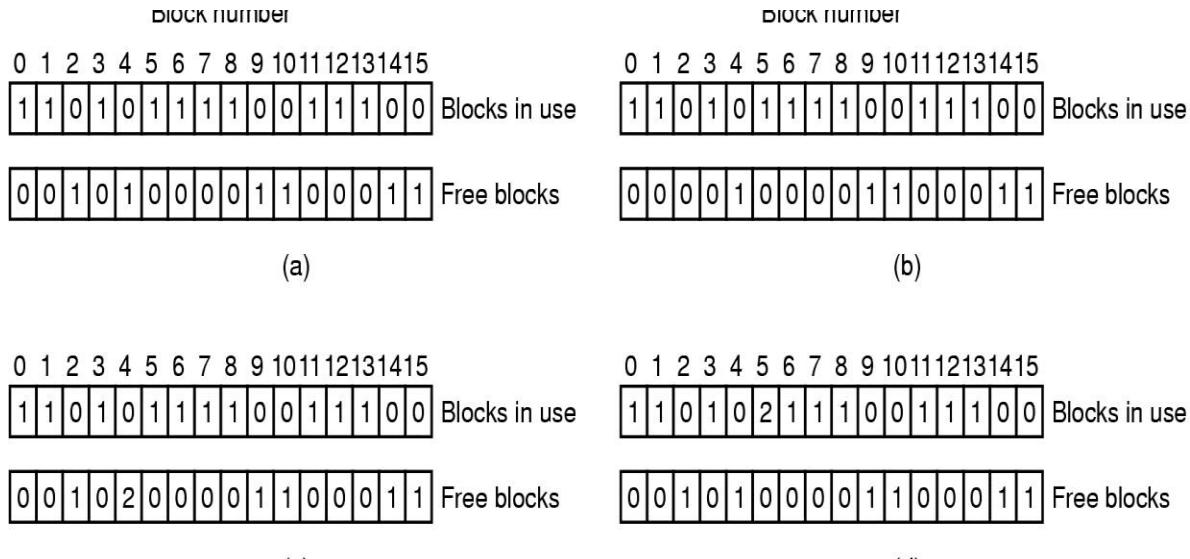


Fig. 6.26 Gjengjet e sistemit. (a) I qendrueshem (b) Missing block (c) bllok i duplikuar ne free list (d) bllok me data i duplikuar.

diskut. Zgjidhja për **missing block** është e drejtepërdrejtë: kontrolluesi i file system-it, thjesht i shton ato ne free list.

Një situatë tjetër mund të jetë ajo e fig. 6.26(c). Ku dallohet një bllok (me numrin 4) qe paraqitet dy here ne free list (duplikimi mund të behet vetëm në qoftë se free list është me të vertetë një list, me bitmap një gje e tille është e pamundur). Zgjidhja ne ketë rast është e thjeshtë: të rindertohet lista.

Gjeja me e keqe, qe mund të ndodhi është qe i njëjtë bllok te dhenash të shfaqet ne dy ose me shume file, siç ka ndodhur ne figuren 6.26(d), me bllokun 5. Në qoftë se, ndonjë nga file-et fshihet, blloku 5 do të vendoset ne free list, duke na çuar ne një situata, ku i njëjtë bllok është dhe ne përdorim, por edhe i lire ne të njëjtën kohe. Veprimi i duhur qe duhet të beje kontrolluesi i file system-it, është të alokoje një bllok të lire, të kopjoje përbajtjen e bllokut 5 ne të, dhe ta fusi kopjen ne një nga file-et.

Ne ketë menyre, përbajtja e informacionit nuk ndryshon, por struktura e file system-it, është të paktën e qendrueshme. Gabimi duhet të raportohet me qellim qe përdoruesi të insepektuje dhe ta korigjoje atë. Përveç kontrollimit për të pare, qe çdo bllok numerohet siç duhet, kontrolluesi i file system-it, kontrollon gjithashtu dhe sistemin e direktive. Ai, gjithashtu përdor një tabelë për numeratoret, por keto jane numeratore për file dhe jo për blloqe. Ai fillon ne direktorine e root-imit, dhe zbret ne menyre rekursive ne peme, duke inspektuar çdo direktori ne file system. Për çdo file ne çdo direktori, ajo inkrementon një numerues për numrin e përdorimeve të atij file. Kujtojme, qe përshkak të linke-ve hard, një file mund të paraqitet ne dy a me shume direktori. Symbolic links nuk numerojnë dhe as nuk shkaktojnë inkrementimin e numeruesit të file-it.

Kur e gjithe kjo përfundon, krijohet një listë, e indeksuar nga i-node-t, qe tregon sa direktori përmban çdo file. Me pas behet krahasimi i ketyre numrave me numrin e linkeve të ruajtura ne vetë i –nodeit. Ky numerim fillon tek 1-ishi, kur krijohet një file dhe behet inkrementimi i tyre sa here qe behet një (hard) link me file-in, ne një file sistem të qendrueshem, të dy numerimet do të bien dakort. Megjithatë, mund të ndodhin dy gabime: numri i linkut ne i-node mund të jetë shume i vogel ose shume i madh.

Në qoftë se numri i linkut është me i madh se numri i hyrjeve ne direktori, atëherë edhe pse mund të behet e gjithe fshirja e file-ve ne direktori, numri pas fshirjes do të jetë një numer jo zero, dhe i-node nuk do të fshihet.

Ky gabim nuk është dhe kaq serioz, por humbet hapesire ne disk me file qe nuk jane ne ndonjë direktori. Mund të rregullohet duke vendosur numrin e linkut dhe të i-node-it ne vleren e duhur.

Gabimi tjetër është pothuajse katastrofik. Në qoftë se dy entry ne direktori janë të lidhura me një file, nderkohe qe i-node tregon qe ka vetëm një, kur ne fakt nuk është fshire asnjë entry e direktorise, por numri i i-node shkon ne 0. Kur një numer i i-node shkon ne 0, file system e shenon atë si jo të përdorshem dhe i liron të gjithe blloqet e tij. Ky veprim mund të ndikoje ne një nga direktoritë qe do të pointojë tek i-node jo me i përdorshem, blloqet, e të cilit mund ti përkasin file-ve të tjere. Përseri zgjidhja do të ishte marra vesh mes numrit të i-node me numrin e entry-ve të direktorive.

Keto dy operime: kontrolli i direktorive, kontrolli i blloqeve, janë të integruara për arsy efiçencë. Behen dhe çekime të tjera. Për shembull direktoritë kane një format të përcaktuar, me numrin i-node dhe me emra ASCII. Në qoftë se, një numer i-node është me i madh se numrat e i-node ne disk, atëherë thuhet se direktoria është demtuar.

Për me tepër, çdo i-node ka një gjendje, disa prej të cilave janë të lejueshme, por të çuditshme, si për shembull 0007, qe nuk i jep as pronarit as grupit të tij të drejtë hyrje, por lejon të jashtmit të lexojne, të shkruajne, dhe të ekzekutojne file. Do të ishte me leverdi, qe të paktën të reportoheshin file, qe i japin me shume të drejta të jashtmeve se sa vetë pronarit. Disa direktori, të themi rrëth 1000 entry, dyshohen të jene të tillë. File të lokalizuara ne direktoritë e user-ave, por qe gjithashtu zotërohen dhe nga superuser-i i tyre dhe qe kane bit-in e SETUID, janë probleme të sigurimit, sepse file të tillë kerkojne veprimimin e superuser-it kur ato ekzekutohen nga ndonjë user.

Ne paragrafet e meparshem, u diskutua mbi mbrojtjen e përdoruesit nga shkatërrimi i sistemit. Disa file system –e, shqetësohen gjithashtu dhe për mbrojtjen e userit nga vetë sistemi. Keshtu, në qoftë se user-i shtyp:

rm* , o për të fshire të gjithe file-t a qe mbarojne me, *o*, por aksidentalisht shtypet

rm* -o kjo komande do të fshije të gjithe file-t ne direktorine përkatëse, dhe me

pas do të ankohet qe nuk gjen dot ,*o*. Ne MS/DOS dhe ne disa sisteme të tjera, kur file-et fshihen, ajo qe ndodh është: një bit vendoset ne direktorine ose ne i-node-in e file-it të

fshire për ta shenuar atë si një file të fshire. Tek një free list, nuk kthehen blloqet e liruara nga fshirja e file-it, sepse ndodhe qe ato mund të nevojiten. Në qoftë se, përdoruesi e zbulon shpejt gabimin, është e mundur të ekzekutohet një program ndihmes, qe ripërtërin file-t e fshira.

6.3.7 Performance e File System

Aksesimi ne disk është shume me i ngadaltë se aksesimi ne memorie. Leximi i një fjale ne memorie mund të zgjasë 10 nsec. Leximi nga diskut mund të behet me një shpejtësi 10Mb/sec, qe është 40 here me i ngadaltë se leximi i fjalet 32bit, kesaj i shtohen 5-10 msec për kohen e kerkimit ne disk dhe të pristes derisa të vije sektori i duhur poshtë kokes lexuese. Në qoftë se nevojitet një file i vetëm, aksesimi ne memorie është ne rendin e miliona hereve me të shpejt se aksesimi ne disk. Si rezultat i kesaj diferenca ne kohen e aksesimit, shume file system jane dizenuar me një variacion optimizimesh, për të përmiresuar performancen. Ne ketë pjese do të trajtohen tre prej tyre.

Cashing

Metoda me e zakonshme e përdorur për të zvogeluar kohen e aksesimit të ne disk është blloku cache, ose buffer cache. Ne kontekst, cache-ja është një bashkesi blloqesh, qe logjikisht duhet t'i përkasin diskut, por mbahet ne memorie për arsyet performance.

Algoritma të ndryshem përdoren për menaxhimin e cache-s, por një i zakonshme do të ishte kontrolli i të gjithe kerkesave të lexuara, për të pare në qoftë se blloku i kerkuar ndodhet ne cache.

Në qoftë se ai është, kerkesa për lexim mund të behet pa nevojen e aksesimit ne disk. Në qoftë se, blloku nuk ndodhet ne cache, fillimisht kerkohet ne cache, e me pas kopjohet aty ku është e nevojshme.

Operimi ne cache tregohet ne fig. 6.27. Meqenese ka shume blloqe ne cache, duhet qe procesi të përfundoje shpejt në qoftë se një bllok i dhene është prezent. Menyra e zakonshme është të ngatërrrosh pajisjet me adresat dhe të shikosh rezultatet ne tabelen hash. Të gjithe blloqet me me të njëjtën vlere hash-i vendosen zinxhir ne një linked list, ne menyre qe të mund të ndiqet njëpasnjëshmeria ne zinxhir.

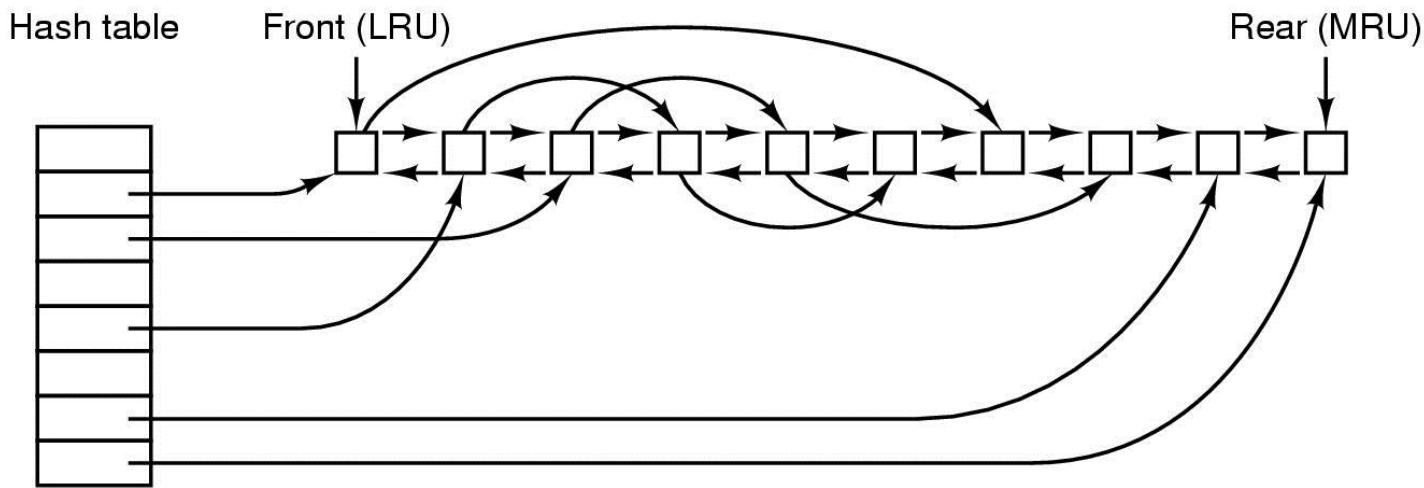


Fig. 6.27 Struktura data ne bufferin e cache-es.

Kur një bllok duhet të ruhet ne një cache të mbushur, disa blloqe duhet të largohen prej saj(dhe të rishkruhen ne disk, në qoftë se jane modifikuar qe kur jane marre heren e fundit). Situata është pak a shume si faqosja (paging), dhe jane të vlefshme të gjithe algoritmat e diskutuara ne kapitullin 3.

Një ndryshim interesant mes faqosjes dhe caching, është se referencat e caches janë relativisht jo të rradhitura, prandaj është e nevojshme qe bllojet të ruhen të gjitha sipas rradhes se LRU-se, me linked lista.

Ne fig 6.27, ne shikojme qe zinxhire, qe startojne ne tabelen hash, ka gjithashtu, një list dy drejtimeshe qe ekzekutohet sipas rradhes se ekzekutimit, me bllokun me shume të përdorur ne fillim të listës dhe me bllokun me pak të përdorur ne fund të listës.

Kur i referohemi një blloku, ai zhvendoset nga pozicioni i tij ne list dhe vendoset ne fund të saj, ne kete menyre ruhet rradha sipas LRU. Por, fatkeqisht ndodh një kurth. Tani, qe kemi situatën me LRU-ne e mundur, shfaqet LRU jo e deshirueshme. Ky problem është i lidhur me shkatërrimet e file system-it, qe u diskutua me pare.

Në qoftë se një bllok kritik, siç është për shembull një bllok i-node, lexohet ne cache dhe modifikohet, por nuk rishkruhet ne disk, ky proces e fut sistemin ne një gjendje jo të qendrueshme. Në qoftë se blloku i-node, vendoset ne fund të zinxhirit të LRU-se, atërehore do të duhet shume kohe të shkoje ne fillim të tij dhe me pas të rishkruhet ne disk. Për me tepër referimi dy here i të njëjtit bllok brenda një intervali të shkurtër është e rradhe. Keto çeshtje, na çojne ne një skeme LRU të modifikuar, duke marre parasysh dy faktore:

A do të duhet blloku përseri shume shpejt?

A është blloku thelbesor ne qendrueshmerine e sistemit?

Për të dyja pyetjet, blloqet mund të ndahen ne kategori si: blloqet i-node, blloqet indirektë, blloqe direktorish, blloqe plot me data dhe blloqe pjeserisht të mbushur me data.

Blloqet qe nuk nevojiten me shkojne ne fillim të listës, keshtu buffer-at e tyre do të përdoren shume shpejt. Blloqe qe mund të nevojiten përseri, siç jane blloqet pjeserisht të mbushura me data, shkojne ne fund të listës dhe do të rrine aty për një kohe të gjatë.

Pyetja e dytë është e pavarur nga e para. Në qoftë se blloku është i rendesishem ne qendrueshmerine e sistemit (çdo gje përvèç blloqeve të datave), dhe ai është modifikuar, ai duhet të shkruhet menjëhere ne disk. Duke i shkruar blloqet kritike shpejt ne disk, ne zvogelojme propabilitetin qe të ndodhe një përplasje ne sistem.

Është e qartë qe blloqet me data mbahen ne cache për një kohe të gjatë derisa të shkruhen ne disk.

Mendoni dike, qe është duke shkruar një liber ne një kompjuter. Edhe pse shkruesi ne menyre periodike jep komandan për te ruajtur materialin ne disk, por ndodh qe i gjithe materiali të jetë ne cache dhe ne disk mos te kte asnje gje. Në qoftë se ndodh një përplasje e sistemit, struktura e file system-it nuk nderpritet, por e gjithe puna do të humbasi.

Kjo situatë nuk para ndodh shume shpesh. Sistemet kane dy menyrat për të zgjidhur ketë problem:

1. sipas UNIX: ekziston një thirrje system, *sync*, qe i detyron të gjithe blloqet e modifikuara të shkruhen ne menyre të menjëherëshme ne disk. Kur sistemi startohet, një program *update*, startohet ne sfond për tu futur ne një loop të pafundme, dhe qendron mes thirrjeve sistem për 30 sec. Si përfundim, jo me shume se 30 sec pune humbasin, përshkak të ndonjë përplasje.
2. Sipas MS/DOS-it: çdo blok i modifikuar duhet të shkruhet ne disk, aq shpejt sa është shkruar. Cache-të ne të cilat, blloqet e modifikuara, shkruhen ne disk ne menyre të menjëherëshme quhen **write-through cache**. Ato kerkojne me shume disk I/O se cache-t jo write through. Diferenca mes ketyre dy cache-ve mund të shikohet kur një program shkruan ne një bllok 1KB, një karakter për një çast kohe. UNIX do t'i mbledhi të gjitha karakteret ne cache dhe do ta shkruaje bllokun ne disk çdo 30 sec ose kurdo qe blloku të zhvendoset nga cache-ja. MS-DOS do të beje një aksesim ne disk për çdo karakter të shkruar. Shumica e programeve bejne një bufferim të brendshem, keshtu qe normalisht ata nuk shkruajne vetëm një karakter, por një rresht ose një njësi të madhe të thirrjes system **write**.

Si pasoje e kesaj diferenca ne strategjine e caching, vetëm heqja e një floppy disk nga sistemi UNIX pa bere një *sync* do të shkaktoje humbje te dhenash, dhe një file system të nderprere gjithashtu. Me MS/DOS-in nuk rezultojne probleme të tillë. Keto strategji të

ndryshme, u zgjodhen sepse UNIX-i u zhvillua ne një ambjent ku të gjithe disjet ishin HDD dhe jo removable disk, kurse MS/DOS-i startoi ne një floppy disk. Nderkohe qe HDD-et u bëne model, parimet e UNIX me eficience me të madhe, u bëne gjithashtu model dhe u përdoren dhe ne Windows për HDD-t.

Leximi i Parakoheshem i Blloqeve (Block Read Ahead)

Një teknike e dytë e përdorur për të përmiresuar performancen e file system-it është të marresh 10 blloqe ne cache, me pare se ato të nevojiten për të rritur ne ketë menyre shpejtësine e gjetjes (hit rate). Një rast i veçantë është leximi i file-ve ne menyre sekuenciale. Kur një file sistem i kerkohet të prodhoje një bllok k ne një file, ai e bën ketë gje, por kur mbaron bën një kontroll të shpejt cache për të pare në qoftë se blloku $k+1$ është aty. Në qoftë se jo, ai skedulon një komande read për bllokun $k+1$, me shpresen qe kur ai të nevojitet të ketë mberritur ne cache.

Kjo strategji është e vlefshme vetëm për file qe lexohen ne menyre sekuenciale, në qoftë se një file aksesohet ne menyre të rastesishme, kjo strategji nuk funksionon me. Për të pare në qoftë se kjo strategji është e vlefshme, file sistem-i analizon aksesimin për çdo model file-esh të hapur. Për shembull, një bit i shoqeruar me çdo file, analizon në qoftë se një file është aksesuar ne menyre sekuenciale, apo ne menyre të rastesishme. Fillimisht, file-it i jepet e drejta për të dyshuar dhe ta vendosi bitin ne menyren e aksesimit sekuencial. Por kur behet kontrolli, biti fshihet. Në qoftë se leximi sekuencial fillon të behet përseni, bit setohet përseni. Keshtu file sistem merr një vendim të arsyeshem, në qoftë se ai duhet të vazhdoje të lexoje ne menyre të parakoheshme apo jo. Në qoftë se vendos gabim, nuk ndodh gje me sistemin vetëm sa humbet pak gjeresi brezi i diskut.

Reduktimi i Levizjes se Krahut të Diskut.

Caching dhe leximi i parakohshem, nuk jane menyrat e vetme të rritjes se performancës se file system-it. Një teknike tjeter e rendesishme është reduktimi i levizjes se krahut të diskut, duke vendosur blloqe, qe mund të aksesohen pothuajse ne menyre sekuenciale, afer njëri-tjetrit, e preferueshme ne të njëtin cilinder. Kur një file output shkruhet, file system-i duhet të alokoje blloqet çdo çast kohe, sa here qe është e nevojshme. Në qoftë se blloqet e lira jane të rregjistruala ne një bitmap, dhe bitmapi është i ruajtur ne memorien kryesore, do të ishte shume e thjeshtë, për të zgjedhur një bllok të lire sa me afer bllokut të meparshem. Me një free list, një pjese e se ciles ndodhet ne disk, do të ishte shume e veshtire për të alokuar blloqet afer njëri-tjetrit.

Megjithatë dhe me një free list mund të behen disa klasa blloqesh, por ne grupe blloqesh të njëpasnjëshme. Në qoftë se sektoret konsiston ne 512 byte, sistemi mund të përdori blloqe 1KB (2 sektore), por të alokoje të dhenat e ruajtura ne disk ne njësi prej 2 blloqesh(4 sektore). Kjo nuk është njësoj si të kemi një bllok ne disk prej 2 KB, nderkohe cache-ja përdor akoma blloqe 1KB dhe transferimet ne disk behen me blloqe 1 KB. Leximi i një file ne menyre sekuenciale, do të zvogeloje numrin e kerkimeve me një

faktor 2, duke përmiresuar ne menyre të konsiderueshme performancen. Një variacion i kesaj metode do të ishte numerimi i pozicioneve të rrotullimeve. Kur behet alokimi i blloqeve, sistemi mundohet të vendosi blloqet e njëpasnjëshme ne një file, ne të njëtin cilinder.

Një tjetër performance e qafes se shishes, ne sisteme qe përdorin i-node ose çdo gje tjetër ekuivalente me i-node-t, është fakti qe leximi i një file të shkurtër kerkon dy aksesime ne disk: një për i-node-t dhe një për blloqet. Vendosja e i-node-ve të zakonshme tregohet ne fig 6.28(a). Ketu të gjithe i-node-t Jane afer fillimit të diskut, keshtu qe distanca mesatare mes një i-node dhe bllokut të tij do të jetë sa gjysma e numrit të cilindrave, duke kerkuar kohe të gjatë kerkimi.

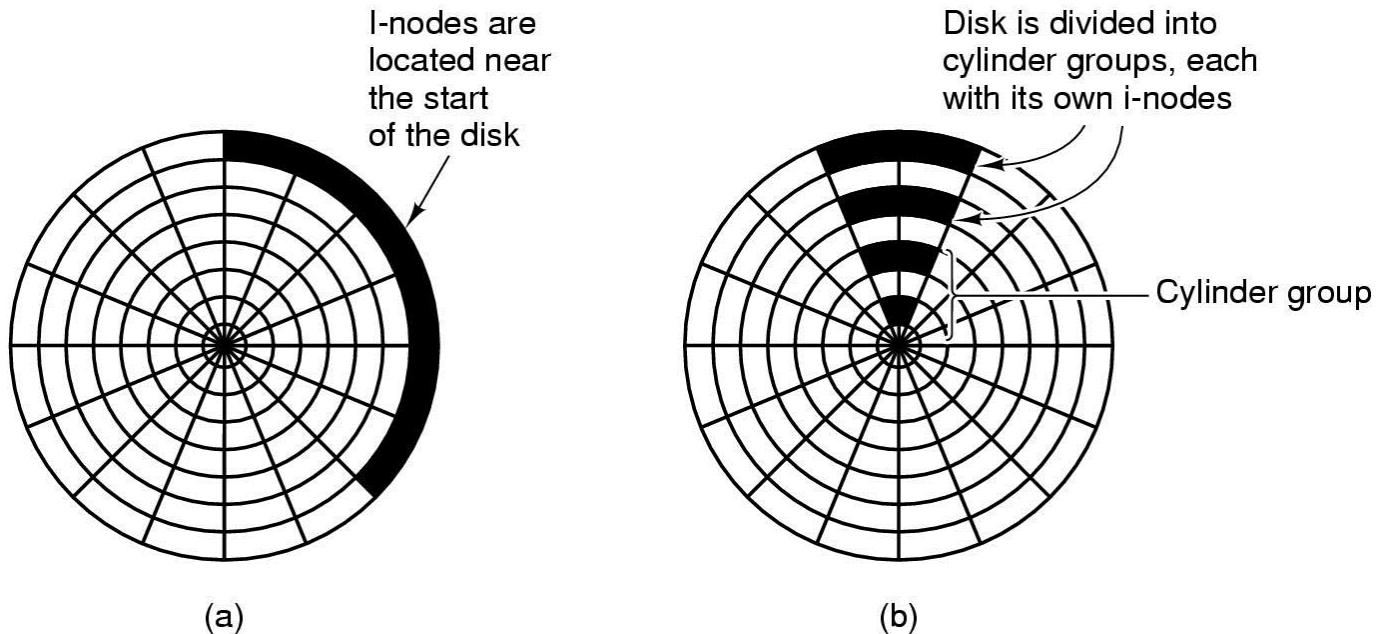


Fig. 6.28 (a) I-node-t të vendosura ne fillim të diskut. (b) disku i ndare ne grupe cilindrash, secili ne blloqet dhe i-node-t përkatëse.

Një përmiresim i thjeshtë i performancës, është t'i vendosesh me mire i-node-t ne mes të diskut, dhe jo ne fillim, duke shkurtuar keshtu mesataren e kerkimit mes i-node-ve dhe blloqeve të tyre me një faktor ose dy. Një tjetër ide, do të ishte ndarja e diskut ne grupe cilindrash, secili me i-nodet, free listën dhe blloqet përkatëse. Kur krijohet një file i ri, mund të zgjidhet çdo i-node. Në qoftë se asnje nuk është i vlefshem, atëherë përdoret një bllok ne një grup cilindrash të afert.

6.3.8 Log-Structured File System

Ndryshimet ne teknologji, jane duke ushtruar presion ne file system-et ekzistuese. Ne veçanti, CPU-t po përpilen të behen me të shpejta, disjet po behen me të medha dhe me të lira (por jo shume të shpejta) dhe memoriet po zmadhohen ne menyre eksponenciale. Parametri qe nuk është përmiresuar, është koha e kerkimit ne disk. Kombinimi i ketyre faktoreve tregon qe përforma e ‘qafes se shishes’ është duke u rritur ne disa file sistem. Ne ketë pjese do të shpjegohet shkurt si funksionon **LFS (Log-structured File System)**.

Idea qe solli LFS-n, është qe CPU-t të behen me të shpejta dhe memoria RAM me e madhe, gjithashtu dhe cache-t jane duke u zmadhuar me shpejtësi. Rrjedhimisht, tani do të jetë e mundur të behet leximi direkt ne cache, pa kerkuar akses ne disk. Keshtu, ne të ardhmen, shumica e aksesimeve ne disk do të jene vetëm për të shkruar, pra mekanizmi i leximit të parakohshem të blloqeve nuk do të duhet me për të përmiresuar performancen.

Marrim rastin me të keq, kur ne shumicen e file system-eve, shkruajtja behet ne pjese të vogla.

Shkrime shume të shkurtra, jane jo shume eficentë, sepse një shkrim ne disk prej 50 msec, ndiqet nga një kerkim 10 msec dhe nga një kohe rroullimi 4 msec. Me keto parametra, eficenca e diskut zvogelohet 1 %.

Për të pare nga vijne, të gjithe shkrimet e shkurtra, krijohet një file ne UNIX. Për të shkruajtur ketë file duhet të shkruhen me pare, i-node për direktorine, bllokun e direktorise, i-node-in për file-in dhe vetë file-in. Nderkohe qe keto shkrime mund të vonohen, i gjithe ky proces bën qe sistemi të haset me probleme serioze qendrueshmerie në qoftë se ndodh një përplasje e sistemit para se të behet shkruajtja. Prandaj shkruajtja e i-nodeve është përgjithesisht me imediate. Për ketë arsy, dizenjuesit e LFS, vendosen të ri-implementojne file system-in e UNIX-it, ne atë menyre qe të shfrytëzohet e gjithe gjeresia e brezit ne disk, edhe ne rastet e aksesimit të shkrimeve të shkurtra të rastit. Idea baze është të strukturohet i gjithe diskut si një log (si një trup i vetëm). Me pas kur ka nevoje për të, të gjithe shkrimet ne pritje të vendosura ne një buffer të memories, grumbullohen ne një segment të vetëm dhe shkruhen ne disk si një segment i vazhdueshem ne fund të log-ut. Një segment i vetëm mund të përbaje i-node, blloqe direktorish, dhe blloqe datash, të gjitha të përziera bashke. Ne fillim të çdo segmenti ka një përbledhje të segmentit, qe tregon se çfare mund të gjendet ne ketë segment. Në qoftë se segmenti mesatar mund të jetë rreth 1 MB, pothuajse i gjithe brezi i diskut mund të përdoret.

Ne ketë dizenjim, *i* node-t akoma ekzistojne dhe kane të njëjtën strukture si ne UNIX, por tani ato jane të shpërdara ne të gjithe log-un ne vend qe të kene një pozicion fiks ne disk. Megjithatë, kur një *i* node lokalizohet, lokalizimi i blloqeve behet ne të njëjtën menyre. Gjetja e një *i* node është shume me e veshtire, sepse adresa e saj nuk mund të llogaritet thjesht nga *i* numrat e saj, si ne UNIX. Për të thjeshtësuar gjetjen e tyre përdoret një hartë *i* node-sh, me indekset e *i* node-ve.

Hyrja e *i*-të ne ketë hartë pointon ne *i* node-in e *i*-të ne disk. Kjo hartë ruhet ne disk, por gjithashtu mund të ruhet dhe ne cache, keshtu pjesa me e madhe e saj do të jetë ne

memorie shumicen e kohes. Kur i node lokalizohet, prej saj mund të gjendet adresa e blloqeve. Vetë blloqet do të jene ne segmentët, diku ne log.

Në qoftë se disqet do të ishin pafundesisht të medha, do të viente përseni përshkrimi i mesipërm. Megjithatë, disqet reale jane të fundem, keshtu qe log-u do të zëj komplet diskun, ne ketë kohe nuk mund të shkruhen segmente të reja ne log. Fatmiresisht, shume segmente ekzistuese mund të kene blloqe qe nuk nevojiten me, për shembull: në qoftë se një file mbishkruhet, i node i tij do të pointoje tanë ne blloqet e reja, por të vjetrat akoma do të zene vend ne segmentet e shkruara me pare. Për të zgjidhur ketë problem LFS ka një **thread pastrues**, qe e kalon kohen duke skanuar log-un ne menyre ciklike, për ta kompaktesuar atë. Ai fillon të lexoje përbledhjen ne segmentin e pare ne log, për të pare cilat i node dhe file ndodhen aty. Me pas kontrollon hartën e i node-ve ekzistuese, për të pare në qoftë se i node-t dhe filet jane akoma ne përdorim. Në qoftë se jo informacioni injorohet. I nodet dhe blloqet qe janë ne përdorim, shkojne ne memorie për tu shkruar ne segmentin tjeter. Segmenti origjinal shenohet si i lire, keshtu log-u mund ta përdori atë për te dhena të reja. Ne ketë menyre pastruesi kalon të gjithe log-un, duke fshire segmente të vjetra dhe duke vodosur ne to te dhena ne përdorim ne memorie, për t'i rishkruar ato ne segmentet e ardhshme. Përfundimisht disku është një buffer i madh ne forme ciklike, me thead-in shkrues, qe shton segmente të reja ne fillim dhe thread-in pastrues qe fshin segmentët e vjetrat nga e kaluara.

Mbajtja e llogarise ne ketë rast është e pavlefshme, meqenese kur një bllok shkruhet ne një segment të ri, i node i file-it (diku ne log), duhet të lokalizohet, dhe të vendoset ne memorie për tu shkruar ne një segmentin e ardhshem. Harta e i node-ve duhet të updateohet për të pointuar ne kopjen e re. Megjithatë është e mundur të behet administrimi, dhe i gjithe ky komplicitet tregon qe ja vlen. Matjet e dhena ne ketë pjese tregojne qe LFS performon nga jashtë UNIX-in, për disa shkrime të shkurtra, nderkohe ka një performance po aq te mire ne lexim dhe ne shkrimet e gjata, si të UNIX-it.

KAPITULLI I SHTATË

Sistemet Operative Multimedial

Filmat digital, video klipet, dhe muzika po behen gjithnjë e me shume një menyre për të paraqitur informacionin dhe argetimin nepërmjet kompjuterit. Audio dhe video files mund të ruen ne një disk dhe mund të luhen (play) serisht kur dicka e tille kerkohet (on demand). Mirepo, karakteristikat e tyre jane shume të ndryshme nga text file-at tradicional. Si rrjedhim, nevojiten file system të rinj ne menyre qe ti mananaxhojne ato. Gjithashtu, ruajta (store) dhe luajta (play) e audio dhe video nxorri ne pah kerkesa të reja ne schedulerin si dhe ne pjeset e tjera të sistemit operativ. Ne paragrafet qe do vijne, ne do të studiojme disa nga keto ceshje dhe implikimet e tyre për sisteme operative qe jane disenjuar për menaxhuar multimedia.

Zakonisht, për filmat digital përdoret emri **multimedia**, qe ne kuptim letrar do të thotë me shume se një mjet (medium). Sipas të ketij përkufizimi, vet ky liber është një pune (detyre) multimediale. Libri, përmban 2 tipe mediash: tekst dhe imazhe (figurat). Megjithatë, pjesa me e madhe e njerezve përdorin termin “multimedia” për të nenkuptuar një dokument qe përmban dy ose me shume media të vijueshme (continuous media), qe do thotë media qe duhet të luhet serisht pas një intervali kohor. Ne ketë liber, termin multimedia do ta përdorim ne ketë sens.

Një term tjetër qe është ka kuptim disi i erret (i dykuptueshem) është “**video**”. Ne sensin teknik, është thjesht porzioni i imazhit të një filmi (movie) (ne të kundert pjeses se zerit)..... Ne fakt, regjistruesit kamera (camcorders) dhe tëlevizionet shpesh kan dy *connectors*, njëri i etiketuar “video” dhe tjetri “audio”, meqene se sinjalat jane të ndare nga njëri tjetri. Mirepo, termi “digital video” zakonisht i referohet produktit të plotë, bashke me imazhet dhe sound. Ne vijim termi “film” (movie) do të përdoret ne sensin e produktit të plotë. Vini re qe ne ketë sens një film nuk ka nevoje të jetë 2ore i gjatë dhe të jetë i prodhuar nga studiot e Hollywood-it me një kosto qe kalon dhe atë të Boeing 747. Një klip prej 30-sec me lajme, qe mund të download-ohet nga faqja e CNN, është gjithashtu një “film” (movie) sipas përkufizimit tone. Keto do ti quajme “video clips” kur i referohemi filmave shume të shkurtër.

7.1 Hyjre ne Multimedia (Parathenie)

Para se të hyjme ne teknologjine e multimedias, do të ishte e dobishme të themi disa fjale për përdorimet e tanishme dhe të se ardhmes të saj. Ne një kompjuter të vetëm, multimedia shpesh ka kuptimin e një luajtjes të një filmi të pararegjistruar nga një **DVD (Digital Versatile Disk)**. DVD-të jane disqe optike qe përdorin po të njëjtën 120-mm boshllek polikarbonat (polycarbonate plastic blanks), ashtu sic përdor dhe CD-ROM, por ato jane të regjistuar ne një densitet me të lartë, duke dhene një kapacitet nga 5 GB deri 17 GB, ne varesi të formatit.

Një tjetër përdorim i multimedia është ai i downloadimit të video klipeve nga interneti. Shume faqe Web kane items qe sapo i klikon, fillon downloadimi i filmave të shkurtër.

Ne një shpejtësi 56Kbps, edhe downloadimi i video klipeve të shkurtër kerkon një kohe të gjatë, por teknologjitet e shpërndarjes të shpjetë (si për shembull **cable TV** dhe **ADSL – Asymmetric Digital Subscriber Loop**) mbizotërojnë, dhe prania e video klipeve ne internet do të rritet tejmase.

Një tjetër sfere ne të cilin multimedia duhet të përkrahet është krijimi i videove. Sistemet e editimit multimedial ekzistojnë dhe për performanca sa me të larta ato kane nevoje të ekzekutohen ne një sistem operativ që mbështet (supports) multimedian aq mire sa një detyre të zakonshme.

Gjithashtu një sfere tjetër ku multimedia po behet e rendesishme është ajo e lojrale kompjuterike. Lojrat shpesh ekzekutojnë video klipe të shkurtra për të përshkruar disa lloj verpimesh. Klipet zakonisht jane të shkurtra, por ka shume si ato dhe klipi i duhur selektohet ne menyre dinamike, ne varesi të veprimeve qe kryen përdoruesi. Keto janë shume të sofistikuara.

Përfundimisht, ceshtja thelbesore e botës se multimedia është **video on demand (video me kerkese)**, me të cilën njerezit do të zgjedhin një filmin që deshirojne vetëm duke përdorur tëlekomenen (ose mouse), dhe ne atë cast do të shfaqet filmi ne ekranin e TV ose monitorin e kompjuterit. Për të mundesuar video on demand (video me kerkese), nevojitet një infrastrukturë e vecantë. Ne Fig. 7-1 ne shohim dy infrastruktura të mundshme me video on demand. Secila përmban tre komponentë esenciale: *një ose disa video servers, një Distribution Network (Rrjet të shpërndare), dhe një kuti (set-top box)* ne çdo shtëpi për të dekoduar sinjalin. **Video server** është një kompjuter shume i fuqishem që mban shume filma ne file sistemin e tij dhe i luan serisht kur ato kerkohen (on demand). Ndonjehere mainframes përdoren si video servers, sepse lidhja e 1000 disqeve tek një mainframe është me e hapur/efikase, sesa lidhja e tyre tek PC. Shume nga materiali qe vijon do të jetë përreth video servers dhe sistemeve operative të tyre.

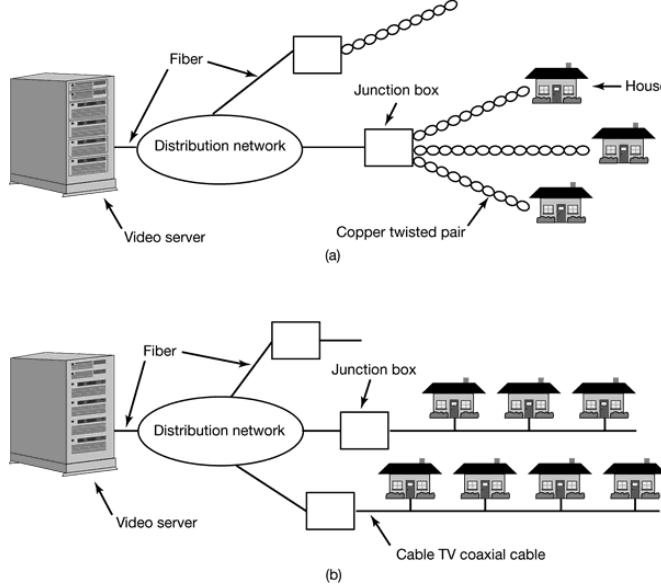


Figura 7-1. Video on demand duke përdorur teknologji shpërndarje të ndryshme lokale. (a) ADSL. (b) Cable TV.

Rrjeti i shpërndarjes (distribution network) midis user-it dhe video serverit duhet të jetë i aftë të transmetoje të dhena me shpejtësi të madhe dhe ne kohe reale. Disenjimi i ketyre rrjetave është intëresant dhe kompleks, por del jashtë qellimit të ketij libri. Ne nuk do të

themi asgje tjetër përreth tyre përvèc se të veme ne dukje qe keto network përdorim gjithmone fibrat optike nga video serveri deri të kutia e bashkimit (*junction box*) të zones ku jetojne klientët. Ne sistemet e ADSL-se, të cilat ofrohen nga kompanitë telefonike, linja ekzistuese e tëlefonit (twisted pair – cift i përdredhur) ofron kilometrin e fundit të transmetimit. Ne sitemet kabllore, të cilat ofrohen nga operatoret kabllor, tëlat ekzistues të TV kabllor përdoren për shpërndarjen lokale (local distribution). ADSL ka avantazhin sepse i jep çdo përdoruesi një kanal të dedikuar, keshtu qe garaton bandwidth, por bandwidth është i voglet disa (megabit/sec) për shkak të kufizimeve të télave tëlefonik. TV kabllor përdor kabllin co-axial me bandwidth të madh disa (gigabit/sec), por shume klientë përdorin se bashku (ndajne – share) të njëjtin kabell, keshtu bandwidthi nuk është i garantuar për secilin nga klientët.

Pjesa e fundit e sistemitës është kutia **set-top box**, ne të cilën përfundon ADSL-ja ose TV kabllor. Kjo pajisje është një kompjuter i zakonshem, me një chip special për de-codim dhe de-compresimin e videove. Kompjuteri përmban minimalisht CPU, RAM, ROM, dhe një nderfaqe për ADSL ose për TV kabllor.

Si alternative e kutise set-top box, mund të përdoret PC ekzistuese e klientit dhe të shfaqet filmi ne monitorin, set-top box merret parasysh nga fakti qe njerezit duan të shohin video on demand ne dhomen e tyre të ndenjës dhe nuk kane kompjuter aty. Nga një përspektive teknike, përdorimi i një kompjuteri ne vend të set-top box, merr me tepër kuptim, sepse kompjuteri është me i fuqishem, ka një disk të madh, dhe rezolucioni i ekranit është i madh. Gjithashtu, ne do të bejme dallimin midis video server dhe procesit të klientit ne skajin e përdoruesit qe dekodon dhe shfaq ne ekran filmin. Ne termat e disenjimit të sistemeve, nuk ka shume rendesi ne se procesi i klienti ekzekutohet ne kutine set-top box, apo ne PC. Për një sistem video-editimi desktop, të gjitha proceset ekzekutohen ne të njëjtën makine, por ne do të vazhdojme të përdorim terminologjine server dhe client, ne menyre qe të jetë e qartë se cfare bën një proces i caktuar.

Duke u kthyer tek multimedia, ajo ka dy karakteristika kyce qe duhen të kuptohen mire:

Multimedia përdor **Data Rates** – jashtëzakonisht të larta.

Multimedia ka nevoje përi ri-luajtje ne kohe reale (**real time playback**).

Niveli i lartë i Data Rates vjen nga natyra e informacionit akustik e vizual. Sytë dhe veshet mund të procesojne sasira shume të medhaja informacioni ne sekonde, dhe duhen të ushqehen ne atë Rate përi të prodhuar pamje të pranueshme. Ne Fig. 7-2 është paraqitur Data rate i disa burimeve multimediale dhe i disa pajisjeve hardwarike. Ne do të diskutojme përi disa nga keto formatë encodimi ne vone ne ketë kapitull. Ajo qe duhet të theksohet është, meqë data rate multimediave është i lartë, ai ka nevoje përi kompressim dhe përi një vend ku të ruhet. Përi shembull një film 2-ore HDTV i pa-kompresuar është një file prej 570 GB. Një video server ruan 1000 file të tille, pra ka nevoje përi 570 TB hapesire, qe do të thotë një sasi jo e parendesishme përi standartet e sotme. Ajo qe duhet të theksohet tjetër është, pa kompresimin e të dhenave, hardware e tanishme nuk mund të ecin (përballojne) me data rate qe prodhohen. Kompresimin e videove do ta shohim me vone ne ketë kapitull.

Kerkesa e dytë qe multimedia vendos ne një sistem, është nevoja e shpërndarjes të të dhenave ne kohe reale. Porcion i videos të një filmi digital konsiston ne disa numra frames përi sekonde. Sistemi NTSC qe përdoret ne Amerike dhe Japoni, ekzekutohet ne 30 frames/sec (29.97 ekzaktësisht), ndersa sistemet PAL dhe SECAM qe përdor pjesa

tjetër e botës, ekzekutohet ne 25 frame/sec (25frame/se ekzaktësish). Framet duhet të shpërndahen ne intervale precise 33msec ose 40msec, respektivisht, ose përndryshe filmi do duket me ndryshime të shpeshta (copetuar).

Source	Mbps	GB/hr	Device	Mbps
Tëlphone (PCM)	0.064	0.03	Fast Ethernet	100
MP3 music	0.14	0.06	EIDE disk	133
Audio CD	1.4	0.62	ATM OC-3 network	156
MPEG-2 movie (640 x 480)	4	1.76	SCSI UltraWide disk	320
Digital camcorder (720 x 480)	25	11	IEEE 1394 (FireWire)	400
Uncompressed TV (640 x 480)	221	97	Gigabit Ethernet	1000
Uncompressed HDTV (1280 x 720)	648	288	SCSI Ultra-160 disk	1280

Figura 7-2. Disa data rates për multimedia dhe për pajisje I/O me performance të lartë. Vini re qe 1 Mbps është 10^6 bits/sec por 1 GB është 2^{30} bytes.

Zyrtarisht NTSC do të thotë National Television Standards Committee, por kur hyri ne skene tëlevizori me ngjyra, doli një shaka për industrine, Never Twice the Same Color. PAL – Phase Alternating Line. Teknikisht është sistemi me i mire. SECAM – Séquentiel Couleur Avec Mémoire, përdoret ne France (dhe u shpik për të mbrojtur prodhuesit e Télevisioneve franceze nga kompeticioni me të huajt. SECAM përdoret gjithashtu ne euopen lindore, sepse kur u shfaq tëlevizioni ne ketë vende, qeveritë komunistë të atëhershme nuk donin qe njerezit të shikon tëlevizonin Gjerman keshtu qe zgjodhen një sistem jo-kompatibel.

Veshi është me i ndieshem sesa syri, keshtu qe një ndryshim (variance) edhe ne rendin e milisecondave ne kohen e shpëndarjes do të ishte e ndieshme. Nryshueshmeria (varianca) ne ratet e shpëndarjes: quhet **jittër** dhe duhet qe ti vendosen kufijtë ne menyre striktë për përfomaanca e mira. Vini re qe jittër nuk është e njëjtë gje si vonesa (delay). Në qoftë se rrjeti shpëndares Fig. 7-1 vonen ne menyre uniforme secilin bit me 5.000 sec, filmi do të filloje pak me vone, por do të duket i qartë. Ndersa në qoftë se vonohen ne menyre rastesore frames me 100-200msec, filmi do të duket si një film i vjetër i Charlie Chaplin.

Vetitë ne kohe reale qe kerkohen për ri-luajtur serisht (playback) multimediat, shpesh ato përshkruen nga parametrat e **cilesise të sherbimit** (QoS - quality of service). Parametrat përfshijne bandwidthin mesatar të disponueshem, pikun e bandwidth, vonesat minimale dhe maksimale (të cilat se bashku vendosin kufijtë e jittër), dhe një probalilitët i humbjes se biteve. Për shembull, një operator rrjeti mund të ofroje një sherbim duke garantuar një bandwidth 4 Mbps, 99% e transmetimit i ka vonesat ne intervalin 105 – 110 msec, dhe rate e humbjes se biteve 10^{-10} , qe do të ishte mese ok për filmat me MPEG-2. Operatori gjithashtu mundet të ofroje, një sherbim me nivel disi me të ulet, me të lire, me një bandwidth 1 Mbps (sh. ADSL), ne ketë rast për cilesine duhet të biem ne një kompromis, mund të ndryshojme resolucionin, duke ulur ndjeshem frame rate, ose të heqim dore nga gjyrat dhe ta shohim filmin ne bardh e zi.

Menyra me e zakonshme për të ofruar garancitë për cilesine sherbimit është të rezervojme një capacitët përpara ne kohe (in advance), për secilin nga klientët. Burimet e rezervuara përfshijne një porcion të CPU, bufferat e memories, capacitëtin e transferimit të diskut, bandwidth e rrjetit. Ne qoftëse një klient i ri kerkon të shikoje një film, por

video serveri ose rrjeti nuk ka kapacitet të mjaftueshem për një klient tjetër, ai duhet të refuzoje klientin e ri ne menyre qe të shmangi degradimin e sherbimit klientëve aktual. Si rrjedhim, serverat multimedial kane nevoje për skema rezervimi të burimeve, dhe për algoritmin e kontrolli të pranimeve (admission control algorithm) ne menyre qe të vendosin se kur ata mundet të menaxhojne me shume detyra.

7.2 SKEDARET MULTIMEDIA

Ne shumicen e sistemeve, një skedar tekst përbehet nga një sekunce byte-sh pa ndonjë strukture qe sistemi operativ njeh apo i intëreson të njohe. Me skedaret multimedia, situata është pak me e komplikuar. Si fillim, video dhe audio jane krejtësisht të ndryshme. Ato kapen nga paisje të ndryshme (CCD chip versus microphone), kane strukture të brendshme të ndryshme (video ka 25-30 frame/sek; audio ka 44,100 kampjone/sec), dhe ato përdoren nga paisje të ndryshme (monitor vs altoparlantë).

Me tej, shumica e filmave të hollywood-it i drejtohen një audience mbare botërore, shumica e të ciles nuk flet Anglisht. Kjo pike zgjidhet ne një ose dy menyra. Për disa shtëtë, një kolone zanore shtese prodhohet, me zera të dubluar ne gjuhen lokale (por jo efektët zanore). Ne japoni, të gjitha tëlevizionet kane dy kanale zeri për ti lejuar shikuesit të degjojne filmat e huaj ne gjuhen origjinale apo atë japoneze. Një buton ne pult përdoret për zgjedhjen e gjuhes. Ne disa vende të tjera përdoret kolona zanore origjinale me titra ne gjuhen lokale.

Disa filma tëleviziv ofrojnë closed-caption titra ne anglisht gjithashtu, për të lejuar folesit Anglisht por me degjim të kufizuar të shohin filmat. Si rezultat filmi dixhital mund të konsistoje nga disa skedare: një video, disa audio, disa titra tekst ne gjuhe të ndryshme. DVD –të kane aftësine të ruajne deri ne 32 gjuhe dhe titra text. Një bashkesi skedaresh multimedia tregohet ne Fig. 7-3. Ne do të shpjegojme kuptimin e “fast forward” dhe “fast backward” me vone ne ketë kapitull.

Si rrjedhoje, një “file system” ka nevoje të përcaktoje disa nen skedare për skedare. Një skenar i mundshem është të menaxhoje çdo nen skedar si një skedar tradicional (për shembull., duke përdorur një nyje *i* për të dalluar blloqet) dhe të mbaje një strukture të dhenash qe liston të gjitha nen skedaret për çdo skedar multimedia. Një metode tjeter është të përdorim një tip nyjesh dy-dimensionale, me çdo kolone duke listuar të gjitha blloqet për nen skedae. Ne përgjithesi, organizimi duhet të jetë i tilë qe shikuesi ne menyre dinamike të zgjedhe cilin skedar audio apo titre të përdore ne momentin qe shihet filmi.

Ne çdo rast, ndonjë menyre për të sinkronizuar nen skedared duhet ne menyre qe kur audio e zgjedhur të luhet të jetë e sinkronizuar me vidjon. Ne se audjon dhe vidjo dalin sado pak nga sinkronizimi, shikuesi degjon fjalet e aktorit para ose pas levizjes se buzeve, gje qe dallohet lehtë dhe është shume acaruese.

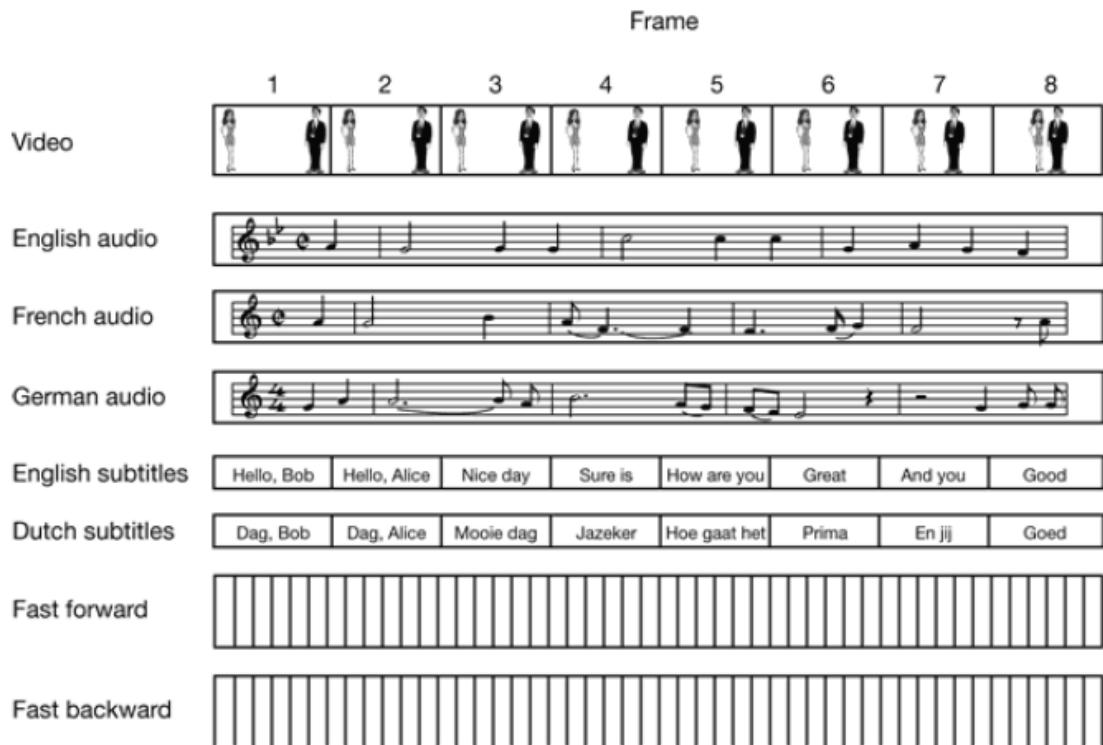


Figure 7-3. Një film mund të konsistoje nga disa skedar.

Për të kuptuar me mire si skedared multimedia organizohen, është e nevojshme të kuptojme si funksionon audjo dhe vidjo dixhitale ne detaje. Ne tani do japim një prezantim të ketyre ceshtjeve.

7.2.1 Enkodimi Audio

Një vale audio (zanore) është një vale një dimensionale akustike (presioni). Kur një vale akustike hyn ne vesh, daullja e veshit dridhet, duke shkaktuar qe pjeset e brendshme të veshit të dridhen bashke me të, dhe duke derguar impulse ne tru. Keto pulse përceptohen si ze nga degjesi. Ne menyre të ngjashme, kur një vale akustike godet mikrofonin, ai gjeneron sinjale elekrik, qe përfaqesojne amplituden e zerit si funksion i kohes.

Brezi i frekuencave të veshit të njëriut varion nga 20 Hz deri 20,000 Hz, megjithese disa kafshe, sidomos qentë, mund të degjojne edhe ne frekuencia me të larta. Veshi degjon ne menyre logaritmike, keshtu qe rapporti i dy zerave me amplituda A dhe B mund të shprehet konvencionalisht ne dB (decibels) sipas formules:

$$dB = 20 \log_{10}(A/B)$$

Ne se ne përcaktojme limitin e poshtëm të degjimit (një presion rreth 0.0003 dyne/cm^2) për një vale sinusoide 1kHz si 0 dB, një bashkebisedim i zakonshem zhvillohet ne 50dB dhe dhimbjet e veshit fillojnë ne 120 dB, një brez dinamik me një faktor 1 milion. Për të shmangur konfuzionin, A dhe B lart Jane *amplituda*. Ne se ne do të përdorim fuqitë, qe Jane propacionale me katrorin e amlitudes, koeficienti i algoritmit do ishte 10, dhe jo 20.

Valet audio mund të konvertohen ne dixhitale nga një ADC (Analog Digital Convertër). Një ADC merr një voltazh elektrik ne hyrje dhe gjeneron një numer binar ne dalje. Ne fig. 7-4(a) ne shohim një shembull të një sinusoide. Për të përfshuar sinjalin dixhital, ne mund ta kampionojme çdo T sekonda, sic tregohet nga gjatësia e vijave ne fig. 7-4(b). Ne se vala zanore nuk është puro sinusoidale, por një superpozim i vales sinusoidale ku komponentja me frekuencë me të lartë përfshesohet nga f , atëherë është e mjaftueshme të bejmë kampionime me frekuencë $2f$. Ky rezultat është provuar matematikisht nga H. Nyquist ne 1924. Kampionimi me i shpeshtë nuk ka vlerë meqene se frekuencat me të larta që do merreshin nuk jane prezentë.

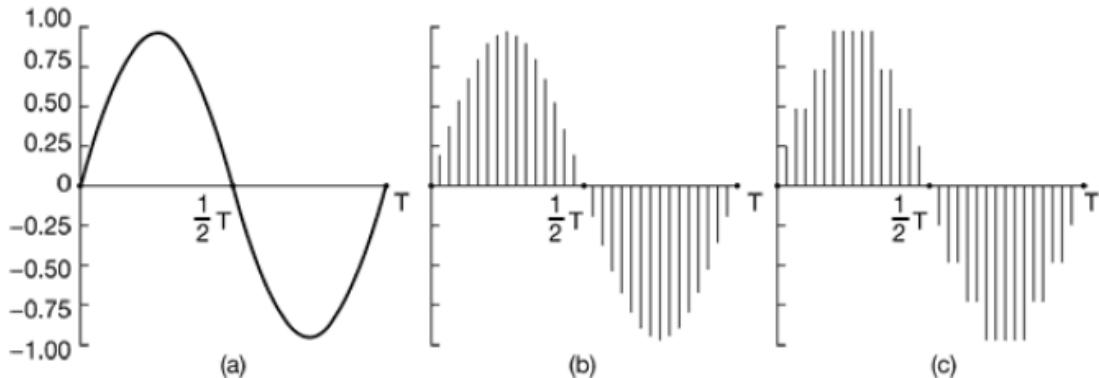


Figure 7-4.(a) Një vale sinusoidale. (b) Kampionimi i vales sinusoidale. (c) Kuantizimi me 4bit.

Kampionet dixhitale nuk jane asnjehere ekzaktë. Kampionet e Fig. 7-4(c) lejojnë vetëm nentë vlera, nga -1.00 deri të +1.00 ne shkalle prej 0.25. Rrjedhimisht, 4 bit nevojiten për të përfshuar të gjitha ato. Me 8-bit kampione lejohen 256 vlera të ndryshme. Me 16 bit kampione mund të marrim 65,536 vlera të ndryshme. Gabimi i futur nga numri i fundem i biteme për kampion quhet zhurma kuantizimi. Ne se është shume e lartë, veshi do e dalloje atë.

Dy shembuj të mirenjohur të zerit të kampionuar Jane tëlefoni dhe audio compact disjet. “Pulse code modulation” përdoret brenda sistemit të telefonik dhe përdor 7-bit (Amerika Veriore dhe Japonia) ose 8 bit (Europa) kampione 8000 here ne sekonde. Ky sistem jep një data rate prej 56.000 bps ose 64.000 bps. Me vetëm 8000 kampione/sek, frekuencat mbi 4kHz humbasin.

CD-të Audio Jane dixhitale me një kampionim 44.100 kampione/sek, e mjaftueshme për të kapur frekuencë deri ne 22.050 Hz, që është e mire për njerezit por e keq për qentë. Kampionet Jane 16 bit secila, dhe Jane lineare mbi brezin e amplitudave. Vini re se 16-bit kampione lejojnë vetëm 65.536 vlera të ndryshme, megjithese brezi dinamik i veshit është rrëth 1 million kur matët me hapa degjimi me të vogjel të mundshem. Pra duke përdorur vetëm 16 bit për kampion shkakton zhurma kuantizimi (megjithese i gjithe brezi dinamik nuk mbulohet). Me 44.100 kampione/sec nga 16 bit, një audio CD ka nevoje për “bandwidth” prej 705.6 Kbps për “mono” dhe 1.411 Mbps për stëreo (shih fig 7-2). Kompresimi i audios është i mundur duke u bazuar ne modelet psikoakustik sipas të cilave punon veshi i njërit. Një kompresim me 10x është i mundur duke përdorur

sistemin MP3 (MPEG layer 3). Player-at e muzikes portabel për ketë format kane qene shume të zakonshem vitet e fundit.

Zeri dixhital mund të proçesohet lehtësisht nga kompjuteri me anen e software-it. Shume programe ekzistojne për PC qe lejojne përdoruesit të regjistrojne, shfaqin, editojne, miksojne dhe të regjistrojne valet zanore nga disa burime. Pothuajse gjithe editimi profesional i zerit është dixhital ditët e sotme.

7.2.2 Enkodimi Video

Syri i njëriut ka vetine qe kur një imazh i shfaqet mbi retine, fiksohet për disa milisekonda përpala se të ndryshohet. Ne se një sekunde imazhes shfaqen çdo 50 ose me shume imazhe për sekonde, syri nuk veren qe po shikon imazhe diskret. Të gjithe filmat video dhe filmat e animuar shfrytëzojnë ketë koncept për të prodhuar pamje të levizshme.

Për të kuptuar sistemet video, është me mire të fillojme me një téllevizor të thjeshtë të stilit të vjetër bardh e zi. Për të formuar imazhin dy-dimensional ne ekran, kamera skanon me shpejtësi një rreze elektronike një-dimensionale si voltazh ne funksion të kohes, duke regjistruar intënsitetin e dritës gjatë rruges. Ne fund të skanimi të quajtur “frame”, rreza rivendoset. Intënsiteti si funksion i kohes transmetohet, dhe receptoret përsërisin proçesin e skanimi për të riformuar figuren. Proçesi i skanimi i përdorur nga kamera dhe marresi tregohen ne Fig. 7-5. (Megjithatë disa kamera CCD intëgrogjne ne vend qe të skanojnë, disa të tjera si dhe monitoret CRT skanojnë.)

Parametrat e skanimi variojnë nga shtëti ne shtët. NTSC ka 525 vija skanimi, një raport horizontal me vertikal 4:3, dhe 30 frame/sek. Sistemi European PAL dhe SECAM kane 625 vija dhe raport 4:3 me 25 frame/sek. Ne të dy sistemet vijat me të sipërme dhe me të poshtme nuk shfaqen (për të afersuar një imazh sa me kator ne një CRT të rrumbullaket). Vetëm 483 nga 525 NTSC linja skanimi (dhe 576 nga 625 PAL/SECAM) shfaqen.

Ndersa 25 frame/sek Jane të mjaftueshme për të kapur levizje të rrjedhshme, ne atë frame rate shume njerez sidomos të vjetrit, do përceptojne imazhin të nderpritet (sepse imazhet e vjetra Jane shuar nga retina para se të rinjtë të shfaqen). Ne vend qe rritet frame rate, qe do të kerkonte përdorimin e me shume bandwidth-i, merrent një rruge tjetër. Ne vend qe të shfaqen vijat e skanimi nga lart posht, ne fillim të gjithe vijat shfaqen tek, pastaj vijat cift. Çdo njëra prej ketyre gjysem frameve quhet një fushe. Ekspérimentët kane treguar qe megjithese njerezit dallojne nderprerje ne 25 frame/sek, ata nuk e dallojne ne se ka të 50 frame/sec. Kjo teknike quhet intërlacing. Téllevizioni nonintërlacing ose video thuhet se është progresive.

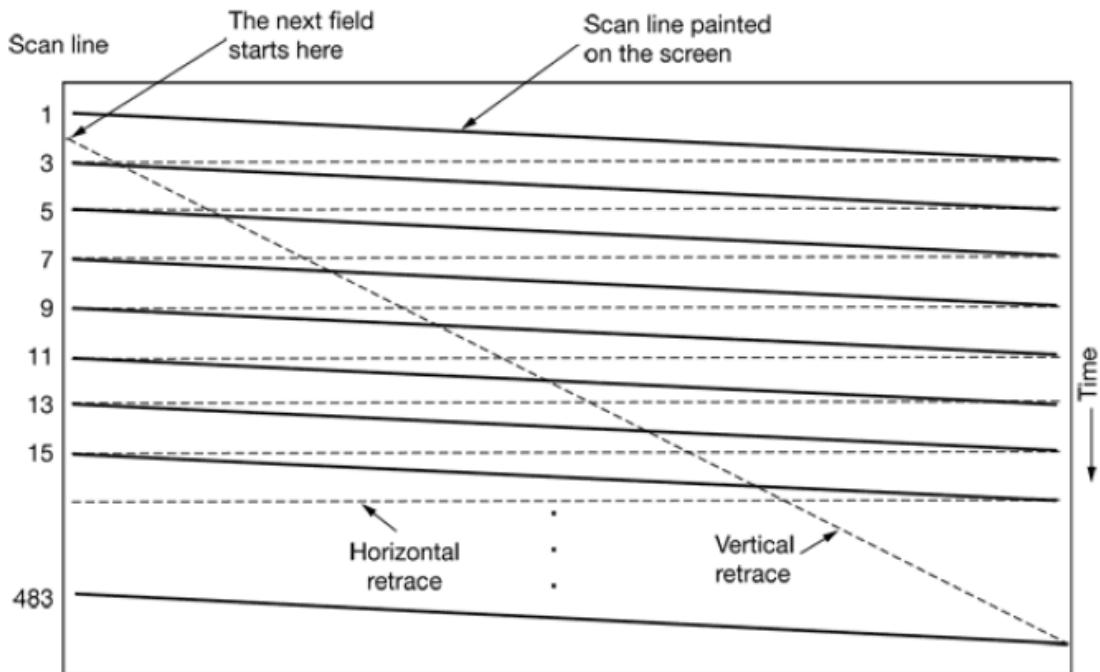


Figura 7-5. Skanimi i përdorur për NTSC video dhe TV.

Video me ngjyra përdor të njëjtin koncept si ajo monokrome, përvet se përdor tre rreze. Secila përdoret për njëren nga njyrat primare: e kuqe, jeshile dhe blu (RGB). Kjo teknike funksionon sepse çdo ngyre mund të formohet nga superpozimi i ngjyres se kuqe, jeshile dhe blue me intënsitet përkatëse. Megjithatë, për të transmetuar një kanal të vetëm, të tre ngjyrat mund të kombinohen ne një sinjal të vetëm.

Për të lejuar transmetimet me ngjyra për tu shfaqur ne marres bardh e zi, të tre sistemet RGB kombinohen linearisht ne një si sinjal i vetëm luminishent (bardhesie), dhe dy krominance (ngjyre), megjithese të gjithe përdorin koeficientë të ndryshem për të ndertuar keto sinjale nga sinjalet RGB. Intëresant është fakti qe syri është me i ndjeshem ndaj sinjalit luminance se atij krominance, keshtu qe i fundit mund të mos transmetohet me saktësi. Gjithashtu sinjali luminance mund të transmetohet ne të njëjen frekuence si ai i vjetri bardh e zi keshtu qe mund të kapet nga tëlevizoret bardh e zi. Të dy sinjalet krominance mund të transmetohen ne breza të ngushtë ne frekuencë të larta. Disa tëlevizore kane butona për kontrollin e “brightness, hue dhe saturation. Kuptimi i luminances dhe kromances është i nevojshem për të kuptuar si funksionon kompresimi i videos.

Deri tani kemi pare video analoge. Tani le të kthehem i të video dixhitale. Përfaqesimi me i thjeshtë i videos dixhitale është një sekuence framesh, secili i përbere nga një rrjetë katroresh prej elementë figurash, ose pixel. Për video me ngjyra, 8 bit për pixel përdoren për çdo njyre RGB, duke dhene 16 million ngjyra, qe është mëse mjaftueshem. Syri i njëriut as nuk mund ti dalloje kaq shume ngjyra.

Për të prodhuar një levizje të rrjedhshme, vidjo dixhitale, ashtu si ajo analoge, duhet të shfaqe 25 frame/sek. Megjithatë, gjersa cilesia e mire e monitorit të kompjuterit mund ta

riskanoje ekranin për imazhet ne video RAM deri 75 here ne sekonde apo me shume. intéracing nuk është e nevojshme. Pra, të gjithe monitoret e kompjuterave përdorin skanimin progresiv. Thjesht duke rivizuar të njëjtën frame tre here rresht mjafton për të menjanuar nderprerjet.

Me fjale të tjera, rrjedhshmeria e levizjes përcaktohet nga një numer i ndryshem imazhes për sekonde, ndersa nderprerja përcaktohet nga numri i vizatimeve të ekranit ne sekonde. Keto dy parametra jane të ndryshem. Një imazh i palevizhem i vizatuar me 20 frame/sek nuk do të prodhohe një levizje aritmike por do të nderpritet sepse një frame do të shuhet nga retina para se të shfaqet tjetra. Një film me 20 frame të ndryshme për sekonde, secila e vizatuar 4 here rresht me 80Hz, nuk do të nderpritet, por levizja do të shfaqet aritmike.

Rendesia e ketyre parametrave behet e qartë kur marrim ne konsiderate bandwidth-in e nevojshem për të transmetuar vidjo dixhitale ne një rrjet. Monitoret e sotëm përdorin të gjithe reportin 4:3 keshtu qe mund të përdorin tuba katodik të prodhuar ne seri të dizenuar për télvezoret. Konfigurimet me të zakonshme janë 640x480 (VGA), 800x600 (SVGA), dhe 1024x768 (XGA). Një XGA shfaq 24 bit për pixel dhe 25 frame/sek, i nevojitet një ushqim me 472 Mbps. Dyfishimi i ratet për të menjanuar nderprerjet nuk është shume tèrheqes. Një zgjidhje me e mire është transmetimi me 25 frame/sek dhe ruajtja ne kompjutér e çdo frame dhe shfaqja dy here. Transmetimet télvezive nuk përdorin ketë teknike sepse télvezoret nuk kane memorie, dhe do të kerkohet konvertimi njehere ne dixhital qe kerkon hardware shtese. Si rrjedhoje, intéracing përdoret për transmetimet télvezive dhe jo për vidjo dixhitale.

7.3 Kompresimi Video

Duhet të jetë e qartë qe manipulimi i matérjalit multimedia ne forme të pa konpresuar është jasht çdo diskutimi – është shume e madhe. E vëtmja shprese është kompresimi masiv, i cili është i mundur. Fatmiresisht, një trupe e madhe kerkimi gjatë ketyre decadave të fundit ka cuar ne shume teknika dhe algoritma kompresimi të cilat e bejne transmetimin multimedia të pranueshem. Ne ketë pjese ne do të studiojme disa metoda për kompresimin e të dhenave multimediale, sidomos imazheve.

Të gjitha sistemet e kompresimit kerkojne dy algoritma: një për kompresimin e burimit të të dhenave, dhe një për dekompresimin të tij ne destinacion. Ne ketë litérature, ketyre algoritmave i referohemi si algoritma enkodimi (encoding) dhe dekodimi (decoding). Ketë terminologji do ta përdorim ketu, gjithashtu.

Keto algoritma kane disa asimetri qe jane të rendeshishme për tu kuptuar. Ne fillim, për shume aplikacione, një dokument multimedia, le të themi, një film do të enkodohet vetëm njehere (kur ruhet ne server-in multimedia) por do të decodohet mijera here (kur të shihet nga klientët). Kjo asimetri do të thotë qe është e pranueshme qe algoritmi i encodimit të jetë i ngadaltë dhe të kerkoste hardware shtese duke lejuar qe algoritmi i

dekodimit të jetë i shpejtë dhe të mos kerkonte hardware të shtrenjtë. Nga ana tjetër, ne multimedia realtime, si përshebull video konferanca, një enkodimi i ngadaltë është i pa pranueshem. Encodimi duhet të behet ne “ajer” ne sistemet real time.

Një asimetri tjetër është qe procesi enkodim/dekodim të mos jetë i kthyshem. Qe do të thotë, kur kompresojme një file, transmetojme atë dhe dekompresojme, përdoruesi pret të marre origjinalin mbrapër shembull, të saktë deri ne bitin e fundit. Me multimedia, kjo kerkese nuk qendron. Ne përgjithesi është e pranueshme të kemi sinjal video pas enkodimit dhe dekodimit paksa ndryshe nga origjinali. Kur dalja e dekodimit nuk është ekzakt si inputi origjinal themi se sistemi është me humbje (lossy). Të gjithe sistemet e kompresimit të përdorur për multimedia janë “loosy” sepse jasin me shume kompresim.

7.3.1 Standarti JPEG

Standarti JPEG (Joint Photographic Experts Group) për kompresimin e figurave fikse me ton të vazhdueshem (për shembull., fotografitë) u zhvillua nga ekspertët e fotografise me bashkunimin e ITU, ISO dhe IEC. Është i rendesishem për multimedia sepse, me një përafrim, standarti për figurat e levizhme MPEG është thjeshtë JPEG për çdo frame vecantë, plus disa veti shtese për kompresimin ndër frameve dhe kompresimin e levizjes. JPEG përcaktohet ne Standartin Nderkombetar 10918. Ka katër gjendje dhe shume opsione, por ne do merremi vetëm me menyren e përdorimit për video 24-bit RGB dhe do të leme jashtë shume nga detajet.

Hapi i pare i enkodimit të imazhi me JPEG është ndarja e blloqeve. Për arsyen thjeshtësie, le të supozojme qe JPEG merr ne hyrje një imazh 640x480 RGB me 24bit/pixel, sic tregohet ne Fig. 7-6(a). Meqë përdorimi i luminances dhe kromoinances jep rezultat me të mire kompresimi, luminanca dhe dy sinjalat e kromances llogaritën nga vlerat RGB. Për NTSC ata quhen Y, I dhe Q, respektivisht. Për PAL ata quhen Y, U dhe V dhe formulat janë të ndryshme. Me poshtë ne do të përdorim emrat NTSC, por algoritmi i kompresimit është i njëjtë.

Matrica të ndryshme ndertohen për Y, I dhe Q, secila me elementet qe varjojne nga 0 ne 255. Me pas, blloqe katrore prej katër pikselash mesatarizohen ne matricat I dhe Q për ti ulur ne 320x240. Ky reduktim është me humbje, por syri zor se dallon atë, meqë ai reagon me shume ndaj luminances se krominances.

Megjithese, i kompreson të dhenat me një faktor dy. Tani 128 i zbritet nga çdo element i tre matricave për të vendosur 0 ne mes të brezit. Ne fund, çdo matrice ndahet ne blloqe 8x8. Matrica Y ka 4800 blloqe; dy të tjera kane 1200 blloqe secila, sic tregohet ne Fig. 7-6(b).

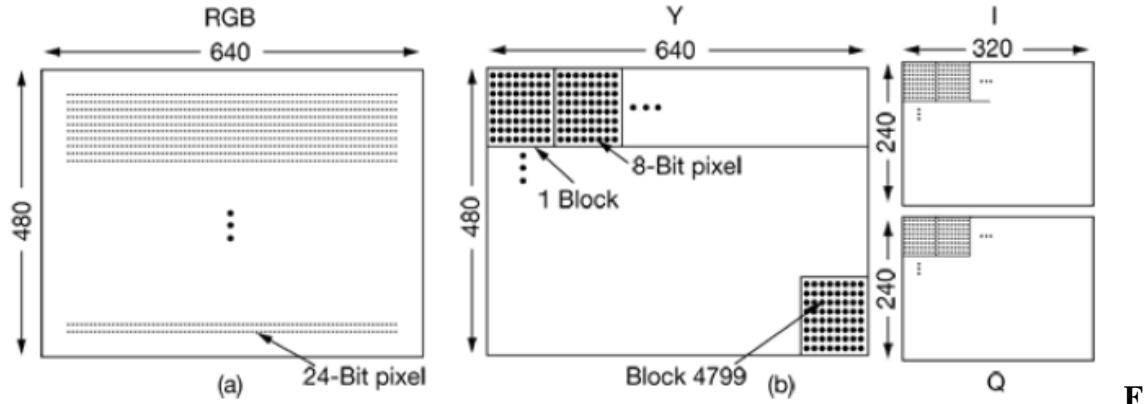


Figura 7-6. (a) RGB input data. (b) After block preparation.

Hapi i dytë i JPEG është ti aplikojme një DCT (Discrete Cosine Transformation) seciles nga 7200 bloqet veconte. Dalja e seciles DCT është një matrice 8×8 me koeficientët DCT. Elementet DCT (0,0) është një mesatare e bllokut. Elementet e tjere tregojne sa shume fuqi spektrale është prezantë ne çdo frekuencë. Ne tëori, një DCT është pa humbje (lossless), por ne praktike përdorimi i numrave me presje të levizhme dhe funksioneve transhendencial gjithmone shfaq disa gabime rrumbullakimesh qe cojne ne humbje informacioni. Normalisht, keto elementë shuhën shpejt me distancen nga origjina, (0,0), sic sugjerohet nga Fig. 7-7(b).

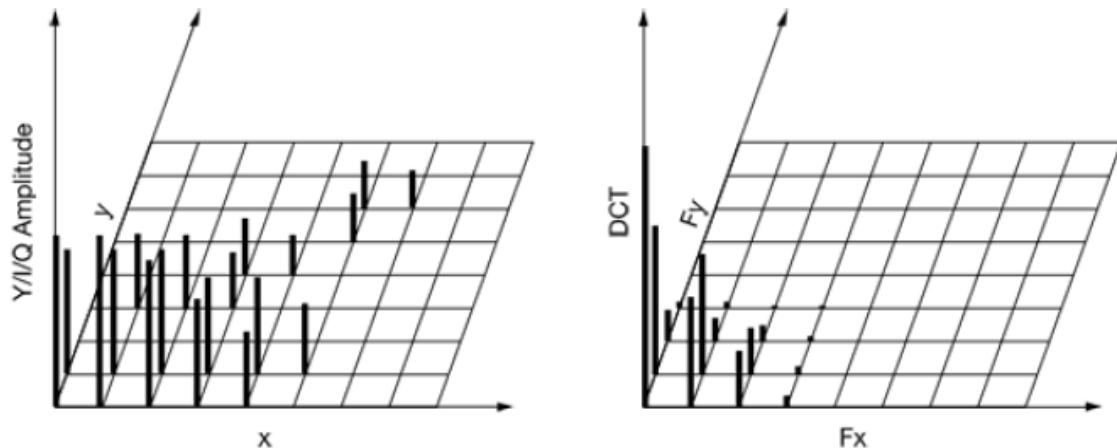


Figura 7-7. (a) Një bllok i matrices Y, (b) Koeficientët DCT.

Kur DCT kompletohet, JPEG leviz ne hapin 3, i cili quhet kuantizim, ne të cilin koeficientët me pak të rendesishem DCT shuhën. Ky transformim (me humbje) behet duke pjesetuar secilen nga koeficientët ne matricen 8×8 DCT nga elementi përkatës ne tabelen e kuantizimit. Vlerat e vleres se kuantizimit nuk jane pjese e standartit JPEG. Çdo aplikacion duhet të ofroje tabelen e vet të kuantizimit, duke i dhene atij mundesine për të kontrolluar raportin humbje-kompresim.

DCT Coefficients								Quantized coefficients								Quantization table							
150	80	40	14	4	2	1	0	150	80	20	4	1	0	0	0	1	1	2	4	8	16	32	64
92	75	36	10	6	1	0	0	92	75	18	3	1	0	0	0	1	1	2	4	8	16	32	64
52	38	26	8	7	4	0	0	26	19	13	2	1	0	0	0	2	2	2	4	8	16	32	64
12	8	6	4	2	1	0	0	3	2	2	1	0	0	0	0	4	4	4	4	8	16	32	64
4	3	2	0	0	0	0	0	1	0	0	0	0	0	0	0	8	8	8	8	8	16	32	64
2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	16	16	16	16	16	16	32	64
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	32	32	32	32	32	32	32	64
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	64	64	64	64	64	64	64	64

F

Figura 7-8. Llogaritja e koeficientëve të kuantizuar DCT.

Hapi 4 është reduktimi vleres (0,0) ne çdo bllok (i atij ne cepin lart majtas) duke zevendesuar atë me vleren qe ndryshon nga bllokuna paraardhes. Meqe keto elementë janë mesataret e bllokut respektiv, ata duhet të ndryshojne gradualisht, keshtu qe duke marrë vlera diferenciale duhet të reduktoje shumicen e vlerave të vogla. Asnjë diferencial nuk llogaritet nga vlerat e tjera. Vlerat (0,0) i referohemi si komponentët DC; dhe të tjerave si komponentë AC.

Hapi 5 linearizon 64 elementet dhe aplikon enkodim të listës. Duke skanuar bllokun nga e majta ne të djathtë dhe nga lart poshtë nuk do të përqendroje zerot bashke, keshtu qe skanimi zig-zag përdoret, sic tregohet ne Fig. 7-9. Ne ketë shembull, kalimi zig-zag prodhon 38 zero rresht ne fund të matrices. Ky string mund të reduktohet ne një count të vetëm duke thene se jane 38 zero.

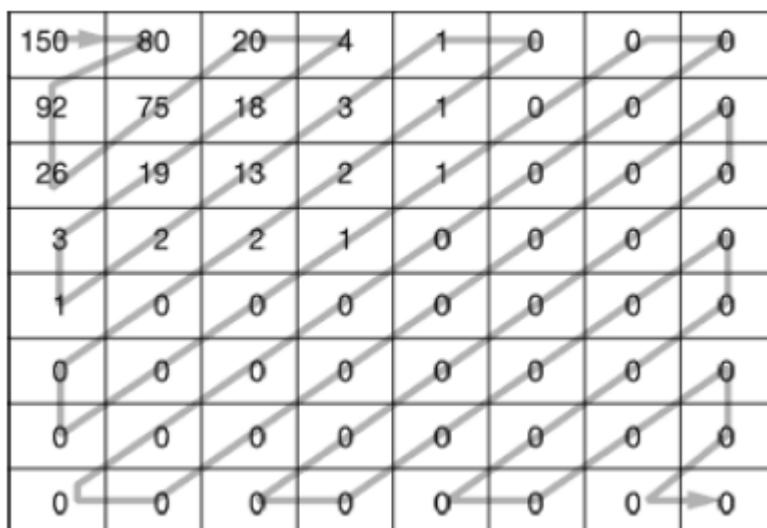


Figura 7-9. Rradha ne të cilën vlerat e kuantizuara transmetohen.

Tani kemi një listë numrash qe përfaqesojne imazhin (ne hapesiren e transformuar). Hapi 6 Huffman enkodon numrat e ruajtjes ose transmetimit.

JPEG mund të duket e komplikuar, por kjo ndodh sepse është vertet i komplikuar. Megjithatë, derisa shpesh prodhon një kompresim 20:1 ose me mire, është shume i përhapur. Dekodimi i JPEG kerkon ekzekutimin e algoritmit mbrapër shembull. JPEG është thuajse simetrik: kerkon po aq për dekodimin sa për enkodimin.

7.3.2 Standarti MPEG

Me ne fund arritem ne zemren e ceshtjes: standarti MPEG (Motion Picture Experts Group). Keto jane algoritmat kryesore për kompresimin e videos dhe kane standarte nderkombetare qe ne 1993. MPEG-1 (Standarti Nderkombetar 11172) u dizenjua për cilesi video regjistruesi (352 x 240 për NTSC) duke përdorur bit rate prej 1.2 Mbps. MPEG-2 (Standarti Nderkombetar 13818) u dizenjua për kompresimin e vidjos me cilesi broadcast me 4 ose 6 Mbps, keshtu qe mund të përfshihej ne kanalet e transmetimit NTSC dhe PAL.

Të dy versionet përfitojne nga dy lloj redundancies panevojshme qe ekzistojne ne filma: hapesinore dhe kohore. redundancy hapesinore mund të përdoret thjesht duke koduar çdo frame vec e vec me JPEG. Kompresim shtese mund të arrihet duke marren ne konsiderate faktin qe frame të njëpasnjëshme jane pothuaj identike (redundancy kohore). Sistemi DV (Digital Video) i përdorur nga vidjo kamerat dixhitale përdor vetëm një skeme të ngjashme me JPEG sepse enkodimi duhet të behet ne kohe reale dhe është shume me i shpejtë se enkodimi i seciles frame vec e vec. Rrjedhoja e ketij vendimi mund të shihet ne Fig. 7-2: megjithese vidjo kamerat dixhitale kane një data rate me të vogel se video e pakompresuar, ata nuk jane aq të mira sa MPEG-2. (Për të bere krahasimin të sinqertë, video kamerat DV kampjonojne luminancen me 8bit dhe çdo chromance me 2 bit, por qendron përseri një faktor pese për kompresimin duke përdor encodim JPEG të ngjashem.)

Për skenat ku kamera dhe sfondi jane të palevizshem dhe një ose dy aktore po levizin rrötull me ngadale, afersisht të gjitha pixelat do të jene identike nga frame ne frame. Ketu, thjesht duke zbritur çdo frame nga e para dhe duke ekzekutuar JPEG ne çdo diferenca do ishte mese e mjartuehsme. Megjithatë ne ato skena ku kamera po “panning” ose zmadhon, kjo teknike deshton keq. Ajo cfare nevojitet është ndonjë menyre për të kompesuar levizjen. Kjo është ajo cka bën MPEG saktësisht; ne fakt, kjo është diferenca kryesore ndermjet MPEG dhe JPEG.

Dalja e MPEG-2 konsiston nga tre tipe të ndryshme framesh qe duhen procesuar nga programi shfaqesh:

I (Intracoded) frames: S

P(Predictive) frames:

B(Bidirectional) frames:

Framet I Jane po figura të koduara me JPEG, duke përdorur luminance me rezolucion të plotë dhe kromance me gjysem rezolucioni gjatë secilit bosht. Është e nevojshme të kemi frame I të shfaqen përiodikisht për tre arsyen. E para, MPEG mund të përdoret për transmetimet télivizive, ku shikuesit qe zgjedhin me deshire. Ne se të gjitha framet varen

nga paraadheset duke shkuar deri të e para, ne se dikush ka humbur framin e pare nuk mund ta dekodoje asnjehere frame pasardhese. Kjo do të bënte të pamundur qe shikuesit të lidhen edhe ne se filmi ka filluar. E dyta, ne se ndonjë nga framet do të merrej gabim, dekodimi i metejshem do të ishte i pamundur. E treta, pa framet I, kur bejme nej fast forward ose rewind, dekoderi duhet të llogarise çdo frame qe ka kaluar qe të ishte ne gjedje të dintë vleren e frames du do të ndale. Me framet I, është e mundur qe kalimi fast forward ose backward deri të një frame I të gjendet dhe të filloje shikimi aty. Për keto arsyet framet I janë të futura ne output çdo një dy here ne sekonde.

Framet P, përkundrazi kodojne diferenca të frameve. Ato bazohen ne idene e macrobloqeve, qe mbulojnë 16×16 pixela ne hapesire luminance dhe 8×8 ne hapesire chromance. Një bllok macro enkodohet duke kerkuar frame paraardhës ose dicka pak të ndryshme nga ajo.

Një shembull ku framet P mund të jene të përdorshme jepet ne Fig. 7-10. Ketu shohim tre frame të një pas njëhem qe kane të njëtin sfond, por ndryshojne ne pozicionin e personit. Blloqet macro qe përbajne sfondin do të përputhen ekzaktësisht, por ato qe përbajne personin do të zhvendosen me një vlerë dhe duhet të gjurmohen.



Figura 7-10. Tre frame video të njëpasnjëshme.

Standarti MPEG nuk specifikon si të kerkojmë, sa larg të kerkojmë, ose sa i mire duhet të jetë përputhja qe të ketë rendesi. Kjo varet nga çdo implementim. Për shembull, një implementim mund të kerkoje një bllok macro ne pozicionin aktual të frameve paraardhëse, dhe të gjithe pozicionet si zhvendosje $+x$ ne drejtim x dhe $+y$ ne drejtim y . Për çdo pozicion, numri i përputhjeve ne matricen e luminances mund të llogaritet. Pozicionet me shume pike mund të deklarohet fitues, duke supozuar se ishte mbi një threshold të parapërcaktuar. Përndryshe, blloku macro thuhet se mungon. Algoritme shume me të komplikuara ekzistojnë, sigurisht.

Ne se një bllok makro gjendet, ai enkodohet duke marre diferenca e vlerave të tij me frame paraardhëse (për luminancen dhe cromancen). Keto matrica të diferenca me pas janë objekti i enkodimit JPEG. Vlera e makro bllokut ne rrjedhen daljes është vektori i levizjes (sa larg blloku macro u zhvendos nga pozicioni meparshem ne secilin drejtim), duke u ndjekur nga diferenca e enkodimit JPEG me atë ne frame paraardhëse. Ne se blloku makro nuk ndodhet ne frame paraardhëse, vlera aktuale enkodohet me JPEG, si ne framet I.

Framet B Jane të ngjashme me P, përvèc qe le ato lejojne referenca ne framen paraardhese dhe pasardhese, edhe ne frame I apo P. Kjo liri shtese lejon përmirsimin e kompresimit të levizjes, dhe është gjithashtu e përdorshme kur objektët kalojne përpara ose pas objektëve të tjera. Për shembull, ne një loje baseball kur “basemen” i tretë gjuan topin ne bazen e pare, mund të jetë ndonjë frame ku topi mbulan koken e basemen t edytë qe po leviz, ne sfond. Ne framen pasardhese koka ku mund pjeserisht e dukshme ne të majtë të topit, me përafersine qe koka rrjedh nga frame qe vjen ku topi ka kaluar koken. Framet B lejojne të bazohen ne një frame të pasuese.

Për të enkoduar një frame B, enkoduesi ka nevoje të mbaje tre frame të dekoduara ne memorie njëkohesisht; të kaluaren, prezentën dhe të ardhmen. Për të thjeshtuar dekodimin, framet duhet të jene prezentë ne rrjedhen MPEG ne rradhe varesie, se ne radhe shfaqje. Pra ehde me kohezim përfekt, kur një video shfaqet ne një rrjet, buferimi kerkohet ne makinen e userit për të rirenditur framet për shfaqje korrekte. Ne saj të kesaj diferenca ndermjet rradhes se varesise, të përpinqesh të luash një film mbrapër shembull nuk do të punoj pa marre ne konsiderate bufferimin dhe algoritma kompleks.

SKEDULIMI I PROÇESEVE MULTIMEDIA

Sistemet operative që mbështësin multimediat ndryshojnë nga ato tradicionalet në tre aspekte: skedulimi proçeseve, sistemi i skedarëve (file) dhe skedulimi i diskut. Ne do të fillojmë këtu me skedulimin e proçeseve dhe do vazhdojmë me të tjerat në vijim.

Skedulimi i Proçeseve Homogjene

Lloj më i thjeshtë i serverëve video është ai që mund të suportojë paraqitjen e një numri të caktuar filmash, të gjithë duke përdorur të njëjtin shkallë frame, rezolucion video, data rate dhe parametra të tjera. Poshtë këtyre kushteve, por një algoritem skedulimi efektiv është si një shoqëruar. Për çdo film, ka një proçes të veçantë (ose thread) puna e të cilit është të lexojë filmin nga disku frame pas frame dhe më pas ta transmetojë tek përdoruesi. Meqë të gjithë proçeset janë njësoj të rëndësishëm, duhet të bëhet e njëjta sasi pune për çdo frame, dhe ti bllokosh kur ka ato kanë përfunduar përpunimin e frame aktual, skedulimi round-robin e bën këtë punë shumë mirë. E vetmja gjë që duhet të shtojmë është standardizimi i algoritmeve skeduluese që është një mekanizëm kohe për t'u siguruar se çdo proçes ekzekutohet në frekuencën korrekte.

Një mënyrë për të realizuar kohën e duhur është duke patur një mastër clock që shënjon, le të themi, 30 herë në sekondë (për NTSC). Në çdo shënjim, të gjitha proçeset ekzekutohen në mënyrë sekuenciale, në të njëjtin rend. Kur një proçes ka mbaruar punën e tij, të gjitha proçeset ekzekutohen përsëri në njëtin rend. Për sa kohë që numri i

procëseve është mjaftueshëm i vogël sa të gjithë punët mund të bëhen një kohë frame, skedulimi round-robin është efiçent.

Skedulimet e Përgjithshme në Kohë-Reale

Fatkeqësisht, ky model zbatohet pak në realitet. Numri i përdorueseve që ndryshojnë si shikues vijnë e shkojnë, madhësia e frame-it varion në natyrën e kompresimit të videos (frame- I janë më të mëdha se frame -P ose -B) dhe filmat e ndryshëm mund të kenë rezolucion të ndryshëm. Si rrjedhojë, procëse të ndryshme duhet të ekzekutohen në frekuencë të ndryshme, me një sasi punë të ndryshme, dhe me deadline (afati i përfundimit) të ndryshme në varësi të punës që duhet të përfundojë.

Këto konsiderata çojnë në një model të ndryshëm: procëse të shumta konkurrojnë për CPU, secili me punën dhe deadline-in e tij. Në modelet pasuese, ne do të supozojmë se sistemi e njeh frekuencën që duhet të ekzekutohet secili procës, dhe cili është deadline pasues. Skedulimi i procëseve të shumtë konkurrues, disa ose të gjithë që kanë një deadline (afati i përfundimit) që duhet të plotësohet është quajtur **skedulim në kohë-reale**.

Si një shembull të ambientit punës së skedulerit multimedia në kohë-reale, konsiderojmë tre procëse *A*, *B* dhe *C* siç tregohet në Fig. 7-11. Procësi *A* ekzekutohet çdo 30msec (afërsisht shpejtësia e NTSC). Çdo frame kërkon 10msec të kohës CPU. Në mungesë të konkurrencës, duhet të ekzekutohet në hovet (bursts) *A*₁, *A*₂, *A*₃, etj., çdonjëri fillon 30msec pas të parit. Çdo hov (burst) i CPU kap një frame dhe ka një deadline; duhet të mbarojë përpara se tjetri të fillojë.

Gjithashtu në Fig. 7-11 tregohen dhe dy procëse *B* dhe *C*. Procësi *B* ekzekutohet 25 herë/sec (per shembull,, PAL) dhe procësi *C* ekzekutohet 20 herë/sec. Koha e llogaritur për frame është treguar si 15 msec dhe 5 msec respektivisht për *B* dhe *C*, vetëm për ta bërë problemin e skedulimit më të përgjithshëm se sa të pasurit të njëjtën të gjithë.

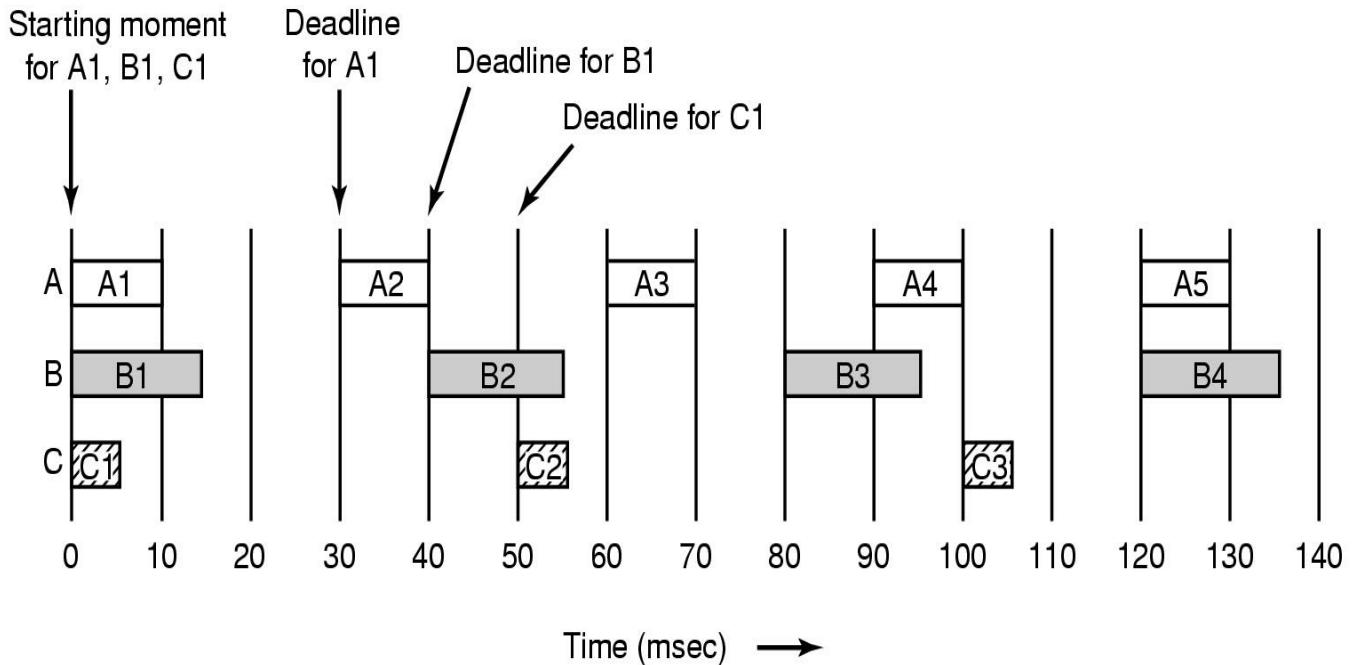


Figure 7-11. Tre proçese periodikë, secili paraqet një film . Frame rate dhe përpunimi i kërkuar për çdo proçes është i ndryshëm për secilin film.

Problemi i skedulimit tani është se si të skedulohen A , B , dhe C për t'u siguruar se ato do të takohen me deadline-ët e tyre respektive. Përpala se të shohim një algoritem skedulimi, ne duhet të shohim në se ky set instrukcionesh janë të skedulueshëm gjithsesi. Referuar seksionit 2.5.4, që në se proçesi i ka përiodën P_i msec dhe kërkesa C_i msec kohë CPU për frame, sistemi është i skedulueshëm vetëm dhe vetëm në qoftë se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Ku m është numri i proceseve, në këtë rast, 3. Vini re që P_i/C_i është vetëm një pjesë e CPU që përdoret nga proçesi i . Per shembull,, në Fig. 7-11, A është duke ngrënë 5/50 të CPU. Së bashku këto pjesë shkojnë në 0.808 të CPU, kështu që sistemi i proceseve është i skedulueshëm

Deri tani ne supozuam se ka një proçes për stream. Në të vërtëtë, duhet të jenë dy (ose më shumë procese) për stream. Per shembull,, një për audio dhe një për video. Ato duhet të ekzekutohet në shkallë (rate) të ndryshme dhe duhet të konsumojnë sasi të ndryshme kohe CPU.

Në disa sisteme kohë-reale, proceset janë të zëvendësueshme dhe në disa të tjera nuk janë. Në sistemet multimedia, proceset janë përgjithësisht të zëvendësueshëm, që do të thotë se një proçes që është në rrezik të humbjes së deadline-it mund në ndërpresë

proçesin në ekzekutim përpara se ai të mbarojë me frame-in e tij. Kur kjo mbaron, proces i mëparshëm vazdon. Kjo sjellje është vetëm multiprogramim, siç e kemi shikuar më parë. Ne do të studiojmë algoritmet skeduluese në kohë-reale preemptable (zëvendësuese) sepse nuk ka kundërshtime të tyre në sistemet multimedia dhe ato janë një përfomancë më të mirë se sa ato nonpreemptable. I vetmi problem është se në se një bufer transmetues është mbushur në pak burst-e (hove), buferi është mbushur nga deadline-et kështu që ai mund të dërgohet tek përdoruesi më një opération të vetëm. Algoritmet kohë-reale mund të janë statik dhe dinamik. Algoritmet statik i caktojnë çdo procesi një prioritet fiks dhe atëherë i jep përparësi skedulimit zëvendësues duke përdorur ato prioritete. Algoritmet dinamike nuk kanë prioritet të fiksuar. Më poshtë ne do të shohim nga një shembull për secilin.

Skedulimi në Shkallë Uniforme

Algoritmi statik klasik i skedulimit zëvendësues në kohë-reale për , proceset periodikë është **RMS** (**Rate Monotonic Scheduling**). Ai mund të përdoret për proceset që plotëson kushtet e mëposhtme:

- Çdo proces përiodik duhet të mbarojë brenda përiodës tij.
- Proceset nuk varen nga njëri tjjetri.
- Çdo proces i nevojitet e njëjtë sasi kohe CPU në çdo burst (hov)
- Çdo proces jopériodik nuk ka deadline (afat përfundimi)
- Proceset zëvendësues ndodhin menjehëre dhe pa overhead.

Katër kushtet e para janë të arsyeshme (të pranueshme). E fundit nuk është, sigurisht, por e bën modelin më të lehtë. RMS funksionon duke i përcaktuar çdo procesi një prioritet fiks të njëvlershme me frekuencën e zhvillimit të ngjarjes triggering. Per shembull,, një proces që duhet të ekzekutohet çdo 30 msec (33 herë/sec) merr prioritetin 33, një proces që duhet të ekzekutohet çdo 40 msec (25 herë/sec) merr prioritetin 25 dhe një proces që duhet të ekzekutohet çdo 50 msec (20herë/sec) merr prioritetin 20. Prioritetet janë kështu lineare me shkallën (rate) (numrin e herëve/sec që procesi ekzekutohet). Kjo është arsyjeja pse quhet shkallë uniforme. Në kohën e ekzekutimit, skeduleri gjithmonë ekzekuton procesin me prioritetin më të lartë që është gati duke zëvendësuar procesin në ekzekutim në se është e nevojshme. Liu dhe Layland provuan se RMS është optimale përgjatë klasës së algoritmeve statik të skedulimit.

Figura 7-12 tregon se skedulimi në shkallë uniforme funksionon në shembullin e Fig. 7-11. Proceset A, B dhe C kanë prioritet statik, përkatësisht 33, 25 dhe 20, që do të thotë se kurdo që A ka nevojë të ekzekutohet, ai ekzekutohet, duke zëvendësuar çdo proces tjeter që aktualisht po përdor CPU. Procesi B mund të zëvendësojë C por jo A. Procesi duhet të presë gjersa CPU nuk është i zënë që të ekzekutohet.

Në Fig. 7-12, fillimisht të tre proceset janë gati të ekzekutohen. Ai me prioritet më të lartë, A, është i zgjedhuri dhe lejohet të ekzekutohet gjersa të përfundojnë në 15 msec, siç tregohet në rreshtin RMS. Pasi përfundon, B dhe C ekzekutohen në atë rend. Së bashku,

këto proçese marrin 30 msec për t'u ekzekutuar, kështu që kur C mbaron, është koha e A për të filluar sërisht. Ky rotacion (altërnim) vazhdon derisa sistemi bëhet i lirë për $t=70$.

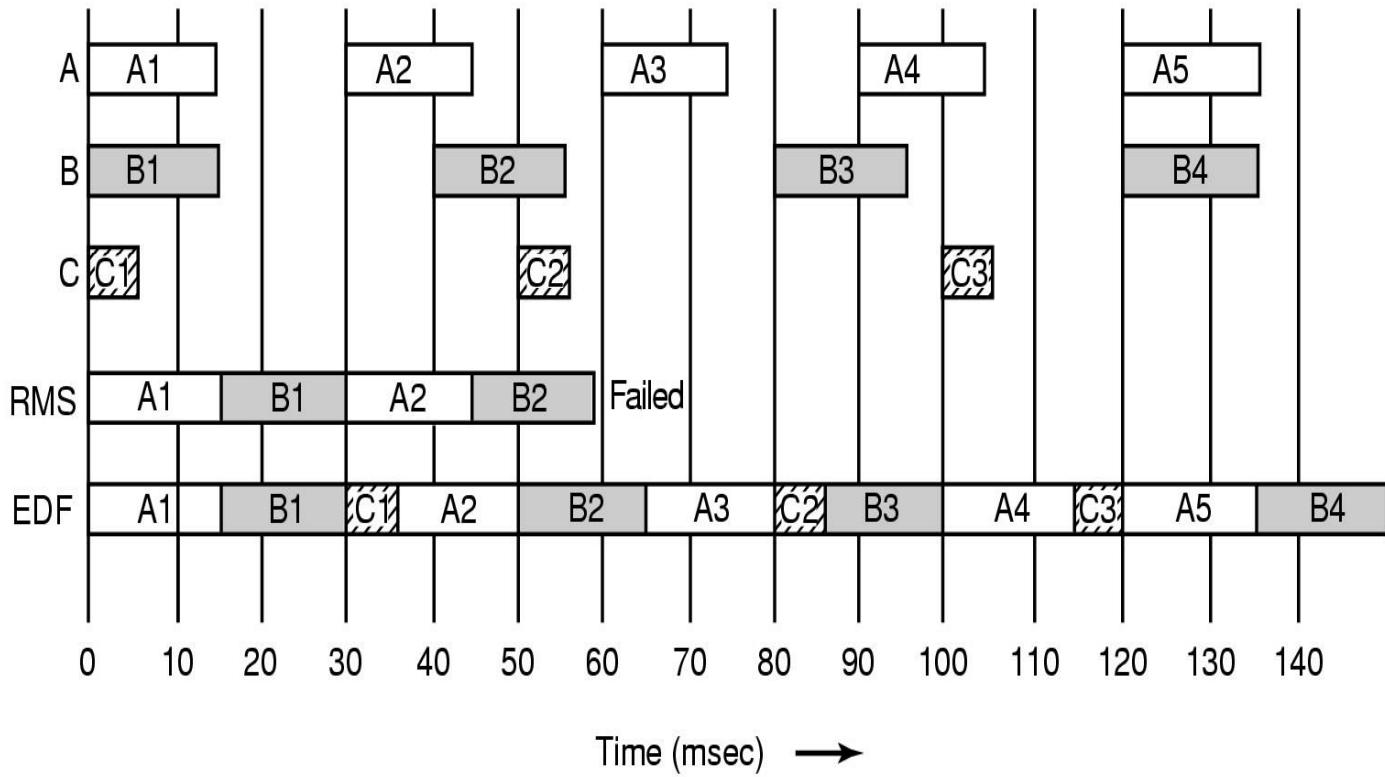


Figure 7-12. Një shembull i skedulimit në kohë-reale të RMS dhe EDF

Për $t=80$ B bëhet i gatshëm dhe ekzekutohet. Megjithatë, për $t=90$, një proçes me prioritet më të lartë, A, bëhet gati, kështu që ai zëvendëson B dhe ekzekutohet deri sa mbaron, për $t=100$. Në këtë pikë sistemi mund të zgjedhë të mbarojë B ose të fillojë C por ai zgjedh proçesin me prioritet më të lartë, B

7.4.4. Skedulimi në fillim i deadline-eve më të hershëm

Një tjetër algoritem populor skedulues në kohë-reale është **Earliest Deadline First**. EDF është një algoritem dinamik që nuk kërkon proçese (përpunim) për të qenë përiodik, siç kërkontë algoritmi me shkalle uniforme. Madje nuk kërkon as të njëjtën kohë ekzekutimi për çdo hov (burst) të CPU, siç bënte RMS. Kurdo që një proçes i nevojitet kohë CPU, ai bën të ditur prezencën dhe deadline-in e tij. Skeduleri ruan një list të proceseve të ekzekutueshëm. Algoritmi ekzekuton proçesin e parë në list, atë që është më afër me

deadline-in. Kurdo që një proces bëhet gati, sistemi kontrollon në se deadline tij ndodhet përpëra ekzekutimit të procesit aktual. Në se po, procesi i ri zëvendëson aktualin.

Një shembull i EDF është dhënë në Fig. 7-12. Fillimisht të gjithë proceset janë gati. Ata ekzekutohen në rendin e deadline-eve të tyre, A duhet të mbarojë për $t=30$, B duhet të mbarojë për $t=40$ dhe C duhet të mbarojë për $t=50$, kështu që A ka deadline-in më të hershem kështu që duhet të fillojë e para. Deri për $t=90$ pengesat janë të njëjtë si tek RMS. Për $t=90$, A bëhet sërisht gati, dhe deadline tij është $t=120$, i njëjtë me deadline e B-së. Skeduleri logjikisht mund të zgjedh se cilin prej të dyve për t'u ekzekutuar, por në praktikë, zëvendësimi i B ka disa kosto lidhur me të, kështu është më mirë që B të vazhdojë të ekzekutohet.

Për të larguar idenë se RMS dhe EDF gjithmonë japin të njëjtën rezultat, le të shohim tani një tjetër shembull, që jepet në figurën e mëposhtme. Në këtë shembull perioda e A, B dhe C janë të njëjtë si më parë, por A tani ka nevojë për 15 msec kohë CPU për çdo hov (burst) në vend të 10 msec. Skedulshmëria teston llogaritjen e përdorimit të CPU si $0.500 + 0.375 + 0.100 = 0.975$. Vetëm 25% e CPU lihet, por në tëori CPU nuk është e mbishkrueshme dhe do të ishte e mundur të siguroje një skeduleri legal.

Figure 7-13. Një tjetër shembull i skedulimit me RMS dhe EDF

Me RMS, prioritet e tre proceseve janë ende 33, 25, dhe 20 vetëm si çështje periodë, jo si kohë ekzekutimi. Kësaj here, B1 nuk mbaron derisa $t=30$, në të cilën A është gati përsëri. Gjithashu A mbaron për $t=45$ dhe B është përsëri gati kështu ka një prioritet më të lartë se C, ai ekzekutohet dhe C humbet deadline-in e tij, RMS dështon.

Dhe tani shiko se si EDF e zgjidh këtë problem. Për $t=30$, ka një “zënkë” midis A2 dhe C1, sepse deadline i C-së është 50 dhe i A2 është 60, C skedulohet. Kjo është e ndryshme nga RMS, ku A ishte fituesi me prioritet më të lartë.

Për $t=90$ A rikthehet gati për të katërtën herë. Deadline i A-së është i njëjtë më atë të procesit aktual (120), kështu që skeduleri ka një zgjidhje për zëvendësim ose jo. Si më parë, është më mirë që të mos ketë zëvendësim në se nuk është e nevojshme, kështu që B3 lejohet të mbarojë.

Në shembullin e Fig. 7-13 CPU është e zënë 100% deri në $t=1500$. Megjithatë, përfundimisht një hapësirë do të përftohet sepse CPU është vetëm 97.5% e shfrytëzuar. Derisa koha e fillimit dhe mbarimit janë shumëfish të 5 msec, hapësira do të jetë 5 msec. Për të realizuar dhe 2.5% e kohës në të cilën CPU nuk është e zënë, hapësira prej 5 msec do të merret për çdo 200 msec, e cila nuk tregohet në Fig. 7-13.

Një pyetje interesante është pse RMS dështon. Parimisht, përdorimi i prioriteteve statik punon vetëm në se përdorimi i CPU nuk është shumë i lartë. Liu dhe Layland (1973) provuan se për çdo sistem të proceseve periodikë, në se

$$\sum_{i=1}^m \frac{Ci}{Pi} \leq m(2^{1/m} - 1)$$

atëherë RMS është e garantuar se do të punojë. Për 3, 4, 5, 10, 20 dhe 100, maksimumet e përdorimit të lejuar janë 0.780, 0.757, 0.743, 0.718, 0.705 dhe 0.696. Kur $m \rightarrow \infty$, maksimumi i përdorimit është një asimptotë $\ln 2$. Me fjalë të tjera, Liu dhe Layland provuan që për tre procese, RMS gjithmonë punon në se përdorimi i CPU është e barabartë ose më pak se 0.708. Në shembullin e parë ishte 0.808 dhe RMS punoi, por vetëm sa ishim me fat. Me perioda dhe kohë ekzekutimi të ndryshme, përdorimi i 0.808 do të dështojë. Në shembullin e dytë, përdorimi i CPU ishte aq i lartë (0.975) sa nuk kishte shpresë se RMS do të punonte.

Në kontrast, EDF gjithmonë punonte për çdo set procesesh të skedulueshme. Mund të arrijë të përdor 100% të CPU-së. Çmimi që do të paguhet është një algoritem më kompleks. Kështu që në serverat video në se përdorimi i CPU-së është poshtë limitit të RMS, atëherë RMS mund të përdoret, në të kundërt duhet të zgjidhet EDF.

KAPITULLI I TETE

MULTIPROCESORET

Nje multiprosesor me kujtese te ndare (shared-memory multiprocessor) eshte nje sistem kompjuterik ne te cilin dy ose me shume CPU ndajne akses te plete ne nje RAM te perbashket. Nje program qe ekzekutohet ne nje nga CPU-te duket si nje hapesire adrese virtuale normale. Karakteristika e vetme, jo e zakonshme qe ky sistem ka eshte qe CPU mund te shkruaje disa vlera ne nje fjale memorje (memory word), dhe me pas lexon fjalen perseri dhe jep nje vlere te ndryshme (sepse nje tjeter CPU e ka ndryshuar ate). Kur organizimi eshte korrekt, kjo karakteristike formon bazen e komunikimit interprocesor: nje CPU shkruan disa te dhena ne memorje dhe nje CPU tjeter lexon te dhenen jashte. Per pjesen me te madhe, sistemet e operimit multiprocesore jane tamam sisteme operimi te rregullta. Ato merren me thirrjet sistem, bejne menaxhimin e memorjes, mundesojne nje file system dhe menaxhojne pajisjet I/O. Megjithate, ndodhen disa zona ne te cilat ato kane tipare unike. Keto perfshijne sinkronizimin e proceseve, menaxhimin e burimeve dhe skedulimin. Me poshte, si fillim do bejme nje veshtrim te permblehdhur te hardware-it te multiprocesorit dhe pastaj shkojme ne funksionet e sistemeve operativ.

8.1.1 Multiprocessor Hardware

Ndone se te gjithe multiprocesoret kane karakteristike qe cdo CPU mund te adresoje te gjitha fjalet e memorjes, disa multiprocesore kane tipar shtese ku cdo fjale memorje mund te lexohet aq shpejt sa cdo fjale tjeter memorje. Keto makina jane quajtur multiprocesore UMA (Uniform Memory Access). Ne kontrast, multiprocesoret NUMA (NonUniform Memory Access) nuk e kane kete tipar. Arsyeva pse ekziston kjo diferenca do te tregohet me vone. Si fillim, ne do te ekzaminojme multiprocesoret UMA dhe me pas do shkojme te multiprocesoret NUMA.

UMA Bus-Based SMP Architectures

Multiprocesoret e thjeshte jane bazuar ne nje single-bus, sic eshte ilustruar ne fig.8-2(a). Dy ose me shume CPU dhe nje ose me shume module memorje, te gjitha perdonin te njejtin bus per komunikim. Kur nje CPU do te lexoje nje fjale memorje, ne fillim ajo kontrollon ne qofte se bus-i eshte i zene apo jo. Ne qofte se bus-i eshte bosh, CPU vendos adresen e fjalet qe do ne bus, kerkon disa sinjale kontrolli dhe pret derisa memorja te vendose fjalen e deshiruar ne bus. Ne qofte se bus-i eshte i zene kur nje CPU do te lexoje ose shkruaje ne memorje, CPU pret pikerisht derisa bus-i te jete bosh. Ne kete menyre zgjidhet problemi me kete projekt. Me dy ose me tre CPU, lufta per bus-in behet e menaxhueshme, me 32 ose 64 ajo behet e pa perballueshme. Sistemi do te jete totalisht i

limituar nga bandwidth-i i bus-it dhe shume nga CPU-te do te jene pa pune shumicen e kohes. Zgjidhja e ketij problemi eshte qe te shtohet nje cache ne cdo CPU, sic pershkruhet ne fig.8-2(b). Cache-ja mund te ndodhet brenda chipit te CPU, afer chipit te CPU, ne bordin e procesorit ose ne disa kombinime te ketyre.

Ne per gjithesi, caching nuk i referohet nje fjale individuale baze por nje bloku prej 32-64 byte. Kur nje fjale adresohet, i gjithe bloku i saj shkon te marre ate (fjalen) ne cache-ne e CPU-se.

Cdo blok cache-je eshte etiketuar sikur cdo blok te ishte read-only (ne kete rast ai mund te jete prezent ne multiple cache ne te njejten kohe) ose si read-write (ne kete rast bloku nuk mund te jete prezent ne ndonje cache tjeter). Në qofte se, nje CPU perpiqet te shkruaje dhe ndodhet ne nje ose me shume remote cache, bus-i hardware detekton shkrimin dhe vendos nje sinjal ne bus per te informuar te gjitha cache-te e tjera mbi shkrimin. Në qofte se, cache-te e tjera kane nje "clean" copy qe mund te jete nje kopje ekzakte e asaj cfare eshte ne memorje, ato pikerisht mund te hedhin kopjet e tyre dhe te lejojne shkruesin (writer) te kape blokun e cache-se nga memorja perpara se ai te modifikohet.

Në qofte se ne disa cache te tjera ndodhet nje "dirty" copy, kjo duhet patjeter te shkruhet perseri ne memorje para se te vazhdoje shkrimi ose kjo te transferohet direkt te shkruesi prane bus-it. Disa cache transferojne protokolle ekzistues. Nje mundesi tjeter eshte pershkrimi i fig.8-2(c), ne te cilin cdo CPU nuk ka vetem nje cache, por gjithashtu ka nje memorje lokale, private ne te cilin ajo akseson mbi nje bus te dedikuar (privat). Per ta perdorur kete konfiguracion ne menyre optimale, kompiluesi duhet te vendose te gjithe tekstin e programit, stringat, konstantet dhe te dhenat e tjera vetem te lexueshme, stacks dhe variablat lokale ne memorien private.

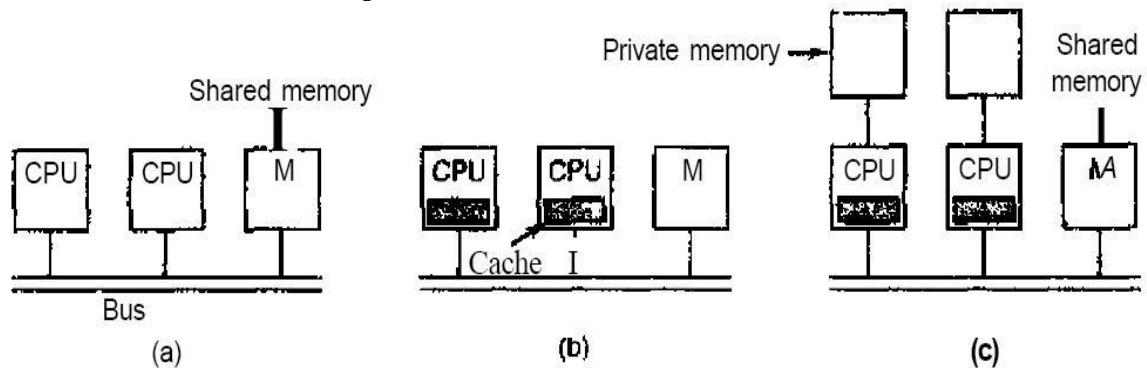


Figure 8-2. Three bus-based multiprocessors, (a) Without caching, (b) With caching., (c) With caching and private memories.

Figura 8-2. Memorja shared perdoret vetem per te shkruajtur variabla shared. Ne shumicen e rasteve, kjo vendosje e kujdeshshme do te zvogelonte shume trafikun ne bus, por kjo do te kerkonte bashkepunim aktiv nga kompiluesi.

UMA Multiprocessors Using Crossbar Switches

Megjithe perdomrimin e caching me te mire, perdomimi i nje single bus e limiton madhesine e nje multiprocesori UMA me rrreth 16 ose 32 CPU. Per te shkuar me tej, nevojitet nje lloj tjeter nderlidhjeje rrjeti. Qarku me i thjeshte per te lidhur n CPU me k memorje eshte **crossbar switch**, i treguar ne fig.8-3. Crossbar switches jane perdomur per dekada brenda centraleve te linjave telefonike per te lidhur nje grup linjash hyrese me nje grup linjash dalese ne nje rruge arbitrale. Ne cdo interseksion te nje linje horizontale (hyrese) dhe vertikale (dalese) ndodhet nje **crosspoint**. Nje crosspoint eshte nje celes (switch) i vogel qe mund te hapet ose te mbyllit elektrikisht, i varur ne se linjat horizontale dhe vertikale jane te lidhura ose jo.

Ne fig.8-3(a) ne shohim tre crosspoints te mbyllura njehersh, duke lejuar lidhjet (CPU, memorje) ndermjet cifteve (00L, 000), (101, 101) dhe (110, U10) ne te njejtën kohë. Shume kombinime te tjera jane gjithashtu te mundshme. Ne fakt numri i kombinimeve eshte i barabarte me numrin e rrugeve te ndryshme qe tete torrra mund te vendosen pa problem ne nje fushe shahu.

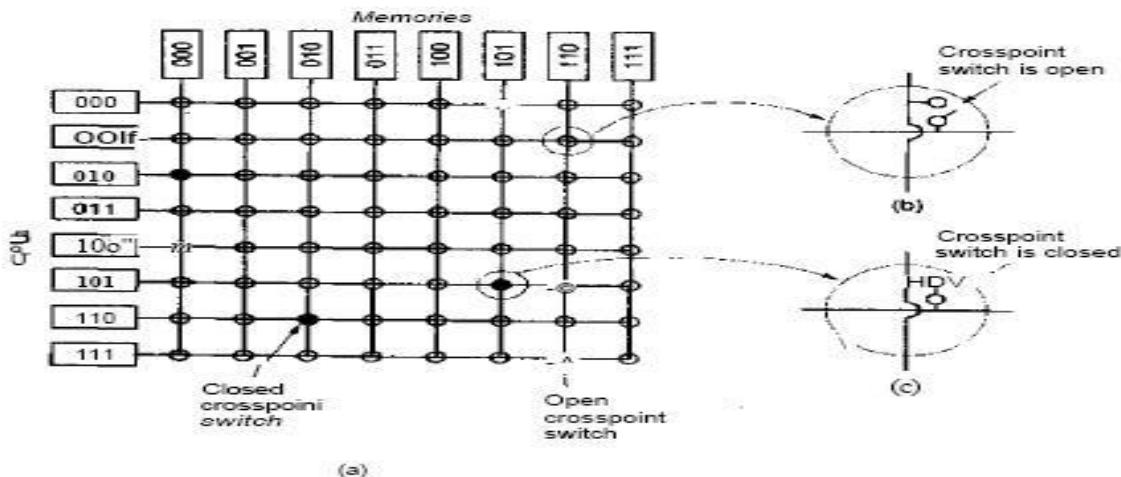


Figure 8-3. (a) An $8 \times K$ crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

Nje nga karakteristikat me te mira te crossbar switch eshte qe kjo eshte nje **nonlocking network** (rrjet jo i mbyllur), qe do te thote se gjithmone asnje CPU nuk do pranoje lidhjen qe i nevojitet sepse disa crosspoint ose linja eshte tashme e zene (supozojme se vete moduli i memorjes eshte i perdorshem, i lire). Gjithashtu asnje planifikim i avancuar nuk nevojitet. Madje në qofte se ngrihen shtate lidhje arbitrimi, eshte gjithmone e mundshme te lidhen pjesa e CPU me pjesen e memorjes.

Nje nga karakteristikat me te keqia te crossbar switch qendron ne faktin se numri i crosspoints rritet me n^2 . Duke patur 1000 CPU-s dhe 1000 module memorje, na duhen 1 milion crosspoints. Nje crossbar switch kaq i madh nuk eshte i realizueshem, pra nuk eshte i mundshem. Megjithate, per sistemet medium-size, nje crossbar switch eshte i realizueshem, i zbatueshem.

UMA Multiprocessors Using Multistage Switching Networks

Nje multiprosesor komplet i ndryshem i bazuar ne nje switch 2×2 te thjeshte tregohet ne fig.8-4(a). Ky switch ka 2 hyrje dhe 2 dalje. Mesazhet e arritura ne cdo linje hyrese mund te transferohen ne cdo linje dalese. Per qellimet tona, mesazhet do te perbehen nga 4 pjese, sic tregohet ne fig.8-4(b). Fusha *Module* tregon se cila memorje perdoret. *Address* specifikon nje adresë brenda nje modulli. *Opcode* jep operacionet, sic jane READ ose WRITE. Ne fund, fusha jo e detyrueshme *Value* mund te permbojte një operand, sic eshte nje fjale 32-bit e cila shkruhet me ane te një WRITE. Switch kontrollon fushen *Module* dhe e perdor ate per te vendosur në qofte se mesazhi do te dergohet ne X ose ne Y. Switch-i jone 2×2 mund te rregullohet ne disa rruge per te ndertuar nje **multistage switching networks**.

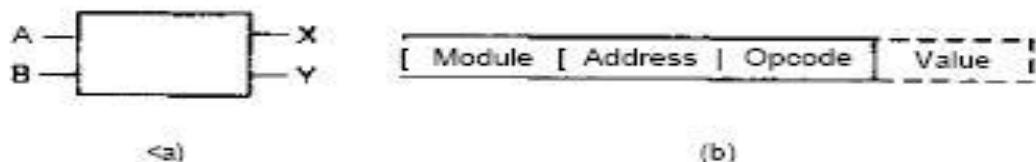


Figure 8-4. (a) A 2×2 switch, (b) A message format.

Nje mundesi eshte klasa ekonomike **omega network** e ilustruar ne fig.8-5. Ketu kemi lidhur 8 CPU, ne 8 memorje duke perdorur 12 switches. Ne pergjithesi, per n CPU dhe n memorje ne do na duhen $\log_2/7$ faza (stages), me $n/2$ switches (celesa) per faze, me nje total prej ($>>/2\log_2<<$ switches, e cila eshte me e mire se n^2 crosspoints, kryesisht per vlera te medha te tij .

Modeli elektrik i rrjetit network (omega network) shpesh quhet **perfect shuffle**, pasi miksimi i sinjaleve ne cdo faze riasemblon nje pako lettrash te prera pergysem, me pas perzihen leter per leter. Per te pare se si punon omega network, supozojme se CPU 01 I do te lexoje nje fjale nga moduli memorjes I 10. CPU dergon nje mesazh READ per te transferuar permajtjen ID110 ne fushen *Module*. Switch merr bitin e pare (nga e majta) te I 10 dhe e perdor ate per routing. Nje 0 cohet ne daljen e siperme dhe nje 1 cohet ne daljen e poshtme. Per sa kohe ky bit eshte 1, mesazhi kalon neper daljen e poshtme te 2D. Te gjithe switches e fazes sekondare, perfishre 2D, perdorin bitin e dyte per routim. Kjo, gjithashtu, eshte nje J, keshtu qe mesazhi tani avancohet ne daljen e poshtme te 3D. Ketu biti i trete testohet dhe duhet te jete 0. Si pasoje, mesazhi shkon ne daljen e siperme dhe arrin ne memorjen 110, sic u deshirua. Rruga e ndjekur nga ky mesazh tregohet ne fig.8-5 nga letra a.

Si mesazh qe leviz permes switching network, bitet ne te majte te numrit te modulit jane jo shume te nevojshme. Per rrugen a, linjat hyrese jane 0 (upper input to ID), 1 (lower input to 2D) dhe I (lower input to 3D), respektivisht. Pergjigja routohet perseri mbapsht, duke perdorur 011, vetem duke lexuar kete nga e majta ne te djathte. Ne te njejtën kohe, CPU 001 kerkon te shkruaje nje fjale ne modulin e memorjes 001. Nje proces analog ndodh ketu, me mesazhe qe routohen neper daljet e siperme dhe te poshtme, respektivisht, te etiketuar nga letra b. Kur kjo arrin, fusha *Module* e saj lexon DOI, duke treguar rrugen qe duhet te ndjeke. Per sa kohe keto dy kerkesa nuk perdorin switches,

linja ose module memorje te njejta, ato mund te vazhdojne ne paralel.

Tani te shohim se c'mund te ndodhe në qofte se CPU 000 kerkon te aksesoje direkt modulin e memorjes 00. Kjo kerkese do te bjere ne konflikt me kerkesen e CPU 001 ne switch 3A. Nje nga keto mund te kete luhatje. Ndryshe nga crossbar switch, omega network eshte nje **blocking network** (rrjet i bllokuar). Jo cdo bashkesi kerkesh mund te plotesohet menjehere. Konfliktet mund te ndodhin gjate perdonimit te nje percjellesi ose nje switch, aq mire sa ndermjet kerkesave *ne* memorje dhe pergjigjeve *nga* memorja.

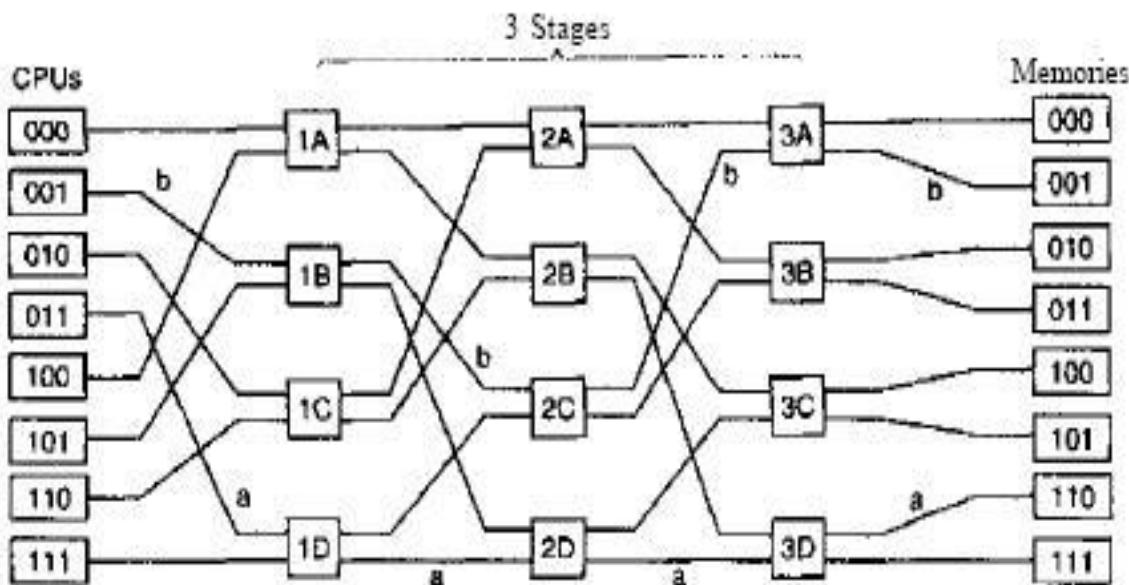


Figure 8-5. An omega switching network.

Eshte forte e pershtatshme te perhapesheren referencat e memorjes uniformisht permis moduleve. Nje teknike e zakonshme eshte perdonimi i biteve low-order si numra te modulit. Konsiderojme, për shembull, nje hapesire adresë te orientuar byte-i, zakonisht 2 bitet low-order do jene 00, por 3 bitet e tjera do jene uniformisht te shperndare. Duke perdonur keto 3 bite si numer moduli, adresat e fjalive te njepasnishesme do jene ne modulet e njepasnishesme. Nje sistem memorje ne te cilin fjalat e njepasnishesme jane ne module te ndryshme do te quhet **interleaved** (i bere ne shtresa). Memorjet interleaved rrisin paralelizmin sepse shumica e referencave te memorjes ndodhen ne adresat e njepasnishesme. Gjtheshtu eshte e mundur te projektohen switching networks te cilat jane nonlocking dhe qe ofrojne rruge te shumta nga cdo CPU ne cdo modul memorje, te cilat lehtesojne dhe trafikun.

NUMA Multiprocessors

Multiprocesoret UMA me nje bus te vetem (single-bus UMA multiprocessors) jane pergjithesisht te limituar ne jo me shume se disa CPU dhe crossbar ose switched multiprocessors. Keto perdonin pajisje hardware te shtrenjta dhe nuk jane shume te

medhenj. Per te marre me shume se 100 CPU, dicka duhet te jepet. Zakonisht, ajo qe jepet eshte idea se te gjitha modulet e memorjes kane kohe te njejte akse simi. Ky koncension con ne idene e multiprocesoreve NUMA, sic tregohet me poshte. Ne ngjashmeri me multiprocesoret UMA, NUMA kane nje hapesire te vetme adresë gjate gjithe CPU-ve, por ne ndryshim nga makinat UMA, akse si per ne modulet e memorjes lokale eshte me i shpejte se akse simi ne largesi. Ne kete menyre te gjitha programet UMA do te ekzekutohen pa problem ne makinat NUMA, por performanca ne keto makina do te jete me e dobet se ne nje makine UMA me te njejten shpejtesi ore. Makinat NUMA kane 3 karakteristika kryesore te cilat edhe i dallojne keto nga multiprocesoret e tjere.

1. Keto kane nje hapesire te vetme adresë e dallueshme ne te gjithe CPU-te.
2. Akse simi i memorjes ne distance (remote memory) behet nga instruksionet LOAD & STORE.
3. Akse simi i memorjes ne distance (remote memory) eshte me i ngadalte se akse simi ne memorje lokale.

Kur koha e akse sit te memorjes ne distance nuk eshte e fshehur (sepse ketu nuk ndodh caching), sistemi quhet **NC-NUMA**. Kur cache-te koherente jane te pranishme, sistemi quhet **CC-NUMA (CACHE COHERENT NUMA)**.

Rruja me e perhapur per ndertimin e multiprocesoreve te medhenj CC-NUMA aktualisht eshte **directory-based multiprocessor**. Ideja eshte qe te kete nje database e cila te treguje se ku ndodhet cdo linje cache-je dhe cfare statusi ka ajo. Kur referohet nje linje cache-je, database mundohet te gjeje se ku ndodhet kjo linje dhe ne se kjo linje eshte clean ose dirty (modified).

Per te patur nje ide me konkrete rreth nje multiprocesori directory-based, le te shohim nje shembull te thjeshte: nje sistem me 256 nyje (a 256 –node system), cdo nyje (node) perbehet nga nje CPU dhe 16 MB RAM te lidhura te CPU permes nje bus-i lokal. Memorja totale eshte 2^{32} byte, e ndare ne 2^{26} linja cache-je me 64 byte secila. Memorja eshte e shperndare ne menyre statike midis nyjave (nodes), me 0-16M ne nyjen 0, 16-32M ne nyjen 1 dhe keshtu me rradhe. Nyjet lidhen me ane te nje rrjeti nderlidhjeje (interconnection network) sic tregohet ne fig.8-6(a). Cdo nyje gjithashtu mban directory entries per $2^{1*} 64$ -byte linjat e cache-se duke perfshire dhe 2^{24} byte memorje. Per momentin, ne do supozojme se nje linje mund te mbahet ne te shumten nje cache.

Per te pare se si punon direktoria, le te marrim nje instruksion LOAD nga CPU 20 e cila i referohet nje linje te cache-se. Ne fillim CPU con instruksionin prezent per ne MMU e saj, e cila perkthen ate ne adresë fizike, domethënë, 0x24000108. MMU-ja e ndan kete adresë ne tre pjesë sic tregohet ne fig.8-6(b).

Ne decimale, tre pjeset perbehen nga: nyja 36, linja 4 dhe offset 8 bit. MMU sheh se fjalë e referuar ne memorje eshte nga nyja 36, dhe jo nga nyja 20, keshtu qe ajo dergon nje mesazh kerkese nepermjet rrjetit te nderlidhjes ne linjen e vete nyjes (node) 36, duke pyetur ne se linja 4 e saj eshte cashed dhe në qofte se po, ku.

Kur kerkesa mberrin ne nyjen 36 gjate rrjetit te nderlidhjes, ajo routohet ne direktorine hardware.

Hardware fut ne tabelen e vet prej 2^{18} entries (hyrje), nje nga linjat e saj te cache-se dhe ekstrakton hyrjen 4 (entry 4).

Ne fig.8-6(c) ne shohim se linja nuk eshte cached, keshu qe hardware terheq linjen 4 nga RAM lokal, e dergon ate mbrapsht tek nyja 20 dhe update-on direktorine e entry 4 per te treguar se tani linja eshte cached ne nyjen (node) 20.

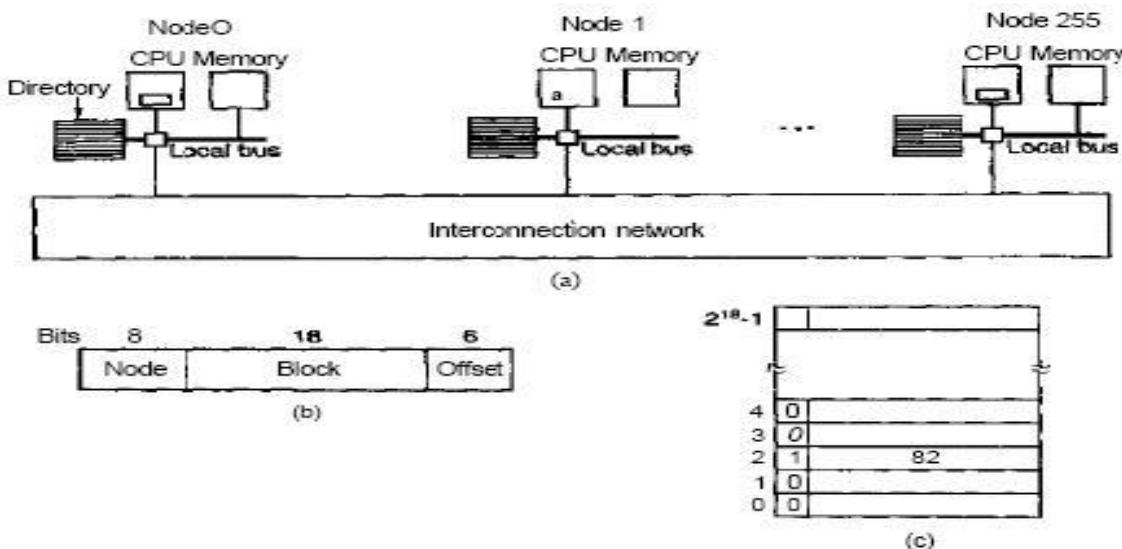


Figure 8-6. (a) A 256-node directory-based multiprocessor (b) Division of a 32-bit memory address into fields, (c) The directory at node 36.

Tani le te shohim nje kerkese te dyte, kete here duke pyetur rreth nyjes 36's linja 2. Nga fig.8-6(c) shohim se kjo linje eshte cached ne nyjen 82. Ne kete pike hardware mund te update-oje direktorine entry 3 qe do te thote se linja tani eshte ne nyjen 20 dhe me pas te dergoje nje mesazh ne nyjen 82, duke njoftuar kete te kaloje linjen ne nyjen 20 dhe te shfuqizoje cache-ne e saj.

Le te llogarisim se sa memorje zihet nga direktorite. Cdo nyje ka 16 MB RAM dhe 2^{18} 9-bit entries per te mbajtur gjurmet e RAM-it. Pra direktoria e siperme ka rreth 9×2^{18} bite te ndara nga 16 MB ose rreth 1.76%, e cila eshte per gjithesish e pranueshme (ndone se ajo duhet te jete memorje me shpejtesi te larte, gje qe rrit koston e saj). Madje me linje cache-je 32-byte overhead-i duhet te jete 4%. Kurse me 128-byte linje cache-je, ajo duhet te jete poshte 1%.

Nje kufizim i qarte i ketij projekti eshte se nje linje (line) mund te jete cached ne nje nyje (node) te vetme. Per te lejuar linjat qe te jene cached ne shume nyje, ne na duhen disa rruge per pozicionimin e gjithe ketyre, për shembull te zhvleresoje ose te update-oje ato ne nje shkrim. Ekzistojne opsione te ndryshme per te lejuar caching ne nyje te ndryshme ne te njejten kohe, por nje diskutim i ketyre mund te trajtohet jashte ketij libri.

8.1.2 Llojet e Sistemeve Operativ ne Multiprocesore (Multiprocessor Operating System Types)

Tani le te shkojme nga multiprocesoret hardware ne multiprocesoret software, ne menyre te vecante, ne sistemet e operimit multiprocesore. Jane te mundshme disa variante. Me poshte ne do te studiojme tre prej tyre.

Cdo CPU Ka Sistemin e Vet Operativ

Rugja me e thjeshte e mundshme per te organizuar nje sistem operimi multiprocesore eshte qe ne menyre statike te ndahet memorja ne aq particione sa procesore ka dhe cdo CPU te kete memorjen e vet private, si dhe kopjen e vet private te sistemit operativ. Ne fakt, n CPU-te me pas do te operojne si n kompjutera te pavarur. Nje optimizim i qarte eshte qe te lejojme te gjitha CPU te ndajne kodin e sistemit operativ dhe te bejne vetem kopje private te te dhenave sic tregohet ne fig 8-7.

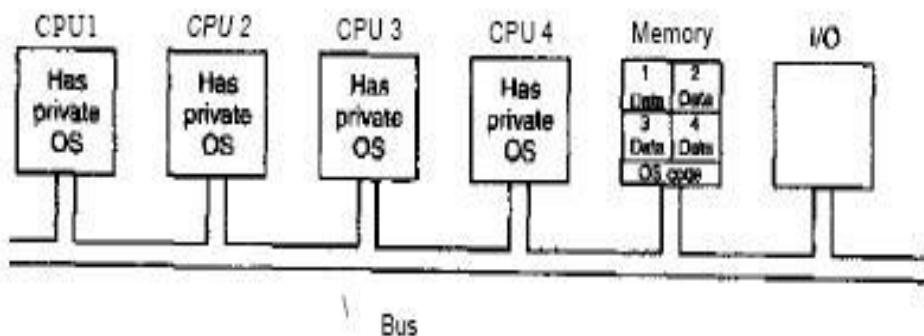


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

Kjo skeme eshte akoma me e mire se duke patur n kompjutera te ndara, sepse kjo i lejon te gjitha makinave te ndajne nje bashkesi disqesh dhe pajisjet e tjera I/O dhe gjithashtu kjo skeme lejon memorjen te ndahet (shared) ne menyre fleksibel. Për shembull, në qofte se nje dite nje program i madh i paperdorur do te ekzekutohet, nje nga CPU do zinte nje pjese extra te madhe te memorjes per kohezgjatjen e ketij programi. Vec kesaj, proceset mund te komunikojne ne menyre eficiente me nje proces tjeter duke patur, le te themi nje prodhues (producer), i cili te jete ne gjendje te shkruaje te dhenat ne memorje dhe nje konsumator per te terhequr te dhenat nga vendi ku prodhuesi i shkruan ato. Eshte nje reference e hollesishme, shume e vlefshme e kater aspekteve te projektit e cila nuk mund te jete shume e qarte.

E para, kur nje proces kryen nje thirrje sistemi, kjo thirrje eshte e zene dhe merret me CPU-ne e vet duke perdorur strukturat e te dhenave ne tabelat e sistemit te vet operativ. E dyta, per sa kohe cdo sistem operativ ka tabelat e veta, ai gjithashtu ka edhe bashkesine e vet te proceseve qe ky sistem i skedulon vete. Ketu nuk ndodh ndarja e proceseve. Në qofte se nje user logon ne CPU 1, te gjitha proceset e tij ekzekutohen ne CPU 1. Si rrjedhoje, mund te ndodhe qe CPU 1 te jete bosh ndersa CPU 2 te jete e mbushur me pune.

E treta, nuk ndodh ndarja e faqeve. Mund te ndodhe qe CPU 1 te kete faqe per te ruajtur ndersa CPU 2 te jete duke faqosur (paging) ne menyre te vazhdueshme. Nuk ka asnje rruge qe CPU 2 te marre hua disa faqe nga CPU 1 derisa shperndarja e memorjes eshte fikse.

E katerta, dhe me e keqja, në qofte se sistemi operativ mban një buffer cache te blloqueve te diskut te perdorur se fundi, cdo sistem operativ e ben kete ne menyre te pavarur nga një tjeter. Keshtu qe mund te ndodhe qe një bllok diskut te jete present ose dirty ne shume buffer cache ne te njejten kohe, duke nga dhene një rezultat te gabuar. Menyra e vetme per te zgjidhur kete problem eshte qe te eleminohen blloqet e cache-se. Per ta bere kete nuk eshte e veshiore por kjo ul shume performancen.

Master-Slave Multiprocessors

Per keto arsy, ky model eshte jo shume i perdorur ndone se ky eshte perdorur ne fillimet e multiprocesoreve, kur qellimi ishte pershatja e disa multiprocesoreve te rinj sa me shpejt te ishte e mundur. Nje model i dyte tregohet ne fig.8-8. Ketu, një kopje e sistemit operativ dhe tabelat e saj jane prezente ne CPU 1dhe jo ne ndonje CPU tjeter. Te gjitha thirrjet sistem jane per te CPU 1 qe te perpunohen atje. Gjithashtu CPU 1 mund te ekzekutoje proceset user në qofte se ka CPU time te mbetur. Ky model quhet **master-slave** ku CPU 1 eshte master dhe gjithe te tjeret jane slaves.

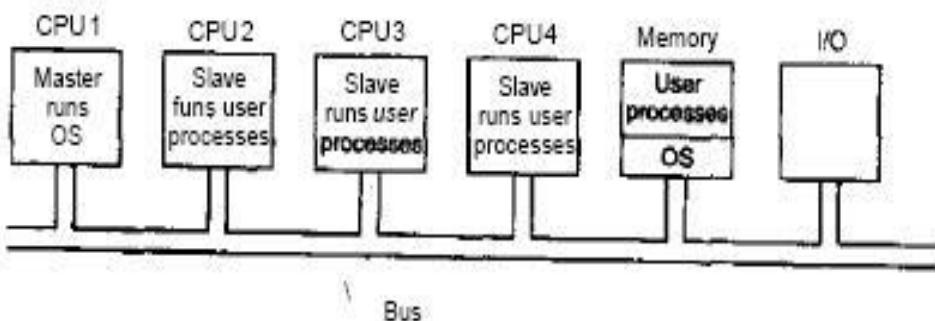


Figure 8-8. A master-slave multiprocessor model.

Modeli master-slave zgjidh shumicen e problemeve te modelit te pare. Ka një strukturë te vetme te dhenash qe mban gjurmen e proceseve ready (gati).

Kur një CPU punon bosh, ajo therret sistemin operativ per një proces qe te ekzekutohet dhe percakton njerin. Keshtu qe asnjeherë nuk mund te ndodhe qe një CPU te jete bosh ndersa një tjeter te jete e mbingarkur ose plot.

Ne menyre te ngjashme, faqet mund te shperndahen per gjithe proceseve ne menyre dinamike dhe ndodhet vetem një buffer cache, keshtu qe nuk mund te kete mosperputhje. Problemi ne kete model eshte se me disa CPU, master-i do jete si një "gryke shisheje". Megjithate, ai mund te merret me te gjitha thirrjet sistem nga te gjitha CPU-te.

Në qofte se, le te themi, 10% e gjithe kohes eshte harxhuar duke u marre me thirrjet sistem, atehere 10 CPU do te mbushin shume master-in, dhe me 20 CPU ai do jete plotesisht i mbingarkuar.

Keshtu qe ky model eshte i thjeshte dhe i zbatueshem per mikroprocesoret e vegjel, ndersa per te medhenjte ai deshton.

Symmetric Multiprocessors

Modeli yne i trete, **SMP** (Multiprocesoret Simetrike) eliminon asimetritet. Ky eshte nje kopje e sistemit operativ ne memorje por edhe ndonje CPU mund ta ekzekutoje ate. Kur behet nje thirrje e sistemit, CPU-ja ne te cilen eshte kryer thirrja e sistemit bie ne Kernel dhe proceson thirrjen e sistemit. Modeli SMP eshte ilustruar ne figuren 8.9.

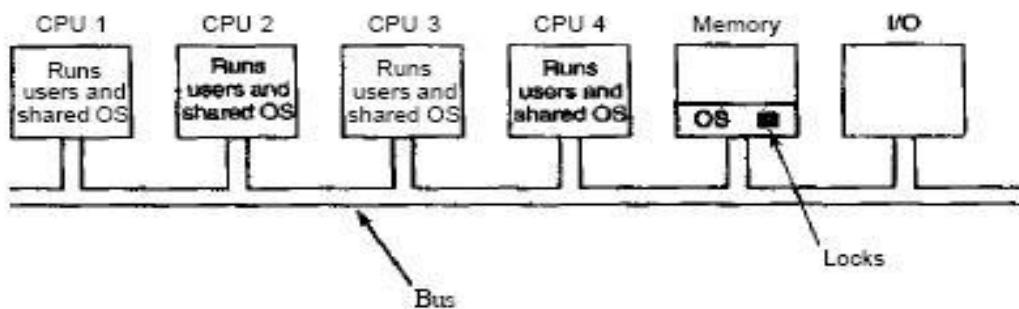


Figure 8-9. The SMP multiprocessor model

Ky model balancon proceset dhe memorjen ne menyre dinamike, perderisa ka nje set tabelash te sistemit operativ. Ai gjithashtu eleminon CPU master bottleneck pasi nuk ka master. Por ai zoteron te tjera probleme. Ne vecanti, ne qofte se dy ose me shume CPU ekzekutojne kodin e sistemit operativ ne te njejten kohe, rezultati do te jete shkatterru. Imagjinoni dy CPU qe punojne njekohesisht duke zgjedhur te njejtin proces per ta ekzekutuar ose ti drejtohen te njejtes faqe te lire te memorjes. Menyra me e mire per te zgjidhur keto problem eshte bashkimi i nje mutex (për shembull, lock) me sistemin operativ, duke e bere te gjithe sistemin nje zone te madhe kritike. Kur nje CPU do te ekzekutoje kodin e sistemit, duhet fillimisht te marre ne zoterim mutex-in. Në qofte se mutex-i eshte i kryer, i duhet te prese. Ne kete menyre, cdo CPU mund te ekzekutoje sistemin operativ, por ne nje kohe te caktuar.

Ky model funksionon, por ka disavantazhe po aq sa modeli master-slave. Serish supozojme qe 10% e kohes totale eshte shpenzuar brenda sistemit operativ. Me 20 CPU do te kete nje rradhe te gjate te CPU-ve qe presin te hyjne ne sistem. Per fat te mire, eshte e thjeshte te permiresohet. Shume pjesa te sistemit operativ jane te pavarura nga njera-tjetra. Per shembull, nuk ka asnjë problem me nje CPU qe ekzekuton skedulerin nderkohë qe nje CPU tjeter po mban nje thirrje te sistemit file dhe nje CPU e trete proceson nje page fault.

Ky rregull con ne ndarjen e sistemit operativ ne rajone kritike te pavarura qe nuk nderveprojne me njera-tjetren. Cdo rajon kritik mbrohet nga mure i tij personal, keshtu qe ne nje kohe te caktuar vetem nje CPU mund ta ekzekutoje ate. Ne kete menyre arrihet me shume parallelizem.

Gjithese si fare mire mund te ndodhe qe disa tabela, si tabelat e procesit te perdoren nga shume zona kritike. Per shembull, nje tabele procesi nevojitet per skedulim, por

gjithashtu per thirrjen e sistemit te ndare dhe gjithashtu permajtjen e sinjalit. Cdo tabele qe mund te perdoret nga shume zona kritike i nevojitet mutex-i i tij personal. Ne kete menyre nje zone kritike mund te ekzekutohet nga nje CPU e vetme ne nje kohe te caktuar dhe nje tabele kritike eshte e aksesueshme nga nje CPU e vetme ne nje kohe te caktuar.

Shumica e multiprocesoreve moderne perdonin kete organizim. Pjesa me e vesh tire ne shkrimin e sistemit operativ nga nje makine, nuk eshte se kodi aktual eshte shume i ndryshem nga nje sistem i rregullt operativ. Pjesa me e vesh tire eshte ndarja ne zona qe ekzekutohen njekohesisht nga CPU te ndryshme pa nderhyre te njera-tjetra jo vetem ne menyra te vesh tira, indirekte, por per me teper cdo tabele e perdonur nga dy ose me shume zona kritike duhet te mbrohet e ndare nga nje mutex dhe te gjitha kodet qe perdonin tabelen duhet te perdonin mutex-in ne menyre korrekte.

Per me teper, duhet treguar kujdes i madh per te shmangur bllokimet. Ne qofte se dy zona kritike kane nevoje per tabelen A dhe tabelen B, dhe njera prej tyre kerkon A te paren dhe tjetra kerkon B te paren, heret a vone nje bllokim do te ndodhe dhe askush s'do ta dije pse. Teorikisht, te gjitha tabelat mund te kerkojne vlera integrale dhe zonave kritike mund t'ju kerkohet te perfshije tabela ne rend rrites. Kjo strategji shmang bllokimet, por i kerkon programuesit te mendoje me shume kujdes cilat tabela i nevojiten seciles zone kritike per te bere kerkesat sipas rradhes se duhur.

Teksa kodi zhvillohet me kalimin e kohes, nje zone kritike mund te kete nevoje per nje tabele qe nuk i nevojitej me pare. Ne qofte se programuesi eshte i ri dhe nuk e kupton logjiken e plote te sistemit, do te tundohet me pas ne marrjen e mutex-it ne tabele ne pikun qe nevojitet dhe e clirojne ate kur nuk nevojitet me. Gjithese si, logjikisht kjo mund te ndodhe duke cuar ne bllokime, per te cilat perdonuesi vihet ne dijeni tek sa sistemi behet i qendrueshem. Marrja e sakte nuk eshte e thjeshte dhe mbajtja per nje periudhe te gjate kohore ne ndryshimin e vazhdueshem te programuesve eshte shume e vesh tire.

8.1.3 Sinkronizimi Multiprosesor (Multiprocessor Synchronization)

CPU-te ne nje multiprosesor zakonisht kane nevoje te sinkronizohen. Sapo pame rastin ne te cilin zonat kritike te kernelit dhe tabelat duhet te jene te mbrojtura nga mutexet. Le te shikojme tani shkurt se si punon aktualisht ky sinkronizim ne nje multiprosesor.

Per te filluar, primitivat e duhura te sinkronizimit jane vertet te nevojshme. Ne se nje proces ne nje multiprosesor ben nje thirrje sistem qe kerkon akse simin e disa tabelave kritike ne kernel, kodi i kernelit vetem mund te ç'aktivizoje interruptet perpara se ta kape tabelen. Ai me pas mund te beje punen e tij duke ditur qe ai do te jete i afte te perfundoje pa vjedhur ndonje proces tjeter dhe duke kapur tabelen perpara se te perfundoje. Ne nje multiprosesor, ç'aktivizimi i interrupteve ndikon vetem tek CPU qe po bente ç'aktivizimin. CPU-te e tjera mund te vazhdojne te ekzekutojne dhe tashme mund te kapin tabelen kritike.

Si pasoje, nje protokoll i pershtatshem mutex duhet te perdoret dhe te respektohet nga te gjitha CPU-te per te garantuar qe mutual exclusion te punoje.

Pjesa kryesore e nje protokolli praktik mutex eshte nje instruksion qe lejon nje fjale memorje te inspektohet dhe te vendoset ne nje operacion te pandashem. Pame se si TSL (Test and Set Lock) ishte perdonur ne fig. 2-22 per te implementuar zonat kritike. Siç diskutuam edhe me pare, ajo çfare ben ky instruksion eshte qe lexon nje fjale memorje dhe e ruan ate ne nje register. Menjehere, ai shkruan nje 1 (ose disa vlera te tjera jozero)

ne fjalen e memorjes. Sigurisht, ai merr dy cikle busesh te ndara per te treguar leximin dhe shkrimin e memorjes. Ne nje multiprosesor, per sa kohe qe nje instruksion nuk mund te ndahet per gjysme, TSL punon gjithmone siç u supozua.

Tani le mendojme se çfare mund te ndodhe ne nje multiprosesor. Ne figuren 8-10 shohim rastin me te keq te kohezimit, ne te cilin fjala 1000 e memorjes, po perdoret si nje bllokues (lock) qe fillimisht eshte 0. Ne hapin 1, CPU 1 lexon fjalen dhe merr nje 0. Ne hapin 2, perpara se CPU 1 te kete mundesine te rishkruaje fjalen ne 1, CPU 2 e merr fjalen dhe gjithashtu e lexon si 0. Ne hapin 3, CPU 1 shkruan nje 1 tek fjala. Ne hapin 4, CPU 2 gjithashtu shkruan nje 1 tek fjala. Te dyja CPU-te marrin mbrapsht nje 0 nga instruksioni TSL, keshtu qe te dyja tani kane akses tek zona kritike dhe mutual exclusion deshton.

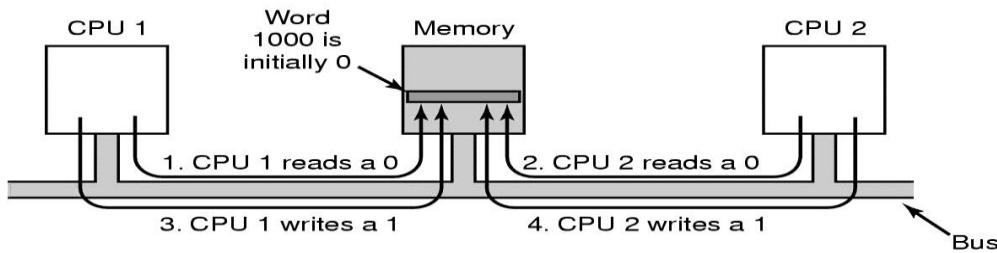


Figure 8-10. The TSL instruction can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

Intruksioni TSL mund te bllokohet ndersa busi nuk mund te bllokohet. Keto 4 hapa tregojne sekuencen e ngjarjeve ku demonstrohet nje deshtim.

Per te ndaluar kete problem, intruksioni TSL se pari duhet te bllokoje busin duke ndaluar CPU-te e tjera qe ta aksesojnë, te beje te dy akse simet e memorjes, dhe me pas te zhbllokoje busin. Zakonisht, bllokimi i busit behet duke kerkuar busin me ane te protokollit kerkese te perdonimit te busit, me pas duke kerkuar disa linja busi speciale derisa *te dy* ciklet te jene kompletuar. Per sa kohe qe kjo linje speciale kerkohet, asnje CPU-je tjeter nuk do ti jepet akse simi i busit. Ky intruksion mund te implementohet vetem ne nje bus qe ka linjat e nevojshme dhe protokollin (hardware) per perdonimin e tyre. Buset moderne i kane keto lethesira, por me perpara ato nuk i kishin keto dhe ishte e pamundur te implementoje nje intruksion TSL ne menyre korakte. Kjo eshte arsyje pse protokolli i petersonit ishte shpikur, per te sinkronizuar teresisht nje software.

Ne se TSL implementohet dhe perdoret korrektesisht, ajo mund ta beje mutual exclusion te punoje. Megjithate, kjo metode mutual exclusion perdor nje **spin lock** sepse CPU e kerkuar vetem vendoset ne nje cikel duke testuar shpejtesine e lock. Jo vetem qe shperdonon kohen e CPU (ose CPU-ve) te kerkuara, por ai mund te vendose nje ngarkese te madhe ne bus ose memorje, duke ngadalesuar punen e gjithe CPU-ve te tjera qe perpiqen te bejne normalisht punen e tyre.

Ne pamje te pare, mund te duket se prezanca e caching duhet te eleminoje problemin e permbajtjes se busit, por nuk eshte e vertete. Ne teori, kur CPU e kerkuar ka lexuar fjalen lock (blloko), ajo duhet te marre nje kopje ne cache-ne e saj. Per sa kohe qe asnje CPU tjeter nuk tenton te perdore lock, CPU e kerkuar duhet te jete e afte te ekzekutoje jashte

cachese se saj. Kur CPU zoteron lock, ajo shkruan nje 1 per ta lene ate, protokolli i cachese automatikisht zhvlereson te gjitha kopjet e tij ne ato pak cache duke kerkuar vlerat korrekte per tu marre perseri.

Problemi eshte qe cache-ja vepron ne blloqe 32 ose 64 byte. Zakonisht, fjalet perreth lock nevojiten per CPU qe po mban lock. Meqene se instruksioni TSL eshte nje shkrim (sepse modifikon lock), atij i duhet nje akses eskluziv ne bllokimin e cachese duke permbajtur lock. Prandaj, çdo TSL shfuqizon bllokimin e mbajtesit te lock ne cache dhe merr nje kopje private, eksluzive per CPU e kerkuar. Sapo mbajtesi i lock has nje fjale te perafert me lock, blloku i cachese zhvendoset ne makine. Si pasoje, i gjithe blloku i cachese qe permban edhe lock ne menyre konstante shkon dhe vjen midis mbajtesit te lock dhe kerkuesit te lock, duke gjeneruar edhe me shume trafik ne bus sesa leximi i nje fjale te vetme lock.

Ne se ne mund te çlirohem i nga te gjitha TSL e shkaktuara te shkruara ne anen e kerkuar, ne mund te reduktojme shperdorimin e cachese ndjeshem. Ky synim mund te arrihet duke bere qe CPU e kerkuar se pari te beje nje lexim te paster per te pare ne se lock eshte i lire. Vetem ne se lock duket te jete i lire duhet te beje nje TSL per ta siguruar ate. Rezultati i ketij ndryshimi te vogel eshte qe shumica e zgjedhjeve tanj eshte lexim ne vend te shkrimit. Ne se CPU qe po mban lock vetem po lexon variablat ne te njejtin bllok cacheje, secili prej tyre mund te kene nje kopje te bllokut te cachese ne menyen e sharuar read-only, duke eleminuar te gjitha transferimet e bllokut te cachese. Kur lock perfundimisht lirohet, zoteruesi ben nje shkrim, i cili kerkon nje akses eksluziv, keshtu shfuqizojme te gjitha kopjet e cacheve te mbetura. Ne leximin tjeter nga CPU e kerkuar, blloku i cachese do te ringarkohet. Theksojme se ne se dy apo me shume CPU po pretendojne per te njejtin lock, mund te ndodhe qe te dy shikojne qe eshte i lire menjehere, te dy bejne nje TSL per ta marre ate. Vetem njera prej tyre do te kryhet me sukses, dhe keshtuqet keto s'kemi kushte shpejtesie sepse detyra e vertete eshte bere nga instruksioni TSL, dhe ky instruksion eshte atomik. Duke pare qe lock eshte i lire dhe duke u perpjekur per ta marre ate menjehere me nje CX u TSL nuk na garanton qe mund ta marrim. Ndonje tjeter mund te fitoje.

Nje rruge tjeter per te reduktuar trafikun e busit eshte perdonimi i algoritmit Ethernet binary exponential backoff (Anderson, 1990). Ne vend qe te zgjedhe vazhdimeshit, si ne fig. 2-22, mund te futet nje cikel vone se midis zgjedhjeve. Fillimisht vonesa eshte nje instruksion. Ne se lock eshte ende i bllokuar, vonesa dyfishohet ne dy instruksione, ne kater dhe keshtu me radhe deri ne disa maksimume. Nje maksimum i ulet jep nje perqjigje te shpejte kur lock eshte lene, por shperdon me shume cikle buseh ne cache. Nje maksimum i larte redukton shperdorimin e cachese por me çmimin qe nuk njofton shpejt qe lock eshte i lire. Ky algoritdem mund te perdoret me ose pa leximet e pastra duke na paraprire instruksioni TSL.

Nje tjeter ide me e mire eshte qe ti japim çdo CPU deshiren per te siguruar mutex variablen e tij privat lock per ta testuar, siç tregohet ne figuren 8-11. Variabla duhet te rrije ne nje bllok cacheje te ndryshem per te ndaluar konfliktin. Algoritmi punon duke pasur nje CPU qe deshton per te siguruar alokimin e lock, dhe e vendos veten ne fund te listes se CPU-ve duke pritur per lock. Kur mbajtesi aktual i CPU del nga zona kritike, ai liron lock privat qe CPU e pare ne liste po testonte (ne cachene e tij). Kjo CPU me pas hyn ne zonen e tij kritike. Kur mbaron pune, ajo liron lock pasues qe eshte duke u perdonur, dhe keshtu me rradhe. Megjithese protokolli eshte disi i komplikuar (per te

evituar vendosjen e dy CPU-ve ne fund te listes njekohesisht), ai eshte efiçent dhe i lire nga starvacioni.

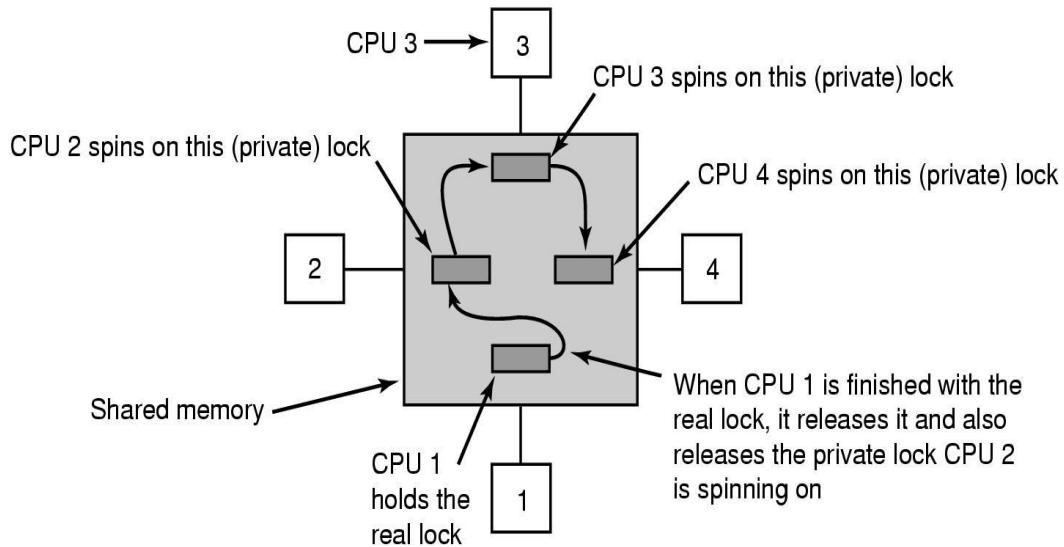


Figura 8-11. Perdorimi i multiple locks per te evituar shperdorimin e cachese.

Spinning kundrejt Switching

Me pare ne kemi supozuar qe nje CPU qe i nevojitet nje mutex lock vetem pret per te, per me teper duke zgjedhur vazhdimisht, ne menyre te nderprere ose duke e vendosur veten ne nje liste priteze te CPU-ve. Ne disa raste, nuk ka nje zgjedhje reale per ti kerkuar CPU vetem te prese. Për shembull. supozojme se disa CPU jane pa pune dhe iu duhet te aksesojne listen e sharuar ready per te zgjedhur nje proces per ta ekzekutuar. Ne se lista ready eshte e bllokuar, CPU nuk mund te nderprese ate çfare ajo po ben dhe te ekzekutoje nje proces tjeter, sepse qe te bejme kete duhet te kemi akses tek lista ready. *Duhet* pritur derisa te mund te sigurohet lista ready.

Megjithate, ne raste te tjera ka nje alternative. Për shembull. ne se disa threadave ne nje CPU iu duhet te aksesojne buferin e file-ve sistem te cachese dhe ai eshte aktualisht i bllokuar, CPU mund te vendose te komutohet (switch) ne nje tjeter thread ne vend qe te prese. Çeshtja se kur te bejme nje spin (rrotullim) dhe kur te bejme nje komutim threadi ka qene nje çeshtje e shume kerkimeve, disa prej te cileve do te diskutohen me poshte. Theksojme se kjo nuk ndodh ne sistemet me nje procesor sepse spinning nuk ka shume kuptim kur nuk ka CPU tjeter per te lene te lire lock. Ne se nje thread perpiqet te siguroje nje thread dhe deshton, ai eshte gjithmone i bllokuar per ti dhene zoteruesit te lock nje mundesi per ta ekzekutuar dhe per ta lene te lire lock.

Duke supozuar qe spinning dhe komutimi i threadit jane te dyja opsione te mundshme, çeshtja eshte si me poshte. Spinning shperdon direkt ciklet e CPU. Testimi i nje lock vazhdimisht nuk eshte nje pune produktive. Komutimi, megjithate, shperdon gjithashtu ciklet e CPU, meqene se gjendja aktuale e threadit duhet te ruhet, duhet te sigurohet lock

ne listen ready, duhet te zgjidhet nje thread, gjendja e tij duhet te ngarkohet, dhe duhet te startoje. Gjithashtu, cacheja e CPU do te permbaje te gjithe blloqet gabim, keshtu qe shume deshetime cacheje te kushtueshme do te ndodhin kur threadi i ri starton ekzekutimin. Deshtimet e TLB jane gjithashtu te ngashme.

Perfundimisht, duhet te behet nje komuntim tek threadi original, me shume deshitime cacheje qe e shoqerojne. Ciklet e shpenzuara duke bere keto dy komutime plus dhe deshtimet e cacheze shperdorohen.

Ne se dihet qe mutexet per gjithesist mbahen për shembull. per 50 μ sek dhe duhet 1 msec per te komutuar nga threadi aktual dhe 1 msec per komutimin mbrapsht, eshte me eficiente qe vetem te bejme spin mbi mutex. Nga ana tjeter, ne se mesatarja e mutex mbahet per 10msec, eshte problem i rendesishem berja e dy komutimeve. Problemi eshte qe zonat kritike mund te variojne konsiderueshem ne kohezgjatjen e tyre, keshtu qe cila eshte rruga me e mire?

8.1.4 SCHE DULIMI MULTIPROCESOR

Ne nje procesor te vetem schedulimi eshte nje dimensional. Pjetja e vetme qe i duhet per gjigjur me doemos eshte: "Cili proces do te ekzekutohet me vone?" Ne sistemet me shume procesor schedulimi eshte dy dimensional. Scheduleri duhet te zgjedhe se cilin proces te ekzekutoje dhe ne cilin CPU ta ekzekutoje. Ky dimension shtese e komplikon shume procesin e schedulimit.

Nje tjeter faktor komplikues eshte se ne disa sisteme, proceset nuk kane lidhje me njeri tjeterin ndersa ne disa te tjre ato grupohen ne grupe. Nje shembull i kesaj situate eshte nje sistem *timesharing* ne te cilin user te vecante inicializojne procese te vecante. Proseset jane te palidhur me njeri tjeterin dhe mund te perzgjidhen panvarisht nga proceset e tjere. Kjo situate ndodh shpesh ne ambjemt per zhvillimin e programeve. Sisteme te medha zakonisht konsistojene ne nje numer te caktuar file header qe permbajne macros, percaktimet e tipeve dhe deklarime variablash qe perdoret nga kodi i file. Kur nje file header ndryshon, i gjithe kodi i tij duhet et rikompilohet. Programi **make** perdoret gjeresisht per te menaxhuar zhvillimin. Kur thirret **make**, ai fillon kompilimin e atyre kodeve te file qe duhet rikompiluar. File objekt qe jane akoma te vlefshem nuk ndryshojne.

Versioni original i **make** bente punen e tij ne menyre sekuenciale, por versionet e reja te dizenuara per shume procesore mund te fillojne te gjithe kompilimin njehersh. Ne se duhen 10 kompilime, nuk ka kuptim te zgjedhesh 9 prej tyre shpejt dhe ta lesh te fundit per me vone, kjo do te sjelle nje performance te ulet pasi puna e kerkuar nga useri nuk do te perfundonte. Ne kete rast proceset duhet te perzgjidhen si grup.

TIMESHARING

Le te shohim rastin e perzgjedhjes se proceseve te pavarura nga njeri tjetri: me vone do te shohim si te perzgjedhim procese te lidhur midis tyre. Algoritmi me i thjeshte perzgjedhes qe merret me proceset e pavarur midis tyre eshte qe te kesh nje sistem te madh strukture te dhenash per proceset gati, ndoshta vetem nje liste, por me shume

mundesi nje bashkesi listash per proceset me priorite te ndryshem sic jepet ne Fig. 8-12(a)

Ketu te 16 CPU-te jane te zena, dhe nje set procesesh te cilet kane priorite te ndryshem pret qe te ekzekutohet. CPU-ja e pare q mbaron punen e saj eshte CPU 4, qe me pas merr rradhet nga te cilet perzgjidhen proceset dhe selekton procesin me prioritet me te larte qe eshte ne kete rast A, sic tregohet ne Fig.8-12(b). Me pas CPU 12 eshte pa pune dhe zgjedh procesin B, sic tregohet ne Fig.8-12(c). Per sa kohe proceset nuk kane lidhje midis tyre, berja e schedulimit ne kete menyre eshte shume e arsyeshme.

Duke pasur nje strukturre te vetme schedulimi te perdorur nga te gjithe CPU-te, kjo ben qe CPU-te te ndajne kohen e perdorimit (timeshare).

Dy disavantazhet e kesaj menyre jane: konfliktet per perdorimin e kesaj strukturre duke pasur parasyphe se numri i CPU-ve do te rritet dhe overhead-i qe krijohet kur kemi nje nderrim midis proceseve kur nje proces bllokohet per I/O.

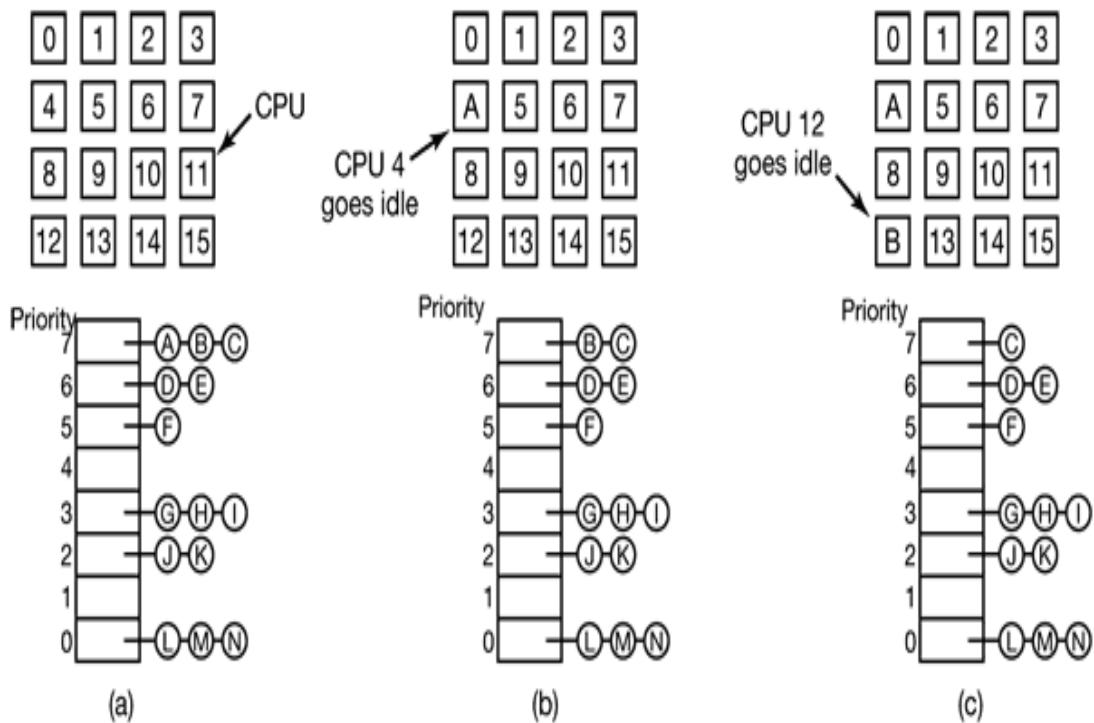


Figura 8-12. Perdorimi i nje strukturre te dhenash per schedulim tek nje multiprocesor.

Gjithashtu eshte e mundur qe nje kontekst switch te ndodhe kur nje proces 'quantum' mbaron. Ne nje multiprocesor, qe ka karakteristika te caktuara jo prezente ne nje uniprocesor, supozojme qe procesi ka nje spin lock, jo i zakonshem ne nje uniprocesor, sic do e tregojme me poshete. Ne nje uniprocesor, spin locks jane jo shume te perdorshem keshtu qe, në qofte se nje proces nderpritet ndone se ka nje mutex, atehere edhe procesi tjeter starton dhe perpiqet te marre mutex-in. Ky do te bllokohet menjehere, keshtu qe ajo pak kohe do te humbase.

Per te rreguluar kete anomali, disa sisteme perdonin **smart scheduling**, ne te cilen nje process qe ka siguruar nje spin lock vendos nje process-wide flag, pra nje flamur qe te tregoje se ky proces tashme ka nje spin lock. Kur procesi liron braven, the lock, ai pastron flag-un. Me pas scheduler-i nuk nderpret nje process duke mbajtur nje spin lock, por me mire merr ate pak kohe per te kompletuar zonen e tij kritike dhe liron braven.

Nje tjeter ceshje qe luan rol ne scheduling eshte fakti qe ndersa gjithe CPU-te jane te njejta, disa CPU Jane akoma me te njejta. Ne menyre te vecante, kur procesi A ekzekutohet per nje kohe te gjate ne CPU *, cache-ja CPU V* do te jene plot me blloqet e A-se.

Në qofte se procesi A kerkon te ekzekutohet serish se shpejti, ai do bez mire te ekzekutohet ne CPU L sepse cache-ja mund te permbate akoma blloqe te A-se. Prania e blloqeve te cache-se do sjelle rritjen e cache hit rate dhe si rrjedhim rritjen e shpejtesise se procesit.

Vec kesaj, TLES gjithashtu mund te permbate faqet e duhura, duke reduktuar gabimet e TLB-se.

Disa multiprocesore perdonin ate qe quhet **affinity scheduling** (lidhje skedulimi). Ketu ideja kryesore eshte qe te behet nje perpjekje serioze per te patur nje process te ekzekutueshem ne CPU-te e njejta. Nje menyre per te krijuar kete arritje eshte qe te perdoret nje algoritem skedulimi me dy nivele (**two-level scheduling algorithm**). Kur nje proces eshte krijuar, ai ngarkohet ne nje CPU. Ngarkimi i proceseve ne CPU eshte niveli i siperi i algoritmit. Si rezultat, cdo CPU merr koleksionin e vet te proceseve.

Skedulimi aktual i proceseve eshte niveli i fundit te algoritmit. Kjo behet nga cdo kompjuter ne menyre te vecante, duke perdonur karakteristikat ose disa arsyet te tjera. Duke u perpjekur te mbajme nje proces ne te njejten CPU, terheqja e cache-se rritet. Megjithate, në qofte se nje CPU nuk ka procese per te ekzekutuar, ai merr nje proces nga nje CPU tjeter duke bere qe te mos punoje bosh.

Skedulimi me dy nivele ka tre te mira. E para, ky shperndan rregullisht ngarkimin barabar ne CPU-te e perdonshme. E dyta, avantazhi qendron ne lidhjen e cache-se kur eshte e mundur. E treta, duke patur cdo CPU listen e saj gati, lufta per kete liste minimizohet sepse perpjekjet per te perdonur listat e CPU-ve te tjera jane relativisht jane te rralla.

Space Sharing

Nje tjeter metode per skedulimin e multiprocesorit, mund te jetë kur procesoret jane te lidhur me njeri-tjetrin ne ndonje lloj menyre. Me perpara ne permendem shembullim e lidhjes ne paralel si nje nga rastet. Gjithashtu mund te ndodhi qe nje proces i vetem mund te lindi shume threads, te cilat punojne se bashku. Per qellimin tone, nje pune qe konsiston ne shume procese te lidhura me njeri-tjetrin, ose nje proces qe konsiston ne shume kernel threads, jane e njejta gje. Ne ketu do ti referohemi threads, si lloj te skedulimit, por materiali mund te perdoret dhe per proceset gjithashtu. Skedulimi i shume threads ne te njejten kohe, nepermjet shume CPU-ve eshte quajtur space **sharing** (ndarje e hapesires).

Algoritmat me te thjeshte per space sharing punojne ne kete menyre. Supozojme se nje grup i tere qe ka lidhje me threads eshte krijuar njekohesht. Ne kohen qe krijohet, skeduleri kontrollon ne qofte se ka po aq CPU te lira sa threads jane. Ne qofte se jane, cdo thread-i i jepet CPU-ja e vet (dedikuar) dhe ata fillojne startimin, ne qofte se nuk ka CPU te mjaftueshme, asnjë nga thread-et nuk fillon startimin derisa te kemi CPU te mjaftueshme. Cdo thread qendron ne CPU-ne e tij deri sa te perfundoje procesin, ne te

njejten kohe CPU-te e lira kthehen mbrapsht ne vendin e CPU-ve te lira. Ne qofte se nje thread bllokohet nga I/O, ai vazhdon ta mbaje CPU-ne te zene, e cila eshte thjesht ne gjendje idle derisa thread te zgjohet. Kur shfaqet thread i radhes aplikohet e njejt procedure.

Ne nje moment kohe te caktuar, nga ana stukturore CPU-ja ndahet ne nje numer pjesesh, ne secilen prej tyre vepron threads i nje procesi. Ne Fig.8-13, per shembull, ne kemi ndarjet te madhesise 4, 6, & dhe 13 te CPU-se, dhe 2 CPU te pashenuara. Me kalimin e kohes, numri dhe madhesia e cdo particioni do te ndryshoje ne varesi te procesit qe eshte apo qe u ekzekutua.

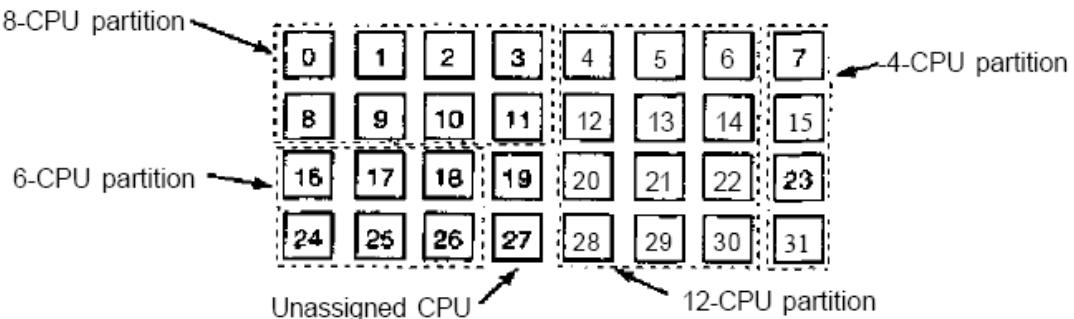


Figure 8-13. A sci of 32 CPUs split imc four partitions, with two CPUs available.

Periodikisht, duhet te kryhen dhe vendimet mbi skedulimin. Ne nje sistem me nje procesor, kryerja e punes me te vogel eshte algoritmi me i njohur per skedulimet qe jane ne radhe. Ne menyre analoge algoritmat per multiprocesorin duhet te zgjedhin procesin qe i nevojitet numri me i vogel i cikleve te CPU-se, i cili eshte ai qe ka CPU-count * runtime me te vogel. Megjithate, ne praktike ky informacion disponohet rralle, keshtu qe algoritmat jane te veshtire per tu kryer. Ne fakt, studimet kane treguar qe, ti sherbesh te parit qe vjen eshte e veshtire.

Ne kete model te thjesht ndarjesh, nje procesh vetem pyet per disa numra CPU-she, dhe i merr ato te gjitha ose pret sa te jene te lira. Nje menyre tjeter eshte qe proceset ne menyre aktive te menaxhojne shkallen e paralelizmit. Nje menyre per te menaxhuar paralelizmin eshte qe te kemi nje server qendor qe mban gjurmet se cili process eshte duke u egzekutuar dhe cili do te egzekutohet, dhe sa eshte minimumi dhe maksimumi i CPU-ve qe kerkohen. Periodikisht, secila CPU pyet serverin qendor se sa CPU mund te perdori. Pastaj rregullon numrin e proceseve duke i shtuar ose pakesuar, per ti njehsuar me ato qe jane te lira. Per shembull, nje Web server mund te kete 1, 2, 5, 10, 20 ose nje numer tjeter threadesh qe jane duke vepruar ne paralel. Ne qofte se aktualisht jane duke vepruar 10 threads dhe papritur ka nje kerkese me te madhe per CPU dhe u eshte thene qe te leshohen vetem 5, kur 5 threads e tjere mbarojne punen e tyre aktuale, atyre u thuhet qe te dalin jashte ne vend qe tu jepet pune e re. Kjo skeme lejon qe ndarja me pjese ne menyre dinamike te perballoje punen qe ngarkohet me mire se nje sistem fiks, sic eshte treguar ne Fig.8-13.

Gang Scheduling

Nje avantazh i qarte i space sharing eshte eleminimi i multiprogramimit, i cili eleminon ne kontekst vartesine nga switching. Megjithate, nje disavantazh po i njellojte eshte koha qe humbet CPU-ja kur bllokohet dhe nuk ka asnje pune tjeter per te bere, derisa te jete gati perseri. Vazhdimeshit, njerezit kane bere perpjekje per te gjetur algoritma qe te skedulojne menjehere, dhe kohen dhe hapesiren, vecanerisht per procese qe krijojne multiple threads, te cilet zakonisht duhet te komunikojne me njeri-tjetrin.

Per te pare se cfare problemesh mund te dalin kur thread-et e nje procesi skedulohen ne menyre te pavarur, marrim ne shqyrtim nje sistem me threads A_f dhe A , qe i perkasin procesit A , dhe threads B_0 dhe B , qe i perkasin procesit B . thread-et A_0 dhe B_0 , jane timeshared ne CPU—ne 0. Thread-et A dhe B jane timeshared ne CPU-ne 1. Thread-et A_0 dhe A nevojitet qe te komunikojne shpesh. Menyra e komunikimit eshte qe A_0 i con nje mesazh A_y , nepermjet A_t , pastaj kthen mbrapsh nje per gjigje tek A_0 , duke u ndjekur nga nje tjeter sekuence e tille. Le te supozojme se, A_a dhe B_x fillojne te paret, sic eshte treguar ne Fig. 8-14.

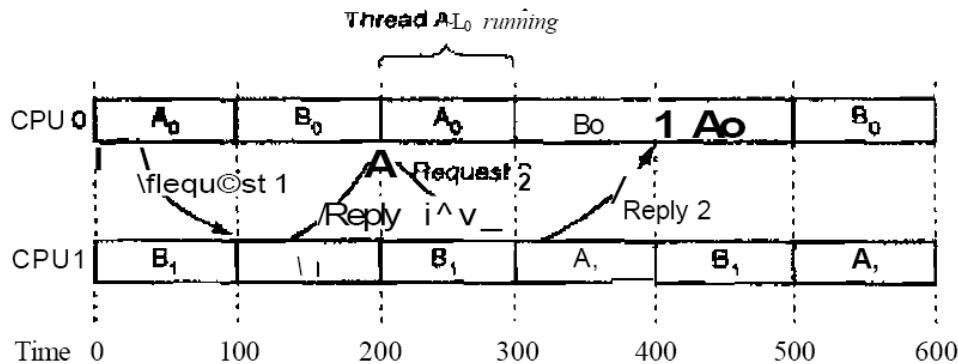


Figure 8-14. Communication between two threads belonging to process A that are running out of phase.

Ne pjesen e kohes 0, A_0 i dergon A_1 nje kerkese, por A_y nuk e gjen ate derisa ai te veproje ne pjesen e kohes 1 duke startuar me 100 msec. Ai dergon menjehere nje per gjigje, por A_0 nuk merr per gjigje derisa ai te filloje te veproje perseri me 200 msec. Rezultati tjeter eshte nje sekuence pyetje-per gjigjesh ne cdo 200 msec, gje e cila nuk eshte e mire.

Zgjidhja e ketij problemi eshte gang scheduling, i cili eshte nje vete rritje e co-scheduling, Gang scheduling perbehet nga tre pjesa:

1. Grupet e threade-ve qe kane lidhje skedulohen si nje pjese e tere, nje gang.
2. Te gjithe pjesetaret e nje gang-u veprojne njeheresh, ne CPU te ndryshme timeshared.
3. Te gjithe pjesetaret e gang fillojne dhe mbarojne ne te njejten kohe se bashku.

Truku per te bere gang te skeduloje te gjithe punen eshte se te gjithe CPU-te Jane skeduluar ne menyre sinkrone. Kjo do te thote qe koha eshte ndare ne quante diskrete sic eshte treguar ne Fig. 8-14. ne fillim te cdo kunati , te gjitha CPU-te Jane riskeduluar dhe

nga nje thread i ri starton ne secilen prej tyre. Me startimin e quantumit qe pason, ndodh nje tjeter skedulim. Nderkohe, asnje skedulim nuk ka perfunduar. Ne se nje thread bllokohet, CPU-ja e tij qendron idle deri ne perfundim te quantumit,

Nje shembull se si punon nje skedulim gang eshte dhene ne Fig. 8-15. Aty ne kemi nje multi processor me gjashte CPU, te cilat perdoren nga pese procesore, nga A tek E, me nje numer total prej 24 threads te gatshem. Gjate kohes se slotit 0, threads A_i deri tek A, jane skeduluar dhe ekzekutuar. Gjate kohes se slotit 1, threads $B_0, B_1, B_2, C_0, C_1, C_2$, jane skeduluar dhe ekzekutuar. Gjate kohes se slotit 2, $Z > s$, pese threads dhe E^0 fillojne te egzekutohen. Gjashte threads qe kane mbetur i perkasin procesit E, i cili vepron ne slotin 3. Me pas cikli perseritet, fillon me slotin 4, njesoj si sloti 0 dhe keshtu me rradhe.

		CPU					
		0	1	2	3	4	5
Time slot	0	A_0	A_1	A_2	A_3	A_4	A_5
	1	B_0	B_1	B_2	C_0	C_1	C_2
	2	D_0	D_1	D_2	D_3	D_4	E_0
	3	E_1	E_2	E_3	E_4	E_5	E_6
	4	A_0	A_1	A_2	A_3	A_4	A_5
	5	B_0	B_1	B_2	C_0	C_1	C_2
	6	D_0	D_1	D_2	D_3	D_4	E_0
	7	E_1	E_2	E_3	E_4	E_5	E_6

Figure 8-15. Gang scheduling.

Ideja e skedulimit gang eshte qe te gjithe threads e nje procesi te veprojne njekohesisht, keshtu ne se nje nga ata con nje kerkese tek nje tjeter, do te marri nje mesazh pothuaj menjehere dhe gjithashtu do te jete ne gjendje te perqigje po menjehere. Ne Fig 8-15, derisa te gjitha threads A veprojne njekohesisht, gjate nje quantumi, ato mund te cojne dhe te marrin nje numer te madhe mesazhesh ne nje quantum, gje qe eleminon problemin e Fig. 8-14.

8.2 MULTIKOMPJUTERAT

Multiprocesoret janë shumë te përhapur dhe shumë “térheqës” në faktin që ata ofrojnë një model të thjeshtë komunikimi: të gjitha CPU-te ndajnë një memorje të perbashkët. Proçeset mund të shkruajnë mesazh drejt memorjes i cili mund te lexohet nga proçese të tjera. Sinkronizimi mund të bëhet duke perdorur mutex-et, semaforet, monitored dhe teknika te tjera te mirë percaktuara. Disavantazhi i tyre me i madh është që multiproçesoret e medhenj paraqesin vështirësi të medha ne ndertim, e cila rrit dhe koston e tyre.

Per të perballuar keto probleme është bërë një kerkim shume i madh ne **multi komjuterat**, te cilet jane CPU *tightly-coupled* qe nuk ndajne memorje. Secili prej tyre ka memorjen e vet sic shihet edhe ne fig. 8.1(b). Keto sisteme njihen edhe me emra te tjere si **kompjutera cluster** dhe **COWS (Clusters of Workstations)**.

Multikomjuterat janë shumë të thjeshtë per tu ndertuar, sepse komponenti baze eshte thjesht një PC stripped-down me adicionin e një karte të nderfaqes se rrjetit. Sigurisht, sekreti per te arritur një performance të larte eshte të projektosh një rrjet nderlidhes dhe një karte nderfaqeje ne një mënyre shumë eficiente. Kjo situate eshte analoge me ndertimin e një memorjeje qe ndahet ose sic quhet ndryshe memorje e “share-uar” ne një multiproçesor. Sidoqoftë, qellimi eshte të dergosh mesazhe ne një rend prej milisekondash, sesa te aksesosh memorjen ne një kohe te rendit *nanosec*, pra kjo gje eshte me thjeshte dhe me lire per tu arritur.

Ne leksionet qe vijojne, ne do të hedhim një veshtrim te shkurter persa i perket hardware te multikompjuterave, vecanerisht hardware te nderlidhjes ose interkoneksionit. Me pas ne do te shohim software, duke filluar me software e rendit te ulet te komunikimit, per te vazhduar me tej me atë të rendit të larte. Do të hedhim veshtrimin gjithashtu edhe se si memorja e share-uar mund te kapet edhe nga sisteme qe nuk e zoterojne ate. Per ta myllur ne do ekzaminojme skedulimin dhe balancimin e ngarkeses (load balancing).

8.2.1 HARDWARE I MULTIKOMPJUTERAVE

Nyja baze e një multikompjuteri konsiston ne: CPU, memorja, nderfaqe rrjeti dhe ndonjehere perfshihet edhe hard disk. Nyja mund te paketohet ne një PC standarte, por pershtatesi grafik (adapteri), tastiera dhe mausi mungojne ne këtë rast. Ne disa raste, PC permban një multiproçesor board 2-way ose 4-way ne vend te një CPU-je të vetme. Por per thjeshtesi ne do te supozojme se çdo nyje ka vetëm një CPU. Multikompjuteri formohet nga organizimi i qindra ose mijera nyjeve së bashku. Me poshte do te permendim menyren se si hardware eshte organizuar.

TEKNOLOGJIA E NDERLIDHJES

Cdo nyje ka një karte nderfaqeje rrjeti e shoqeruar me një ose dy kabllo (fibra) qe dalin jashte saj. Keto kabllo lidhin nyjet me switchet. Ne një sistem të vogel mund te kete një switch ku jane lidhur të gjitha nyjet sipas topologjise STAR si ne Fig 8-16(a). Kjo e fundit aplikohet ne rrjetet Ethernet.

Si alternative e një projektimi switch single, nyjet mund te formojne një unaze, me dy kabllo qe dalin jashte kartes se nderfaqes se rrjetit, ku njera del nga e majta dhe tjetra nga e djathta, sic shihet ne Fig 8-16(b). Ne ket lloj topologjje nuk nevojiten switch-e dhe kjo eshte arsyesa pse nuk Jane paraqitur ne figure.

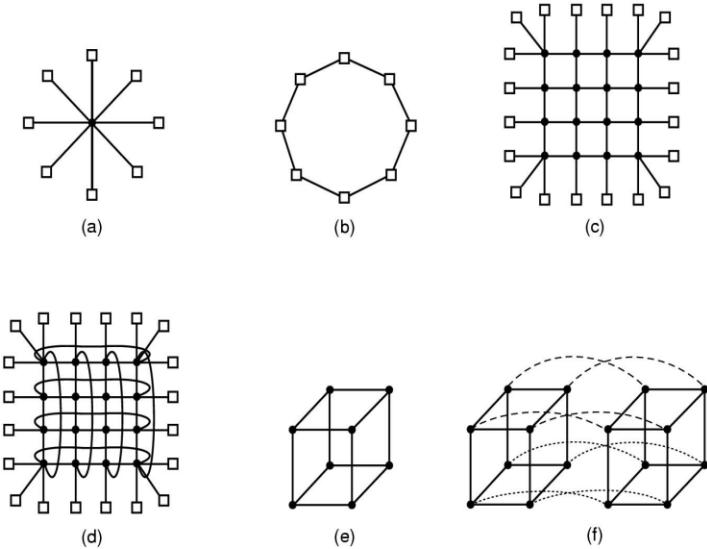


Figure 8-16. Topologji te ndryshme nderlidhjeje.(a) Nje switch i vetem(single switch).(b) Unaza.(d) Nje torus double.(e) Kubi.(f) Nje hypercube 4D.

Grid ose **mesh** është nje projektim dydimensional i cili perdoret ne shume sisteme komerciale, Fig 8-16(c). Karakterizohet nga një rregullsi shume e lartë dhe nje lethesi shume e madhe per tu rritur në permasa. Ka nje **diameter** që nenkupton rrugen me te gjate ndermjet dy nyjeve dhe rritja e te cilit eshte ne relacion me numrin e nyjeve si vijon: rrenja katrore e numri te nyjeve. Nje variant i grid eshte **double torus** si ne Fig 8.16(d), i cili eshte një grid me ane te lidhura. Jo vetem qe eshte me tolerant ndaj gabimeve se gridi, por edhe diametri eshte më i shkurter sepse qoshet e kunderta mund te komunikojne vetem në dy kercime.

Kubi i Fig 8.16(e) eshte nje topologji e rregullt tre-dimensionale. Ne kemi ilustruar një kub te permasave $2 \times 2 \times 2$, por ne per gjithesi ajo mund te jete e permasave $k \times k \times k$. Ne fig 8-16(f), ne kemi nje kub kater-dimensional i cili rrjedh nga ai tre-dimensional me nyjet koresponduese të lidhura. Ne mund te bejme një te permasave pese-dimensionale duke klonuar strukturen e Fig 8-16(f) dhe te lidhim nyjet koresponduese per te formuar nje bllok prej kater kubesh. Për të arritur ne dimensionin e gjashte, ne mund te replikoni bllokun e kater kubeve dhe te nderlidhim nyjet koresponduese, dhe keshtu me rradhe. Nje kub n -dimensional i formuar ne këtë menyre është quajtur hypercube. Shume kompjutera paralele e perdonin kete topologji sepse diametri rritet ne menyre lineare me dimensionalitetin. Me fjalë te tjera diametri eshte logaritmi me baze 2 i numrit te nyjeve, për shembull nje hypercube 10-dimensional ka 1024 nyje, por diametrin vetem 10 duke shkaktuar vonesa të medha.

Në multikompjutera perdoren dy skema switching. Ne te paren çdo mesazh ndahet fillimisht (ose nga perdoruesi i software ose nga nderfaqja e rrjetit) ne një “mase”me permasa maksimale e quajtur **pakete**. Skema e switching, e quajtur **store-and-forward packet switching**, konsiston në dergimin ose injektimin e paketes ne switchin e pare nga nyja burim e bordit te nderfaqes se rrjetit, sic e shohim ne Fig 8-17(a). Bitet vijne njehersh të gjithë dhe kur ka ardhur e gjithe paketa, ajo kopjohet ne switchin e rradhes gjate rruges sic tregonet ne Fig 8-17(b). Kur paketa vjen ne switchin e pozicionuar ne nyjen destinacion, si ne Fig 8-17(c), paketa kopjohet në ate nyjen e nderfaqes se rrjetit dhe aktualisht ne RAM-in e tij.

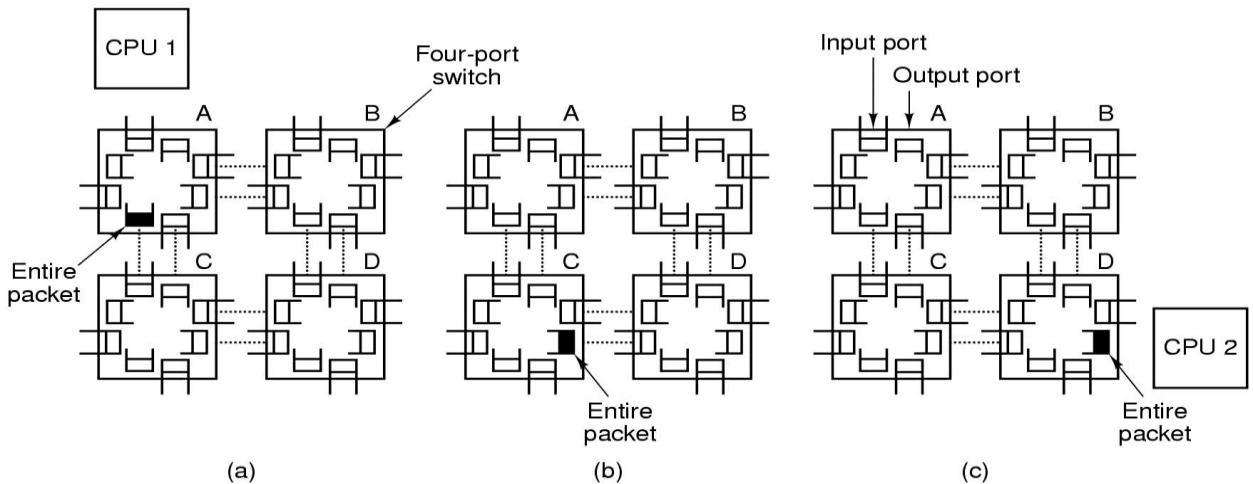


Figura 8-17. Store-and-forward packet switching.

Pavarsisht nga fakti qe store-forward switching është fleksibel dhe eficiente, ajo ka disavantazhin kryesor te rritjes se vonesave gjate rrjetit nderlidhes. Supozojme qe koha qe na duhet për te levizur nje pakete ne një kercim eshte T nsec. Vonesa per gjate ketij rrjeti koneksioni është $4T$ sepse paketa duhet te kopjohet kater here për te kaluar nga CPU 1 ne CPU 2 (drejt A-se, C-se, D-se dhe drejt destiancionit qe eshte CPU-ja), dhe per më teper nuk mund te behet asnje kopje derisa paketa e meparshme te kete perfunduar. Nje rruge eshte projektimi i nje rrjeti hibrid, duke adaptuar disa nga cilesite e circuit + packet switching. Per shembull, çdo pakete mund te ndahet logjikisht ne njesi me te vogla. Sa me shpejt qe njesia e pare te arrije te switchi, ajo mund te levizi drejt switchit tjeter, madje edhe me shpejt se mberritja e “bishtit” (tail) te paketes.

Alternativa tjeter switching, **circuit switching**, konsiston në paravendosjen e rruges (path) ndermjet switchit te pare dhe atij destinacion. Pra me tu vendosur rruga (path) bitet “fluturojne” nga burimi te destinacioni pa nderprerje. Nuk ka buffera ndermjetes ne switchet nderpreres me njeri tjetrin. **Circuit switching** kerkon fazën setup, e cila kerkon pak kohe, por me perfundimin e kesaj faze shpejtesia fillon e rritet. Me dergimin e paketes rruga qe ishte e paracktuar per te, tashme prishet. Variacioni i **circuit switching**, i quajtur **wormhole routing**, copeton çdo pakete ne subpaketa dhe lejon injektimin e subpaketes se pare, perpara ndertimit të plote të rruges (path).

NDERFAQET E RRJETIT

Te gjitha nyjet ne nje multikomputer kane një bord, të quajtur plug-in, i cili permbar lidhjen e nyjeve ne rrjetin nderlidhes ku puna e te cilit eshte te “mbaje” nje multikomputer kompakt. Menyra se si keto borde jane ndertuar dhe se si ato lidhen ne CPU-ne qendrore dhe Ram-in paraqesin implikacione te medha per sistemin operativ. Ne do te japid nje veshtrim të shkurter te ketyre çeshtjeve me poshte.

Virtualisht i gjithe bordi i nderfaqes se multikomjuterave permbar RAM-in i cili ruan paketat hyrese dhe dalese. Zakonisht, nje pakete dalese duhet te kopjohet ne bordin e nderfaqes se Ram-it perpara se ajo te dergohet ne switchin e pare. Arsyja pse eshte projektuar ne këtë menyre eshte per shkak se rrjetet nderlidhese gezojne vetine e

sinkronizimit, domethënë sa po te kete filluar transmetimi i paketes fluturimi i biteve behet me një shpejtesi konstante. Në qofte se paketa eshte ne memorjen kryesore, shpejtesia konsante e fluturimit te elektroneve nuk garantonohet per shkak te trafikut të të dhenave ne busin e memorjes. Këtë problem mund ta eliminojme duke perdorur një RAM te dedikuar per këtë pune. Ky projektim paraqitet ne Fig 8-18.

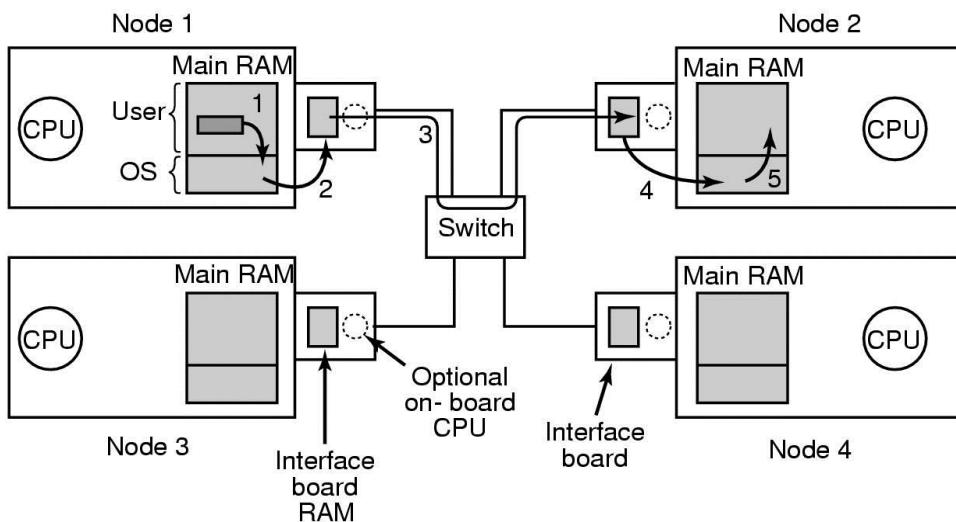


Figura 8-18.Pozicioni i bordit te nderfaqes se rrjetit.

I njejti problem ndodh edhe me paketat hyrese. Bitet “fluturojne” ne rrjet me një shpejtesi konstante dhe të larte. Në qofte se se bordi i nderfaqes se rrjetit nuk mund ti ruaje paketat ne kohe reale ndodh qe te dhenat mund te humbasin. Serish ketu paraqitet rrezikshmeria ne kalimin nga busi i sistemit (për shembull busi i PCI) ne memorjen kryesore. Perderisa bordi i rrjetit eshte vendosur ne menyre tipike brenda busit te CPU-se, kjo eshte lidhja e vetme qe ajo ka me me memorjen kryesore, keshtu qe konkurimi qe lind midis busit, diskut dhe cdo pajisjeje tjeter I/O eshte i pashmangshem. Eshte me e sigurte te ruash paketat qe vijne ne RAM-in privat te bordit te nderfaqes, se sa ti kopjosh me vone ato ne memorjen kryesore ose RAM-in kryesor.

Bordi i nderfaqes mund te kete një ose me shume kanale te DMA-se ose një CPU komplete ne bord. Kanalet DMA mund te kopojne paketa ndermjet bordit te nderfaqes dhe memorjes kryesore ne një shpejtesi te larte duke kerkuar transferimin e blloqeve ne busin e sistemit, cfare i thjeshton punet ne transferimin e disa fjalave pa i kerkuar busit te beje nje gje te tille per çdo fjalë te marre veçmas. Disa borde nderfaqesh kane një CPU te vetme dhe te plote ne to, duke perfshire edhe një ose me shume kanale DMA. Ky projektim do ti sjelle shume lehtesi bordit te rrjetit nga CPU-ja pasi ajo kryen mjaft detyra si:

Te realizoje një transmetim te qendrueshem (reliable), domethënë te mos lejoje humbjen e paketave.

Multicasting (te degjoje një pakete ne me shume se një drejtim). Te kujdeset per një mbrojtje sa me te mire ne një sistem me multiprocësore. Sidoqoftë te pasurit dy CPU do te thote qe nevojitet sinkronizimi per te shmangur kushtet e një gare, domethënë shtohet puna per sistemin operativ.

8.2.2 Low-level communication Software

Armiku me i madh ne performancen e komunikimit ne sistemet me multikomjutera eshte kopjimi i paketave. Ne rastim me te mire, do te kete vetem nje kopje nga RAM-i te bordi i nderfaques në nyjen burim, nje kopje nga bordi i nderfaques burim te ai destinacion dhe nje kopje nga atje ne RAM-in destinacion, pra ne total kemi tre kopje. Sidoqofte ne shume sisteme gjendja eshte edhe me keq. Ne vecanti ne se bordi i nderfaques eshte përkthyer (mapped) ne hapesiren viruale te adresave te kernelit dhe jo ne hapesiren virtuale te adresave te userit, procesi user mund te dergoje nje pakete duke perdorur vetem nje thirrje sistemi, qe i kalon kontrollin kernelit. Kerneli nga ana e tij mund te kopjoje paketat ne memorjen e tyre perkatese ne I/O, për shembull, te shmange *page faults* gjate transmetimit ne rrjet. Nga ana tjeter kerneli ne destinacion nuk di ku te vendose paketen pasi i duhet ta ekzaminoje njehere ate. Ne Fig 8-18 jane paraqitur pese hapat e kopjimit.

Në qofte se, performance dominohet nga kopjet qe behen nga dhe drejt RAM-it, kopjet ekstra nga dhe drejt kernelit mund te dyfishojne vone sen e tipit end-to-end dhe te presin bandwidth-in ne mes. Per te shmangur kete problem te performances shume multikompjutera perkthejne bordin e nderfaques ne menyre direkte ne hapesiren user per te lejuar procesin te vendose paketat direkt ne bord pa ndermjetesimin e kernelit. Edhe pse kjo perqasje ndihmon ne performance paraqet dy probleme.

Se pari, ç'ndodh ne se disa procese po ekzekutohen ne nyje dhe kerkojne te aksesojne rrjetin per dergimin e paketave? Cila prej tyre e merr bordin e nderfaques ne hapesiren e e tij te adresimit? Ne se vetem nje proces e merr bordin po te tjerat si i dergojne paketat? C'fare ndodh ne se bordi eshte perkthyer (mapped) ne hapesiren virtuale te adresimit te A-se dhe nje pakete vjen per B-ne, vecanerisht ne rastin kur A-ja dhe B-ja kane posesues te ndryshem ku asnjeri prej tyre nuk do te vije ne ndihme te njeri tjetrit?

Nje zgjidhje eshte te perkthyerit (mapping) ne të gjitha proceset qe kane nevoje per te, por duhet mbajtur parasysh shmangia e kushteve te konkurencës. Për shembull ne se A-ja dhe B-ja kerkojne te njejtin buffer ne bordin e nderfaques, ndodh nje perplasje apo jo? Ketu nevojitet nje mekanizem sinkronizimi, për shembull mutexet, por keto funksionojne vetem ne rastet kur proceset bashkeveprojne me njeri tjetrin. Si perfundim mund te themi qe perkthimi i bordit te nderfaques ne hapesiren user funksionon vetem ne rastin kur ekzekutohet vetem nje proces ne cdo nyje, perkundrazi merren masa paraprake.

Se dyti eshte qe kerneli mund te kerkoje vete aksesin e rrjetit nderlidhes per vete, për shembull, te aksesoje sistemin e file-ve ne nje nyje te larget. Qe kerneli te aksesoje edhe nderfaqen e rrjetit, ku kjo e fundit perdoret nga cdo perdorues nuk eshte nje ide e mire, madje edhe ne sistemet *timesharing*. Supozojme qe ndersa bordi eshte perkthyer ne hapesiren user, vjen nje pakete e tipit kernel. Ose marrim rastin tjeter kur procesi user ka derguar nje pakete ne nje makine te larget duke pretenduar se eshte e tipit kernel. Si perfundim themi se projektimi me i thjeshte duhet te kete dy borde nderfaqesh te rrjetit, nje e perkthyer ne hapesiren user per te dhena aplikative dhe tjetra është e perkthyer ne kernel ku perdoret nga sistemi operativ. Shume multikompjutera aplikojne këtë lloj modeli.

Nyja ne nderfaqen e komunikimit te rrjetit

Nje çeshtje tjeter eshte menyra se si ti cojme paketat ne bordin e nderfaques. Menyra me e shpejte eshte te perdoresh çipin e DMA-se ne bord per te kopjuar paketat nga RAM-i. Problemi qe ekziston me kete lloj perqasjeje lidhet me dy vecorite e DMA-se, domethënë ajo perdon adresat fizike dhe ekzekutohet ne menyre te pavarur nga CPU-ja. Per me teper, edhe pse nje proces user mund ta dije me siguri adresen virtuale te cdo pakete qe ajo mund te dergoje, por ne per gjithesi ajo nuk e di adresen fizike. Te besh nje thirrje sistemi per te bere perkthimin virtual-fizik eshte jo shume e deshirueshme pasi qellimi yne ishte të shmangnim thirrjet e sistemit per çdo pakete te derguar.

Ne se sistemi operativ vendos te zevendesoje nje faqe, ndersa cipi i DMA-se kopjon nje pakete nga kjo e fundit, do te ndodhe transmetimi i te dhenave te gabuara. Per me teper ne se sistemi operativ zevendeson nje faqe, ndersa cipi i DMA-se po kopjon nje pakete hyrese ne te, ne kete rast jo vetem qe paketa hyrese do te humbase por edhe faqja e memorjes do te shkaterrohet.

Keto probleme mund te shmanget duke pasur thirrjet e sistemit per pin dhe unpin te faqeve ne memorje. Sidoqofte, nevoja per te aktivizuar thirrjet e sistemit per pin-in dhe unpin-in e faqeve qe permbajne paketat dalese, kushton shume. Në qofte se paketat jane te vogla, për shembull 64 bytes ose me pak, ngarkesa per pin-in dhe unpin-in e çdo bufferi eshte e ndaluar. Per paketa te medha, 1 KB ose me shume, mund te bejme tolerime. Per permasa te ndermjetme ajo cfare mund te themi eshte qe kemi vartesi ndaj hardware.

Teorikisht, ndodh i njejt problem me DMA-ne nga disku ose pajisje të tjera, por duke pasur parasysh faktin qe keto te fundit vendosen nga sistemi operativ ne bufferat e kernelit, eshte shume e thjeshte per sistemin te shmange paging e bufferave. Problemi qe qendron ketu eshte qe perdonuesi vendos dhe menaxhon DMA-ne, dhe nderkohe sistemi operativ nuk e di qe zhvendosja e faqes mund te jete fatale, ndersa per pajisjet I/O ai eshte ne dijeni të kesaj gjeje pasi ato sigurojne vete fillezen e tyre. Arsyja pse perdonim bufferat e kernelit per diskun I/O dhe jo per komunikimin e multiprocesoreve, eshte qe nje vonese prej 20 *microsec* eshte e tolerueshme ne rastin e nje disku por jo te nje komunikimi proçes-proçes.

Problemi i DMA-se mund te shmanget duke e qene procesi user ai qe starton pin e pare ne faqe dhe pyet per adresen e tij fizike. Paketat dalese kopjohen nje here atje dhe me pas ne nderfaqen e rrjetit, por kjo kopje ekstra eshte po aq e keqe si te kopjosh ne kernel.

Duke pasur parasysh keto arsy, te pedoresh I/O te programuar nga dhe drejt boardit te nderfaques eshte rruga më e sigurte, perderisa cdo page fault e hasur konsiderohet thjesht si page fault e CPU-se dhe menaxhohet me menyren e zakonshme nga sistemi operativ. Kur ndodh nje page fault, cikli i kopjimit ndepritet menjehere dhe vazhdon ne gjendjen e pritis derisa sistemi operativ te merret me këtë problem. Nje skeme me e sofistikuar eshte të perdoresh I/O e programuar per paketat e vogla dhe DMA-ne ne pin-in dhe unpin-in per paketa me te medha.

Në qofte se se boardet e nderfaques se rrjetit kane CPU-ne e tyre keto te fundit e rritin shume shpejtesine. Sidoqofte, duhet pasur kujdes ne shmangien e kushteve te konkurencës midis CPU-se qendrore dhe CPU-ve on-board. Nje rruge per te shmanjur keto probleme ilustrohet ne Fig 8-19, ne te cilën fokusohemi ne nyjen 1 qe dergon paketa dhe nyja 2 qe i merr ato, por jo domosdoshmerisht te kene lidhje midis tyre. Calesi i

sinkronizimit te struktura se te dhenave per derguesin eshte send ring (unaza e dergeses); per marresin eshte receive ring (unaza marrese). Te gjitha nyjet jane te pajisura me te dyja llojet e unazave duke pasur parasysh faktin qe ato edhe dergojne edhe marrin. Cdo pakete ka hapesire per n-paketa. Gjithashtu, kemi edhe një bitmap per cdo unaze me n-bitë, te cilat mund te jene ose te ndara ose te integruar ne unaza, ku jepin informacion se kush slot te unazes jane te vlefshme.

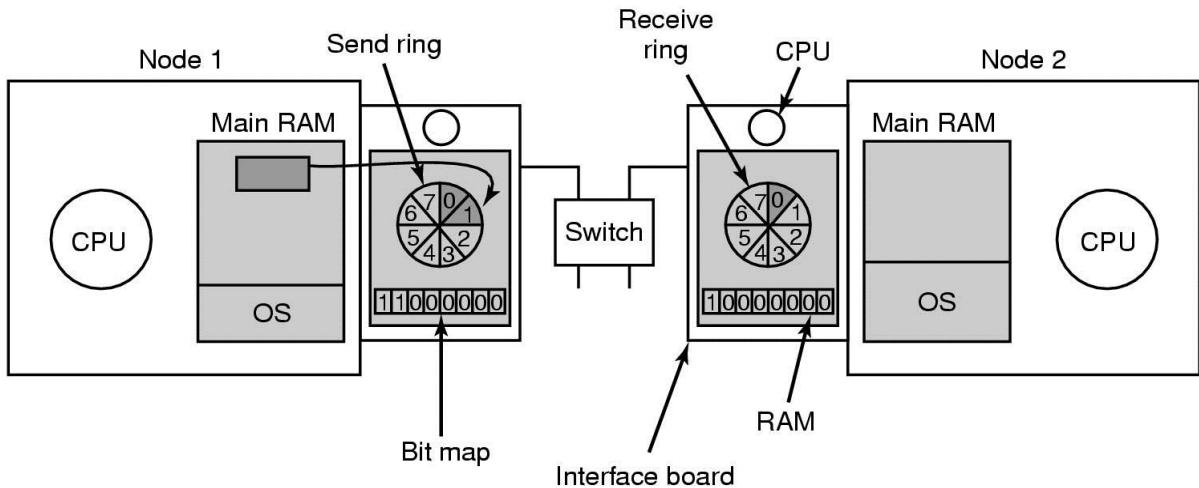


Figura 8-19. Perdorimi i unazave send&receive per te koordinuar CPU-ne kryesore me CPU-ne onboard.

Kur derguesi ka një pakete te re per te derguar ai kontrollon njehere per te pare ne se ka ndonje slot te disponueshem ne unazën derguese. Ne se nuk ka te tille, atehere ai duhet te prese ne menyre qe te parandaloje mbivendosjen. Ne se ka një slot te lire, ai e kopjon paketen ne slotin tjeter te disponueshem, pasi kjo pune ka perfunduar ai vendos bitin korespondues ne bitmap. Kur CPU on-board eshte e lire, ajo kontrollon unazën derguese. Në qofte se ajo permbo paketa, merr paketen me te gjate dhe e transmeton. Kur kjo gje behet ai pastron bitin korespondues ne bitmap. CPU-ja qendrore eshte ajo qe vendos bitet dhe CPU-ja on-board ajo qe i fshin. Ne këtë rast nuk kemi kushte konkurence. Unaza marrese punon ne nje menyre tjeter, me nje CPU on-board qe vendos nje bit per te lajmeruar ardhjen e paketes dhe CPU-ne qendrore, qe fshin bitin per te lajmeruar kopjimin e paketes dhe lirimin e bufferit.

Kjo skeme mund te funksionoje edhe pa pasur nevojen e programimit te I/O nga ana e CPU-se qendrore. Ne kete rast unaza derguese nuk permbo paketen por një pointer te paketes ne memorjen kryesore (RAM-in kryesor). Kur CPU on-board eshte gati te transmetoje paketen, ai e ngarkon paketen ne bordin e nderfaqes duke perdorur I/O e programuar ose DMA-ne. Ne te dyja rastet kjo perqasje funksionon vetem kur faqja qe permbo paketen eshte e njohur per tu pinuar.

8.2.3 Software i komunikimit te nivelit te ulet

Proceset ne CPU te ndryshme ne nje multikompjuter komunikojne duke derguar mesazhe njeri tjetrit. Ne pamje te pare ky kalim mesazhesh ndermjet proceseve eshte i ekspozuar

ndaj proceseve user. Me fjalë të tjera sistemi operativ siguron dergimin dhe marrjen e ketyre mesazheve dhe ato që realizojnë thirrjet perkatese janë procedurat e librarise. Ne një forme me të sofistikuar mesazhi aktual dergohet në një menyre të fshehte për perdoruesin duke krijuar idene e një thirrjeje procedure, ne vend të një komunikimi ne largesi. Ne do ti shohim keto dy metoda me poshtë.

Dergo dhe merr

Ne menyre të permblledhur sherbimet e siguruara të komunikimit mund të reduktohen në dy thirrje librarish (library call), një për dergimin e mesazheve dhe një për marrjen e tyre. Thirrja për dergimin e mesazhit eshte:

send (dest, &mptr);

dhe ajo për marrjen e mesazhit:

receive (addr, &mptr);

Formerit (I meparshmi), dergon mesazhin e pointuar nga *mptr* drejt një procesi i identifikuar nga *dest* dhe shkakton bllokimin e thirresit derisa mesazhi të dergohet. Latteri (I dyti) shkakton bllokimin e thirresit derisa te vije mesazhi. Kur ndodh kjo mesazhi kopjohet në buferin e e pointuar nga *mptr* dhe ne kete rast thirresi del nga gjendja e bllokuar. Parametri *addr* specifikon adresen ne të cilën marresi po degjon. Gjithsesi ka mjaft variante të ketyre dy procedurave dhe parametrave të tyre.

Një çeshtje ka të bëje me menyrën se si kryhet adresimi. Perderisa multikompjuterat janë statike me një numer fiks CPU-sh, menyrat e lehta per të bere adresimin, eshte të besh një *addr* me dy pjesë, një për numrin e CPU-së dhe tjetra për numrin e procesit ne CPU-në e adresuar. Ne kete menyre cdo CPU mund të menaxhoje adresat e veta pa pasur konflikte.

Thirrjet bllokuese kundrejt thirrjeve jo bllokuese

Thirrjet e pershkruara me siper quhen thirrje bllokuese (ndonjehere quhen edhe thirrje sinkronizuese). Kur një proces therret *send*, ai specifikon destinacionin dhe një buffer për ta derguar në atë destinacion. Gjate kohës që dergohet mesazhi, procesi dergues qendron në gjendje të bllokuar. Instruksioni që pason thirrjen *send* nuk ekzekutohet derisa ka perfunduar komplet dergimi i mesazhit, sic shihet në Fig 8-20 (a). Te njejtën gjë themi edhe për thirrjen *receive* e cila nuk kthen kontroll derisa mesazhi të jetë marre dhe eshte futur në bufferin e mesazhit, i pointuar nga parametri. Edhe ne kete rast procesi qendron në gjendjen e pezulluar derisa mesazhi mberrin, edhe pse mund të vazhdoje për ore të tera. Ne disa sisteme marresi mund të specifikoje edhe derguesin që deshiron.

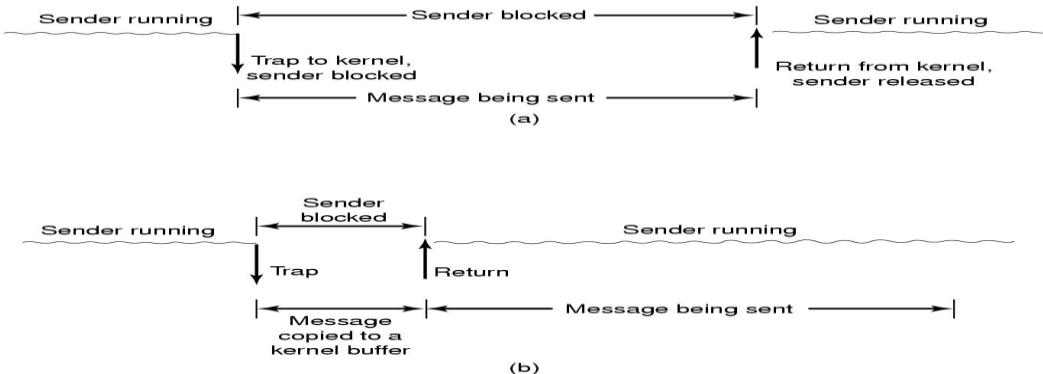


Figura 8-20.(a) Nje thirrje send bllokuese.(b) Nje thirrje send jo bllokuese.

Nje alternative e thirrjeve bllokuese jane thirrjet jo bllokuese (nonblocking ose asynchronous calls). Pra ne se send eshte jo bllokuese, ajo i kalon kontrollin thirresit menjehere, perpara dergimit te mesazhit. Avantazhi i kesaj skeme eshte se procesi dergues mund te vazhdoje llogaritjet ne paralel me transmetimin e mesazheve. Zgjidhja ndermjet primitivave bllokuese dhe atyre jo bllokuese behet nga projektuesit e sistemit, pamvaresisht se ne disa sisteme te dyja llojet jane te disponueshem dhe perdoruesit jane te lire te zgjedhin.

Sidoqofte avantazhi qe kemi me primitivat jo bllokuese ne lidhje me performancen permbyset nga nje disavantazh i madh: derguesi nuk mund te modifikoje bufferin e mesazhit derisa mesazhi te jete derguar. Mund te kete pasoja te renda ne rastin kur procesi mbishkruan mesazhin gjate transmetimit. Me keq akoma, procesi dergues nuk e ka idene se kur kryhet transmetimi, ne kete menyre ai nuk e di se kur eshte me i sigurte riperdorimi i bufferit.

Ketu mund te permendim tre zgjidhje. E para ka te beje me kernelin, pra ai kopjon mesazhin ne nje buffer te tij te brendshem dhe me pas lejon procesin te vazhdoje, sic tregohet ne Fig 8-20(b). Nga pikepamja e derguesit kjo eshte njesoj si thirrja bllokuese: sa me shpejt derguesi te marre kontrollin ai eshte i lire te riperdore bufferin. Disavantazhi i kesaj metode eshte se prape derguesi nuk mund te beje gje derisa mesazhi te jete derguar perfundimisht. Ne shume nderfaqe rrjeti, mesazhi duhet te kopjohet ne nje buffer transmetimi hardware-rik ne cdo rast, keshtu qe kopja e pare shperdorohet. Dhe kopja qe behet me vone mund te ndikoje ne uljen e performances.

Zgjidhja e dyte ka te beje me nderprerjen e derguesit kur dergimi i mesazhit ka perfunduar, duke lajmeruar ate se bufferi tashme eshte i disponueshem. Ketu nuk nevojitet asnje kopje, c'ka redukton kohen. Por interruptet e nivelit te ulet e bejne programimin e veshtire dhe subjekt kushtesh konkurence çfare i ben ata jo shume produktiv dhe te pamundur per t'iu nenshtuar debugger-it.

Zgjidhja e trete ka te beje me percaktimin e bufferit si read-only derisa te perfundoje dergimi i mesazhit. Ne rast se bufferi eshte riperdorur perpara se mesazhi te dergohet, behet nje kopje. Problemi qe lind me kete lloj zgjidhje eshte se, edhe pse buferi eshte izoluar ne faqen e tij, kopjimi do te ndodhe edhe ne se shkruhet ne variablat me te aferta. Gjithashtu ketu duhet edhe pak administrim pasi dergimi i mesazhit tani perfshin edhe statusin read/write te faqes. Perfundimisht mund te themi se faqja duhet rishkruar prape.

Persa i perket anes se dergimit kemi keto zgjidhje:

Blocking send (CPU eshte e disponueshme gjate transmetimit te mesazhit).

Nonblocking send with copy (koha e CPU-se shperodrohet per kopjet ekstra).

Non blocking send with interrupt (programimi eshte shume i veshtire).

Copy on write (lind nevoja e kopjeve ekstra).

Ne kushte normale preferohet me shume zgjidhja e pare, sidomos ne rastin kur kemi shume threade, ku ne cdo rast ndersa nje thread eshte i bllokuar gjate dergimit, vazhdon puna e threadeve te tjere. Gjithashtu nuk lind nevoja e menaxhimit te ndonje bufferi te kernelit. Sic mund ta shohim nga krahasimi qe mund ti bejme dy Fig 8-20(a) dhe 8-20(b) Dergimi i mesazhit mund te behet me shpejt ne qofte se nuk ka nevoje per kopje.

Duke pasur parasysh qe send mund te jete bllokuese ose jo, te njejten gje mund te themi edhe per receive. Nje thirrje bllokuese mund te pezulloje thirresin derisa te perfundoje dergimi i mesazhit. Per shume threade te disponueshem kjo eshte nje perqasje e thjeshte. Nje menyre tjeter do te ishte, nje receive jo bllokuese qe thjesht i tregon bufferit vendodhjen e bufferit dhe i kthen kontrollin derguesit menjehere. Nje interrupt perdoret per te sinjalizuar mberritjen e mesazhit. Por interruptet jane shume te veshtire per tu programuar dhe shume te ngadalte, keshtu qe do ishte me mire te mendonim nje menyre tjeter per te sinjalizuar mberritjen e mesazhit. Si alternative permendim proceduren *poll* e cila jep informacion ne lidhje me mesazhet qe jane ne pritje. Ne qofte se kemi kete gjendje, atehere thirresi therret *get-message* e cila kthen mesazhin e pare te kthyer.

Nje opson tjeter eshte skema ne te cilin mberritja e nje mesazhi shkakton krijimin spontan te nje threadi te ri ne hapesiren e adresimit te procesit marres. Nje thread i ri i krijuar ne kete menyre quhet pop-up. Ai ekzekuton nje procedure te specifikuar me pare dhe parametri i tij eshte nje pointer te mesazhi pasardhes. Ne perfundim te procesimit te mesazhit, ai shkaterohet automatikisht.

Nje variant i kesaj skeme eshte te ekzekutosh kodin e marresit me interruptet pa pasur nevojen e krijimit te threadeve pop-up. Per ta bere kete skeme me te thjeshte, mesazhi nga ana e tij permban adresen e menaxhuesit te interrupteve, pra kur vjen mesazhi menaxhuesi (handler) mund te thirret me pak instruksione. Fitorja me e madhe ketu qendron se tashme nuk ka nevoje me per kopjime. Menaxhuesi e merr mesazhin nga bordi i nderfaqes dhe e proceson ate me nje shpejtesi marramendese. Kjo skeme njihet me emrin **active messages**, e cila funksionon vetem atehere kur derguesi dhe marresi kane besim te njeri tjetri plotesisht.

8.2.4 Remote procedure call

Skema e siperpermendor ka nje te mete te madhe: modeli baze mbi te cilin eshte ndertuar i gjithe komunikimi eshte I/O. Procedurat send dhe receive perfshijne I/O dhe shumica e njerezve besojne se I/O eshte modeli i gabuar i programimit.

Birrel dhe Nelson sygjeruan se mund te lejohej thirrja e procedurave te lokalizuara ne CPU te tjera. Kur nje proces ne makinen 1 therret nje procedure ne makinen 2, procesi thirres ne makinen 1 eshte i pezulluar dhe ne makinen 2 fillon ekzekitimi i procedures perkatese. Informacioni nga thirresi te marresi, mund te transmetohet ne parametra dhe kthehet si nje rezultat procedure. Nderkohe per programuesin i gjithe procesi i transmetimit te mesazhit dhe pajisjet I/O jane te padukshme. Kjo teknike eshte quajtur **RPC** (Remote Procedure Call) dhe eshte baza e software ne shume multikompjutera. Tradicionalisht procedura thirrese quhet *client* ndersa ajo e thirrur quhet *server*.

Ideja e RPC-se eshte qe te beje qe nje thirrje e larget procedure te duket si lokale. Ne formen me te theshte, programi klient qe te beje nje thirrje te larget duhet te jetet i pajisur me nje procedure librarie e quajtur *client stub*, qe perfaqeson proceduren e serverit ne hapesirene e adresimit te klientit. Ne menyre te ngjashme mund te themi edhe per serverin i cili eshte i pajisur me proceduren e quajtur *server stub*. Keto procedura e fshehin faktin qe thirrja e procedures nga klienti te serveri eshte jo lokale.

Hapat qe e kryejne RPC tregohen ne Fig 8-21. Hapi 1 eshte thirrja e client stub nga ana e klientit. Kjo thirrje eshte nje thirrje lokale procedure ku parametrat jane te ruajtur ne stak si normalisht. Hapi 2 eshte paketimi i parametrave ne mesazh dhe aktivizimi i thirrjes se sistemit per dergimin e mesazhit nga client stub. Paketimi i paketave quhet **marshaling**. Hapi 3 ka te beje me dergimin e mesazhit nga makina client drejt asaj server, nepermjet kernelit. Hapi 4 ka te beje me kalimin e paketes hyrese ne server stub nepermjet serverit. Hapi 5 eshte thirrja e procedures se serverit nga server stub.

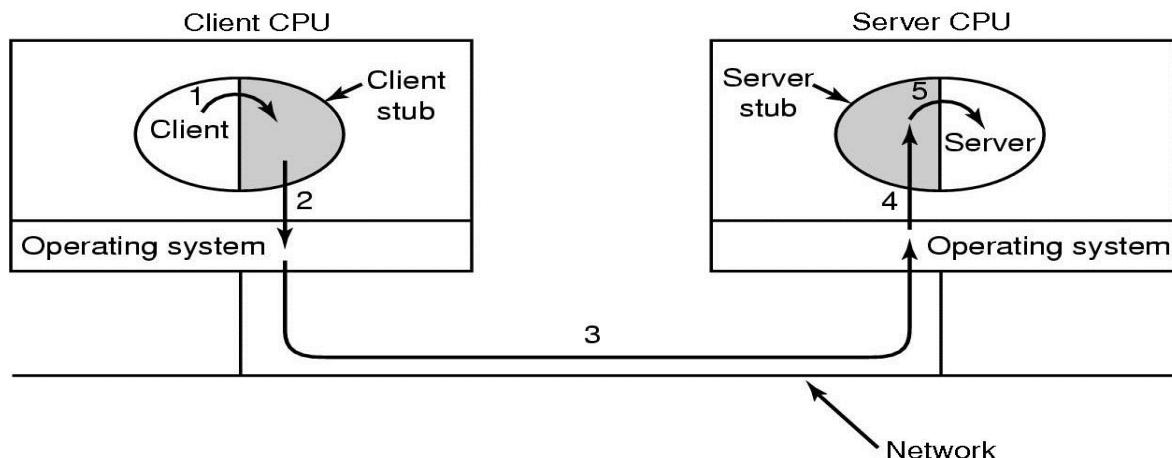


Figura 8-21.Hapat qe ndiqen ne berjen e remote procedure call.

Ajo çka do te dallojme ketu eshte se procedura e klientit, e shkruar nga perdoruesi, thjesht ben nje thirrje normale procedure te client stub e cila ka te njejten emer si te procedures se serverit. Perderisa procedura e klientit dhe client stub jane ne njejten hapesire adresimi atehere kalimi i parametrave behet ne menyren e zakonshme. Ne kete menyre ne realizojme komunikimim thjesht duke bere nje thirrje procedure ne vend te perdorimit te I/O dhe send&receive.

Ceshtjet e implementimit

Pavarisht favoreve te RPC ajo bart edhe mjaft te meta, ku me e madhja eshte perdorimi i parametrave pointer. Normalisht kalimi i pointerit ne nje procedure nuk paraqet ndonje veshtiresi. Procedura e thirrur mund te perdore pointerin ne te njejten menyre si thirresi, pasi te dyja ndajne te njejten hapesire adresimi. Ne RPC perdorimi i pointerave eshte i pamundur pasi klienti dhe serveri ndajne hapesira te ndryshme adresimi.

Problemi i dyte qe haset eshte qe ne gjuhet e programimit si C, ne mund te lejojme te shkruajme nje procedure e cila llogarit produktin e brendshem te dy vektoreve pa

specifikuar gjatesine e secilit. Secili prej tyre mund te perfundoje nepermjet nje vlore speciale e cila njihet vetem nga procedura thirrese dhe ajo e thirrur. Ne keto kushte eshte gati e pamundur per client stub te specifikoje parametra si për shembull, gjatesia e vektoreve.

Problemi i trete eshte pamundesa per te gjetur gjithmone tipet e parametrave, madje edhe kur na jepet nje specifikim formal ose kodi vete. Nje shembull do te ishte funks *printf* i cili mund te kete nje variacion tipesh, numrash dhe gjatesish te parametrave. Te therrasesh *printf* si procedure te larget (remote procedure) eshte e pamundur pasi gjuha C eshte permisive.

Keto probleme qe u permenden me siper nuk do te thone qe RPC eshte jo e perdonshme, por ato u permenden per te marre masat e duhura ne shambahien e problemeve ne rastin kur ajo gjen aplikim.

8.2.5 Distributed Shared Memory

Shume programues akoma preferojne nje model te memorjes se share-uar ne multikompjuterat. Madje mund te krijohet iluzioni i shared memory kur ajo nuk ekziston realisht duke perdonshme nje teknike te quajtur **DSM (Distributed Shared Memory)**. Ne kete model çdo faqe eshte lokalizuar ne nje nga memorjet e Fig 8-1.Cdo makine ka memorjen virtuale dhe PT e vet. Kur CPU-ja ben nje LOAD ose STORE ne nje faqe qe nuk e ka atehere ajo ekzekuton nje instruksion trap te sistemi operativ. Sistemi operativ me pas lokalizon faqen dhe i kerkon CPU-se te c'hardezoj faqen dhe ta dergoje ate ne rrjetin nderlidhes. Kur faqja mberrin ajo perkthehet dhe instruksioni qe kishte ndaluar tani mund te filloje. Ne fakt ajo c'ka ben sistemi operativ eshte te ngarkoje faqet nga nje RAM me i larget sesa nga disku lokal. Pra te perdonuesi krijohet iluzioni se kemi shared memory ose memorje te share-uar.

Diferencia qe eksiston midis shared memory aktuale dhe DSM ilustrohet ne Fig 8-22. Ne Fig 8-22 (a) shohim nje multiprocessor te vertete me shared memory fizike e cila eshte implementuar nga hardware-ri. Ne Fig 8-22(b) shohim DSM, e implementuar nga sistemi operativ. Ne Fig 8-22 (c) shohim nje forme tjeter te shared memory e implementuar nga shtresa te larta te software.

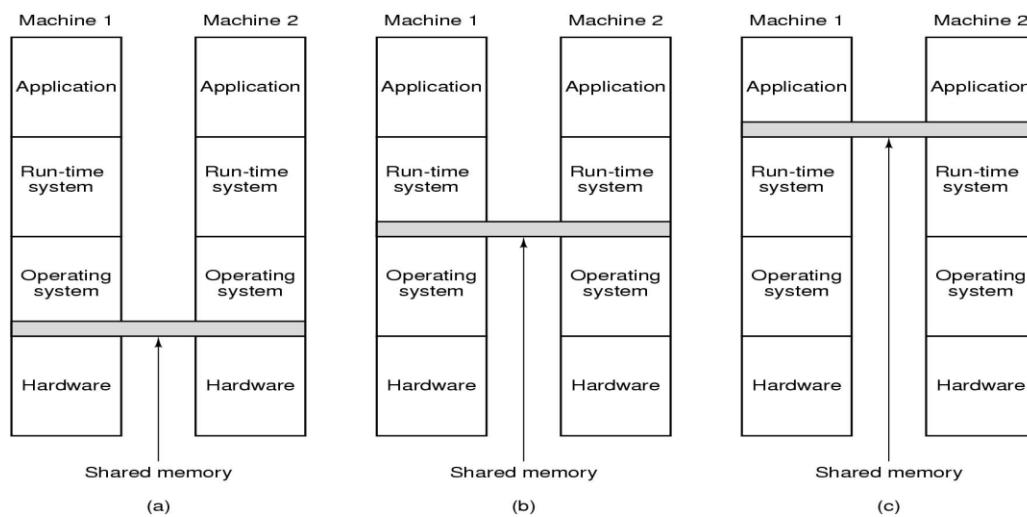


Figura 8-22.Shtresa te ndryshme ku mund te implementohet memorja e sharuar.(a) Hardueri.(b) Sistemi operativ.(c) Softueri i nivelit user.

Le te shohim me ne detaje se si funksionon DSM. Ne sistemet e ketij tipi, hapesira e adresimit eshte e ndare ne faqe te cilat jane shperndare ne te gjitha nyjet e sistemit. Kur CPU-ja i referohet nje adrese jo lokale, ndodh nje trap dhe software-i i DSM merr faqen qe permban adresen, dhe ben te rifilloje instruksioni qe kishte ndaluar, i cili pritet te perfundoje me sukses. Ky koncept ilustrohet ne Fig 8-23(a), per nje hapesire adresimi me 16 faqe dhe 4 nyje ku secila mund te mbaje 4 faqe.

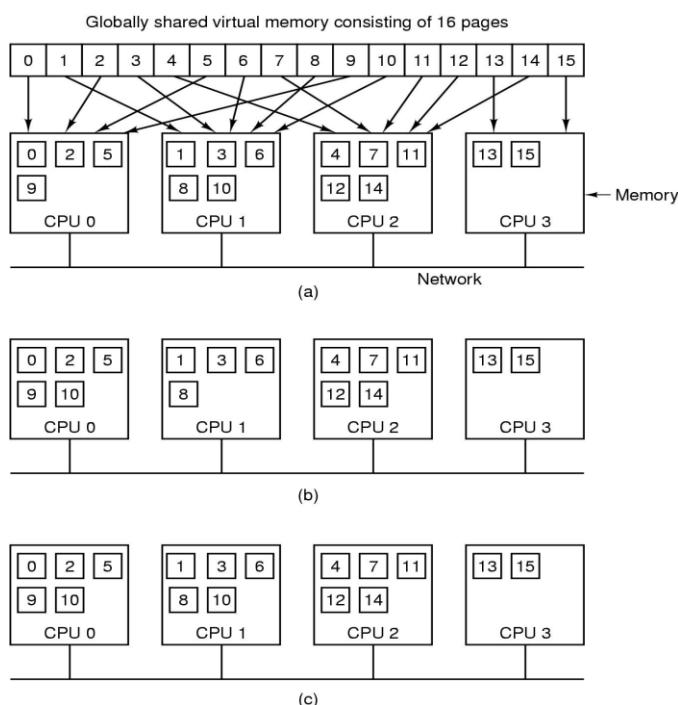


Figura 8-23.(a) Faqet e hapesires se adresimit te shperndara ndermjet kater makinave.(b) Situata pasi CPU-ja referon faqen 10.(c) Situata kur faqja 10 eshte read-only dhe perdoret replikimi.

Ne kete shembull, ne se CPU-ja, I/O referohet instruksioneve ose te dhenave ne faqen 0, 2, 5 ose 9, atehere ky referim behet ne menyre lokale. Referencat ne faqet e tjera shkaktojne trape. Për shembull, referencia te nje adrese ne faqen 10 do te shkaktoje nje trap ne software e DSM, i cili do te shkaktoje levizjen e fakes 10 nga nyja 3 ne 0, sic e shohim ne Fig 8-23(b).

Replikimi

Nje permiresim qe i behet sistemit baze qe mund te permiresoje performancen ne menyre te ndjeshme eshte replikimi i faqeve, programeve tekst, konstanteve ose strukturave te tjera te te dhenave qe jane vetem te lexueshme (read-only). Për shembull ne faqen 10 ne Fig 8-23(c), eshte nje sesion i programit tekst, perdorimi i tij nga CPU 0 rezulton nga kopja qe i eshte derguar asaj, pa ndikuar fare ne kopjen origjinal qe eshte ne CPU-ne 1,

sic e shohim edhe ne Fig 8-23 (c). Ne kete menyre CPU-ja 0 dhe 1 mund ti referohen fakes 10 sa here te kene nevoje pa pasur nevojen e trapeve per te ngarkuar faqen e memorjes qe mungon.

Nje mundesi tjeter eshte te mos replikojme vetem faqet read-only por te gjitha llojet e faqeve. Per sa kohe qe behen veprime leximi, nuk ka ndonje difference persa i perkthet replikimit te faqeve read-only apo atyre read-write.

False sharing

Sistemet DSM jane te ngjashem me multiprocesoret ne disa aspekte. Ne te dyja sistemet, kur referohen nje fjale jo lokale te memorjes, pjesa e memorjes qe permbojne fjalen ngarkohet nga pozicioni i saj lokal dhe vendoset ne makine duke realizuar referencen (memorja kryesore ose cache-ja). Nje çeshtje shume e rendesishme eshte madhesia e njesise se memorjes. Ne multiprocesoret, permusat e bllokut te caches varojne nga 32-64 bytes. Ne sistemet DSM, gjatesia e njesise duhet te jete shumefishi i permases se fakes (MMU punon vetem me faqe), por gjithsesi ajo mund te jete 1, 2, 4 ose me shume faqe.

Te pasurit permasa te medha te fakes te DSM ka avanazhet dhe disavantazhet e veta. Avantazhi me i madh eshte se koha startup per transferimet e rrjetit eshte shume e rendesishme, për shembull te transferosh 1024 dhe 4096 bytes kerkojne pak a shume te njejtën kohe. Kur eshte nevoja te levizim nje pjese te madhe te hapesires se adresimit, numri i transferimeve mund te reduktohet duke transferuar te dhena ne njesi te medha. Kjo cilesi eshte shume e rendesishme pasi shume programe duan qe pasi te bejne referencen te nje fjale ne nje faqe, ata mund te bejne referime te shume faqeve te tjera ne te ardhmen.

Por ne anen tjeter, rrjeti do te jete i zene per nje kohe te gjate me transferime te medha, dhe bllokon faulte te tjera te shkaktuara nga procese te tjera. Ne Fig 8-24 paraqitet nje problem tjeter i quajtur **false sharing**. Ketu paraqitet nje faqe qe permbojne dy variabla te share-uar te palidhura me njera tjetren, A dhe B. Procesi 1 perdor A-ne ndersa procesi 2 B-ne. Ne keto kushte, faqja qe permbojne te dyja variablat "udheton" midis dy makinave.

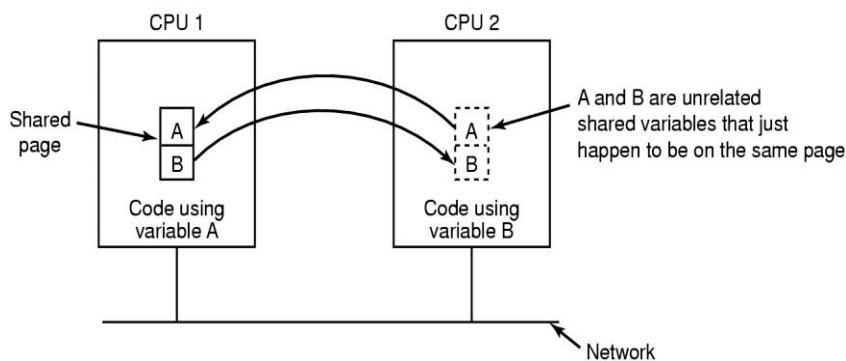


Figura 8-24. False sharing e fakes qe permbojne dy variabla te palidhura.

Problemi qe haset ketu eshte se, edhe pse variablat jane te palidhura me njera tjetren, ato do te ndodhen aksidentalish ne te njejtën faqe dhe kur procesi perdor njerën prej tyre, ai perdor edhe tjetren. Sa me e madhe te jete permasa e fakes efektive aq me shume false

sharing do te ndodhin dhe e kunderta. Ne sistemet e zakonshme te memorjes virtuale nuk ndodh asnje fenomen analog me false sharing.

Ky problem mund te reduktohet duke perdorur kompilatore te “zgjuar” qe vendosin variablat respektivisht ne hapsirat e tyre te adresimit. Por deri tani flasim vetem teorikisht.

Arritja e nje konsistence sekuenciale

Ne se faqet qe shkruhen nuk jane te replikueshme, atehere arritja e konsistencies nuk eshte problem. Atje kemi vetem nje kopje te cdo faqeje te shkrueshme dhe ajo leviz ne te dyja drejtimet ne menyre dinamike. Perderisa nuk eshte gjithmone e mundur te parashikosh se cilat faqe jane te shkrueshme, ne shume sisteme DSM, kur nje proces perpiqet te lexoje nje faqe te larget, behet nje kopje lokale dhe keto dy kopje, ajo lokale dhe ajo e larget vendosen respektivisht ne MMU-te e tyre me vetine read-only. Cdo gje eshte ne rregull per sa kohe qe te gjitha referencat jane vetem te lexueshme.

Sidoqofte, ne se nje process perpiqet te shkruaje ne nje faqe te replikuar, ngrihet problemi i konsistencies, sepse te ndryshosh nje kopje dhe te lesh te tjerat eshte e papranueshme. Kjo situate eshte analoge me ate c’ka ndodh ne nje multiprocessor kur nje CPU perpiqet te ndryshoje nje faqe qe eshte presente ne shume cache. Zgjidhja per CPU-ne eshte qe, perpara se te beje shkrimin ne faqen perkatese, te sinjalizoje ne bus duke i treguar CPU-ve te tjera te injorojne kopjet e tyre te bllokut te caches. Sistemet DSM punojne ne te njejten menyre. Perpara se nje faqe e share-uar te shkruhet, dergohet nje mesazh te te gjitha CPU-te qe mbajne nje kopje te faqes, per ti treguar atyre qe duhet te mos perkthejne dhe te injorojne faqen. Pasi te gjitha CPU-te jane perqjigjur qe c’hartezimi ka perfunduar, CPU-ja origjinale mund te fillojë shkrimin.

Gjithsesi eshte e mundur te tolerojme shume kopje te faqeve te shkrueshme ne rrethana te caktuara. Nje menyre eshte te lejojme procesin te kerkoje nje lock ne porcionin e hapesires se adresimit virtual, dhe me pas te performoje operacionet e leximit dhe te shkrimit ne memorjen e lock-uar. Ne kohen kur eshte liruar lock, ndryshimet mund te kapin edhe kopjet e tjera. Per sa kohe qe vetem nje CPU mund te lock nje faqe ne nje moment te dhene, arrijme ne perfundimin qe kjo skeme e arrin konsistencen.

8.2.6 Skedulimi ne Multikompjutera

Ne nje multiprocesor, te gjitha proceset ruhen ne te njejten memorje. Kur CPU-ja perfundon punen qe ka ne dore, ajo merr nje proces tjetër dhe fillon e ekzekuton. Ne multikompjuterat situata paraqitet ndryshe. Cdo nyje ka memorjen e vet dhe setin e saj te proceseve, për shembull CPU-ja 1 nuk mund te ekzekutoje procesin e nyjes 4 pa bere njeherë kompromis per ate. Kjo diferenca do te thote qe skedulimi ne multikompjutera eshte me i thjeshte, por alokimi i proceseve ne nyjet perkatese eshte me i rendesishëm. Skedulimi i multikompjuterave eshte pak a shume i ngjashem me skedulin e multiprocesoreve por jo te gjitha algoritmat e multiprocesoreve aplikohen te multikompjuterat. Algoritmi me i thjeshte ne multiprocesoret: mirembajtaja e nje liste te

vetme te proceseve gati, nuk funksionon pasi cdo proces ekzekutohet nga CPU-ja vetem ne se ai eshte i lokalizuar ne kete te fundit. Sidoqofte, kur krijohet nje proces i ri, duhet gjetur vendi se ku do te vendoset.

Duke qene se cdo nyje ka proceset e veta, mund të perdoret çdo algoritdem skedulimi lokal. Sidoqofte, eshte e mundur te perdoret skedulimi gang, ne te njejten menyre qe perdoret ne multiprocesoret. Parimi eshte i njejte, te vendosesh nje marreveshje fillestare ne lidhje me percaktimin e slotave te kohes se ku do te ekzekutohen proceset dhe nje menyre se si te kordinojme fillesen e slotave të kohes.

8.2.7 Load Balancing

Pra per skedulimin ne multikompjutera me vendosjen e procesit në nyje, mund te perdonim çdo algoritdem skedulimi, ne rast te kundert mund te perdonim skedulimin gang. Por eshte shume e rendesishme menyre se si caktohen proceset per çdo nyje. Kjo eshte ne kontrast me sistemet e multiprocesoreve ku te gjitha proceset jane në te njejten memorje dhe mund të skedulohen me deshire nga cdo CPU. Për të percaktuar vendosjen e proceseve ne nyjet perkatese perdoren algoritmat e njohur si **processor allocation algorithms** (algoritmat e alokimit te procesorit). Nder qellimet e tyre mund te permendim, minimimi i shperdorimit te ciklevet CPU-ve dhe i bandwidthit total te komunikimit dhe te qenit i drejte me perdonuesit&proceset.

Algoritmi Graph-Theoretic Deterministic

Nje klase shume e perhapur e algoritmeve aplikohet ne sistemet qe kane te njohur CPU-ne dhe specifikimet e memorjes dhe nje matrice e cila jep informacion për te dhenat mesatare midis dy proceseve. Ne se numri i proceseve eshte me i madh se numri i CPU-ve, atehere çdo CPU-je i caktohen disa procese. Ne kete menyre ne minizojme trafikun ne rrjet.

Sistemi mund te perfaqsohet si një graf i peshuar, ku çdo rreth perfaqeson një proces dhe shigjeta perfaqeson rrjedhjen e mesazheve midis dy proceseve. Matematikisht problemi shtrohet ne gjetjen e rrugeve per copetimin e grafit ne k subgrafe, te cilat i nenshtrohen specifikimeve perkatese. Shigjetat qe lidhin dy subgrafe perfaqesojnë te dhenat e rrjetit. Qellimi ne kete rast do të ishte te gjesh particionin e duhur, qe minimizon trafikun e te dhenave ne rrjet gjithmone duke ju pershtatur specifikimeve. Ne Fig 8-25 shohim nje sistem prej nente procesesh.

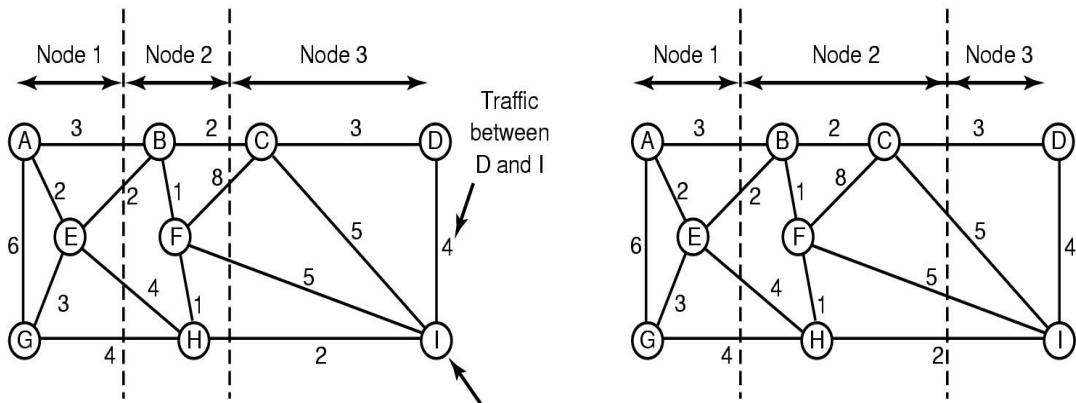


Figura 8-25. Dy menyrat e alokimit te 9 proceseve ne 3 nyje.

Ne Fig 8-25(a) kemi copetuar grafin me proceset A, E dhe G ne nyjen 1, B, F, H ne nyjen 2 dhe C, D, I ne nyjen 3. Ne Fig 8-25 kemi nje particion tjeter i cili ka vetem 28 njesi te dhenash te rrjetit. Kjo zgjidhje kerkon me pak komunikim.

Algoritmi Sender-Initiated Distributed Heuristic

Te shohim tani disa algoritma te shperndare. Për shembull, një algoritem thote qe kur një proces krijohet, ai ekzekutohet ne nyjen qe e krioi perndryshe ajo nyje mbingarkohet. Ne se ndodh kjo, nyja zgjedh një nyje tjeter ne menyren random dhe e pyet ate per ngarkesen qe ka. Ne se ngarkesa e kesaj nyje eshte me poshte se vlera threshold, atehere dergojme procesin atje, ne se jo, atehere zgjedhim një makine tjeter per te analizuar. Ky i fundit nuk zgjat per gjithmone. Ne se nuk gjendet një host i pershtatshem brenda N kontrolleve, algoritmi perfundon dhe procesi ekzekutohet ne makinen origjine, sic tregohet ne Fig 8-26.

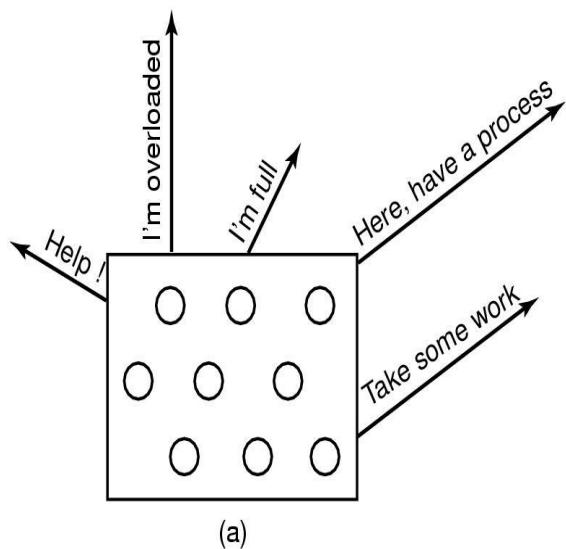


Figura 8-26. Nje nyje e mbingarkuare e cila eshte ne kerkim te nje nyje tjeter per ndihme.

Ideja eshte qe nyjet e superngarkuara te mos marrin pune tjeter persiper. Duhet nenvizuar fakti qe, ne kushte te nje ngarkese te madhe, te gjitha makinat do te fillojne kontrollet ne makinat e tjera duke gjetur nje menyre per ti lene atyre pak pune. Por shume pak procese e realizojne kete gje dhe mund te themi qe, zakonisht mbizoteron mbingarkesa.

Algoritmi Receiver-Initialed Distributed Heuristic

Parimi i punes se ketij algoritmi eshte, kur nje proces perfundon, sistemi kontrollon ne se ai ka pune mjaftueshem. Ne se jo, ai kerkon nje makine tjeter ne menyren random dhe i kerkon atij pune. Ne se ai nuk gjen serish pune ne makinat e kontrolluara N, atehere nyja pushon se pyeturi perkohesisht, ben çdo pune qe ka ne dore dhe perpiqet serish kur procesi tjeter perfundon. Ne se serish nuk ka pune te disponueshme, makina kalon ne gjendjen idle. Pas nje intervali fiks, ai fillon serish analizen ose kontrollin. Pra ky algoritmet inicializohet nga nje marres *underloaded*, domethënë pa ngarkese.

Avantazhi i ketij algoritmi eshte se ai nuk vendos ngarkese ekstra ne sistem ne kohe kritike. Algoritmi sender-initialed ben nje numer te madh analizash, kur sistemi mund ti toleroje ato, kur eshte goxha i ngarkuar. Me kete algoritmet, me sistemin shume te ngarkuar, shansi qe makina te kete pune jo te mjaftueshme eshte shume e vogel. Sidoqofte kur kjo ndodh, do ishte shume e lehte per te marre me shume pune persiper. Sigurisht, kur ka pak pune per te bere, algoritmi R-I krijon nje trafik te konsiderueshem analizash kur makinat ne menyre te “deshperuar” kerkojne pune. Pra eshte me mire te kemi nje rritje te ngarkeses ne kushtet e nje nen-ngarkese sesa ne rastin e nje mbingarkese. Ne mund ti kombinojme keto dy algoritma, domethënë makinat do te heqin qafe nje pjese te punes kur ka shume dhe do te kerkojne me shume kur nuk ka pune mjaftueshem.

Algoritmi Bidding

Nje klase tjeter algoritmash perpiqet ta ktheje sistemin kompjuterik ne nje ekonomi miniature, me shites, bleres sherbimesh dhe çmime te vendosura nga ngarkesa dhe kerkesa. Protagonistet me te rendesishem ne ekonomi jane proceset, te cilet duhet te blejne kohen e CPU-se per te bere punen e tyre dhe nyjet te cilat nxjerrin ne ankand ciklet e tyre per ofruesin me te larte.

Cdo nyje lajmeron cmimin e vet ne nje file publik te lexueshem. Ky cmim nuk eshte i garantuar por eshte nje indikacion per vleren e sherbimit. Nyje te ndryshme kane cmime te ndryshme ne varesi te shpejtesise, permasave te memorjes, prezences se hardware-it te shpejte floating-point dhe tipareve te tjera. Nyja mund te publikoje edhe sherbime te tjera si koha e pergjigjes.

Kur nje proces do te krijoje nje proces femije, ai shikon se kush ofron sherbimin qe ai kerkon. Me pas ai percakton setin e nyjeve me sherbimet perkatese. Nga ky set ai zgjedh kandidatin me te “mire”, domethënë ate me te shpejten, me performance me te mire, ate qe i pershtatet me mire aplikacionit te tij. Ai gjeneron cmimin i cili mund te jete me i larte ose me i ulet se cmimi i publikuar.

KAPITULLI I NËNTË

SECURITY (Siguria e Sistemit)

9.1 Ambjenti i Sigurise

Shume veta i përdorin termat “siguri” dhe “mbrojtje” të pandashme. Kjo do të hapte gjithmone debat për problemet kryesore ne administrimin e file-ve dhe mirembajtjen e tyre. Për ti shmangur keto keqkuptime, ne përgjithesi përdorim termin “**siguri**”, për ti përfshire të gjithe problemet dhe përdorim termin “**mbrojtje e mekanizmit**” për ti referuar mekanizmit specific të sistemit operativ e përdorur kjo për të mbikqyrur informacionin ne kompjuter. Siguria ka tre aspekte qe duhen pare, te cilat jane: threats, intruders, accidental data loss.

9.1.1 Threats

Për një prespektive të sigurt sistemet kompjuterike kane tre objektiva kryesore, për kercenimet e ndryshme si ne figuren 9-1. E para është **të dhenat e besueshme**, qe ka si qellim te dhenat qe i kemi te fsheta te mbeten te tilla. Me e specifikuar, ne qofte se zoteruesi i disa te dhenave, ka vendosur qe keto te dhena te shfaqen per nje numer te caktuar njerezish, sistemi duhet te garantoje qe mos te ndodhi, qe leja tu jepet njerezve te paautorizuar nga ai. Por minimalisht zoteruesi i ketyre te dhenave duhet te percaktoje kush do ti shikoje keto te dhena dhe sistemi e perforcon ate.

Objektivat	Threat
Të dhenat e besueshme	Ekspozimi i të dhenave
Integriteti i të dhenave	Tampering with data
Disponueshmeria e sistemit	Denial of service

Figura 9-1. Objektivat e sigurise dhe thretet.

Objektivi e dytë është **integriteti të dhenave**, nenkupton qe përdoruesit e paautorizuar nuk duhet të jene ne gjendje të modifikojne asnjë të dhene pa leje. Modifikimi i të dhenave nenkupton jo vetëm ndryshimin e tyre por edhe fshirjen e tyre ose të shtimit të të dhenave fallco. Ne se një sistem nuk garanton qe keto te dhena te mbeten te pandryshueshme derisa vete zoteruesi i tyre te vendosi ti ndryshoje, qellimi mund të rezultoje i pavlefshem.

Objektivi e tretë, është **Disponueshmeria e sistemit**, nenkukpton qe asnjëri nuk mund të nderhyje ne sistem për ta bere atë të papërdorshem. Keto sulme të **Denial of service** vijne gjithmone duke u rritut. Për shembull, ne se një kompjuter është një Internet Server, dergimi i i nje fluksi te madh kerkesash nga ana e klientëve mund ta ngrijë sistemin,

sepse CPU-ja angazhohet plotesh per egzaminimin e tyre dhe berjen e tyre te pavlefshme. Mund te themi se ne qofte se koha per te perpunuar nje kerkesa per leximin e nje faqe Web, eshte 100 μ sec, atehere çdokush qe do te dergonte 10.000 kerkesa ne sekonde do ta ngrinte sistemin.

Një tjetër akspekt i sigurise është problem i privatesise: ti mbrosh personat nga një keqperdorim i informacionit qe egziston per ta. Kjo shkurtimisht ka te beje me problem morale dhe legale. Një detyre e rendesishme i paraqitet dhe shtetit për të parandaluar ose për të marre masat e ndryshme për persona qe krijojne difekte ne sisteme të ndryshme.

9.1.2 Intruders (Sulmuesit e sistemit)

Shume njerez jane të respektueshem ndaj ligjit dhe për keta nuk mund të kemi preoukupime të thyerjes se sigurise, shume të tjere jane të prirur të thyejne rregullat dhe ligjet dhe shumica e bejne për qejf ose ana financiare. Ne letërsine e sigurise , thuhet qe njerezit qe vijne rrötull bisneseve qe nuk kane të bejne me to, quhen intruders ose zakonisht thuhet adversaries. Aktet e sulmuesve ndahen ne dy kategori, kjo përcakton dhe natyren e sulmuesit të sistemit. Sulmuesit pasive, thjesht kerkojne te lexojne informacionin e fileve. Sulmuesit aktive jane me të keqinjte, ata synojne të modifikojne informacionin e fileve. Kur eshte duke u programuar nje sistem per te qene te sigurte ndaj sulmeve, është e rendesishme të parashikohet lloji i sulmit dhe detaje të tij. Disa kategori jane :

1. Prishjet rastesore nga përdorues të paaftë. Shume njerez disponojne kompjutera ne tavolinat e tyre, të cilët jane të lidhur me një server ku informacioni i tyre ndahet (share) midis klientëve të ndryshem, keshtu qe se paku ata do të behen kurioz dhe do të deshirojne të lexojne e-mailet e te tjereve, ne qofte se nuk jane vendosur bariera per ta ndaluar kete fenomen. Për shembull, shume sisteme UNIX, e ka default qe file-t e rinj qe jane krijuar mund te jene te lexueshem nga te gjithe.
2. Nderhyrje nga te brendshmit. Studentet, programuesit e sistemeve, operatoret dhe personeli tjeter teknik shpesh konsiderohen si persona, qe e marrin si sfide thyerjen e sistemeve te sigurise. Ata shpesh jane persona shume te kualifikuar dhe shpenzojne shume kohe për ta arritur ate.
3. Veprime të ndryshme për të patur fitime financiare. Disa programues bankash sulmojne banken ne te cilen ata kane punuar, ne menyre qe të mbajne prej klientëve disa të dhjetrat e centëve por qe ne shume vetë ato arrijne shifra të medha.
4. Spiunime ushtarake ose tregetare. Spiunimi ne ketë rast vjen shume i përgatitur se është me një pikesnim të percaktuar nga nje kundershtar, rezulton ne vjedhje programesh të huaja, transferta bankare, plane bisnesi, projekte nderkombetare, etj.. Keto nderhyrje kane te bejne me nderhyrjet e fshehta ne sistem ose me vendosjen e antenave per thithjen e valeve elektromanjetike.

Duhet te jete e qarte qe eshte komplet ndryshe te ruhesh nga sulmet e nje shteti per te vjedhur sekretet ushtarake dhe te ruhesh nga studentat te cilet perpiqen te fusin nje mesazh per te qeshur ne sistem. Ne tjeter lloj nderhyrje qe eshte shfaqur ne vitet e fundit jane viruset. Ne parim nje virus eshte nje pjese kodi qe zevendeson vetveten dhe sjell demtime. Ne kete kuptim, programuesi i nje virusi eshte nje intruder, shpesh me aftesi

shume te medha. Ndryshimi ndermjet nje intruder dhe nje virusi eshte se i pari eshte nje person qe po perpiqet te hyj ne sistem dhe ta demtoje ate, ndersa i dyti eshte nje program i shkruar nga nje person dhe pastaj i lancuar me shpresen per te bere deme, pra ne qofte se intruder perpiqet te shkaterroj nje gje specifike, virusi ben deme me te per gjitheshme.

9.1.3 Humbja aksidentale e të dhenave

Ne shtim të threat-ve, kur sistemi sulmohet, mund të kemi humbje të të dhenave. Disa nga rastet me tipike të humbjes se të dhenave ne menyre aksidentale jane:

1. Aksidentet natyrore si zjarri, përmbytjet, lufta qe mund të shkaktoje deme ne kompjuterat qe ndodhen ne atë ambjent.
2. Gabimet hardware-rike ose Software-rike: keqfunkcionimi i CPU, disjet e palexueshem për shkak të demtimeve sipërfaqesore, gabimet ne nderlidhje...etj
3. Gabimet njerezore: deklarime të pasakta të të dhenave, kerkimi i fileve ne disqe të gabuar, ekzekutim i gabuar i programit, humbja e disqeve ose disketave, etj..

Disa nga keto gabime mund të shmangen ne se kemi një mirembajtje të të dhenave, të dyfishuara (backup), të mbajtura ne kushte të tjera nga ato qe mbahen të dhenat origjinale. Mbrojtja e të dhenave nga humbja aksidentale e tyre, mund të rezultoje me e lodhshme se sa mbrojta kunder sulmuesve të sistemit.

9.2 BAZAT E KRIPTOGRAFISE

Disa njohuri rreth kriptografise mund të nevojiten për të kuptuar disa pjese të ketij kapitulli dhe të disa të tjereve vijues. Gjithsesi ky liber nuk ka për qellim të kryeje një diskutim serioz rreth kesaj çeshtje. Për të interesuarit rreth kesaj teme, ekzistojne shume libra të tjere qe diskutojne me hollesisht (si Kaufman et al,1995; apo Pfleeger,1997). Me poshtë do të zhvillojme një diskutim të shpejtë rreth kriptografise, për lexues qe nuk jane shume të familiarizuar me të.

Qellimi i Kriptografise është qe të marre një mesazh ose file, të quajtur **plaintext** (tekst i thjeshtë) dhe ta enkriptoje atë ne një **ciphertext** (tekst të koduar), pra ta kodoje, ne menyre qe vetëm persona të autorizuar të dine ta konvertojne kodin ne mesazhin fillestar, pra të kuptojne përbajtjen e mesazhit. Kurse për të gjithe të tjeret **ciphertext** -i është thjesht një grumbull i pakuptueshem bitesh. Megjithese, për fillestaret ne ketë fushe, mund të duket e çuditshme, algoritmet enkriptues dhe dekriptues duhet qe gjithmone të jene publike (të njohura). Përpjekjet për ti mbajtur të fshehta keto algoritme gjithmone deshtojne, dhe s`bejne asgje tjetër veçse i jepin njerezve qe përpilen ti mbajne të fshehtë një ndjesi sigurie të rreme. Ne ketë fushe (kriptografi), kjo takte quhet **Security By Obscurity** (siguri nga paqartësia) dhe përdoret vetëm nga amatoret. Çuditërisht ne ketë kategori hyjne dhe shume korporata të medha multinacionale qe duhet të jene me të vetëdishme (qe takтика është e kotë).

Ne të vertetë, fshehtësia varet nga parametrat e algoritmit të quajtur **keys** (çelës). Ne qoftë se P është file plaintext, K_E është key i enkriptimit, C është ciphertext dhe E është algoritmi enkriptues, atëherë :

$$C = E(P, K_E)$$

Ky është përkufizimi i enkriptimit, qe tregon se Ciphertext-i merret duke përdorur algoritmin enkriptues E , me plaintext P dhe key e enkriptimit K_E si parametra.

Ne analogji me sa thame me sipër kemi dhe: $P = D(C, K_D)$, ku D është algoritmi i dekriptimit dhe K_D është key (çelësi) dekriptues. Kjo domethene qe për të marre plaintext-in P nga ciphertext-i C dhe key K_D të dekriptimit, duhet përdorur algoritmi D me C dhe K_D si parametra. Lidhja ndermjet pjeseve të ndryshme tregohet ne figuren 9.2.

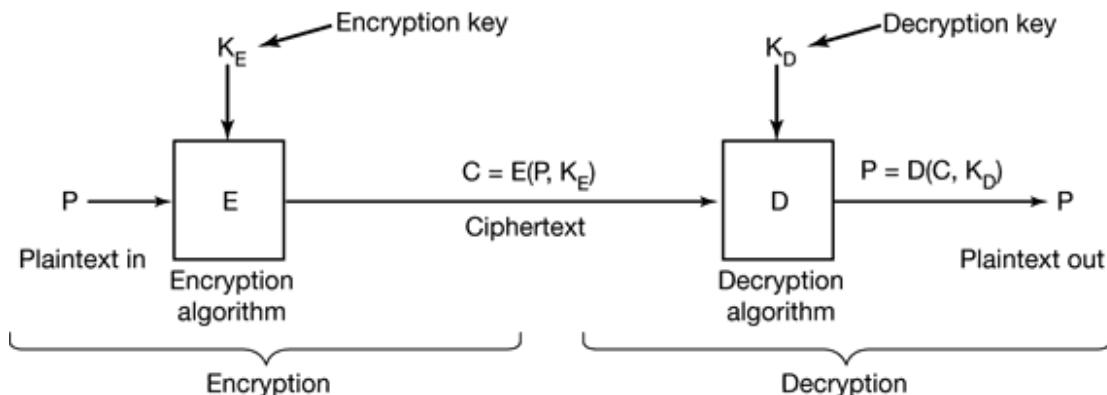


Figura 9-2. Lidhja ndermjet plaintext-it dhe ciphertext-it.

9.2.1 Kriptografia Secret-Key (me çelës të fshehtë)

Për të qene sa me të qartë, mendojme për një algoritem enkriptues ne të cilin çdo shkronjë zevendesohet me shkronjë tjeter të ndryshme, për shembull të gjitha A -të zevendesohen nga Q -të, të gjitha B -të zevendesohen nga W -të, të gjitha C -të zevendesohen nga E -të dhe keshtu me rradhe:

plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ciphertext: QWERTYUIOPASDFGHJKLZXCVBNM

Ky sistem i përgjithshem quhet një zevndesim monoalfabetik ku çelës është stringa me 26 germa qe i korrespondon gjithe alfabetit. Çelësi i enkriptimit është *QWERTYUIOPASDFGHJKLZXCVBNM*. Për çelësin e mesipërm, plaintext-i *ATTACK* do të transformohej ne ciphertext-in *QZZQEA*. Çelësi i dekriptimit na tregon se si behet kthimi nga ciphertext ne plaintext. Ne ketë shembull çelësi i dekriptimit është *KXVMCNOPHQRSZYUADLEGWBUFT* sepse një A ne ciphertext është një K ne plaintext, një B ne ciphertext është një X ne plaintext e keshtu me rradhe.

Ne pamje të pare ky sistem mund të duket i sigurtë, pasi megjithese kriptanalisti e njeh sistemin e përgjithshem (zevndesimet germe për germe), ai nuk e di se cili nga $26! \approx 4 \times 10^{26}$ çelësat e mundur është ne përdorim. Megjithatë, në qoftë se çuditërisht jepet një

ciphertext me gjatësi të vogel, kodi mund të thyhet lehtësisht. Sulmi baze (kryesor) ndaj kodit, përfiton nga vetitë statistike të gjuheve natyrale. Ne Anglisht për shembull e është shkronja me e përdorshme e cila ndiqet nga **t**, **o**, **a**, **n**, **i** etj. Kombinimet me të përdorshme me dy shkronja, të cilat quhen **digrams**, jane **th**, **in**, **er**, **re**, etj. Duke përdorur ketë lloj informacioni, thyerja e kodit behet me e lehtë.

Shume sisteme kriptografike, si ky i mesipërm, kane vetine qe kur çelësi i enkriptimit është i dhene, lehtësisht gjendet ai i dekriptimit dhe anasjelltas. Sisteme të tille quhen **Secret-Key Cryptography** ose **Symmetric-Key Cryptography**. Megjithese kodet monoalfabetike me zevendesime jane të padobishme, njihen të tjere algoritme me çelësa simetrik qe Jane relativisht të sigurtë, në qoftë se çelësat kane gjatësi të mjaftueshme. Për një siguri të madhe (serioze) duhet të përdoren çelësa 1024 bitesh, të cilët jepin një hapesire kerkimi prej $2^{1024} = 2 \times 10^{308}$ çelësash. Çelësa me të shkurtër mund të pengojne amatoret, por jo autoritetet e specializuara ne fushen e Kriptografise.

9.2.2 *Kriptografia Public-Key (me çelës publik)*

Sistemet Secret-Key jane shume eficentë pasi sasia e llogaritjeve të nevojshme për të enkriptuar ose dekriptuar një mesazh është e menaxhueshme, por ato kane një të dobësi të madhe: Qe si derguesi ashtu dhe marresi duhet të zotërojne secret-key-n qe përdorin. Madje mund të duhet edhe të takohen fizikisht ne menyre qe përdoruesit të shkembejne ketë secret-key. Për të anashkaluar ketë problem përdoret **Kriptografia Public-Key** (Diffie dhe Hellman, 1976). Ky sistem ka vetine qe përdor çelësa të ndryshem për enkriptimin dhe dekriptimin, dhe kur jepet një çelës enkriptimi i zgjedhur mire, është virtualisht e pamundur të zbulohet çelësi i dekriptimit korrespondues. Ne keto kushte çelësi i enkriptimit mund të behet publik dhe vetëm çelësi privat i dekriptimit mbahet i fshehtë.

Sa për të dhene një përshtypje rrëth kriptografise public-key, shqyrtojme dy pyetjet e meposhtme :

Pyetja 1: Sa bejne $314159265358979 \times 314159265358979$?

Pyetja 2: Sa është rrenja katrore e $3912571506419387090594828508241$?

Ne përgjithesi një nxenesi të klases se gjashtë po ti japesh një laps dhe një letër, dhe po ti premtosh një akullore të madhe si shpërbirim për përgjigjen e saktë, për 1-2 ore e zgjidh pyetjen e pare. Ndersa shumices se të rriturve po tu japesh një laps, letër dhe premtimin për përgjysmimin e taksave për gjithe jetën, mund ta kene të pamundur të zgjidhin pyetjen e dytë pa përdorur makine llogaritëse, kompjuter apo ndihma të tjera të jashtme. Megjithese ngritja ne kator dhe rrenja katrore jane veprime inverse të njëra-tjetres ato ndryshojne shume ne kompleksitetin e llogaritjes. Kjo lloj asimetrie formon bazen e Kriptografise Public-Key. Enkriptimi kryen veprimin e thjeshtë kurse dekriptimi pa çelësin (e njohur) na bën të kryejme veprimin e veshtire.

Një sistem Public-Key i quajtur **RSA**, shfrytëzon faktin qe shumezimi i numrave të medhenj është me i thjeshtë se faktorizimi i tyre, veçanërisht kur gjithe veprimet arithmetike behen duke përdorur module arithmetike dhe të gjithe numrat e përfshire kane qindra shifra. Ky sistem përdoret gjeresisht ne botën e kriptografise. Përdoren gjithashtu edhe sisteme qe bazohen ne algoritma diskretë. Problemi kryesor i

kriptografise public-key është fakti qe ai është me mijera here me i ngadaltë se kriptografia simetrike.

Kriptografia public-key punon ne menyre të tille qe secili zgjedh një çift çèlesash (public key, private key) dhe publikon çelesin e tij publik. Çelesi publik është çelesi i enkriptimit, kurse çelesi privat është ai i dekriptimit. Zakonisht çelesi gjenerohet ne menyre automatike, mundesisht me një password qe zgjidhet nga përdoruesi i cili futet ne algoritmen. Për ti derguar një përdoruesi një mesazh të fshehtë, enkriptohet mesazhi me public-key e marresit. Meqë vetëm marresi ka private key, atëhere ai mund të dekriptoje mesazhin.

9.2.3 Funksionet One-Way (me një drejtim)

Ekzistojne situata të ndryshme, të cilat do ti shohim me tej, ne të cilat na nevojitet të kemi një funksion f , i cili ka vetine qe kur jepet f dhe parametri i tij x , llogaritja e $Y = f(x)$ behet lehtësisht, por kur jepet vetëm $f(x)$ llogaritja e x behet e pamundur. Një funksion i tille i ndryshon bitet ne një menyre të nderlikuar. Ky ndryshim mund të filloje qe nga inicializimi i y ne x ($y=x$). Me pas mund të kemi një cikel qe përsëritet sa here qe kemi bit 1 ne x , ku ne çdo përsëritje ndryshohet rradha e biteve y ne një menyre qe varet nga përsëritja, duke i shtuar një konstante të ndryshme ne çdo përsëritje, dhe përgjithesisht duke i miksuar (përzier) bitet rrenjësisht. (biti i pare behet i fundit etj)

9.2.4 Nenshkrimet Dixhitale

Shpesh është e nevojshme qe një dokument të nenshkruhet ne menyre dixhitale. Për shembull mendojme për një klient banke qe nepërmjet një email-i, i kerkon bankes të bleje disa aksione për të. Një ore pasi porosia është derguar dhe ekzekutuar, çmimi i aksioneve bie. Tani klienti mohon të ketë bere një porosi me email. Banka mund ta printoje email-in por klienti mund të ankohet se banka e ka falsifikuar email-in me qellim përfitimi. Si mundet një gjykatës të dij se kush po thotë të vertetën?

Nenshkrimet dixhitale na mundesojne nenshkrimin apo firmosjen e mesazheve email dhe dokumenteve të tjera dixhitale ne një menyre qe nuk mund të mohohet me pas nga derguesi. Një menyre e zakonshme është qe fillimi qe fillimi i dokumentit e marre me anen e një algoritmi **hashing** (përpunues, ndares) me një drejtim. (Pra nga fillimi i dokumentit ne fund të tij). Ky funksion përpunues prodhon një rezultat me gjatësi të rregullueshme, të pavarur nga madhesia e dokumentit original. Funksionet përpunuese me të përhapura qe jane sot ne përdorim Jane **MD5 (Message Digest)** i cili prodhon një rezultat prej 16-bytesh (Rivest 1992), dhe **SHA (Security Hash Algorithmn)** i cili prodhon një rezultat prej 20-bytesh (NIST 1995).

Hapi tjetër është ai i përdorimit të kriptografise public-key siç u përshkrua me lart. Me pas derguesi, aplikon mbi **hash** çelesin e tij privat ne menyre qe të marre D (**hash**). Kjo vlore e quajtur **signature block**, (blloku i nenshkrimit/firmes) i shtohet dokumentit dhe i dergohet marresit, si tregohet ne Fig. 9-3. Aplikimi i D ne **hash** shpesh referohet si dekriptimi **hash** por ne fakt nuk është e vertet një dekriptim pasi **hash**-i nuk është enkriptuar me pare. Është vetëm një trasformim matematik i tij.

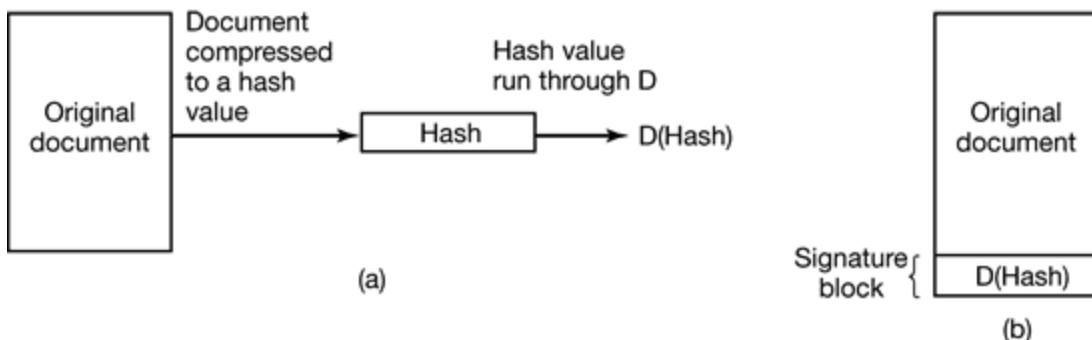


Figura 9-3. (a) Kryerja e nenshkrimit të bllokut. (b) Ajo çfare marresi merr

Kur dokumenti dhe hash-i arrijne ne destinacion, ne fillim marresi kryen copezimin (**hashing**) e dokumentit duke përdorur **MD5** ose **SHA**, si folem me sipër. Marresi me pas përdor mbi bllokun e nenshkrimit, çelesin publik të derguesit me qellim qe të marre E (D (hash)). Për pasoje, ai enkripton hash-in e dekriptuar, duke fshire dekriptimin dhe duke marre serish hash-in. Në qoftë se hash-i i llogaritur nuk përputhet me hash-in e bllokut të nenshkrimit, atëherë dokumenti, blloku i nenshkrimit ose të dy njëkohesisht janë falsifikuar ose ndryshuar aksidentalisht. Vlera e kesaj skeme qendron ne faktin se kriptografia public-key i aplikohet vetëm një pjese relativisht të vogel të dhenash, **hash-it**. Vereni me kujdes se kjo metode punon vetëm ne se për të gjithe X:

$$E(D(x)) = x$$

Nuk mund të sigurohet qe të gjithe funksionet enkriptuese do ta kene ketë veti, përderisa gjithçka qe ne kerkojme nga **Kriptografija** është:

$$D(E(x)) = x$$

Ku E është funksioni enkriptues dhe D është funksioni dekriptues. Ne përfundim, qe të përfitojme vetine e nenshkrimit, rradha e veprimeve nuk duhet të ketë rendesi, domethene qe si D dhe E duhet të jene funksione nderrues. Fatmiresisht algoritmi **RSA** e ka ketë veti.

Për të përdorur ketë skeme nenshkrimi, marresi duhet të njohe çelesin publik të derguesit. Disa përdorues i publikojne çelesat e tyre publik nepër faqet e tyre web. Të tjere jo, pasi kane frike se mos ndonjë nderhyres futet dhe fshehtësisht ndryshon çelesin e tyre. Për ta nevojitet një mekanizem alternativ për shpërndarjen e çelesave të tyre publik. Një metode e zakonshme është qe derguesit ti ngjisnin një **certifikatë** mesazhit. Kjo certifikatë do të ketë emrin e përdoruesit, çelesin e tij publik si dhe nenshkrimin e një pale të tretë të besueshme. Pasi marresi të marre çelesin publik të pales se tretë të besueshme, ai mund të pranoje certifikata nga të gjithe derguesit qe përdorin ketë pale të tretë të besueshme për të gjeneruar certifikatat e tyre.

Me sipër treguam sesi kriptografia public-key mund të përdoret për nenshkrimin dixhital. Gjithsesi ja vlen qe të theksohet se ekzistojne dhe skema të tjera qe nuk përfshijnë ketë kriptografi.

9.3 Vertetimi i përdoruesit

Tani qe kemi një formim të pastër, le të shikojme kriteret e sigurise ne sistemet operative. Kur një përdorues log-ohet ne një kompjuter, sistemi operativ mundohet të përcaktoje cili përdorues është. Ky proces quhet vertetimi i përdoruesit. Mainframe-et e pare si ENIAC nuk kishin system operativ, prandaj e linin të lire proceduren login.

Mainframe-et e mepassheem balch dhe sistemet timesharing zakonisht kane një procedure log-in për vertetimin e përdoruesit. Minikompjuterat e pare (PDP-1 dhe PDP-8) nuk kishin procedure log-in por me përhapjen e UNIX ne minikompiuterin PDP41 kjo procedure nevojitej përseni. Kompjuterat e pare personale (Apple 11 dhe IBM PC) nuk e kishin ketë procedure por ata me sistem operativ me të sofistikuar si Windows 2000 përfshinin një log-in të sigurt. Kur përdorim një kompjuter personal për të aksesuar ne LAN gjithmone përfshihet një log-in. Ceshtja e një log-in të sigurtë kalon nepër disa cikle dhe është një ceshtje me rendesi. Duke pasur të përcaktuar hapin e vertetimit, hapi tjetër është të gjejme një rruge të thjeshtë për të arritur tek vertetimi.

Shume metoda vertetimi bazohen ne një nga tre principet e meposhtme:

1. Dicka qe përdoruesi di
2. Dicka qe përdoruesi ka
3. Dicka qe përdoruesi është

Keto principe shoqerojne skema të ndryshme vertetimi me kompleksitet dhe siguri të ndryshme. Persona të cilët kerkojne të shkaktojne probleme ne një sistem, me pare duhet të log-ohen ne ketë sistem. Keta persona quhen hacker, për ndër të programuesve të medhenj. Ne dallim nga hacker-at e vertetë do të përdorim termin crackers, ne sensin orgjinal dhe do të quajme ata njerez qe mundohen të thyejne sistemet kompjuterike .

9.3.1 Vertetimi duke përdorur password

Forma me e përdorshme e vertetimit është të lejoje përdoruesin të shkruaje një emer log-in dhe një password. Mbrojtja me pagese është e lehtë për tu kuptuar dhe për tu implementuar. Implementimi me i thjeshtë mban një listë cift të log-in name dhe password. Emri log-in i shkruar shikohet ne listë dhe password i shkruar krahasohet me password-in e vendosur. Ne qoftë se përpushten, log-in pranohet, ne të kundert log-in refuzohet. Kuptohet qe kur shkruhet password-i, kompjuteri nuk i tregon karakteret.

Ne Windows 2000, karakteret e shkruar tregohen ne forme ylli, ndersa ne UNIX, nuk tregohet asgje. Keto skema kane cilesi të ndryshme. Ne skemen e Windows 2000 është me lehtë për përdoruesit qe harrojne, pasi ata shikojne sa karaktere kane shtypur por kjo zbulon gjatësine e password-it, e cila sherben për përgjuesit. Ne figuren 9-4 (a) tregohet një log-in i sukseshem. Ne figuren 9-4 (b) tregohet një deshtim nga një cracker për tu log-uar ne sistemin A. Ne figuren 9-4 (c) tregohet një deshtim i një cracker për tu loguar ne sistemin B.

LOGIN: ken
PASSWORD: FooBar
SUCCESSFUL LOGIN

(a)

LOGIN: carol
INVALID LOGIN NAME
LOGIN:

(b)

LOGIN: carol
PASSWORD: Idunno
INVALID LOGIN
LOGIN:

(c)

Figura 9-4. (a)Log-in i suksesshem, (b)Login refuzohet pasi futet emri
(c)Login refuzohet pasi futet emri dhe passwordi.

Ne figuren 9-4(b), sistemi ankohet kur shkruajme një emer log-in të pavlefshem. Ky gabim bën qe cracker-at të provojne emra të tjere log-in derisa të gjejne atë të vlefshmin. Ne fig. 9-4(c), cracker-i gjithmone pytet për password dhe nuk merr feedback derisa emri log-in të jetë i vlefshem.

Menyra si hyjne ne sistem crackers

Shume crackers hyjne ne sistem duke përdorur shume kombime të emrit log-in dhe passwordit, derisa gjejne kombinimin e vlefshem. Shume persona, ne një forme apo ne një tjetër, vendosin emrin e tyre si emer log-in. Të pajisur me një liber me emra për femijen qe do të linde dhe me një numerator telefoni, crackers mund të kompilojne një listë me emra log-in tepër të vlefshem për zonen ku ndodhen. Por sigurimi i emrit log-in nuk është i mjaftushem. Duhet të gjendet dhe password-i. Shtrohet pyetja: Sa e vesh tire është kjo?

Puna klasike për sigurine e password-eve u be nga Morris dhe Thompson (1979) ne sistemet UNIX. Ata kompiluan një listë me password-e si: emer-mbiemer, emer-rruge, emer-qyteti, etj. Me pas i krahasuan listën e tyre ne qoftë se kishte përputhje. Ne mbi 86% të rasteve kishin përputhje. Një rezultat i ngjashem me ketë u morr edhe nga Klein(1990). Një survejim ne 1997 ne distriktin finanziar të Londres tregoi se 82% e password-eve mund të zbuloheshin lehtësisht. Password-et me të përdorshem ishin terma seksuale, shprehje abuzive, emra njerëzish. Ne ketë menyre një cracker mund të komplioje një listë me password-e dhe emra log-in pa ndonjë pune të lodhshme.

Një cracker Australian ka shkruar një program i cili sistematikisht kryen thirrje ne të gjithe numrat e telefonit dhe me pas mundohet të hyje ne sistem duke përdorur zbulues password-i. Kur puna realizohet me sukses programi informon përdoruesin. Midis shume sistemeve qe theu ky cracker ishte dhe një kompjuter ne Arabine Sauditë, i cili i lejoi atij të përfitonte numra kartash krediti. Një alternative është dhe lidhja e kompjuterave ne internet. Çdo kompjuter ne internet ka një IP adresë 32 bit të gjatë e cila përdoret për të identifikuar përdoruesit e ndryshem.

Njerëzit zakonisht i shkruajne keto adresa ne forme decimale si w. x y. z. ku secila prej katër komponentëve të IP adrese është një integer nga 0 ne 255. Një cracker mund të testojne lehtësisht ne qoftë se disa kompjutera kane një IP adresë të caktuar duke shkruar:

ping w. x. y. z.

Ne qoftë se kompjuteri është ne pune, ai do të përgjigjet dhe programi *ping* do të tregojë se sa i gjatë ishte intervali i kohes ne milisekonda vajtje-ardhje. Është e lehtë të shkruhet një program për të realizuar ping ne një numer të madh IP adresash.

Ne qoftë se kompjuteri është ne pune, cracker mund të hyje ne të duke shkruar:

telnet w.x.y.z.

Ne qoftë se lidhja e kerkuar vendoset, cracker mund të filloje me shkrimin e emrave log-in dhe password-eve nga lista e tij. Megjithatë crackeri mund të jetë ne gjendje të hyje ne system dhe të kape password-in i cili ndodhet ne */etc/passwd*. Me pas ai do të mbledhe informacione statistike rreth emrave log-in të përdorur me shpesh dhe këtë informacion e përdor për të siguruar emra log-in ne te ardhmen.

Shume telnet deamons hyjne ne lidhjet TCP pas disa përpjekjeve të deshtuara për log-in, kjo për të vonuar crackers. Crackers reagojnë duke startuar ne paralel shume threads, të cilët punojne ne makina të ndryshme ne të njëjtën kohe. Qellimi i tyre është të befne sa me shume përpjekje ne sekonde. Ne vend qe të kryejne ping ne makina sipas IP adresave të rregullta, crackers mund të kapin një kompani specifike.

Për të gjetur se cilen IP adresë përdor për shembull University of Foobar ne *foobar.edu* duhet të shkruaje:

dnsquery foobar.edu

dhe me pas ai do të marre një listë me IP adresat e tyre. Duke pasur 65,536 IP adresa dhe duke njobur 2 byte-et e pare të një IP adresë të një kompanie është e lehtë të kryesh ping ne te gjithe 65,536 adresat për të pare se cila përgjigjet dhe cila pranon lidhje. Një cracker me një linjë të shpejtë ADSL mund të programoje programe për të thyer sisteme qe punojne pa nderprerje.

Jo vetëm password-et e përdoruesve jane të thyeshme por edhe root password mund të thyhen lehtë. Ne vecanti disa instalime nuk shqetësohen për të ndryshuar password-in default me të cilin sistemet jane pajisur. Një astronom nga Berkeley, Cliff Stoll, vezhgoi parregullsi ne sistemin e tij dhe ngriti një kurth për cracker-in i cili u mundua të thyente sistemin e tij. Ai vezhgoi një skeme të cilen cracker-i e kishte shkruar. Me ketë skeme cracker-i kishte thyer sistemin e Lawrence Berkeley Laboratory (LBL) dhe po e përdorte për të thyer sisteme të tjera si ai i departamentit të energjisë ne U.S.

```
LBL> telnet elksi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Figura 9-5: Skema si cracker theu sistemin e dep.të energjise U.S.

Është e mundur qe të përdoret një software i cili quhet, Packet sniffer i cili mbikeqyr të gjitha paketat hyrese dhe dalese ne rrjet për të kerkuar rreth një modeli të vecantë. Një model interesant dhe special është kerkimi për persona të cilët realizojne log-in ne makina ne distance. Ky informacion mund të dergohet ne një file ku cracker-i e përzgjedh me vone. Ne ketë menyre një cracker i cili thyen një sistem me siguri të dobet mund ta përdore ketë ne thyerjen e sistemeve me siguri me të lartë. Shume thyerje realizohen nga përdorues të cilët përdorin script-e qe gjenden ne internet. Keto script-e përdorin sulme të forta ose përdorin bug-e të njojur ne programe specifike. Hackers-at e vertetë u referohen si script kiddies.

Zakonisht një script kiddy nuk ka ndonjë plan. Ai vetëm kontrollon për makina të cilat jane të lehta për tu thyer. Disa script-e zgjedhin rastesisht një rrjet për ta sulmuar, duke përdorur një numer rrjeti të rastesishem (ne pjesen me të sipërme të IP adres). Me pas kontrollojne të gjitha makinat e rrjetit për të pare se cila mund të përgjigjet.

UNIX Password Security

Disa sisteme të vjetër operative i mbajne filet password ne disk ne një forme jo te enkriptuar, por te mbrojtur nga një sistem mbrojtës me mekanizem të zakonshem. Duke i pasur keto password-e ne disk ne forme jo te enkriptuar përbën një rrezik sepse shume persona kane akses mbi to. Kjo mund të përfshije administratoret e sisemit, operatoret e makinave, personelin, menaxheret dhe madje dhe sekretaret. Një zgjidhje e mire e cila përdoret ne UNIX punon si me poshtë. Programi log-in pyet përdoruesin për të shkruar emrin dhe password-in. Password-i menjehere enkriptohet duke e përdorur atë për të enkriptuar një bllok të dhenash. Me pas programi log-in lexon password file i cili është një seri linjash ASCII, një, për një përdorues, derisa gjen linjën qe i përket emrit të përdoruesit. Ne qoftë se password-i i enkriptuar përmban ne linjë password-in e sapo llogaritur, log-in pranohet. Ne të kundert ajo refuzohet. Kjo skeme ka një avantazh sepse askush, qoftë dhe super përdorues, nuk mund të shoh password-in e ndonjë përdoruesi pasi keto nuk regjistrohen ne një form jo te enkriptuar ne sistem. Megjithatë dhe kjo skeme mund të sulmohet. Një cracker ka ndertuar një fjalor password-esh të ngjashhem me atë të Morris dhe Thompson. Keto password-e mund të enkriptohen duke përdorur algoritma të njojur. Nuk ka rendesi se sa zgjat ky proces sepse ai kryhet gjatë thyerjes se sistemit. Tani të pajisur me një listë të ciftit password, password i enkriptuar, crackers sulmjoine. Ai lexon password file dhe nxjerr të gjitha password-et e enkriptuar. Keto krahasohen me password-et e enkriptuara me pare nga cracker dhe keshtu emri log-in dhe password-i i enkriptuar tashme jane të njojur. Një shell script i thjeshtë mund të automatizoje ketë proces. Duke njojur ketë mundesi Morris dhe Thompson përshkruan një teknike e cila e bënte sulmin pothuajse të pa përdorshem. Ideja e tyre ishte qe të bashkonin një numer të rastesishem prej n-bit, i quajtur salt, me password-in. Ky numer i rastesishem ndryshon sa here qe ndryshon password-i.

Ky numer vendoset ne file-in password ne forme të enkriptuar, keshtu qe kushdo mund ta lexoje atë. Ne vend të vendosjes se password-it të enkriptuar ne file-in password,

password-i dhe numri i rastesishem me pare lidhen pastaj enkriptohen se bashku. Rezultati i enkriptimit ruhet ne file-in password, si ne fig 9-6. Secili përdorues ka një linjë ne file, me tre hyrje të ndara me presje: emri log-in, salt dhe password+salt i enkriptuar. eDog4238 paraqet rezultatin e password-it të lidhur të Bobb. Ky është rezultati i enkriptuar dhe qe ruhet ne fushen e tretë të Boob entry. Tani marim ne konsiderate rastin kur një cracker kerkon të ndertoje një listë me password-e, ti enkriptoje dhe ti ruajne ato ne një lloj file, keshtu qe çdo password i enkriptuar mind të shihet lehtësisht.

Bobbie, 4238, e(Dog4238)
Tony, 2916, e(6%%TaeFF2918)
Laura, 6902, e(Shakespeare6902)
Mark, 1694, e(XaB@Bwcz1694)
Deborah, 1092, e(LordByron,1092)

Figura 9-6, Përdorimi i salt për të mbrojtur implikimin e password-eve të enkriptuar.

Përmirsimi i sigurise se password-it

Edhe pse pajisja e password-it me salt mbronte sistemet nga cracker, kjo nuk funksinonte për David i cili kishte si paswword David. Një cracker mund të provoje si mundesi pikerisht emrin dhe ta gjeje menjehere password-in. Disa kompjutera kane një program qe gjeneron fjale të cilat shqiptohen lehtë por nuk kane kuptim të cilat mund të përdoren si password. Kriteret qe plotëson një password i mire:

- 1.Password duhet të ketë minimumi shtatë karaktere
- 2.Password duhet të ketë shkronja të medha dhe të vogla
- 3.Password duhet të ketë të paktën një shifer ose një karakter special.
- 4.Password nuk duhet të ketë fjale direkrorish, emra njerezish, etj.

One-Time Password

Forma me ekstreme per ndryshimin e passwordit eshte, **one-time password**.

Kur përdoren one-time password përdoruesi merr një liber i cili përban një listë me password-e. Secili log-in përdor password-in pasardhes ne listë. Ne qoftë se zbulohet një password, nuk ka ndonjë problem pasi heres tjetër duhet të përdoret një password tjetër. Sugjerohet qe përdoruesi nuk duhet ta humbase librin e password-eve. Ky liber nuk nevojitet ne skemen e propozuar nga Leslie Lamport, e cila lejonte një përdorues të log-ohet ne një rrjet të sigurtë duke përdorur one-time password. Kjo metode përdoret për të

lejuar përdoruesit të logohen ne një server ne internet edhe pse keqberesit mund të shohin dhe kopojne trafikun.

Algoritmi bazohet ne një funksion $y = f(x)$ i cili ka vecorine qe kur jepet x mund të gjendet lehtë y por dhenia e y nuk bën të mundur gjetjen e x . Hyrja dhe dalja duhet të jene të të njëjtës gjatësi, për shembull 128 bit. Përdoruesi zgjedh një password sekret të cilin e memorizon. Ai gjithashtu zgjedh dhe një integer n , i cili tregon se sa one-time passwords është i aftë të gjeneroje algoritmi.

Shembull:

Konsiderojme $n = 4$, ne qoftë se passwordi sekret është s , passwordi i pare jepet duke perdonur funksionin n here: $P_1 = f(f(f(f(s))))$

Password-i i dytë jepet duke përdorur funksionin $n-1$ here: $P_2 = f(f(f(s)))$

Ajo qe vlen të theksohet ketu është se duke dhene çdo password si një sekuence, është e lehtë të llogarisesh një password të meparshem ne një sekuence numerike por është e pamundur të llogarisesh password-in e ardhshem. Kur përdoruesi deshiron të logohet për here të pare, ai dergon emrin log-in ne server, i cili i përgjigjet duke i derguar një integer ne file-in password L. Makina e përdoruesit përgjigjet me Pi, i cili llogaritet lokalisht se nga cila shtypet ne spot. Me pas serveri llogarit A^i dhe e krahason me vleren e vendosur ne file-in password.

Në qoftë se vlerat përputhen, og-in pranohet, integeri rritet ne 2, F1 kalon P ne file-in password. Ne logimin tjetër serveri i dergon përdoruesit integerin 2 dhe makina e përdoruesit llogarit V. Serveri llogarit Pi dhe e krahason me hyrjen ne file-in password. Në qoftë se vlerat përputhen logimi pranohet, integeri kalon ne 3 dhe F2 kalon P ne file-in password, e keshtu me rradhe.

Kur të gjitha n password-et jane përdorur, serveri i ri inicializon një secret key të ri.

Vertetimi Challenge-Response

Një variant ne idene e passwordeve është qe të mundeshosh një listë të gjatë pyetjesh dhe përgjigjesh të cilat gjenerohen nga përdoruesit dhe vendosen ne server ne forme të enkriptuar. Pyetjet duhet të zgjidhen keshtu qe përdoruesit nuk kane pse ti shkruajne ato. Pyetje të mundeshme jane:

1. Kush është motra e Marjolein?
2. Ne cilen rruge ndodhej shkolla jote fillore?
3. Cilen lende jepte Woroboff?

Gjatë log-in serveri pyet njëren prej tyre rastesisht dhe kontrollon përgjigjen. Për ta bere me praktike nevojiten shume pyetje-përgjigje. Një variant tjetër është **challenge-response**. Kur përdoret kjo metode, gjatë lidhjes përdoruesi zgjedh një algoritem, për shembull x^2 . Kur përdoruesi logohet, serveri i dergon përdoruesit një argument, për shembull 7, përdoruesi shtyp 49. Algoritmi mund të jetë i ndryshem ne mengjes apo ne darke, ne ditë të ndryshme të javes, e keshtu me rradhe.

Ne qoftë se terminali i përdoruesit ka fuqi llogaritëse, si PC, mund të përdoret një forme me e fuqishme challenge-response. Përdoruesi zgjedh një secret key , k, i cili sillet nga sistemi i serverit. Një kopje e tij cohet ne kompjuterin e përdoruesit. Ne momentin e logimit, serveri dergon një numer të rastesishem ne kompjuterin e përdoruesit i cili llogarit $f(rt)$ dhe e kthen mbrapa atë. Me pas serveri e bën vet llogaritjen dhe kontrollon në qoftë se rezultati i kthyer përputhet me llogaritjet. Avantazh i kesaj skeme është se në qoftë se një përgjues shikon dhe regjistron të gjithe trafikun ne të dy drejtimet, ai nuk do të mesoje gje e cila mund ta ndihmoje heren tjetër.

9.3.2 Verifikimi duke përdorur një objekt fizik

Metoda e dytë për verifikimin e përdoruesit është të kontrollosh për objekt fizik të ciu përdoruesit e kane, me mire se sa dicka qe ata dine. Celsa e dyerve Jane përdorur për shekuj për ketë qellim. Ne ditët e sotme objekti fizik i cili përdoret shpesh është karta plastike. Kjo kartë vendoset ne një lexues i cili shoqerohet me një terminal ose kompjuter. Përdoruesi jo vetëm qe duhet të vendose kartën por duhet të shtype dhe password-in. Kjo behet për të shmangur përdorimin e kartës kur ajo humbet ose vidhet. Nga ky kendveshtrim, përdorimi i një ATM-je fillon me logimin e përdoruesit ne kompjuterin e bankes nepërmjet një terminali duke përrdorur një kartë dhe një password .

Kartat plastike ndahen ne dy kategori:

- 1.kartat me shirita magnetik
- 2.kartat me chip

Kartat me shirita magnetike mbajne deri ne 140 byte informacion, i cili shkruhet ne një shirit magnetik i mbyllur nga mbrapa kartës. Ky informacion mund të lexohet nga një terminal dhe të dergohet ne kompjuterin qendror. Informacioni përmban password-in e përdoruesit duke bere të mundur qe terminali të identifikoje përdoruesin edhe ne qoftë se lidhja me kompjuterin qendror nuk egziston.

Kartat magnetike nuk Jane tepër të sigurta sepse pajisjet për lexim/shkrimin e tyre Jane të lira dhe të përhapura.

Kartat me chip përbajne një qark të integruar (chip). Keto karta mund të ndahen ne dy nenkategori:

2. a) kartat e ruajtjes se vleres
2. b) smart kard

Kartat e ruajtjes se vleres permbajne një sasi të kufizuar memorieje (me pak se 1kB) duke përdorur teknologjine EEPROM, e cila lejon të ruhet një vlere kur karta hiqet nga lexuesi. Nuk kemi CPU ne kartë keshtu qe vlera e regjistruar mund të ndryshohet nga një CPU e jashtme e cila ndodhet ne lexues. Keto karta zakonisht përdoren si karta telefonike me

parapagese. Kur një thirrje kryhet, telefoni dekrementon vleren e kartës. Për ketë arsyé keto karta prodhohen nga një kompani për tu përdorur vetëm ne makinat e asaj.

Ne ditët e sotme pune me e sigurt është realizuar me smart cards të cilat kane një CPU 8 bit 4MHz, 16 KB ROM, 4KBEEPROM, 512 B RAM dhe një kanal komunikimi me lexuesin 9600 bps. Smart cards përdoren për të mbajtur parate si kartat e ruajtjes se vleres, por me siguri me të lartë. Kartat mund të mbushen me leke ne ATM ose ne shtëpi duke përdorur një lexues të aprovuar nga banka.

Kur futet karta ne një tip lexuesi, përdoruesi mund të vertetoje kartën për të térhequr një sasi parash nga karta duke detyruar kartën të dergoje një mesazh të enkriptuar.

Ky tip lexuesi kthen ketë mesazh ne banke për të kredituar sasine e parave të paguara. Një avantazh i madh për smart cards është se nuk kerkojne një lidhje on line me banken. Keto karta jane përhapur kudo. Ato kane dhe përdorime të tjera potenciale. Interesi jone ketu eshtë se si ato përdoren për vertetim të logimit të sigurt. Koncepti baze është i thjeshtë; smart card është një kompjuter i vogel, i sigurtë i cili mund të komunikojë me kompjuterin qendror. Variante të ndryshme vertetimi përdoren për smart cards. Një challenge-response i thjeshtë mund të punoje: serveri dergon një numer të rastesishem 512 bit ne smart card, e cila shton dhe password-in e përdoruesit 512 bit qe ndodhet ne EEPROM. Shuma me pas ngrihet ne katorr dhe 512 bitet e mesit dergohen mbrapsht ne server, i cili di password-in e përdoruesit dhe mund të llogarise kur rezultati është i saktë dhe kur jo.

Sekuенca tregohet ne Fig. 9-7. Ne qoftë se një përgjues shikon të dy mesazhet, ai nuk është ne gjendje të kuptoje apo të regjistroje informacion për të ardhmen sepse heres tjetër një numer i rastesishem dhe i ndryshem 512bit do të dergohet. Një disavantazh i protokollit kriptografik (i fshehur grafikisht) fiks është se pas një fare kohe ai mund të thyhet, duke riprodhuar karat e papërdorshme. Një rruge për të anashkaluar ketë disavantazh është përdorimi i ROM ne kartë jo për protokoll kriptografik por për interpretuesin Java. Protokolli real i kriptogrofuar me pas shkarkohet ne kartë si një program Java dhe ve ne pune interpretuesin.

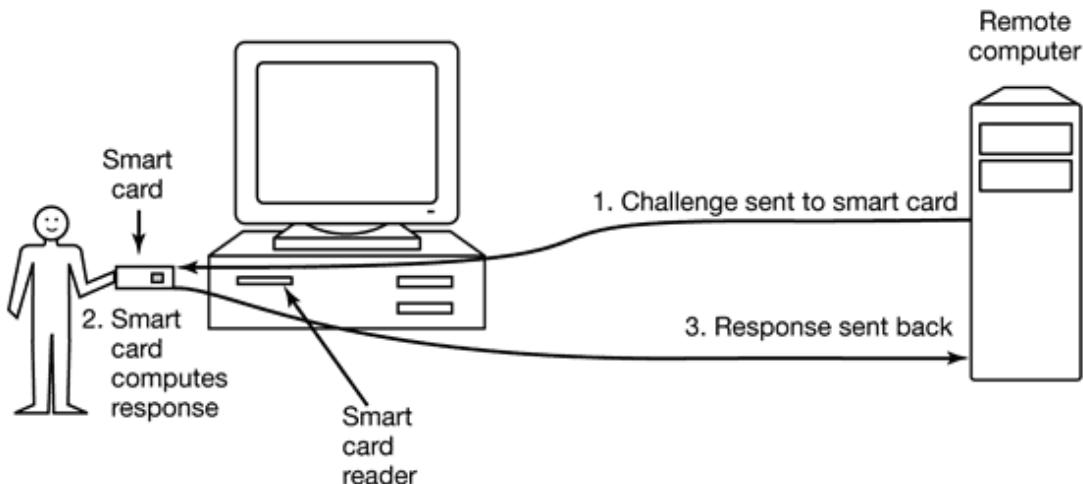


Figura 9-7. Përdoruesi i smart card për vertetim.

Ne ketë menyre sapo të thyet një protokoll, një protokoll i ri mund të instalohet kudo ne botë menjehere. Një disavantazh i kesaj përqasje është se kartat behen me të ngadalshme, por me rritjen e teknologjise kjo metode është fleksible. Një disavantazh tjetër i smart card është se humbja apo vjedhja behet subjekt për analiza sulmi.

9.3.3 Vertetimi duke përdorur Biometrics

Metoda e tretë e vertetimit mat karakteristikat fizike të përdoruesit, të cilat jane të veshtira e imitohen. Keto karakteristika quhen **biometric** (Pankanti et al., 2000). Per shembull, një lexues fingerprint ose voiceprint ne një terminal mund të vertetoje identitetin e përdoruesit .

Një sistem biometrik ka dy pjese: registruesin dhe identifikuesin. Gjatë regjistrimit karakteristikat e përdoruesit maten dhe rezultati dixhitalizohet. Me pas tiparet e rendesishme regjistrohen ne një rekord i cili i bashkangjitet përdoruesit. Rekordi mund të mbahet ne një data baze qendrore ose mund të ruhet ne një smart card. Pjesa tjetër është identifikuesi. Përdoruesi del ne pah dhe gjeneron një emer log-in. Me pas sistemi bën llogaritjet përseni. Në qoftë se përputhen vlerat, logimi pranohet ne të kundert ai refuzohet. Emri log-in nevojitet sepse llogaritjet nuk jane ekzakte keshtu qe është e veshtire ti indeksosh përdoruesit pastaj të kerkosh indeksin.

Karakteristikat e zgjedhura duhet të kene ndryshueshmeri të mjaftueshme qe sistemi të dalloje midis shume personave pa bere gabime. Gjithashtu karakteristikat nuk duhet të ndryshojne vazhdimesht. Le të shikojme disa karakteristika biometrike qe përdoren sot. Analiza e gjatësise se gishtit është tepër praktike. Kur përdoret kjo teknike secili terminal ka një pajisje si ne Fig. 9-8. Përdoruesi vendos doren e tij ne të dhe gjatësia e gishtave të tij matet dhe kontrollohet ne data baze.

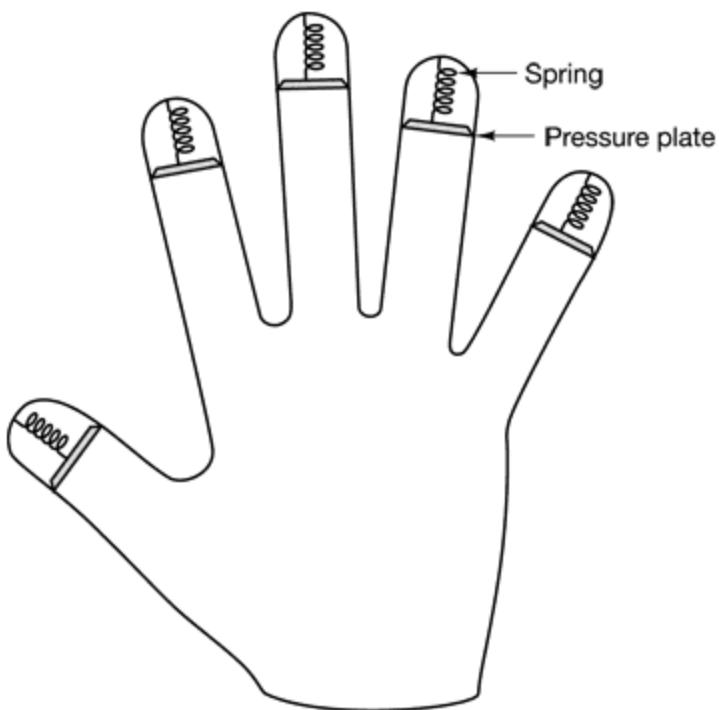


Figura 9-8, Një pajisje për matjen e gjatësise se gishtave.

Edhe gjatësia e gishtave nuk është përfekte. Sistemi mund të sulmohet duke përdorur modele duarsh prej allcie. Një karakteristikë tjetër biometrike është analiza e retines se syrit. Çdo person ka një model të ndryshem të eneve retinale të gjakut, madje dhe për binjaket. Ata mund të fotografohen nga një kamera 1 metër larg subjekti. Sasia e informacionit ne skanimin retinal është me i madh se ne fingerprint dhe mund të kodohet ne 256 byte.

Një teknike disi ndryshe është dhe ajo e analyzes se firmes (nenshkrimit). Përdoruesi firmos emrin e tij me një stilolaps special të lidhur me terminalin dhe kompjuteri e krahason atë me një model të njobur qe ruhet ne një smart card online.

Nuk është me rendesi krahasimi i firmes, por krahasohet levizja e stilolapsit dhe shtypja qe kryhet gjatë shkrimit. Një falsifikues i mire mund të jetë ne gjendje të kopjoje firmen, por nuk mund të ketë te dhena për shtypjet qe behen apo për shpejtësine e shtypjes.

Një metode qe mbështet ne minimun një hardware special është zeri. Gjithshka qe nevojitet është një mikrofon dhe pjesa tjetër është software. Ne kontrast me sistemin e identifikimit të zerit, i cili përpinqet të zbulojë cfare po thotë përdoruesi, ky sistem përpinqet të përcakoje cili është ai qe po flet. Disa sisteme mbështesin qe përdoruesi të thotë një password sekret, por kjo mund të anullohet nga një përgjues i cili regjistrон password-et dhe i përdor me vone ato. Disa sisteme të avancuara thone dicka dhe i kerkojne përdoruesit ta përserise, duke përdorur tekste të ndryshem për çdo log-in.

9.3.4 Kundermasat

Instalimet kompjuterike qe kane nevoje për siguri, zakonisht ndjekin hapa për të kryer vertetimin. Për shembull një kompani duhet të ketë një linjë veprimi e cila lejon punonjësit të logohen nga ora 8 A.M deri ne oren 5 P.M, nga e hena ne të premte. Një prove për tu loguar ne ndonjë orar tjetër do të konsiderohet si përpjekje për të thyer sistemin. Linjat telefonike Dail-up mund të behen të sigurta duke ndjekur rrugen me poshtë:

Kushdo mund të telefonoje dhe të logohet, por pas një logimi të suksesshem, sistemi menjehere nderpret lidhjen dhe therret përdoruesin ne një numer të rene dakort. Kjo mase tregon se një sulmues nuk mund të provoje të thyeje sistemin nga një linjë telefonike. Gjatë ngjarjeve me apo pa thirrje, sistemi kerkon të paktën 5 sekonda të kontrolloje password-in e shkruar ne linjën dail-up dhe duhet ta rrise ketë interval pas disa përpjekjeve të njëpasnjëshme të pasuksesshme për log-in.

Të gjithe log-in duhet të regjistrohen. Kur një përdorues logohet, sistemi duhet të njoftojë kohen dhe terminalin e logimit të meparshem, keshtu qe sistemi mund të zbulojë thyerjet e mundshme. Hapi tjetër është joshja drejt grackes për të kapur ata qe nderhyjne.

Një metode është të përdoresh një emer log-in special dhe një password të thjeshtë. Sa here qe dikush logohet duke përdorur keto të dhena, njoftohen specialistët e sigurise se sistemit. Të gjitha komandat e shkruara nga thyesi jepen ne një monitor të sigurt dhe shohim ekzaktësisht se cfare është duke bere thyesi.

9.4 SULMET NGA BRENDË SISTEMIT

Një cracker i logg-uar ne një kompjuter mund të shkaktoje demtime. Ne qoftë se kompjuteri ka një siguri të mire ai vetëm mund të demtoje përdoruesin sistemi i të cilit u thye. Por kjo thyerje mund të sjelle thyerje të tjera.

9.4.1 Trojan Horses

Një sulm i brendshem jane dhe kuajt e trojes (Trojan Horses), të cilët pajisin programet me kode të cilët kryejne pune të pa pritshme dhe të pa deshirua nga ne. Ky funksion mund të jetë modifikim, shuarje ose enkriptim i file-eve të perdoruesit. Për të vene ne pune një trojan programuesi i tij duhet të ketë programin qe do të ekzekutohet. Një rruge e shperndarjes se trojan është interneti ku ka lojra, mp3, etj. Kur vepron një trojan, thirret mje procedure e cila mund të beje çdo gje qe bën edhe përdoruesi si: fshin file, hap lidhjet e rrjetit, etj.

Ky marifet nuk përfshin autorin e trojanit i cili thyen sistemin. Ka rruge të tjera për të dredhuar viktimen ne një trojan të ekzekutueshem. Shume përdorues UNIX kane një variabel, \$PATH, i cili kontrollon se cilat direktori janë kerkuar për një komande. Kjo mund të shfaqet duke dhene ne shell komanden :

```
echo $PATH
```

Një setting i mundshem për përdoruesin *ast* ne një sistem të vecantë konsiston ne direktoritë :

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/binyX 11 :/usr/ucb:/usr/man\  
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Përdoruesit e tjere kane rruge të ndryshme kerkimi. Kur përdoruesi shkruan *prog* ne shell, shell-i ne fillim shikon ne qoftë se ka ndonjë program të emertuar */usr/ast/bin/prog*. Ne qoftë se po ai ekzekutohet. Ne qoftë se jo shell-i provon */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog*, e keshtu me radhe, duke provuar të gjitha direktoritë. Supozojme sikur njëra nga direktoritë të jetë e pambrojtur. Cracker ka mundesi qe të vendose programin e tij ne ketë direktori. Ne qoftë se kjo është ndodhja e pare e programit ne listë, ai do të ekzekutohet dhe trojan-i do të veproje. Programet me të zakonshem ndodhen ne */bin* ose */usr/bin*, keshtu vendosja e trojan-it ne */usr/bin/XH/ls* nuk sjell demtim për një program pasi ne fillim duhet të gjendet programi specifik.

Megjithëtë supozojme se një cracker vendos *la* ne */usr/bin/XIL*. Ne qoftë se përdoruesi shkruan *la* ne vend të *ls* tani fillon të veproje trojan-i duke kryer korruptim të sistemit dhe me pas shfaq mesazhin se *la* nuk ekziston. Duke futur trojan ne direktori të komplikuara, të cilat zor se shihen nga përdoruesit dhe duke u dhene atyre emra qe mund të prezantonje gabime shkrimi të zakonshme, përbën një shans të favorshem qe dikush ti therrase ato heret a vone. Ky dikush mund të jetë super përdorues duke bere qe trojanet të kene mundesine të zevendesojne *Mn/h* me një version qe përmban një kale troje. Një përdorues keqberes por legal, Mai, mund të instaloje një trap për një super përdorues, si me poshtë.

Ai vendos një version të *h* i cili përmban një trojan ne direktorine e tij dhe bën dicka të dyshimtë e cila është e sigurt qe do të sulmoje super përdoruesin, dicka si fillimi i 100 compute-bound proceseve ne të njëjtën kohe. Shanset Jane qe super përdoruesi do ta zgjedhe trojanin, për të pare cfare Mai ka ne direktorine e tij duke shkruar:

```
cd /usr/mal  
Is-I
```

Disa shell provojne direktorine lokale përparrë se të punojne ne *\$PATH*. Super përdoruesi mund të përfshije trojan-in e Mai-it me fuqitë e super përdoruesit. Trojnan-i mund të kryeje */usr/mal/bin/sh* SETUID root. Ajo qe i nevojitet Jane dy thirrje sistem: chown për të ndryshuar përkatësine e */txr/mal/bin/sh* ne root dhe chmodl për të vendosur bitin e tij SETUID. Tani Mai mund të behet super përdorues. Ne qoftë se Mai gjen veten **frequently short of cash**, ai mund të përdore një nga trojan-et për të ndihmuar **his liquidity position**. Trojani kerkon për të pare ne se viktima ka një program bankar online të instaluar, si *Quicken*.

Ne qoftë se po, trojan-i drejton programin të transferoje shuma të hollash nga llogaria e viktimes ne një llogari false. Me pas trojan-i ul zerin e modemit, telefonon një numer pagese. Ne qoftë se përdoruesi është online kur trojan-i filloj, përdoruesi nuk do të kuptoje dhe mund të qendroje ne linjë për ore të tëra.

9.4.2 Login i rreme

Dicka qe mbështëtët nga trojanet është **login spoofing**. Kjo metode punon si vijon: Normalisht kur askush nuk logohet ne një terminal UNIX ose ne LAN, shfaqet pamja si ne Figuren 9-9(a). Kur një përdorues shkruan emrin login sistemi e pyet për password-in. Në qoftë se eshtë korrekt, përdoruesi logohet dhe fillon punen shell-i. Supozojme se Mai shkruan një program për të shfaqur pamjen e Fig. 9-9(b). Ajo duket si pamja ne Fig. 9-9(a), përvèc se nuk është një program logimi ne veprim por një program i rreme.



Figura 9-9, (a) Logim korrekt, (b) logim i rreme

Kur përdoruesi shkruan emrin login, programi i përgjigjet duke e pyetur për password-in dhe caktivizon përsritjen. Pas mbledhjes se emrit login dhe password-it, ato shkruhen ne një file dhe programi i rreme dergon një sinjal për të vrare shell-in e tij.

Ky veprim logon out Mai dhe zhvendos programin real për të filluar dhe shfaqet pamja si ne Fig. 9-9(a). Përdoruesi supozon se Mai ka bere një gabim drejtshkrimor dhe logohet përseni. Ketë rradhe do të funksionoje. Por nderkohe Mai merr një cift të ri emer logimi, password. Duke u loguar ne shume terminale dhe duke startuar login spoofer ne të gjithe terminalet ai mund të mbledhe të gjithe password-et. E vetmja menyre e vertetë për tu mbrojtur nga kjo është të një sekunce logimi e cila fillon me një celes kombinimi të cilin përdoruesit nuk e arrijne dot. Windows 2000 përdor CTRL-ALT-DEL për ketë qellim. Ne qoftë se një përdorues jep ne terminal komanden CTRL-ALT-DEL, përdoruesi korrent logg-out dhe programi sistem i logimit fillon punen. Nuk ka asnje menyre qe të kalohet ky mekanizem.

9.4.3 Bombat logjike

Një tjetër sulm nga brenda sistemit ne ketë kohe është dhe bomba logjike. Kjo pajisje është një pjese kodit e shkuar nga një programues i një kompanie dhe i vendosur ne menyre të fsheht ne sistemin operativ të prodhuar. Për aq kohe sa programuesi e ushqen me daily password, kodi nuk bën asgje.

Megjithatë, ne qoftë se programuesi nxehet dhe heq dore nga premisat e kodit pa e vrare mendjen, ditën tjetër apo javen tjetër bomba logjike humbet rrezikshmerine e saj. Shume variante të kesaj skeme jane të mundshme. Një ceshtje e famshme e bombave logjike lidhet me pagat. Ne qoftë se numri personal nuk shfaqet ne të për dy perioda pagash të njëpasnjëshme, bomba **went off**. **Going off** mund të përfshije pastrimin e diskut, shuarjen e file-eve të rastesishem.

Ne ceshtjen tjetër kompania kryen një zgjidhje të ashpër rreth thirrjes se police-s (I cili mund ose jo të rezultoje shume muaj me vone por nuk i rivendos file-at e humbur).

9.4.4 Trap Doors

Një tjetër boshlek ne siguri i cili shkaktohet nga brenda është dhe **trap door**. Ky problem krijohet nga futja e kodit ne sistem nga një programues sistemi për të anashkaluar disa kontolle. Për shembull, një programues mund të shfoje kod ne programin e logimit për të pranuar dike të logohet me emrin "zzzzz" dhe s'kemi probleme me password-in. Kodi normal ne programin e logimit mund të jetë si ne fig. 9-10(a). Trap door do të jetë si ne fig. 9-10(b).

Thirrja e *stremp* kontrollon në qoftë se emri i logimit është zzzzz. Në qoftë se po logimi është i suksesshem. Në qoftë se kodi i trap door futet nga një programues i cili punon për një fabrike kompjuterash, programuesi mund të logohet ne të gjithe kompjuterat e prodhuar nga ajo kompani. Trap door anashkalon të gjithe procesin e vertetimit.

```
while (TRUE) {
    printf("login: ");
    get_string(name);
    disable_echoing();
    printf("password: ");
    get_string(password);
    enable_echoing();
    v = check_validity(name,
password);
    if (v) break;
}
execute_shell(name);

while (TRUE) {
    printf("login: ");
    get_string(name);
    disable_echoing();
    printf("password: ");
    get_string(password);
    enable_echoing();
    v = check_validity(name,
password);
    if (v || strcmp(name, "zzzzz") ==
0) break;
}
execute_shell(name);
```

Figura 9-10. (a) Kod normal (b) Kod me trap door

Një menyre me të cilën kompanitë parandalojnë trap doors është të kesh **code review**. Me ketë teknike programuesi mbaron se shkruari kodin dhe testimini e moduleve, modulet kontrollohen nga një kod database. Periodikisht programuesit ne skuader paraqesin kodet qe kane shkruar duke treguar dhe funksionet qe kryejne keto kode, rresht pas rreshti. Kjo rrit mundesine e kapjes se rreshtave ku mund të kemi trap door.

9.4.5 Buffer Overflow

Kapja e burimeve nga sulmet lidhet me faktin qe virtualisht të gjithe sistemet operative dhe shumica e programeve sistem shkruhen ne gjuhen C. Fatkeqsisht asnjë kompilues C **does array bounds checking**. Sekuenca e kodit nuk është illegal, nuk kontrollohet.

```
int i;  
char c[1024];  
i = 12000;
```

$c[i] = 0;$

Ne figuren 9-11(a), shohim një program ne funksionim, me variabla lokale ne stack. Ne disa pika programi therret proceduren A, si ne figuren 9-11(b). Sekuensa standarte e thirrjes nderpritet duke kthyer adresen ne stack. Me pas kontrolli transferohet ne A, i cili dekrementon stack pointerin për të lokalizuar variablat lokale.

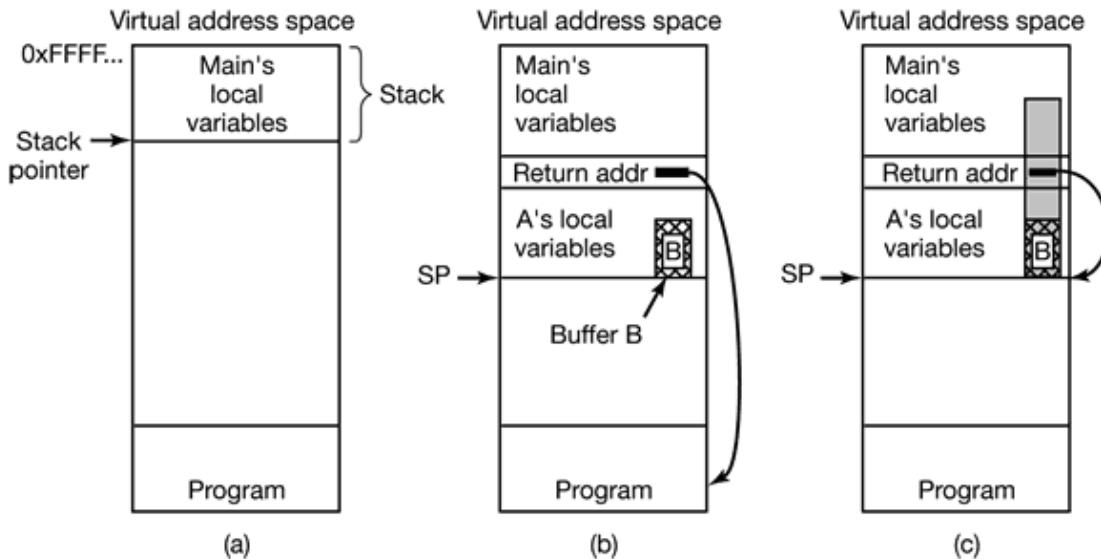


Figura 9-11. (a) Programi ne funksionim, (b) Pasi thirret procedura A,(c) Buffer overflow

Supozojme se puna A përfshin file path-in e plotë dhe me pas hapet ose mund të modifikohet. A ka një buffer me madhesi fikse B, për të mbajtur emrin e file-it, si ne fig.9-11(b). Duke përdorur një buffer me madhesi fikse për të mbajtur file name është me e thjeshtë për tu programuar sesa të përcaktohet për here të pare madhesia aktuale dhe me pas të alokohet dinamikisht memoria e mjaftueshme. Kur procedura kopjon file name ne buffer, emri tejkalon bufferin dhe mbishkruan memorien ne zonen gri si ne fig.9-11(c). Në qoftë se file name është i mjaftueshem, ai gjithashtu mbishkruan adresen e kthimit, keshtu qe kur kthehet A, adresa e kthimit meret nga mesi i file name. Në qoftë se adresa është random, programi do të kerceje ne adresen random dhe mund të krijoje konflikt me disa instruksione.

Po në qoftë se file name nuk përmnjaje adresa të rastesishme? Po ne qoftë se përban një program binar?

Ajo qe mund të ndodhe është se kur Programi kthen A, tani B fillon ekzekutimin. Një mashtrim i njëjtë punon jo vetëm me file name. Ajo punon me stringa shume të gjata, user input ose me dicka ku programuesi ka krijuar një buffer me madhesi fikse. Funksioni gets i librarise C lexon një stringe nga një buffer por pa kontroll overflow duke u bere subjekt sulmesh.

Disa kompilues përcaktojnë përdorimin e gets dhe shqetësohen për të. Tani vjen pjesa me e keqe. Supozojme qe programi qe do te sulmohet është SETUID root ne UNIX. Kodi i futur mund të beje një cift thirrjes sistem për të konvertuar shell-in e sulmuesit ne disk ne STUID root, keshtu qe kur të ekzkuohet ai do të ketë fuqi superuser. Tani mund

të realizohet map ne një file special të përgatitur ne një librari shared e cila mund të kryeje çdo lloj demi. Ose mund të ekzekutoje një thirrje sistem për mbuluar programin korent me shell, duke krijuar një shell me fuqi superuser. Një fraksion substancial i problemeve të sigurise lidhen me ketë gje dhe është e veshtire të fiksosh keto gabime sepse jane shume programe C ne përdorim. Përcaktimi se një program ka probleme buffer overflow është i thjeshtë. Në qoftë se kodi burim është i vlefshem sulmi është gjithmone i thjeshtë sepse layout-i i stack-ut njihet gjatë ecurise. Nga sulmi mund të mbrohem duke duke fiksuar kodin.

9.4.6 Sulmet e përgjithshem ndaj sigurise

Rruja e sotme e testimit të sigurise se sistemit është punesh një grup eksperti të cilët njihen si **tiger teams** për të pare në qoftë se thyejne dot sistemin. Ne shume vite pune keto skuadra kane zbuluar një numer fushash ne të cilat sistemet jane të dobeta. Me poshtë jepet një listë sulmesh të zakonshme të cilat shpesh rezultojnë të sigurta ne thyerjen e sistemeve.

1. Kërkesa e faqeve ne memorie, hapesira e diskut dhe leximi i tyre. Shume sisteme nuk i shuanje ato përparrë se ti lokalizojne. Keto mund të kene shume informacion me interes të shkruar nga dikush qe i zotëronte me përparrë.
2. Kryerja e thirrjeve sistem ilegale ose legale me parametra ilegale apo thirrjet sistem legale me parametra legale por të pa arsyeshem. Shume sisteme mund të behen konfuze.
3. Startimi i logimit dhe shtypja e DEL, RUBOUT ose BREAK gjate sekunes se logimit. Ne disa sisteme programi i kontrollit të passwordit do të vritet dhe logimi mund të konsiderohet i sukseshem.
4. Prova për të modifikuar struktura komplekse të sistemeve operative të cilat mbahen ne user space. Ne disa sisteme për të hapur një file, programi nderton një strukture të madhe të dhenash qe përmban emrin e file-it dhe parametrat e tjere të cilët ja kalon sistemit. Gjatë kohes kur file-i lexohet apo shkruhet, sistemi ndonjehere update-on vetë strukturen. Ndryshimi i ketyre fushave mund të shkatërrje sigurine.
5. Shiko për manuale qe thone“ Mos e bez X gje ”. Provo sa me shume variante të X qe të jetë e mundur.
6. Bind programuesin e sistemit të shtoje një trap door duke anashkaluar kontrollet e sigurise për përdoruesit me emrin tuaj të logimit.
7. Depërtuesi mund të gjeje sekretarine e administratorit të sistemit dhe të paraqitet si një përdorues i dobet i cili ka harruar password-in dhe i duhet shpejt. Sekretaria ka një akses mbi të gjitha llojet e informacionit.

9.4.7 Të metat e famshme të sigurise

Ashtu si industria e transportit kishte deshtimin ne Titanic dhe ndertuesit e sistemeve operative, gjithashtu kane disa gjera të cilat nuk i marrin parasysh.

Të meta e famshme ne UNIX

Sherbimi *Ipr, ne UNIX*, i cili printon një file ne printer ka si mundesi të levize file-in pas printimit. Ne versionet e para ishte e mundur qe *Ipr* të përdorej nga të gjithe dhe me pas sistemi leviz file-in password. Një menyre tjeter për të hyre ne UNIX ishte lidhja e një file ne direktorine e punes të quajtur *core*. Sulmuesi me pas nderhyn me force ne berthamen e programit SETUID, program te cilin sistemi e shkruajti ne file *core*, i cili ndodhet ne maje te filet password. Ne kete menyre, një user mund te zevendesoje një File password me permbytjen e disa stingave sipas zgjedhjes se tij. (e.g., command arguments).

Një tjeter e metë ne UNIX përfshin komanden

```
mkdir foo
```

Mkdir i cili është një SETUID program ne varesi të root, si fillim kriohet një nyje **i** për direktorine foo nepërmjet thirrjes sistem mknod dhe ndrzhon përkatësine ne foo nga ajo efektivja UID.

Kur sistemi ishte i ngadalshem ishte e mundshem për përdoruesit të levizin shpejt direktorine i-node dhe të krijojne një lidhje me file-in password nen emrin foo pas mknod por para chown. Kur mkdir kryen chown, ai pajis përdoruesin me përkatësine e file-it password.

Të metat e famshme ne TENEX

Sistemi operativ TENEX ishte i përdorur ne kompjuterat DEC-10. Nuk u përdor gjatë. TENEX suportonte paging. Për të lejuar përdoruesit të monitoronin sjelljen e programeve të tyre ishte e mundur të udhezohej sistemi të theriste një funksion përdoruesi se here qe kishim page fault. TENEX përdorte password për të mbrojtur file-at. Për të aksesuar një file programi duhet të paraqiste password-in e posacem ne sistemin operativ ne momentin qe hapej file-i. Sistemi operativ kontrollon password-in, shkronjë me shkronjë, duke ndaluar aty ku kemi gabim. Për të thyer TENEX nderhyresi duhet ta pozicionoje mire password-in.

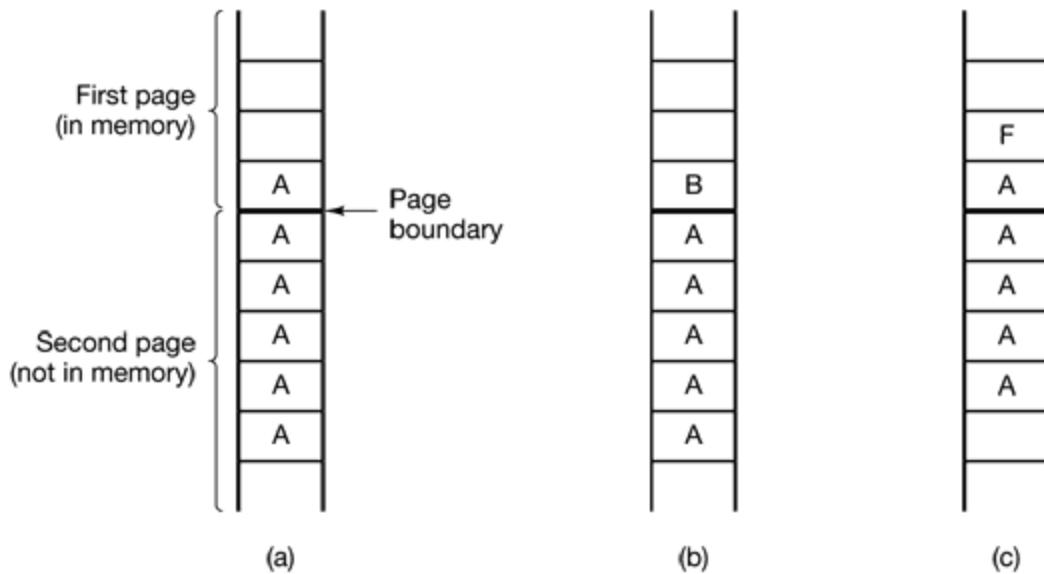


Figura 9-12. Problem ne password-in e TENEX

Hapi tjetër është të sigurohesh qe faqja e dytë nuk është ne memorie ,për shembull duke iu referuar shume faqeve të tjera duke bere qe faqja e dytë të largohet duke hapur vend për ato. Tani programi përpinqet të hape file-in e viktimes duke përdorur me kujdes password-in qe i bashkangjitet. Ne qoftë se karakteri i pare i password-it të vertetë nuk përkon me atë qe shkruan nderhyresi, sistemi ndalon kontrollin e password-it dhe jep mesazhin ILLEGAL PASSWORD. Në qoftë se shkronja e pare e password-it real përkon me atë të shkruar, sistemi vazhdon me leximin dhe ndodh një page fault, për të cilin nderhyresi lajmerohet. Në qoftë se shkronjat e para nuk përkojne nderhyresi ndryshon password.

Të metat e famshme të sigurise ne OS/360

Përshkrimi qe jepet është i thjeshtë por paraqet esencen e të metave. Ne ketë sistem ishte e mundur të startoje leximin ne një tape dhe me pas të vazhdoje me llogaritje gjatë kohes qe driveri transferonte data për ne user space. Mashtrimi ketu ishte qe të startoje me kujdes leximin e tape-it dhe me pas të kryeje një thirrje sistem e cila përbante strukturen e të dhenave të përdoruesit. Sistemi operativ ne fillim verifikonte se a ishte me të vertetë password-i i saktë. Me pas lexohej përseni emri i file-it për aksesim aktual. Fatkeqsisht kur sistemi shkonte të merrte emrin e file-it për here të dytë, emri i file-it mbishkruhej nga tape drive. Sistemi me pas lexonte file-in e ri për të cilin nuk kemi password prezent.

9.4.8 Principet e dizenjimit të sigurise

Tani është e qartë se dizenjimi i një sistemi operativ të sigurt nuk është një problem i vogel. Ne ketë problem është punuar për shume dekada por me pak sukses. Kerkuesit kane përcaktuar disa principe të përgjithshme të cilat mund të përdoren si guide për ndertimin e sistemeve operative të sigurt.

1. Dizenjimi i sistemit duhet të jetë publik. Duke pretenduar se nderhyresit nuk e dine se si funksionon sistemi, ky eshte vetem një iluzion per dizenjuesit e tij. Sulmuesit e sistemit heret apo vone do ta zbulojne dhe mbrojtja e sistemit eshte e kompromentuar ne kete moment.
2. Default-i nuk duhet aksesuar. Gabimet ne të cilët refuzohet aksesimi do të raportohen me shpejt se sa ne atë qe lejohet aksesimi.
3. Kontrollo për current authority. Sistemi nuk duhet kontrolloje për leje, duke llogaritur qe aksesimi eshte i lejuar, pasi e ka marre informacionin ai e kryen kete veprim. Shume sisteme kontrollojnë për leje kur një file hapet. Kjo do të thotë se një përdorues i cili hap një file dhe e mban të hapur për jave të tëra do të ketë akses edhe mbasi zoteruesi i file e ka ndryshuar mbrojten e filet ose e ka fshire ate.
4. Ti jepet secilit proçes privilegjet me të paket të mundshem. Në qoftë se një editor ka vetëm autoritetin për aksesim file-ash për ti edituar, editoret me Trojan nuk do të jene ne gjendje të bejne shume deme. Ky princip implikon një skeme mbrojtjeje të mire.
5. Mekanizmi i mbrojtjes duhet të jetë i thjeshtë, uniform dhe i ndertuar ne numrin me të vogel të mundshem të shtresave. Siguria si ajo e korrigimit nuk është një vecori shtese.
6. Skema e zgjedhur duhet të jetë psikologjikisht e pranueshme. Në qoftë se përdoruesit ndiejne qe mbrojtja e file-eve të tyre është një pune e madhe, ata nuk do ta kryejne ketë pune.

Ne ketë listë do të shtojme dhe një princip tjeter :

Mbaje dizenjinin të thjeshtë.

Ne qoftë se sistemi është elegant dhe i thjeshtë, i dizenjuar nga një arkitekt dhe ka pak principe udhezimi, ka një shans për të qene i sigurtë. Mund të disenjohet një sistem me shume vecori por ky është një system i madh. Sistemet e medha janë potencialisht të pasigurt.

9.5 SULMET NGA JASHTË SISTEMIT

Kërcënimet e diskutuara në seksionin e mëparshëm ishin kryesisht të shkaktuara nga brenda, domethene të bëra nga user-i i loguar tashmë në sistem. Megjithatë, për makinanat të lidhura në Internet ose në ndonjë rrjet tjeter, ka një rritje të kërcënimeve nga jashtë. Një kompjuter në rrjet mund të sulmohet nga një tjeter kompjuter në distancë i cili ndodhet po në këtë rrjet. Pothuajse në të gjitha rastet, një sulm i tillë konsiston në disa kode të transmetuara përgjatë rrjetit drejt makinës objektiv dhe i ekzekutuar atje shkakton dëmtim. Sa më shumë kompjutera t'i bashkohen Internetit, aq më i madh është mundësia për dëme. Më poshtë ne do të shohim disa aspekte të sistemeve operative për këto kërcënime nga jashtë, duke u fokusuar së pari tek viruset, depërtimet (worms), kodet mobile (mobile code) dhe applet-et e Java-ës.

Është e vështirë të hapësh një gazetë në këto ditë, pa lexuar rrëth një tjeter virus kompjuteri ose depërtimi (worm) duke sulmuar botën e kompjuterave. Ata janë qartazi

një problem madhore sigurie për individët dhe kompanitë e ngashme. Në seksionin pasues ne do të vërejmë se si ato punojnë dhe çfarë mund të bëhet rreth tyre.

Do të hezitoja të shkruaja këtë paragraf në shumë detaje, që të mos i jap disa njërzve mendime të pa vlera, por ekzistojnë libra që japid më shumë hollësi duke përfshirë madje dhe kode reale (per shembull Ludwig, 1998). Gjithashtu Interneti është i tejmbushur me informacion rreth viruseve, kështu që “gjeniu është tashmë jashtë shishes”. Veç kësaj, është e vështirë për njerëzit që të mbrojnë veten e tyre kundër viruseve ne qofte se ata nuk dinë se si ata punojnë. Përfundimisht, ka shumë mendime të gabuara rreth viruseve që qarkullojnë, që nevojitet të korrigohen.

Ndryshe nga sa po themi, programues lojërash, autorët e viruseve të suksesshme, nuk priren drejt publicitetit, mbasi produkti i tyre e bën debutimin. Duke u bazuar në të dhënat e pakta që ka, duket se më së shumti janë studentë të shkollave të larta ose kolegjeve ose të apo diplomuarit që shkruajnë këto virusë si një sfidë teknike, duke mos imagjinuar se një sulm virusi mund të shkaktoj po aq viktima të mbledhura së bashku, sa një uragan ose një termet. Le ta quajmë antihero-in tonë Virgil, shkruesin (autorin) e viruseve. Ne qofte se Virgil është tipik, synimi tij është që të krijoj një virus që përhapet shpejt, është e vështirë të zbulohet dhe ta heqësh qafe apo të zbulohet.

Gjithsesi, çfarë është një virus? Shkurtimisht, një virus është një program që mund të riprodhojë vetveten duke vënë kodin e tij në një tjetër program, i ngashëm me riprodhimin e viruseve biologjik. Veç kësaj, virusi mund të bëjë dhe gjëra të tjera veç riprodhimit të vetveteve. Depërtimet (worm) janë si viruset por vet-përsëriten (kopjohen). Ky ndryshim nuk do të na interesojë tanë këtu, kështu që do të përdorim termin “virus” përfshirë të dyja, për momentin. Do të shohim rreth worm-eve në seksionin 9.5.5.

9.5.1 SKENARI I DËMIT QË SHKAKTON VIRUSI

Për sa kohë që një virus është një program, mund të bëjë çdo gjë që bën një program. Per shembull, mund të shtyp një mesazh, të shfaq një imazh në monitor, të luaj një muzikë ose diçka tjetër të padëmshëm. Fatkeqësisht, mundet gjithashtu të fshijë, modifikojë, shkatërrojë ose të vjedh skedarë (duke i dërguar diku tjetër). Kërcënimi (shantazhi) është gjithashtu një mundësi. Imagjinoni një virus që inkripton (encrypt) të gjitha fluturimet në hard disk-un e viktimateve, më pas shfaq mesazhin e mëposhtëm:

PËRSHËNDETJE NGA INKRİPTIMI KRYESOR

PËR TË BLERË ÇELËSIN E DEKREPTIMIT PËR HARD DISK-un TUAJ, JU LUTEM DËRGONI \$100 NË TË HOLLAT, FATUR TË PA SHËNUAR NË KUTINË 2154, PANAMA CITY, PANAMA. JU FALEMINDERIT. E VLERËSOJMË AKTIVITETIN TUAJ.

Tjetër gjë që mund të bëj një virus është ta shndërrojë kompjuterin tuaj të papërdorshëm për aq kohë sa virusi është aktiv (në funksionim). Kjo është quajtur **mohim i sulmit të shërbimit (denial of service attack)**. Rruga më e zakonshme është konsumimi i

burimeve në mënyrë jo të kujdeshme, të tillë si CPU ose mbushja e disk-ut me hedhurina. Këtu është një program një-rresht i përdorur për të fikur çdo sistemi UNIX:

```
main () { while (1) fork ();}
```

Ky program krijon procese derisa procesi tabelë (table) është plot, duke parandaluar çdo proces tjetër nga fillimi. Tani imagjinoni një virus që infekton çdo program në sistem me këtë kod. Të mbrohesh kundër këtij problemi, shumë sisteme moderne UNIX limitojnë numrin e fëmijëve që një proces mund të ketë njehersh.

Madje dhe më keq, një virus mundet për një kohë të gjatë të dëmtojë hardware-in e kompjuterit. Shumë kompjutera modern e mbajnë BIOS-in në flash ROM, në të cilin mund të rishkruhet nën kontrollin e një programi. Një virus mund të shkruajë rastesisht hedhurina në flash ROM kështu që kompjuteri tashmë nuk do të boot-ojet. Ne qofte se chip-i i flash ROM është në një socket, zgjidhja e problemit kërkon hapjen e kompjuterit dhe zëvendësimin e chip-it. Ne qofte se chip-i i flash ROM është i lidhur me bordin prind, mundet që i tërë bordi duhet të zëvendësohet. Padyshim jo një eksperiencë e dëshirueshme.

Një virus gjithashtu nxirret me një synim të caktuar. Një kompani mund të lëshojë një virus që të kontrolloj ne qofte se është duke operuar në një fabrikë konkurrente dhe jo me një administrator sistemi të loguar tashmë në të. Ne qofte se, lëshimi ishte i lirë, do të ndërhyjë në procesin e prodhimit, duke reduktuar kualitetin e produktit, si rrjedhim shakton probleme për konkurrentin. Në të gjithë rastet e tjera nuk do të bënte asgjë, duke e bërë të vështirë për t'u zbuluar.

9.5.2 SI FUNKSIONOJNË VIRUSET.

Mjaftueshmë për një skenar potencial dëmtimi. Tani le të shohim se si viruset punojnë. Virgil shkruajti virusin e tij, ka të ngjarë në gjuhën assemblér, dhe më pas me kujdes e futi në një program në kompjuterin e tij duke përdorur një mjet (tool) të quajtur **droppér**. Ky program i infektuar më pas shpërndahet, ndoshta duke e postuar atë në një bord buletini ose në një koleksion software pa pagesë në Internet. Programi mund të jetë një lojë e re interesante, versione pirate të disa software-ve komerciale, ose çdo gjë të tjetër që mund të konsiderohet e dëshirueshme. Njerëzit më pas fillojnë të transferojnë (download-ojnë) programin e infektuar në kompjuterin e tyre.

Sapo të instalohet në makinën “viktim”, virusi qëndron në gjumë derisa programi i infektuar ekzekutohet. Sapo të fillojë, ai zakonisht fillon duke infektuar programet e tjera në makinë dhe pastaj ekzekuton **payload**-in e tij. Në shumë raste, payload-i mund të mos bëj asgjë derisa një datë e caktuar ka kaluar për t'u siguruar se virusi është përhapur gjerësisht përpara se njerëzit fillojnë të njoftohen për të. Në datën e zgjedhur mund edhe të dërgohen një mesazh politik.

Në diskutimin e mëposhtëm, ne do të vërejmë 7 lloje virusesh bazuar në atë çfarë infektojnë. Këto janë virusë shoqërues, programe të ekzekutueshme, memorie, sektor boot, device driver, macro dhe kod burim. Nuk ka dyshim që tipe të tjera do të shfaqen në të ardhmen.

Companion Viruses (viruset shoqërues).

Një *virus companion* në të vërtëtë nuk infekton një program, por nis të operojë kur programi është i supozuar të fillojë. Koncepti është i thjeshtë, le të shpjegohet më një shembull. Në MS-DOS, kur përdoruesi shtyp;

Prog

MS-DOS-i në fillim kërkon për një program të quajtur *prog.com*. Ne qofte se, nuk gjendet një i tillë, shikon për një program të quajtur *prog.exe*. Në Windows, kur përdoruesi klikon në Start dhe më pas Run, ndodh e njëjtë gjë. Në ditët tona, shumica e programeve janë skedarë *.exe*; skedarët *.com* janë shumë të pakët.

Supozoni se Virgil e di se shume njërež ekzekutojnë *prog.exe* nga një prompt MS-DOS-i ose nga RUN-i i Windows-it. Ai më pas mund të lëshojë lehtësisht një virus të quajtur *prog.com* i cili do të ekzekutohet kur çdo kush do të provojë të ekzekutojë *prog* (veç në se ai shtyp emrin e plotë: *prog.exe*). Kur *prog.com* ka mbaruar detyrën e tij, ai më pas vetëm sa ekzekuton *prog.exe*.

Një sulm disi i lidhur me desktop Windows, i cili përmban shortcut-et (linke simbolike) e programeve. Një virus mund të ndryshojë qëllimin e një shortcut-it për ta bërë argument për virusin. Kur përdoruesi klikon dy herë në një ikonë, virusi është ekzekutuar. Kur kjo është bërë, virusi vetëm ekzekuton qëllimin origjinal të programit.

Program viruset e ekzekutueshëm.

Një hap më lart drejt kompleksitetit janë viruset që infektojnë programet e ekzekutueshëm. Më të thjeshtit e këtyre virusave, vetëm sa mbishkruanjnë programet e ekzekutueshëm me veten e tyre. Këta janë quajtur *viruse të mbishkruajtshëm*. Infektimi logjik të një virusi të tillë është dhënë në Fig. 9-13.

```
#include <sys/types.h>           /* kokë POSIX standarte */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                /* për thirrjet lstat për të parë në se skedari
                                 eshtë symlink */
search (char *dir_name)
{
    /* kontrolli rekursiv (përsëritës) për të
       ekzekutueshmet */

DIR *dirp;                      /* pointer drejt një stream direktorie të hapur */
struct dirent *dp;               /* pointer në hyrje të një direktorie*/
dirp = opendir(dir_name);        /* hap këtë direktori */
```

```

if (dirp == NULL) return;                                /* dir nuk mund të hapet; harroje */
éhile (TRUE) {
    dp = readdir(dirp);                                /* lexo hyrjen e direktorisë pasuese */
    if (dp == NULL) {                                   /* NULL do të thotë ne mbaruam */
        chdir ("..");
        break;
    }
    if (dp->d_name[0] == '.') continue;               /* kalo . dhe .. direktoritë */
    lstat{dp->d_name, &sbuf);                         /* a është hyrja një link simbolik? */
    if (S_ISLNK(sbuf.st_mode)) continue;                /* kapërce link-un simbolik */
    if (chdir{dp->d_name} == 0) {                      /* në se chdir arrihet me sukses, duhet të
                                                               ketë një dir */
        search(".");
    } else {                                         /* jo (file), infektoje atë */
        if (access(dp->d_name,X_OK) == 0) /* në se është i ekzekutueshëm,
                                               infektoje atë */
            infect(dp->d_name);
    }
    closedir(dirp);                                  /* dir i përpunuar; përfundo dhe kthehu */
}
}

```

Figure 9-13. Një procedurë rekursive që gjen skedarë të ekzekutueshëm në një sistem UNIX

Programi kryesor i këtij virusi do të kopjojë në fillim programin binar të tij në një rresht duke hapur *argv* [0] dhe duke e lexuar atë për një ruajtje të sigurt. Më pas i duhet të përshkuajë të tërë file-in sistem duke filluar nga direktoria root (rrënëjë) duke ndryshuar direktorinë root dhe duke thirrur *search* me direktorinë root si parametër.

Procedura rekursive *search* shqyrton një direktori duke e hapur atë, më pas lexon hyrjen një herë në kohë duke përdorur *readdir* derisa një NULL është përgjigjur duke treguar që nuk ka porta. Në se porta (hyrja) është një direktori, proçesohet duke e ndryshuar atë dhe më pas thirret rekursivisht *search*; në se është një file i ekzekutueshëm, infektohet duke thirrur *infect* me emrin e skedarit për ta infektuar si parametër. Skedarët që fillojnë me “.” kapércehen për të shmangur problemet me direktoritë me . dhe .. Gjithashtu, link-et simbolike kapércehen sepse programi supozon që ai mund të hyjë në një direktori duke përdorur thirrjen e sistemit *chdir* dhe më pas të kthehet atje ku ishte duke shkuar sipas .., diçka që mban si një link të fortë por jo link simbolik. Një program amator mund të mbajë gjithashtu me raste link-e simbolike.

Procedura aktuale e infektimit, *infect*, vetëm sa hap skedarin e emëruar në parametrat e tij, kopjon virusin e ruajtur në një rresht (grup) mbi skedar dhe më pas mbyll skedarin.

Ky virus mund të përmisohet në mënyra të ndryshme. Së pari, një test mund të futet brenda *infect* për të gjeneruar një numër të rastit dhe më pas të kthehet në rastet më të

shumta pa bërë asgjë. Në, le të themi, një thirrje nga 128, infektimi do të zinte vend, në këtë mënyrë do të reduktonte shanset për një detektim (zbulim) të hershëm, përpala se virusi të ketë patur një shans të mirë për tu përhapur. Viruset biologjike kanë të njëjtat tipare: ata që vrasin shpejt viktimat e tyre, nuk përhapen aq shpejt sa ata që prodhojnë një ngadalësi, vdekje të ngadaltë, duke i dhënë viktimate shanse të shumta për të përhapur virusin. Një plan alternativ do të ishte një normë e lartë infektimi (le të themi, 25%) por një ndërprerje në numrin e skedarëve të infektuar në të njëjtën kohë për të reduktuar veprimtarinë e diskut dhe si rrjedhojë do të ishte më pak e dukshme.

Së dyti, *infect* (*infektimi*) mund të kontrollojë në se skedari është tashmë i infektuar. Infektimi i të njëjtit skedar për së dyti vetëm sa do të humbtë kohë. Së treti, shkalla mund të merret për të mbajtur kohën e modifikimit të fundit dhe madhësinë e skedarit në të njëjtën mënyrë sikur do të ndihmoj të fshehje infektimin. Për programet më të mëdha se virusi, madhësia do të mbetet e pandryshuar, por për programet më të vogla se virusi, programi tanë do të zmadhohej. Meqenë se shumica e viruseve janë më të vegjël se shumica e programave, ky nuk është një problem.

Ndonë se ky program nuk është shumë i gjatë (programi i plotë është më pak se një faqe C dhe programi burim kompilohet në më pak se 2KB), një version i kodit assemblér i tij mund edhe të shkurtohet. Ludwig (1998) shkroi një program në kod assemblér për MS-DOS që infektonte të gjithë skedarët në direktorinë e tij dhe ishte vetëm 44 byte kur assemblohej.

Më vonë në këtë kapitull do të studiojmë programin antivirus, që është një program që kap dhe fshin viruset. Megjithatë, është e rëndësishme të përmendet logjika e Fig. 9-13, e cila përdor një virus që të gjejë të gjithë skedarët e ekzekutueshëm për t'i infektuar ato, gjithashtu mund të përdoret nga një program antivirus për të kapur të gjithë programet e infektuar për të fshirë këto viruse. Teknologjia e infektimit dhe disinfectimit shkojnë krah për krah, kjo është arsyaja pse është e nevojshme për t'i kuptuar në detaje se si viruset funksionojnë për të qenë të aftë që t'i luftojmë ato efektivisht.

Nga këndvështrimi i Virgil, problemi i mbishkrimit të një virusi është, se është shumë e lehtë për t'i zbuluar. Në fund të fundit, kur një program i infektuar ekzekutohet, mund të përhapë virusin më shumë, por nuk bën atë që duhet të bëjë dhe përdoruesi do ta vërejë këtë menjehershë. Si pasojë, shumë viruse sulmojnë veten e tyre në program dhe bëjnë punën e tyre, por lejojnë që programi të punojë normalisht më pas. Këto viruse njihen si **viruse parazit**.

Viruset parazitë mund të sulmojnë veten e tyre në fillim, në fund ose në mes të programit të ekzekutueshëm. Në se një virus sulmon veten e tij para programit, ai duhet të kopjojë së pari programin në RAM, ta shkruajë veten e tij në fillim të skedarit, dhe më pas të kopjojë programin nga RAM-i duke ndjekur veten e tij, siç tregohet në Fig. 9-14(b). Fatkeqësisht, programi nuk do të ekzekutohet në adresën virtuale të tij të re, kështu që dhe virusi duhet të zhvendos programin meqë ai ka lëvizur, ose ta fus fshehtas prapa në adresën virtuale 0 pasi ka mbaruar ekzekutimin e tij.

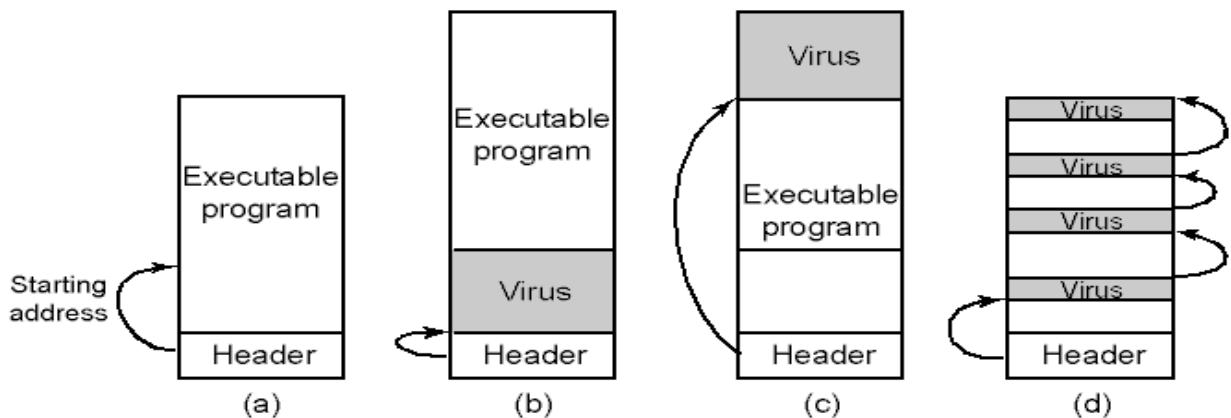


Figure 9-14. (a) Një program i ekzekutueshëm. (b) me një virus në fillim. (c) me një virus në fund. (d) me një virus të përhapur në hapësirën e lirë brenda programit.

Për të shmangur kompleksitetin e tepërm të opsiioneve kërkuar nga ky ngarkim i përparëm, shumë viruse janë ngarkues të fundëm, duke sulmuar veten e tyre në përfundim të ekzekutimit të programit, duke ndryshuar fillimin e fushës së adresave në header për të shënuar fillimin e virusit, siç tregohet në Fig. 9-14(c). Virusi tani do të ekzekutohet në një adresë virtuale të ndryshme i varur nga programi i cili po ekzekutohet, por e gjitha kjo do të thotë që Virgil duhet të sigurojë virusin e tij pozicionim të pavarur. Kjo nuk është e vështirë për një programues me eksperience.

Formati kompleks i programit të ekzekutueshëm, i tillë si skedarët, *exe* në Windows dhe pothuajse në të gjithë formatet binar Unix modern, lejojnë që një program të ketë tekst të shumëfishtë (përbërës) dhe segmentet data...

Viruset Rezident në Memorie

Deri tani ne kemi supozuar se kur një program i infektuar është ekzekutuar, virusi procedon, kalon kontrollin në programin real dhe del. Në kundërshtim, një virus me qëndrim në memorie qëndron në memorie gjithë kohën, për më tepër duke u fshehur në fillim të memories ose ndoshta poshtë, përgjatë vektorëve interrupt, në 100 bytet e fundit të cilët zakonisht nuk janë në përdorim. Një virus i “zgjuar” mundet madje dhe të modifikojë bitmap-in e RAM-it të sistemit operativ, për ta bërë sistemin të mendojë se memoria e virusit është e zënë, për të shmangur vështirësinë i të qenit i mbishkrujtshëm.

Një virus tipik rezident-memorie kap një nga trap-et ose vektorët e interrupt-eve duke kopjuar përbajtjen e pikënisjes së variablate dhe duke vendosur adresën e tij atje, duke e drejtuar kështu atë trap ose duke e penguar atë. Zgjidhja më e mirë është që sistemi të thërrasë trap-et. Në këtë mënyrë, virusi merr ekzekutimin (në mënyrën kernel) në çdo thirrje të sistemit. Kur kjo kryhet, ai vetëm sa i kërkon sistemit real të thirrur nga kërcimet në adresën e ruajtur të trap-it.

Pse një virus duhet të ekzekutohet në çdo thirrje të sistemit?

Për të infektuar programin, natyrisht. Virusi vetëm sa pret deri sa një thirrje sistemi *exec* vjen dhe atëherë, duke ditur që skedari në krah është një binar i ekzekutueshëm, e infekton atë. Ky proces nuk kërkon aktivitet masiv të diskut të Fig. 9-13 kështu që është më pak e dukshme. Të kapurit e gjithë thirrjeve të sistemit gjithashtu i jep virusit mundësi të mëdha për të përgjuar në data dhe kryen të gjitha dëmet e mundshme.

Viruset e Sektorit Boot

Sikurse e diskutuam në Kapitullin 5, kur shumica e kompjuterave janë të ndezur, BIOS-i lexon të dhënat (diskun, regjistrimin, pllakën) e master boot nga fillimi i diskut të boot-it brenda RAM-it dhe e ekzekuton atë. Ky program përcakton cili particion është aktiv dhe e lexon në sektorin e parë, sektorin boot nga ai particion dhe e ekzekuton atë. Atë program që përmes tij ka qarkon sistemin operativ ose sjell në loader (ngarkim) përmes tij ngarkuar sistemin operativ. Fatkeqësisht, një nga shokët e Virgil hodhi idenë e krijimit të një virusi që mund të mbishkruante master boot rekord ose sektorin boot, me një rezultat shkatërrues. Këto viruse, të quajtur **boot sektor viruses**, janë shumë të zakonshëm.

Zakonisht, një virus sektor boot-i {që përfshin viruset MBR (Master Boot Record)}, në fillim kopjon sektorin boot të vërtetë në një vend të sigurt të diskut kështu që ai mund të boot-ojë sistemin operativ kur ai të ketë mbaruar. Programi Microsoft i formatimit të diskut, *fdisk*, kapërcen track-un (pistën) e parë, kështu që ai është një vend i fshehtë i mirë në makinat Windows. Një opsjon tjeter është të përdorë çdo sektor të lirë të diskut dhe më pas të freskojë (update) listën e sektorit të keq që të shënojë “strofullën” si të dëmtuar. Në fakt, në se virusi është i madh, ai mundet gjithashtu të maskojë pjesën tjeter të tij si një sektor të keq. Në se direktoria root është mjafreshëm e madhe dhe në një vend të fiksuar, siç është në Windows 98, fundi i direktorisë root (rrënje) është gjithashtu një mundësi. Një virus vërtetë agresiv mundet madje të caktoj një hapësirë normale të diskut përmes sektorin e vërtetë të boot-it dhe veten e tij dhe të freskojë (update) bitmap-in e diskut ose në përputhje me listën e lirë. Të bësh këtë kërkon njohuri të strukturës së datave në brendësi të sistemit operativ, por Virgil kishte një profesor shumë të mirë përkushtuar e tij të sistemeve operative dhe studionte shume.

Kur kompjuteri është boot-uar, virusi kopjon veten e tij në RAM, në fillim ose ndërmjet vektorëve të interrupt-eve. Në këtë pikë makina është në formën kernel (mode kernel), me MMU off, jo sistem operativ dhe jo me një program antivirus në veprim. Në kohët e pjeseshme (të pritjes) përmes sektorit boot-on sistemin operativ, zakonisht duke qëndruar rezident në memorie.

Një problem, gjithesesi, është se si të merret kontrolli përsëri më vonë. Mënyra e zakonshme është të shfrytëzohen njohuritë specifike se si sistemi operativ menaxhon vektorët e interrupt-eve. Për shembull, Windows nuk i mbishkruan të gjithë vektorët e interrupt-eve në një goditje (blow). Në vend të kësaj, ai ngarkon driver-at e njësive një herë në kohë dhe secila kap vektorin e interrupt-it që ka nevojë. Ky proces mund të marrë një minutë.

Ky projektim i jep virusit manovrimin që i nevojitet. Ai fillon me kapjen e të gjithë vektorëve të interrupt-eve siç tregohet në Fig. 9-15 (a). Sapo driver-i ngarkohet, disa nga vektorët mbishkruhen, por në se ora e driver-it është ngarkuar më parë, pasi te nisi virusi do të jetë plot me interrupt-e clock-u. Humbja e interrupt-it të printerit është treguar në Fig. 9-15(b). Sapo virusi të vërejë që një nga vektorët e tij të interrupt-it është mbishkruar, ai mund ta mbishkruajë atë vektor përsëri, duke e ditur që tani është e sigurtë. Rikapja e printerit është treguar në Fig. 9-15(c). Kur çdo gjë është ngarkuar, virusi kthen të gjithë vektorët e interrupt-eve dhe mban vetëm trap-in e thirrjes së sistemit për veten e tij. Në fund të fundit, të pasurit kontroll në çdo thirrje të sistemit është më mirë se sa të paturit kontroll pas çdo përdorimi të floppy disk-ut, por gjatë boot-imit, ai nuk mund të marr riskun e humbjes së kontrollit përgjithmonë. Në këtë pikë ne kemi një virus rezident në memorie në kontroll të thirrjeve të sistemit. Në fakt, kjo është mënyra si vijnë në jetë shumica e viruseve rezident në memorie.

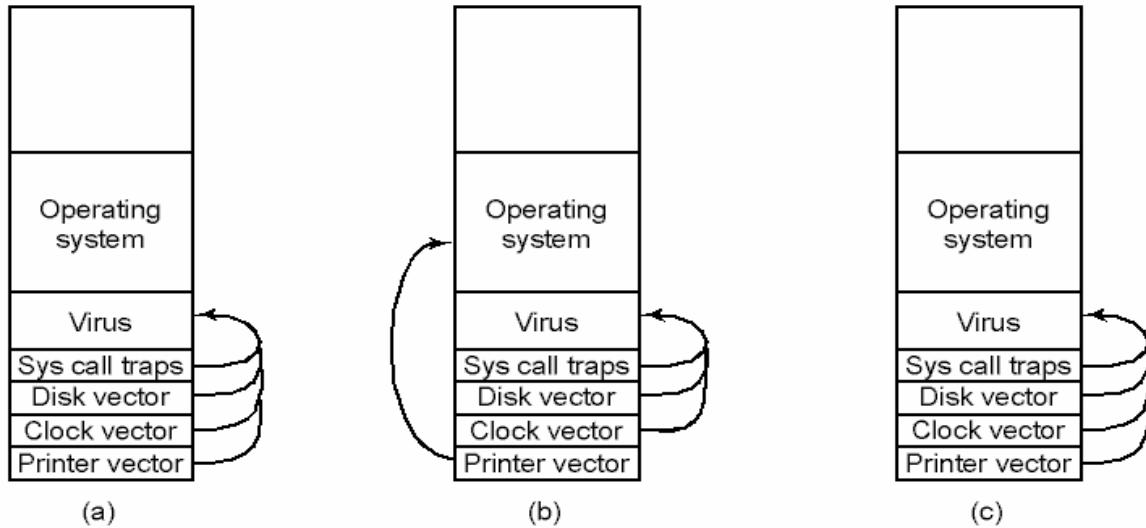


Figure 9-15. (a) Pasi virusi ka kapur të gjithë vektorët e interrupt-eve dhe të trap-eve. (b) Pasi sistemi operativ ka rimarrë vektorin e interrupt-it të printerit. (c) Pasi virusi ka vërejtur humbjen e vektorit interrupt të printerit dhe e ka rikapur atë.

Viruset e Driver Dvice

Të hysh në memorie në këtë mënyrë është si të eksplorosh një shpellë, ju duhet të kaloni ndërmjet gjarpërimeve dhe kujdes rreth diçkaje që mund të bjerë poshtë ne toke dhe ne kokën tënde njekohesisht. Do të ishte shumë më e lehtë ne qofte se sistemi operativ do të ngarkonte virusin zyrtarisht. Me pak punë, ky qëllim mund të realizohej. Marifeti është të infektosh driver-in e një pajisje (device driver), duke të drejtar në një **virus device**

driver. Në Windows dhe në disa sisteme Unix, device driver janë pikërisht programe të ekzekutueshme që qëndrojnë në disk dhe ngarkohen në kohën e boot-imit. Në se njëri nga ata infektohet duke përdorur viruset parazit, virusi gjithmonë do të jetë zyrtarisht i ngarkuar në kohën e boot-imit. Driver-at punojnë në formën kernel dhe pasi një driver është ngarkuar, ai është thirrur, duke i dhënë virusit një mundësi për të kapur vektorin e trap-eve të thirrjes së sistemit.

Viruset Makro

Shumë programe, si Word dhe Excel, e lejojnë përdoruesin të shkruajë macro për të grupuar disa komanda që më vonë mund të ekzekutohen vetëm me një tastë të vetëm. Makrot mundet gjithashtu të lidhen në njësinë menu, kështu që kur një nga ato është selektuar, makro-ja ekzekutohet. Në *Microsoft Office*, makrot mund të përmbajnë një program të tërë në Visual Basic, i cili është një gjuhë programimi i plotë. Makrot janë interpretuar madje dhe si të kompiluara, por kjo vetëm sa ndikon në shpejtësinë e ekzekutimit, jo në atë çfarë ato bëjnë. Meqenë se makrot mund të jenë dokumente specifik, Office i ruan makrot për çdo dokument me vetë dokumentin.

Tani vjen problemi. Virgili shkruan një dokument në Word dhe krijon një makro që e lidh në funksionin OPEN FILE. Makro-ja përmban një virus makro. Më pas ai e poston (e-mail) dokumentin tek viktima, i cili natyrisht e hap atë (duke pranuar se programi i postës, e-mail nuk e ka bërë tashmë këtë gjë për të). Hapja e dokumentit shkakton ekzekutimin e makros OPEN FILE. Meqenë se makro-ja mund të përmbajë një program arbitrar (të diktuar), ai mund të bëj çdo gjë, si infektimi i dokumenteve të tjera Word, të fshijë skedarë dhe më tepër. Në të gjitha paanësite e Microsoft-it, Word jep një shenjë paralajmëruese kur hap një skedar me macro, por shumë përdorues nuk e kuptojnë se çfarë do të thotë kjo dhe vazhdojnë gjithsesi ta hapin. Veç kësaj, është e pranueshme që dokumentet mund të përshtijnë gjithashtu edhe makro. Dhe ka të tjera programe që nuk e japid këtë sinjal (paralajmërim), duke e bërë dhe më të vështirë për të zbuluar virusin.

Me rritjen e email-it, dërgimi i virusave të futur në makro është një problem i madh. Të tillë virusë e kanë më të lehtë te shkrujanë se sa të fshehin sektorin e vërtetë të boot-it diku në një bllok listë të dobët, duke fshehur virusin midis vektorëve të interrupt-eve, dhe të kapin vektorin e trap-eve të thirrjeve të sistemit. Kjo do të thotë që gjithnjë e më shumë njërej me aftësi të pakta mund të shkrujnë tanë virusë, duke zbritur kualitetin e përgjithshëm të produktit dhe dhënien e programuesve të virusave një emër të keq.

Viruset e Kodeve Burim

Viruset parazit dhe të sektorëve boot, kanë një platformë të lartë specifik; viruset e dokumenteve janë disi më të pakët kështu që (Word punon në Windows dhe në Macintosh, por jo në Unix). Viruset më shumë të mbartshëm nga të gjithë janë viruset e kodeve burim (Source Code Viruses). Imagjino virusin e Fig. 9-13, por me një modifikim që në vend të kërkimit të skedarëve të ekzekutueshëm binar, kërkon për një program në C, një ndryshim vetëm në një rreshth (thirrja në Access). Procedura e infektimit duhet të ndryshojë për te futur rreshthin

```
#include <virus.h>
```

në fillim të çdo programi burim në C. Një tjetër ndërhyrje është e nevojshme, rreshti

```
run_virus();
```

për të aktivizuar virusin. Të vendosësh se ku duhet vënë ky rresht kërkon disa aftësi për të analizuar kodin C, meqenë se ai duhet të jetë në një vend që lejon procedurat e thirrjeve dhe gjithashtu jo në një vend ku kodi mund të vdes. Vendosja në mes të një komenti përmes teprer nuk punon, dhe të vendosurit brenda një cikel mund të jetë një gjë shumë e mirë. Thirrja e supozuar mund të jetë një vendosje e përshtatshme (per shemull, pikërisht përparrë fundit të *main* ose përparrë gjendjes *return* në se ka një të tillë), kur programi është kompiluar, ai tani përban virusin, të marrë nga *virus.h*.

Kur programi ekzekutohet, virusi do të thirret. Virusi mund të bëj çdo gjë që ai mund të dojë, per shembull të kërkojë përmes programe të tjera në C që t'i infektojë. Në se gjen një të tillë, ai vetëm sa përfshin dy rreshtat e dhënë më sipër, por kjo do të funksionojë vetëm në një makinë lokale, ku *virus.h* është supozuar të jetë instaluar tashmë. Kjo arrihet duke përfshirë kodin burim të virusit si një inicializim string karakteresh, më mirë si një listë e 32-bit integer hexadecimal përmes parandaluar ndonjë nga figurimet jashtë asaj që ai bën. Ky string ndoshta do të jetë përmes shumë kohë i qartë, por sot me kode multimegalinë, duhet të jetë lehtësisht i kalueshëm.

9.5.3 Si Përhapen Viruset

Ka disa skenarë të mundshëm përmes përhapjen. Le të fillojmë më modelin klasik. Virgil shkruajti virusin e tij, e vendosi atë në disa programe që i ka shkruajtur (ose vjedhur), dhe më pas fillon shpërndarjen e programit, per shembull, duke e vendosur në një Web site shareware. Përfundimisht, dikush e download-on programin dhe më pas e ekzekuton atë. Në këtë pikë, ja disa mundësi. Për ta filluar, virusi ndoshta ka infektuar më shumë skedarë në hard disk, pikërisht në rastin kur viktima ka vendosur te ndajë disa nga këto më vonë me shokë. Ai gjithashtu mund të provojë të infektojë sektorin e boot-it të hard disk-ut. Sapo sektori boot është infektuar, është e lehtë të fillosh një virus rezident-memorie, model-kernel në boot-in pasues.

Veç kësaj, virusi mund të kontrollojë përmes parës së se ka ndonjë floppy disk në drive, dhe në se është kështu, infekton skedarët e tij dhe sektorin boot. Floppy disqet janë një objektiv i mirë, sepse ata lëvizin nga një makinë në tjetrën më shumë se sa një hard disk. Në se në një sektor boot të floppy disk është i infektuar dhe më pas ai disk është përdorur në një boot të ndryshëm në një makinë tjetër, ai mund të fillojë të infektojë skedarë dhe sektorin boot të hard diskut në atë makinë. Në të shkuarën, kur floppy disqet ishin mjeti kryesor transmetimi mesatare përmes programet, ky mekanizëm ishte rruga kryesore e përhapjes së virusave.

Në ditët tona, metoda të tjera janë të vlefshme përmes Virgilin. Virusi mund të shkruhet përmes verifikuar, në se makina e infektuar është në LAN. Ky infektim nuk do të zgjatet me mbrojtjen e skedarëve, por kjo mund të pjesë, duke bërë që programet e infektuar të sillen në mënyrë pazakonte. Një përdorues që përdor një program të tillë do të pyeste administratorin e sistemit përmes ndihmës. Administratori atëherë do të provonte programin me probleme përmes parës se ç'po ndodh. Në se administratori e bën këtë ndërsa logohet si

një superuser, virusi tashmë mund të infektojë sistemet binare, device driver, sistemin operativ dhe sektorët e boot-eve. Të gjithë e marrin një gabim të tillë dhe të gjitha makinat në LAN janë të rrezikuar.

Zakonisht makinat në LAN kanë autorizim për t'u loguar në një makinë të largët nëpërmjet Internetit ose një korporate private ose madje, dhe autorizim për të ekzekutuar komanda të largëta pa u loguar. Kjo aftësi siguron më tepër mundësi për të përhapur viruset. Si rrjedhim një gabim i pavetdijshëm (i pa te keq) mund të infektojë të tërë kompaninë. Për të parandaluar këtë skenar, të gjitha kompanitë duhet të kenë një politikë (linjë veprimi) të përgjithshme duke treguar administratorit për të mos bërë kurrë gabime. Një rrugë tjetër për të përhapur viruset është të postosh një program të infektuar në një newsgroup në USENET ose në bordin e buletinit në cilin postohen rregullisht programet. Gjithashtu është e mundur të krijosh një faqe Web-i që kérkon një browser special reklamimi për ta shfaqur dhe më pas të sigurohesh, që janë të infektuar.

Një sulm i ndryshëm është të infektosh një dokument dhe më pas të dërgosh me e-mail tek shumë njerëz ose newsgroup USENET, zakonisht si një attachment. Edhe pse njerëzit që kurrë nuk do të ëndërronin në ekzekutimin e një programi të disa të huajve, dërgimi i tyre nuk mund të realizoj atë klikim në attachment për të hapur atë, mund të çojnë në një virus në makinen e tyre. Për ta përkeqësuar, virusi mund të shikojë për librin e adresave të përdorueseve dhe më pas ta postojë atë vet tek çdonjëri në librin e adresave, zakonisht me një rresht Subjekti që duket e pranueshme ose interesante, si;

Subject: Change of plans	(ndryshimi i planeve)
Subject: Re: that last email	(lexo email e fundit)
Subject: The dog died last night	(geni ngordhi mbrëmë)
Subject: I am seriously ill	(jam shumë sëmurë)
Subject: I love you	(të dua)

Kur mbërrin e-mail-i, marrësi e sheh se dërguesi është një shok ose një koleg, kështu nuk dyshon në ndonjë shqetësim. Sapo e-mail hapet, është tepër vonë. Virusi “TË DUA” që u përhap anembanë botës në Qershori të 2000 funksioni në këtë mënyrë dhe shkaktoi një dëm me vlerë biliona dollar.

Diçka e ngjashme me përhapjen e tanishme të viruseve aktive është përhapja e viruseve teknologjik. Janë grupe autorësh virusesh të cilët komunikojnë nëpërmjet Internetit dhe ndihmojnë njëri-tjetrin për të zhvilluar teknologji të reja, tools-e dhe viruse.

Ndoshta shumica e këtyre janë hobist (i bëjnë për qejf) se sa “kriminelë” profesionist, por efekti mund të jetë shkatërrues. Një kategori tjetër e shkruajtësve të (autorëve) viruseve është ushtria, të cilët e shohin virusin si një armë lufte, e aftë ta bëj të pafunkzionueshëm kompjuterin e armikut.

Një tjetër çështje referuar përhapjes së viruseve është shhangia e zbulimit. Burgu ka famën e keqe për sa i takon lehtësirave të informatikës, kështu që Virgil do preferonte ti shhangtë ata. Në se ai e poston virusin e parë, nga makina e tij e shtëpisë, ai po merr përsipër një risk të sigurtë. Në se sulmi është i suksesshëm, policia mund ta kap duke parë mesazhin e virusit me gjurmën e kohës (timestamp) më të hershëm, sepse kjo është ndoshta më e aferta me burimin e sulmit.

Për të zvogëluar ekspozimin e tij, Virgil duhet të shkojë në një Internet kafe në një qytet të largët dhe të logohet atje. Ai mund ta mbart virusin në një floppy disk dhe ta lexojë atë vetë ose ne qofte se makinat nuk kanë të gjitha floppy disk drivers, pyet punonjësen për të lexuar në skedarin *book.doc* kështu që ai mund ta printojë atë. Sapo të jetë në hard disk-un e tij, ai i riemëron skedarin në *virus.exe* dhe e ekzekuton atë, duke infektuar të tjerë LAN-in me një virus që sulmon dy javë më vonë, pikërisht në rastin kur policia vendos për t'i kërkuar linjës ajrore një listë të të gjithë njërzve të cilët kanë fluturuar në atë javë. Një alternativë është të harrosh floppy disk-un dhe të marrësh virusin nga një site FTP (File Transfer Protocol) të largët. Ose të sjellësh një laptop dhe ta vësh në prizë në një Ethernet ose port USB që Internet Kafe ka siguruar për turistët me laptop që duan të lexojnë e-mail-et e tyre çdo ditë.

9.5.4 Teknikat Antivirus dhe Anti-Antivirus

Viruset tentojnë të fshihen dhe përdoruesit tentojnë ti gjejnë ato, e cila çon drejt një loje si macja me miun. Le të shohim tani këtu disa nga këto probleme. Për të shmangur shfaqjen e një liste dosjesh, një viruse shoqëruesh, kodin burim të virusit, ose skedarë të tjera që nuk do të jenë, mund të kthehen në një bit të fshehur në Windows ose të përdorë emër skedari filluar me karakterin në Unix. Më tepër e sofistikuar do të ishte të modifikoje explorer e Windows ose *ls* e Unix për tu shmangur nga listimi i skedarëve emri i të cilit fillon me *Virgil-*. Viruset mundet gjithashtu të fshihen në një vend të pazakonte dhe të padyshimtë, të tillë si në një sektor të dëmtuar në disk ose në regjistrin Windows. ROM përdoret për të mbajtur BIOS dhe memoria CMOS janë gjithashtu të mundeshme, ndonëse formuesi e ka të vështirë të shkruajë dhe më i fundit është shumë i vogël.

Skaneri i virusave

Është e qartë, varieteti i përdoruesve mesatar nuk është per te gjetur viruset qe mundohen te fshihen, kështu që tregu eshte zhvilluar ne drejtimin e **antivirus software**. Poshtë ne do të shohim se si punojnë këto software. Kompanitë e e software-ëve kanë laborator në të cilën shkencëtar të përkushtuar punojnë me orë të gjata për të kapur dhe kupuar viruset e rinj. Hapi i parë është të kesh një virus që infekton një program që nuk bën asgjë, shpesh njihen si **goat file**, për të marrë një kopje të virusit në formën e tij më të saktë. Hapi tjetër është të bësh një listim të saktë të kodit të virusit dhe të futësh atë brenda database të virusave të njojur. Kompanitë konkurrojnë në madhësinë e database-it të tyre. Shpikja e virusave të rinj vetëm për të fryrë database-in tënd nuk konsiderohet argëtim.

Menjehërë apo një antivirus instalohet në makinën e klientit, gjëja e parë që bën është skanimi i të gjithë skedarëve të ekzekutueshmëm në disk duke parë për ndonjë virus në database-in e virusave të njojur. Shumica e kompanive antivirus ka një Web site prej të cilët klientët mund të download-ojnë llojin e zbulimit të virusave të rinj në database-in e tyre. Në se përdoruesi ka 10.000 skedarë dhe database-i ka 10.000 virus, disa programe Inteligjentë janë të nevojshme për ta bërë të ecë më shpejt.

Meqenëse variantet minor (të vegjël) e njohjes së viruseve shfaqen në të gjithë kohën, nevojitet një kërkim fuzzy, kështu një 3-bytesh ndryshim në një virus nuk e lejon të ikë zbulimi. Gjithesesi, kërkimet fuzzy nuk janë vetëm më të ngadaltë se kërkimet ekzakte, por mund të dalë një alarm i pavërtetë, kjo është, paralajmërim për skedarët legjitim që ndodh të përmbajë disa kode paksa të ngjashëm me një virus të reportuar në Pakistan, 7 vjet më parë. Çfarë supozohet të bëj përdoruesi me mesazhin:

KUJDES! Skedari xyz.exe mund të përmbajë virusin lahore-9x. Fshije?

Sa më shumë virusë në database dhe sa më i gjerë kriteri për deklarimin e një sulmi, më shumë alarme false do të ketë. Në se ka vertet shume, përdoruesi nuk do të ketë durim dhe do të dorëzohet. Por në se skaneri i viruseve insiston në një ballafaqim të ngushtë, ai mund të humbasë disa virusë të modifikuar. Marrja e të saktës është një balance ndihmuese delikate. Idealja, laboratori duhet të provojë të gjejë të identifikojë disa kode bërthame (pjesë qëndrore-core) në virus që nuk ndryshojne dhe e përdor këtë si një nënshkrim virusi për ta këruar (skanuar).

Në se disku ishte deklaruar i lirë nga viruset javën që shkoi nuk do të thotë se është ende i tillë, kështu që skaneri i viruseve duhet të jetë aktiv herë pas here. Në se skanimi është i ngadaltë, është më eficent të kontrollosh vetëm ato skedarë që kanë ndryshuar që nga data e fundit e skanimit. Shqetësimi është, një virus i zgjuar do të rivendos datën e një skedari të infektuar në datën origjinale të tij për të shmangur zbulimin. Përgjigjja e programit antivirus ndaj kësaj është të kontrollojë datën e mbylljes së direktorisë se ndryshimit më të fundit. Përgjigjja e virusit ndaj kësaj është rivendosja e datës së direktorisë po aq mirë. Ky është fillimi i lojës macja-me-miun referuar sa më sipër.

Një tjetër rrugë për programet antivirus për të zbuluar skedarët e infektuar është të regjistrojë dhe të ruajë në disk gjatësinë e të gjithë skedarëve. Në se skedari është zmadhuar që nga kontrolli i fundit, mundet të jetë infektuar, siç tregohet në Fig. 9-16(a-b). Sidoqoftë, një virus Inteligjent mund të shhang zbulimin duke ngjeshur (compress) programin dhe duke e mbushur skedarin në gjatësinë origjinale të tij. Që të funksionojë kjo skemë, virusi duhet t'i ketë të dyja procedurat e kompresimit dhe dekompresimit, siç tregohet në Fig. 9-16(c).

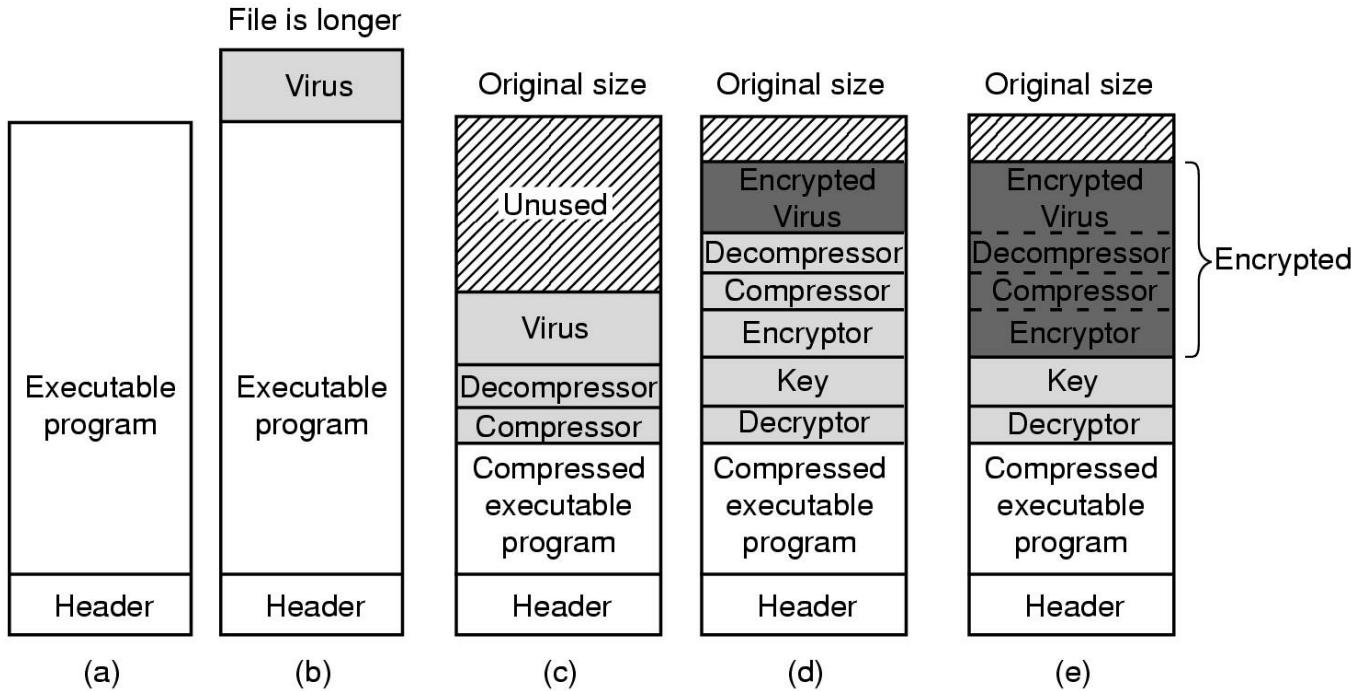


Figure 9-16. (a) Një program. (b) Një program i infektuar. (c) Një program i infektuar i kompresuar. (d) Një virus i inkriptuar. (e) Një virus i kompresuar me një kod të kompresuar të inkriptuar.

Një tjetër mënyrë për virusin që të provojnë t'i shmanget zbulimit është ta bëjnë të sigurtë paraqitjen e tyre në disk duke mos u shfaqur aspak si përfaqësim i tij në database-in e software-it të antivirusit. Një rrugë për të realizuar këtë “gol” është të einkriptojë vtveten me nga një çelës (zgjidhje) të ndryshëm për çdo skedarë të infektuar. Përpara se të bëj një kopje të re, virusi gjeneron një çelës enkriptimi prej 32-bit te rastit, per shembull, sipas XOR-ing koha aktuale me përbajtje të hequr, le të themi, fjalë memorie 72.008 dhe 319.992. Më pas XOR-s eshte i koduar me këtë çelës, fjalë pas fjale për të prodhuar virusin e encrypt-uar të ruajtur në skedarin e infektuar, siç ilustrohet ne Fig. 9-16(d). Çelësi ruhet në skedar. Për një destinacion të fshehtë, vendosja e çelësit në skedar nuk është idealja, por “goli” këtu është të pengoj skanimin e virusit. Sigurisht, për ta ekzekutuar, virusi më parë duhet të decrypt-ojë veten e tij, kështu që nevojitet një procedurë dekriptimi në skedar.

Kjo skemë nuk është ende perfekte sepse procedurat e kompresimit, dekompresimit, enkriptimit dhe dekriptimit janë të njëjtë në të gjitha kopjet, kështu që programi antivirus vetëm sa i përdor si shenja (nënshkrim) të virusit për ta kërkuar. Fshehja e kompresimit, dekompresimit dhe enkriptimit është e lehtë, mjafton të enkriptohen gjatë pjesës tjetër të virusit siç tregohet në Fig. 9-16(e). Megjithatë kodi i dekriptimit nuk mund të enkriptohet. I duhet të ekzekutohet në të vërtëtë në hardware për të dekriptuar pjesën tjetër të virusit kështu që duhet të jetë prezent në formën të kuptueshme tekste. Programet antivirus e dinë këtë gjë, kështu që ata kërkojnë për procedurën e dekriptimit.

Gjithsesi, Virgil argëtohet duke pasur fjalën e fundit, kështu që ai procedon si vijon. Supozojmë se procedura e dekriptimit ka nevojë të kryej këtë llogaritje.

$$X = (A + B + C - 4)$$

Kodi i assemblimit i hapur për këtë kalkulum për dy-adresa të përgjithshme kompjuteri tregohet në Fig. 9-17(a). Adresa e parë është burimi, adresa e dytë është destinacioni, kështu që *MOV A, R1* zhvendos variablin *A* në regjistrin *R1*. Kodi në Fig. 9-17(b) bën të njëjtën gjë, vetëm se është më pak eficent për shkak të instruksionit NOP (no operation) i shpërndarë me kodin real.

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Figure 9-17. Shembull i një virusi polimerfik.

Por nuk kemi mbaruar ende. Është gjithashtu e mundur të maskosh kodin e dekreptimit. Ka shumë mënyrë për të shfaqur NOP. Per Shembull, shtimi i 0 në një regjistër, zhvendosja e bitit të tij të majtë 0 dhe kapërcimi në instruksionit tjetër pa bërë asgjë. Në këtë mënyrë programi në Fig. 9-17(c) është funksionalisht i njëjtë me atë të Fig. 9-17(a). Kur kopjon veten e tij, virusi mund të përdorë Fig. 9-17(c) në vend të Fig. 9-17(a) dhe të punojë më pas kur ekzekutohet. Një virus që ndryshon formë në secilin kopje është quajtur **virus polimorfik**.

Tani supozojmë që *R5* nuk nevojitet gjatë kësaj pjese të kodit. Atëherë Fig. 9-17(d) është ekuivalente me Fig. 9-17(a). Përfundimisht, në shumë raste është e mundur të shkëmbesh (swap) instruksionet pa ndryshuar qëllimin e programit, kështu që ne përfundojmë me Fig. 9-17(e), si një tjetër fragment kodi që është logjikisht ekuivalent me Fig. 9-17(a). Një pjesë kodit që mund të ndryshojë formën e një sekuese të një instruksioni makine pa ndryshuar funksionalitetin e tij është quajtur **mutation engine**, dhe virusë të sofistikuar i përbajnë ato për të t'i ndryshuar formën dekreatorit nga kopja në kopje. Mutation engine vetë mund fshihet duke u enkriptuar përgjatë trupit të virusit.

Të pyesësh një software antivirusi të “varfér” për të realizuar (provuar) që Fig. 9-17(a) deri Fig. 9-17(b) janë të gjithë funksionalisht ekuivalent është e tepërt. Software antivirus mund të analizoj kodin për të parë se çfarë bën, dhe madje mund të provojë të stimuloj funksionimin e kodit, por të kujtoj atë mund të ketë mijëra virusë dhe mijëra skedarë për të analizuar, kështu që ai nuk ka shumë kohë për testin ose do të operojë jashtëzakonisht shumë ngadalë.

Më tutje, ruajtja brenda variablit *Y* ishte bërë pikërisht për ta bërë më të vështirë faktin se kodi referuar *R5* është një kod i vdekur, kështu që nuk bën asgjë. Në se fragmente të tjera kodesh lexojnë dhe shkruajnë *Y*, kodi do dukej fare mirë i pranueshëm. Një mutation engine i shkruajtur mirë që gjeneron kode të mira polimorfike mund të japë një shkruajtës software antivirusi nightmare. Ana e vetme e ndritshm e është se një motorr (engine) i tillë është e vështirë të shkruhet, kështu shokët e Virgil e përdorin të gjithë kodin e tij, që do të thotë se nuk janë shumë të ndryshëm njehershëm në qarkullim –akoma.

Deri tani kemi provuar të njohim viruset në skedarët e ekzekutueshëm të infektuar. Veç kësaj, skaneri i antivirusit duhet të kontrollojë MBR, sektorin boot, listën e sektorit të dëmtuar, ROM, memorien CMOS, etj, por çfarë ndodh, në se një virus rezident në memorie eshte aktualisht duke vepruar? Ai nuk do të zbulohet. Akoma më keq, supozojmë që virusi në punë është duke monitoruar të gjitha thirrjet e sistemit. Ai lehtësisht mund të zbulojë se programi antivirus është duke lexuar sektorin boot (për të kontrolluar për virus). Virusi nuk bën thirrje sistemi. Në vend të saj ai vetëm sa rikthen sektorin e vërtet të boot nga vendi i tij i fshehtë në bllok listën e dëmtuar. Gjithashtu ai bën një shënim mendor për të riinfektuar të gjithë skedarët kur ka mbaruar skanimi i virusit.

Për të parandaluar të qenit i mashtruar nga një virus, programi antivirus mund të bëj lexim të fortë në disk, duke shmangur sistemin operativ. Megjithatë kjo kërkesë ka ndërtuar deivce driver për IDE, SCSI dhe për disqe të zakonshëm, duke e bërë programin antivirus më pak të të mbartshëm (portativ) dhe subjekti të dështojë në kompjutera me disqe jo të zakonshëm.

Mënjanimi i Viruseve.

Menjanimi i viruseve eshte me i lehte se te mundohesh ti kapesh ato pasi te kene infektuar një kompjuter. Më poshtë janë disa direktiva për përdoruesit individual, por gjithashtu disa gjëra që industritë në tërësi mund të bëjnë për të reduktuar problemin e shumtë.

Çfarë mund të bëj një përdorues për të shmangur një infektim virusi?

Së parë të zgjedhë të një sistem operativ që ofron një shkallë të lartë sigurie dhe ndan përdoruesit e ndryshëm me password logimi, si dhe administratorin e sistemit. Sipas këtyre kushteve, një virus që hyn disi vjedhurazi nuk mund të infektojë sistemin binar.

Së dyti, instalo vetëm software të blerë nga fabrikant (prodhues) të besueshëm. Downloadimi i software-eve nga faqet Web është një sjellje e rrezikshme.

Së treti, ble një paketë të mirë software antivirusi dhe përdore sipas udhëzimeve. Sigurohu të marrësh një update të rregullt nga Web site i prodhuesit.

Së katërti, mos kliko në attachment e e-mail-eve dhe thuaji njérëzve qe mos t'i dërgojnë ato. E-mail-et që dërgohen si tekste të dukshme të qarta ASCII janë gjithmonë të sigurta por attachment mund të inicializojnë një virus kur hapen.

Së pesti, bëj një backup të vazhdueshëm skedarëve kyç në një mjedis të jashtëm, si floppy disk, CD ose tape. Kështu ne qofte se zbuloni një virus ju keni një mundësi për të kthyer skedarët ashtu siç ishin para se ato të infektoheshin.

Rikthimi nga një sulm virusi.

Kur zbulohet një virus duhet menjeherë të ndalet puna në kompjuter. Kompjuteri duhet të riboot-ojet nga një CD-ROM ose floppy disk që ka qenë gjithmonë i mbrojtur nga shkrimet dhe që mban të tërë sistemin operativ për të kaluar në sektorin boot, kopjimi në hard disk të sistemit operativ, dhe njësia disk, të gjitha këto tani mund të jenë të infektuara. Më pas një program antivirus duhet të ekzekutohet nga origjinali i tij CD-ROM, meqë hard disku gjithashtu mund të jetë infektuar.

Programi antivirus mund të zbulojë disa viruse dhe mund të jetë në gjendje t'i eliminojë ato, por nuk mund të sigurojë një mbrojtje të plotë. Kështu që mënyra më e mirë për t'u mbrojtur në të këtë pikë është ruajtja e të gjithë skedarëve që nuk mund të përmbajnë viruse (si skedarët ASCII dhe JPEG). Ato skedarë që përmbajnë viruse (si skedarët Word) duhet të konvertohen në një format tjetër që nuk mund të përfshijnë, të tillë si teksti ASCII. Këto skedarë duhet të ruhen në një mjedis të jashtëm. Më pas hard disku duhet të riformatohet. Është shumë e rëndësishme që MBR dhe sektori boot të jenë plotësisht të fshirë. Më pas sistemi operativ duhet të riinstalohet nga një CD-ROM origjinal.

9.5.5 Depërtimet (worm) në Internet

Worm-i konsiston në dy programe, bootstrap (shiriti boot) dhe Worm-i i duhur. Bootstrapi ishte 99 rreshta në C i quajtur ll.c. Ai ishte kompiluar dhe i ekzekutuar në sistem qe poi nenshtrohej një sulmi. Ndersa egzekutohet, ai lidhet me makinën nga i cili erdhi, ngarkon worm kryesor dhe e ekzekuton atë. Pasi kalon disa shqetësime për të fshehur ekzistencën e tij, worm më pas kontrollon hostin e ri duke rrëmuar në tabelat për të parë çfarë makine ishte lidhur hosti dhe tenton të përhapë bootstrap-in në ato makina. Tre metoda ishin provuar për të infektuar makinën e re. Metoda e parë ishte të provoj të ekzelutojë një shell të largët duke përdorur komandën *rsh* pa një vërtëtim të mëtejshëm. Në se kjo funksionon, shell-i largët ngarkon programin worm dhe vazhdon të infektojë makinat e reja nga aty ku ndodhet.

Metoda e dytë bën të përdor një program prezent në të gjitha sistemet BSD të quajtur *finger* që lejon një përdorues kudo në internet të shkruajë,

finger name@site

për të shfaqur informacion rreth një personi në një instalim të veçantë, të hollësishëm. Metoda e tretë varet nga një “bug” në sistemin email, *sendemail*, i cili lejon worm të postojë një kopje të bootsrap dhe ta ekzekutojë.

Sapo vendoset, worm provon të thyej password-in e përdoruesit. Çdo password i thyer lejon worm-in të logohet në çdo makinë që i zoti i password-it e kishte në llogari.

Sa herë që një worm fiton akses në një makinë të re, ai kontrollon për të parë në se ndonjë tjetër kopje worm-i ishte e aktivizuar tashmë atje. Në se po, kopja e re hiqet në të shumtata e rasteve.

9.5.6 Mobile Code

Viruset dhe worm-et janë programe që futen në një kompjuter pa dijeninë e të zotit dhe kundër vullnetit tij. Ndonjeherë, megjithatë, me ose pa qëllim importojnë dhe ekzekutojnë kode të huaja në makinat e tyre. Zakonisht ndodh kështu. Në të shkuarën (që, në botën e Internetit, do të thotë vitin që shkoi) të shumtë e faqeve Web ishin vetëm skedarë statik HTML me pak imazhe të bashkuar. Në ditët tona, gjithnjë e më shumë, shumë faqe Web përmbajnë programe të vogla të quajtura **applet-e**. Kur një faqe Web që përmban aplete është shkarkuar (downloaded), apletet janë terhequr dhe egzekutuar. Per shembull, një mund të përmbajnë një forme që duhet plotesuar, plus ndihmë bashkëvepruese në plotesimin e saj. Kur forma plotesohet, mund të dërgohet diku përmes Internetit për tu egzekutuar.

Një tjetër shembull në të cilën programet janë të transportuar nga një makinë në tjetrën për ekzekutim në makinën destinacion janë **agents** (agjent, spiunë). Këto janë programe që janë lëshuar nga përdoruesi për të përbushur disa detyra dhe më pas të raportojnë. Per shembull, një agjent mund të pyetet për të kontrolluar disa faqeve Web të udhëtim për të gjetur udhëtimin më të lirë nga Amsterdami në San Francisko. Sapo mbërrin në secilin site, agjenti ekzekutohet atje, merr informacionin që i nevojitet, më pas lëviz në Web site-in tjetër. Mbasi ka mbaruar në të gjitha, ai mund të kthehet në shtëpi dhe të raportojë për atë çka mësuar.

Shembulli i tretë i kodeve mobile është një skedarë PostScript që duhet të printohet në një printer PostScript. Një skedarë PostScript është në fakt një program në gjuhën e programimit PostScript që është i ekzekutuar brenda printerit. Ai normalisht i tregon printerit për të hartuar, vizatuar grafikë dhe më pas t'i mbush ato, por mund të bëj fare mire çdo gjë tjetër që do.

Nga keto diskutime të gjatë për viruset dhe worm, bëhet e qartë se kodet e huaja po të ekzekutohen në makinën tuaj mund të përbëjnë një rrezik. Megjithatë, disa njëre zë dëshirojnë gjithsesi t'i ekzekutojnë këto programe të huaja, kështu që lind pyetja: "Mundet që kodet mobile të ekzekutohen të sigurta?" pyetja e shkurtër është: "Po, por jo lehtësisht.". Problemi më thelbësor është se kur një proces importon një applet ose kode të tjera mobile ne hapsiren e tij te adresave dhe i egzekuton ato, ky kode eshte duke u ekzekutuar si një pjesë e një procesi përdoruesi të mirëqënë dhe ka të gjithë fuqinë që përdoruesi ka, duke pëfshirë aftësinë e të lexuarit, të shkruarit, fshirjes ose enkriptimit e skedarëve në diskun e përdoruesit dhe më shumë.

Kohë më parë, sistemet operative krijonin koncepte procesesh që ngrinin mure midis përdoruesve. Ideja ishte që secili proces kishte hapësirën e adresave mbrojtëse të tij dhe UID e tij, duke lejuar të prekej skedari dhe burime të tjera përmes tij, por jo një përdoruesi tjetër. Për të siguruar mbrojtjen përsëri një pjesë e procesit (appleti) dhe pjesa tjetër, konspekti i procesit nuk ndihmonin. Thread-ët lejojnë depërtime të shumëfishta të kontrollit brenda një procesi, por nuk bën asgjë për të mbrojtur një thread kundër një tjetri.

Teorikisht, ekzekutimi i secilit applet si një proces i ndarë ndihmon paksia por është shpesh jo efikas. Per shembull, një web page mund të përmbajë dy ose më shumë applete që bashkëveprojnë njëri me tjetrin dhe me të dhënat në web page. Browseri i web-it mund të ketë nevojë të bashkëveprojë më applete, të nisë dhe t'i ndalojë, ti ushqejë data e tyre dhe kështu me radhë. Në se secili applet është vendosur në procesin e tij, e gjithë gjëja

nuk do të funksionojë. Për më tepër, të vendosurit e një appleti në hapësirën e adresave të tij nuk e bën më të vështirë për appletin për të vjedhur ose dëmtuar datat. Është edhe më e lehtë në se asnjë nuk po sheh atje.

Metoda të reja të ndryshme e sjelljeve me apletët (dhe me kodet mobile në përgjithësi) është propozuar dhe implementuar. Më poshtë do të shohim tre nga këto metodat: sandboxing, interpretimi (interpretation) dhe code signing (nënshkrimi i kodit). Çdonjëra ka të avantazhet dhe disavantazhet e tij.

Sandboxing

Metoda e parë, e quajtur **sandboxing**, tenton të kufizojë çdo applet drejt një vargu të kufizuar adresash virtuale në një kohë ekzekutimi (run time). Ai punon duke ndarë hapësirën e adresave virtuale në shtresa (zona) më madhësi të barabartë, të cilët ne do t'i quajmë sandboxes. Çdo sandbox mund të ketë tipar që të gjitha adresat e tij ndajnë disa stringe të biteve high-order. Për një hapësirë adresë 32 bit, ne mund ta ndajmë atë deri në 256 sandboxes në 16-MB vijë kufiri, kështu të gjitha adresat brenda një sandbox ka një 8 bit-ësh të sipërm unik. E njëvlershme me te, ne mund të kemi 512 sandboxes në 8-MB kufi, me çdo sandbox që ka një 9-bit prefix adresë. Madhësia e sandbox-it duhet të zgjidhet aq e madhe sa të mbajë applet-in më të madh pa humbur shumë hapësirë adresë virtuale. Memoria fizike nuk është një problem në se faqja e kërkuar është prezente, siç është zakonisht. Çdo applet i jepen dy sandbox, një për kodin dhe një për datat, siç tregohet për rastin e 16 sandbox-eve të 16-MB secila, Fig. 9.18(a).

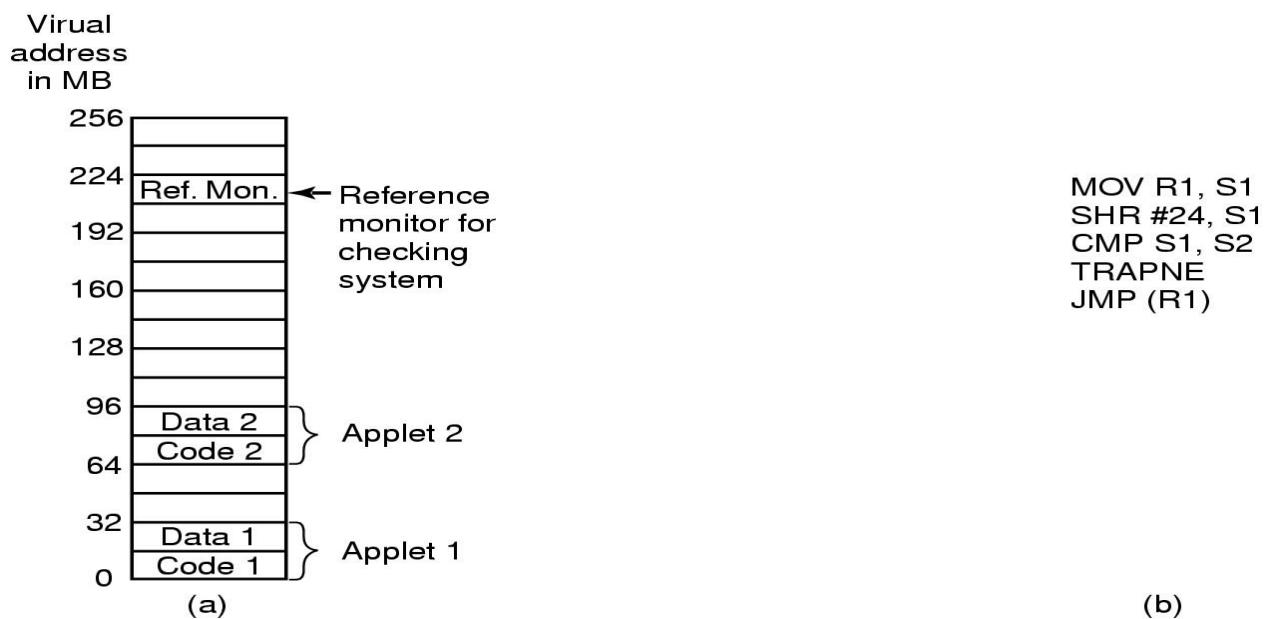


Figure 9-18. (a) Memoria e ndarë në 16MB sandboxes. (b) Një mënyrë e kontrollimit të një instruksioni për vlefshmëri.

Ideja kryesore që fshihet mbrapa një sandbox-i është të garantojë që një applet nuk mund të kërcejë në një kod jashtë kodit tij sandbox ose datave referencë jashtë të dhënave të tij sandbox. Arsyja e të pasurit e dy sandbox-eve është te parandalojë një applet nga modifikimi i kodit të tij gjatë ekzekutimit për të marrë këtë frenim. Duke parandaluar të gjitha përfshirjet brenda kodit sandbox, ne eliminojmë rrezikun e vetë-modifikimit të kodit. Për sa kohë një applet është i mbyllur në këtë rrugë, nuk mund të dëmtojë browser ose applet-et e tjera, viruset plant (“e mbjellë”) në memorie, ose dëme të ndryshme në memorie.

Një problem që duhet të zgjidhet është se çfarë ndodh kur një applet provon të bëj një system call? Zgjidhja këtu do të ishte e qartë. Instruksioni i thirrjes së sistemit zëvendësohet nga një thirrje të një moduli special i quajtur një **këshilltar (vëzhgues) referencë** (reference monitor) në të njëjtën mënyrë sikur kontrolli i adresave dinamike të ishte shtesë. Një tjetër rrugë, reference monitor ekzaminon çdo tentativë thirrje dhe vendos në se është e sigurt të kryhet. Në se thirrja mendohet e pranueshme, thirrja lejohet të proçedojë. Në se thirrja shihet si e rrezikshme ose reference monitor nuk mund ta tregojë, appleti vritet. Në se reference monitori mund të tregojë cili applet e thirri atë, një reference monitor i vetëm diku në memorie mund të perballoje te gjithe kerkasat nga te gjithe aplletet. Normalisht një reference monitori mëson rrith lejeve nga një skedar konfigurimi.

Interpretimi

Rruja e dytë e ekzekutimit të applet-eve të pasigurta është ekzekutimi i tyre në mënyrë interpretive dhe ti lejosh të kenë kontroll në hardware. Kjo është një rrugë e përdorur nga web browser. Appletet web page zakonisht shkruhen në Java ose në një gjuhë skriptive e nivelit të lartë si një TCL e sigurtë ose Javascript. Apletet java në fillim kompilohen në një gjuhë makine stack-oriented virtuale e quajtur **JVM (Java Virtual Machine)**. Janë këto appletet JVM që vendosen në web page. Kur ato download-hen, ato futen brenda një interpretuesi JVM brenda browser-it siç tregohet në Fig. 9-19.

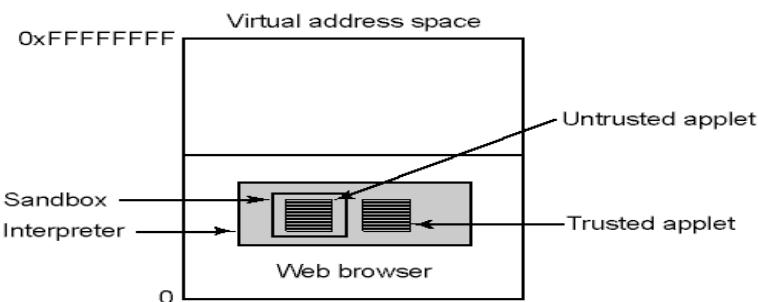


Figure 9-19. Appletet mund të interpretohen nga një Web browser.

Avantazhi i ekzekutimit të kodeve të interpretuara mbi kodet e kompliuara, është se çdo instruksion është i ekzaminuar nga interpretuesi përpara se të ekzekutohet. Kjo i jep mundësi interpretusit të kontrollojë në se adresa është e vlefshme. Veç kësaj, thirrjet e sistemit janë gjithashtu të zëna dhe të interpretuara. Se si manovrohet me këto thirrje është detyrë e politikës së sigurimit. Per shembull, në se një applet nuk është i besueshëm (ato që vijnë nga Interneti) ai mund të vendoset në një sandbox për të kufizuar sjelljen e tij.

Code signing (kodet e nënshkruara)

Një rrugë tjetër lidhur me sigurinë applet-eve është të dish nga vijnë ato dhe të pranosh vetëm applete nga burime të besueshme. Më këtë orvatje, një përdorues mund të mbroj një list applet-esh të besueshme tregtare dhe të ekzekutosh vetëm këto applete nga këta shitësa. Në këtë orvatje, asnjë mekanizëm aktual sigurie nuk është prezantë në kohën e ekzekutimit. Appletet nga shitës të besueshëm, ekzekutohen si të tillë dhe kodet nga një tjetër shitës nuk ekzekutohet si i plotë ose në rrugë të kufizuar.

Që ta bëjmë këtë skemë që të funksionojë, duhet të ketë një rrugë për përdoruesin për të përcaktuar që një applet është shkruajtur nga një shitës i besueshëm dhe nuk është modifikuar nga askush pasi është prodhuar. Kjo bëhet duke përdorur një nënshkrim dixhital, i cili i lejon tregtarëve të nënshkruajnë (firmosin) applet-in në një mënyrë të tillë që modifikime të ardhshme mund të zbulohen.

Kodi i nënshkruar (i firmosur) bazohet në një kriptograf çelës-publik. Një applet nga një shitës, në mënyrë tipike një kompani software-sh, gjeneron një çelës (çelës privat), duke bërë publik të mëparshmin dhe duke e ruajtur me zell të fundit. Për të nënshkruar një applet, për të marrë një numër 128-bit ose 160-bit, varet se ku përdoret në MD5 ose SHBA. Më pas ai nënshkruan vlerën ngatërruese (hash) duke e inkriptuar atë me kodin e tij privat. Ky nënshkrim shoqëron appletin kudo që ai shkon.

Kur përdoruesi merr appletin, browseri llogarit funksionin hash në mënyrën e tij. Më pas ai dekripton nënshkrimin shoqërues duke përdorur kodin publik të shitësit dhe më pas krahason vlerën e funksionimit hash të deklaruar nga tregetari me atë që vetë browseri ka llogaritur. Në se ato përputhen, appleti pranohet si i vërtëtë. Përndryshe refuzohet si falsifikim. Përfshirja e matematikës e bën atë së tepërmë të vështirë për çdokënd për tu ngatërruar më appletin, në një mënyrë të tillë që funksioni hash i tij do të llogarisë funksionin hash që përmban duke dekriptuar nënshkrimin e vërtëtë. Është po ashtu e vështirë për të gjeneruar një nënshkrim të ri false, që llogaritet pas çelësit privat. Proçesi i nënshkrimit dhe verifikimit është ilustruar në Fig. 9-20.

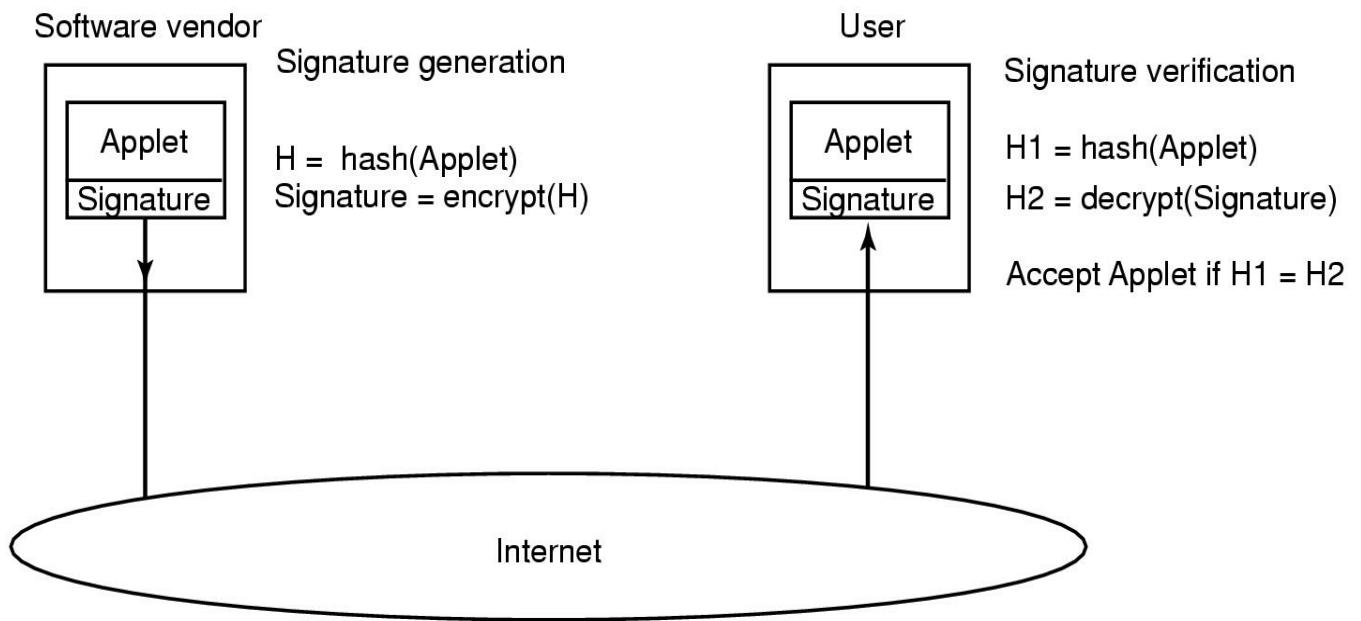


Figure 9-20. Si funksionojë kodet e nënshkruar.

KAPITULLI I DHJETË

UNIX DHE LINUX

10.2 NJË VESHTRIM MBI UNIX

Në këtë seksion do të japim një hyrje të përgjithshme mbi UNIX dhe mënyrën e përdorimit të tij, për ato përdorues që nuk janë familjarizuar akoma me të. Megjithëse versione të ndryshme të UNIX ndryshojnë në mënyrë të lehtë, materiali i prezantuar këtu përdoret në të gjitha versionet. Këtu do të fokusohemi në atë se si duket UNIX në terminal. Në seksionet pasardhëse do të fokusohemi në thirrjet sistem dhe mënyrën se si ato punojnë në brendësi.

10.2.1 Synimet e UNIX

UNIX është një sistem interaktiv i projektuar për të mbajtur shumë procese dhe shumë përdorues në të njëjtën kohë. Ai ishte projektuar nga programuesit, për programuesit, për tu përdorur në një mjeshtëri i cilin shumica e përdoruesve janë relativisht të sofistikuar dhe të angazhuar në projektet e zhvillimit të software-ve. Në shumicën e rasteve, një numër i madh programuesish janë aktiv në bashkëveprim për të prodhuar një sistem të vetëm, kështu që UNIX ka lehtësira të mëdha për të lejuar njerëzit të punojnë së bashku dhe të ndajnë informacionin në mënyrë të kontrolluar. Modeli i një grupei programuesish me ekspériencë duke punuar së bashku të myllur, për të prodhuar software të avancuara është shumë ndryshe nga modeli i kompjuterave personalë me një fillestar që punon i vetëm me një word procesor, dhe kjo diferençë dallohet qartë kudo në UNIX nga fillimi në fund.

Cfarë është ajo që një përdorues i mirë do në një sistem?

Duke filluar me më të preferuarën, që ai të jetë sa më i thjeshtë, elegant dhe i qëndrueshëm. Për shembull, në nivelin më të ulët, një file duhet të jetë një bashkësi byte-sh. Duke pasur klasa të ndryshme file-ash për aksesim të vazhdueshëm, të rastesishëm, të akorduar, të largët etj.. (siç bëjnë mainframe) ato vetëm na pengojnë. Në mënyrë të ngjashme në se komanda

Is A*
do të thotë listo të gjithë file-t duke filluar nga "A", rrjedh që komanda

rm A*
duhet të zhvendosë të gjithë file-t duke filluar nga "A" dhe të mos zhvendosë një file emri i së cilës përmban një "A" dhe një astëriks(*). Kjo karakteristikë shpesh quhet edhe parimi i të paktën një surprize (principle of least surprise).

Një tjetër gjë që programuesit me ekspériencë duan është fuqia dhe fleksibiliteti. Kjo do të thotë që një sistem duhet të ketë një numër të vogël elementesh bazë në mënurë që të kombinohen në një pafundësi mënyrash për të përbushur kërkesat e aplikacionit. Një nga parimet e UNIX është që çdo program duhet të kryejë vetëm një punë, por këtë punë ta bëjë shumë mirë. Kështu që kompilatorët nuk prodhojnë lista sepse programe të tjera mund ta bëjnë më mirë këtë punë.

Përfundimisht, shumicës së programuesve nuk ju pëlqejnë tëpricat e padobishme. Pse shtypim *copy* kur *cp* mjafton? Për të nxjerrë të gjithë rreshtat që përban stringa "ard" nga një file, programuesi në UNIX shkruan:

```
grep ard f
```

Ruga tjetër është që programuesi në fillim duhet të zgjedhë programin *grep* (pa argumentë), dhe më pas *grep* duhet të prezantojë veten. Më pas, grep kërkon menjeherë për emrin e file-s. Pastaj pyet në se ka ndonjë emër tjetër file. Përfundimisht, ai përmbledh shkurt se çfarë do të bëjë dhe pyet në se çdo gjë është në rregull. Ndërsa kjo lloj ndërsa që përdoruesi mund të jetë ose jo e përshtatshme për njerëzit e thjeshtë filletarë, ajo është irrituese për programuesit ekspertë.

10.2.2 Ndërfaqet në UNIX

Një sistem UNIX mund të shikohet si një lloj piramide, siç ilustrohet në figurën 10-1. Në fund është hardware, i cili konsiston në CPU, memorje, disqe, terminale dhe paisje të tjera. Duke vazhduar më lart kemi sistemin operativ UNIX. Funksioni i tij është që të kontrollojë hardware dhe të ofrojë një ndërfaqe për thirrjet sistem për të gjithë programet. Këto thirrje sistem i lejojnë programet user të krijojnë dhe menaxhojnë proceset, file-at dhe burime të tjera.

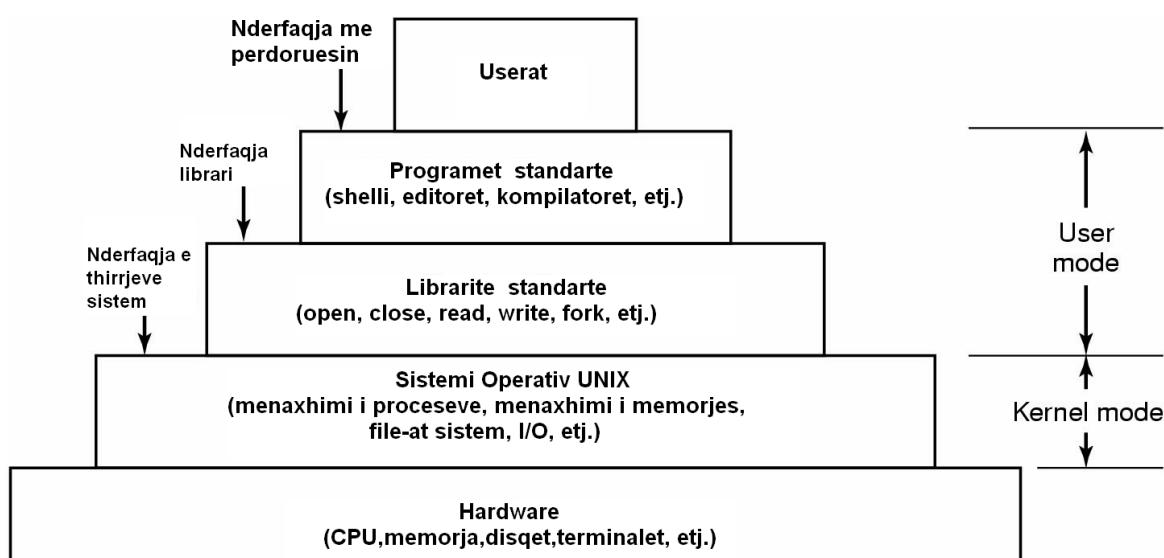


Figura 10-1. Shtresat në një sistem UNIX

Programet i bëjnë thirrjet sistem duke vendosur argumentat në regjistra (ose ndonjeherë në stack), duke lëshuar një trap instruction për të kaluar nga mënyra user në mënyrën kernel për të startuar UNIX. Meqënë se nuk ka ndonjë mënyrë për të shkruar një trap instruction në C, ofrohet një librari, me një procedurë për çdo thirrje sistem. Këto procedura janë të shkruara në gjuhën asembler, por mund të thirren nga C. Çdo procedurë fillimisht vendos argumentat në vendin e duhur, dhe më pas ekzekuton trap instruction. Kështu që për të bërë një thirrje sistem *read*, një program në C mund të thérresë librarinë e procedurës *read*. Pra, është ndërfaqja e librarisë dhe jo ndërfaqja e thirrjes sistem, që specifikohet nga POSIX. Me fjalë të tjera, POSIX tregon se cila librari procedurash duhet të ofrohet për t'iu përshtatur sistemit, çfarë janë parametrat e tyre, çfarë duhet të bëjnë dhe çfarë rezultati duhet të kthejnë. Ai nuk i referohet njëlloj thirrjes aktuale sistem.

Veç sistemit operativ dhe librarisë së thirrjeve sistem, të gjitha versionet e UNIX ofrojnë një numër të madh programesh standarte, disa prej të cilëve specifikohen nga standarti 1003.2 i POSIX, dhe disa prej të cilëve ndryshojnë mes versioneve të UNIX. Këto përfshijnë komand procesorin (shell-in), kompilatorët, editorët, editorët tekst si dhe shërbimet e manipulimit të file-ve. Janë këto programe që një përdorues kërkon në një terminal.

Kështu që mund të flasim për tre ndërfaqe të ndryshme në UNIX: ndërfaqja e thirrjeve të vërteta sistem, ndërfaqja e librarive si dhe ndërfaqja e formuar nga një bashkësi standartesh programesh. Megjithëse e dyta ka të bëjë me atë se çfarë mendon një user i rastesishëm mbi "UNIX", në fakt, ai nuk ka pothuajse asgjë për të bërë me sistemin operativ dhe mund të zëvendësohet lehtësisht.

Disa versione të UNIX për shembull, kanë zëvendësuar ndërfaqen e tastierës me ndërfaqe mausi duke mos ndryshuar sistemin operativ në tërsi. Eshtë pikërisht ky fleksibilitët që e bën UNIX kaq populor, dhe e ka lejuar atë t'ju mbijetojë shumë mirë ndryshimeve thelbësore në teknologji.

10.2.3 Shelli i UNIX

Shumë sisteme UNIX kanë një ndërfaqe grafike me përdoruesin, që u përhap dhe tek Macintosh si dhe më vonë tek Windows. Megjithatë, programuesit e vërtetë ende preferojnë një ndërfaqe *command line* të quajtur *shell*. Kjo ndërfaqe është shumë e shpejtë për tu përdorur, më shumë e fuqishme, lehtësisht e zgjatshme, si dhe nuk e bën përdoruesin të përdorë mausin gjatë gjithë kohës. Më poshtë do të përshkruhet shkurtimisht Bourne shell (*sh*). Qëkur, shumë shell-e të rinj kanë qenë shkruar (*fev/i*, *bash*, etj.). Megjithëse UNIX mbështet plotësisht një mjedis grafik (X Windows), përsëri shumë programues bëjnë panele komandimi (Windows console) dhe veprojnë si të kenë një gjysmë duzine me terminale ASCII, secili duke u ekzekutuar në shell.

Kur shell-i startohet, ai inicializon veten, më pas shtyp një karakter **prompt**, shpesh një "%" ose "\$", dhe pret që përdoruesi të shtypë në ekran një komandë.

Kur përdoruesi formon komandën, shell-i nxjerr fjalën e parë prej saj, duke supozuar që është emri i një programi për tu ekzekutuar, kërkon për këtë program, dhe në se e gjen, e ekzekuton. Shell-i pastaj ndërpërt vetveten derisa programi të përfundojë, kohë në të cilën ai provon për të lexuar komandën pasardhëse. Ajo çfarë është e rendësishme këtu është thjesht vrojtimi që shell-i është një program user i zakonshëm.

Gjithçka që i duhet është të lexojë dhe shkruajë në terminal, dhe fuqia për të ekzekutuar programet e tjera.

Komandat mund të marrin argumente, të cilat i kalohen programit të thirrur si karaktere string. Për shembull, komanda

```
cp sre dest
```

thërrit programin *cp* me dy argumentë, *sre* dhe *dest*. Ky program interpretohet që, e para, (*sre*) të jetë emri i një file ekzistues. Bën një kopje të këtij file dhe thërret kopjen *dest*.

Jo të gjithë argumentat janë emra file-sh. Në

```
head -20 file
```

argumenti i parë -20, i tregon *head* që të printojë 20 rreshtat e parë të file-it, në vend të numrit të zakonshëm 10, të rreshtave. Argumentat që kontrollojnë operacionet e komandës, ose specifikojnë një vlerë të mundshme quhen **flag-e**, dhe me marrëveshje shënohen me të trashë. Kjo trashësi përdoret për të evituar dykuptimësite, sepse komanda

```
head 20 file
```

është e lejueshme, dhe i tregon *head* që të printojë 10 rreshtat e parë të file-s të quajtur 20, dhe më pas të printojë 10 rreshtat e file-s së dytë të quajtur *file*. Shumë komanda të UNIX pranojnë shumë flage dhe argumentë.

Për ta bërë më të lehtë për të specifikuar shumë emra file-sh, shelli pranon **karakteret magjike**, shpesh të quajtura **wild cards**. Një asteriks për shembull, përfshin të gjitha stringat e mundëshme, kështu që

Is *.c

i tregon *ls* të listojë të gjithë file-t emri i të cilave mbaron me .c. Në se emërtimet e file-ve x.c, y.c dhe z.c ekzistojnë të gjitha, komanda e mësipërme është ekuivalente me

Is x.c y.c z.c

Një tjetër wild card është çështja e emërtimit, që përputhet me ndonjë karakter. Një listë karakteresh brenda një kutie mbajtëse zgjedh një prej tyre, kështu që

Is [ape]*

liston të gjithë file-t që fillojnë me "a", "p", ose "e".

Një programi si shelli nuk i duhet të hapë terminalin në rregull për të shkruar apo lexuar nga ai. Më mirë, kur ai (ose ndonjë program tjetër) të startohet, ai ka automatikisht akses në një file të quajtur **standart input** (për lexim), një file të quajtur **standart output** (për të shkruar dalje normale), dhe një file të quajtur **standart error** (për të shkruar mesazhet gabim). Normalisht, të treja nuk paraqiten në terminal, kështu që leximi nga standart inputi vjen nga tastjera dhe shkrimi i standart output dhe standart error dalin në ekran. Shumë programe në UNIX lexojnë nga standart inputi dhe shkruajnë në standart output në mënyrë default. Për shembull,

```
sort
```

thirr një program sort, që lexon reshtat nga terminali (derisa user-i të shtypë CTRL-D, për të treguar fundin e file-s) i rendit ato nga ana alfabetike dhe shkruan rezultatin në ekran.

Gjithashtu është e mundur që të riadresojmë standart inputin *dhe* standart outputin, që shpesh është e dobishme. Sintaksa për riadresimin e standart inputit përdor shenjën (<) të shoqëruar nga emri i file-s hyrëse. Në mënyrë të ngjashme, standart outputi adresohet duke përdorur shenjën më e madhe(>). Eshtë i ndaluar adresimi i të dyave me të njëjtën komandë. Për shembull, komanda

```
sort <in>out
```

e bën *sort* të marrë një input nga file *in*, dhe të shkruajë outputin e tij tek file *out*. Meqenë se standart errori nuk u riadresua, në ekran shfaqet ndonjë mesazh gabimi. Një program që lexon inputin e tij nga standart inputi, bën disa përpunime në të, dhe shkrimi i ouputit në standart ouput është quajtur **filtër**.

Konsideroni komandën e mëposhtme e cila konsiston në 3 komanda të ndara:

```
sort <in>temp; head -30 <temp; rm temp
```

E para ekzekuton *sort*-in, duke marrë inputin nga *in* dhe duke e shkruar outputin në *temp*. Kur kjo ka mbaruar, shelli ekzekuton *head*, duke i treguar atij që të printojë 30 rreshtat e parë të *temp* dhe ti printojë ato në standart output. Në fund file e përkohëshme *temp* është zhvendosur.

Ndodh shpesh që programi i parë në panelin e komandës (command line) të përdorë outputin që është përdorur si input në programin pasardhës. Në shembullin e mësipërm, ne përdorëm file-n *temp* pér të mbajtur këtë output. Megjithatë, UNIX ofron një ndërtim të thjeshtë pér të bërë të njëjtën gjë. Në komandën

```
sort <m | head -30
```

vija vertikale, e quajtur **pipe symbol**, do të thotë merr outputin nga *sort* dhe përdore atë si input tek *head*, duke eleminuar kështu nevojën pér krijimin, përdorimin dhe zhvendosjen e file-s së përkohëshme. Një grupim komandash të lidhura nga pipe symbol, quhet **pipeline**, dhe mund të mbajë arbitrarisht shumë komanda. Një pipeline me katër komponentë tregohet nga shembulli i mëposhtëm:

```
grep ter *.t | sort | head -20 | tail - 5 >foo
```

Këtu të gjithë rreshtat përbajnjë stringën "ter" dhe të gjitha file-t që mbarojnë me *t* janë shkruar në standart output, aty ku edhe janë ruajtur. 20 rreshtat e parë selektohen nga *head*, i cili më pas ia kalon *tail*, i cili shkruan 5 të fundit (rreshtat 16-20 në listën ku ruhen) në *foo*. Ky është një shembull që tregon se si UNIX ofron blloqe të zakonshme ndërtimi, çdonjëri prej të cilëve bën një punë.

UNIX është një sistem multiprogramimi me qëllim të përgjithshëm. Një user i vetëm mund të ekzekutojë disa programe njehersh, secilin si një proces të ndarë. Sintaksa e shellit pér ekzekutimin e një procesi në background është të ndjekë me imtësi komandën e tij. Kështu që

```
wc -|<a>b &
```

ekzekuton programin e numërimit të fjalëve, *wc*, pér të numëruar numrin e rreshtave (-l flaget në inputin *a*), duke shkruar rezultatin në *b*, por duke i bërë këto në background.

Sapo komanda shtypet, shelli shtyp promptin dhe është gati për të pranuar dhe mbajtur komandën pasardhëse. Pipeline gjithashtu mund të vendoset në background, për shembull, me anë të komandës

```
sort <x | head &
```

Shumë pipeline mund të ekzekutohen në background në të njëjtën kohë.

Eshtë e mundur të vendosim një listë me komandat e shellit në një file dhe më pas të starojmë shellin me këtë file si një standart input. Shelli (i dytë) i shqyrton ato me qetësi, sikur të ishin shtypur komandat nga tastjera. File-t që përbajnë komandat e shellit quhen skripte shelli (shell scripts). Skriptet e shellit mund të caktojnë vlerat e variabave të shellit dhe më pas ti lexojnë ato më vonë. Ato mund të kenë gjithashtu parametra, dhe ti përdorin *if*, *for*, *while* dhe konstrukte të rastit. Kështu që një skript shelli është një program i shkruar në gjuhën shell. Shelli Berkley C është një alternativë shelli që ka qenë dizenuar për të bërë skripte shelli (dhe gjuhën e përgjithëshme të komandave) e ngjashme me programet e C në shumë aspekte. Meqenë se shelli është një program user i dytë, shumë njërz të tjerë kanë shkruar dhe shpërndarë versione të tjera të shelleve.

10.2.4 Programet e dobishme në UNIX

Ndërfaqja me përdoruesin në UNIX konsiston jo vetëm tek shelli, por dhe në një numër të madh programesh standarte të dobishme. Afersisht këto programe mund të ndahen në 6 kategori si më poshtë:

Komandat e manipulimit të file-ve dhe direktive.

Filtrat.

Mjetet (tools) e zhvillimit të programit siç janë editorët dhe kompilatorët.

Përpunuesit e tekstit.

Administrimi i sistemit.

Të tjera.

Standarti 1003.2 i POSIX specifikon sintaksën dhe semantikat e 100 prej tyre, kryesisht në tre kategoritë e para. Ideja e standartizimit të tyre është që çdonjëri të mund të shkruajë skripte në shell, ti përdorë këto programe dhe ato të punojnë në të gjitha sistemet UNIX. Përveç këtyre standardeve të dobishme, ka edhe shumë programe (aplikacione) siç janë: Web browsers, image viewers, etj.

Le të konsiderojmë disa shembuj të këtyre programeve, duke filluar me manipulin e file-ve dhe direktive:

```
cp a b
```

kopjon file-n *a* tek *b*, duke lënë file-n original të padëmtuar. Në kontrast me të,

```
mv a b
```

kopjon *a* tek *b* por zhvendos vlerën origjinale. Më mirë të themi kjo komandë zhvendos file-n sesa bën një kopje të saj në kuptimin e zakonshëm. Disa file mund të jenë të lidhura duke përdorur *cat*, e cila lexon çdo input të file-ve dhe i kopjon ato në standart output, njëra pas tjetrës. File-t mund të zhvendosen me anë të komandës *rm*. Komanda *chmod*

lejon përdoruesit të ndryshojë renditjen e biteve për të modifikuar të drejtat e aksesimit. Direktoritë mund të krijohen me *mkdir* dhe mund të zhvendosen me *rmdir*. Për të parë listën e file-ve në një direktori, mund të përdoret komanda *ls*. Ajo ka një numër të madh flagesh për të kontrolluar sa më shumë detaje rrëth file-ve të shfaqur (për shembull, madhësia, grupi, data e krijimit, etj.), për të përcaktuar rregullat e renditjes (për shembull, nga alfabeti, hera e fundit e modifikimit, ndryshimi, etj.), për të specifikuar paraqitjen në ekran dhe shumë të tjera.

Ne kemi parë tashmë disa filtra: *grep* nxjerr rreshtat e përmbajtur nga një strukturë e dhënë nga standart inputi ose një apo më shumë file input; *sort* rendit inputin dhe shkruan outputin në standart output; *head* nxjerr rreshtat e parë të inputit të tij; *tail* nxjerr rreshtat e fundit të inputit të tij. Filtra të tjerë të përcaktuar nga standarti 1003.2 janë *cut* dhe *paste*, të cilat lejojnë kolonat e tekstit të priten dhe të ngjiten në file; *od* e cila konvertion inputin e saj në tekstin ASCII, oktal, decimal apo hekzadecim; *tr*, e cila bën përkthimin e karaktereve (për shembull, nga shkronjë e vogël në të madhe); dhe *pr* që formaton outputin për printerin, duke pasur mundësinë për të printuar kokat e faqeve, numrin e tyre dhe kështu me rradhë.

Kompiluesit dhe mjetet e programimit përfshijnë *cc*, që thërret kompilatorin e C dhe *ar*, e cila grumbullon procedurat e librarisë në file të arkivuara.

Një tjetër mjet i rëndësishëm është *make*, e cila përdoret për të mirëmbajtur programet e mëdha, kodet burim të të cilave konsistonë në shumë file. Zakonisht, disa prej tyre janë **file-t header**, të cilat përbajnë tipin, variablin, dhe deklarime të tjera. File-t burim shpesh i përbajnjë këto duke përdorur një direktivë të veçantë *include*. Në këtë mënyrë, dy ose më shumë file burim mund të share-ojnë të njëjtat deklarime. Megjithatë në se një file header është i modifikuar, është e nevojshme të gjejmë të gjitha file-t burim që varen nga ai, dhe më pas ta rikompilejmë. Funksioni i *make* është që të mbajë gjurmën e çdo file që varet nga headeri, dhe të bëjë të mundur që të gjitha komplimet e nevojshme të bëhen automatikisht. Pothuajse të gjitha programet në UNIX, duke përjashtuar ato të voglat, bëhen që të kompilohen me komandën *make*.

Një përzgjedhje e programeve të dobishme në POSIX tregohet në figurën 10-2, me një përshkrim të shkurtër për secilin. Të gjitha sistemet UNIX i kanë këto programe dhe shumë të tjera.

Programi	Përdorimi i zakonshëm
<i>cat</i>	Lidh shumë file në standart output
<i>chmod</i>	Ndryshon mënyrën e mbrojtjes (protect) së file-s
<i>cp</i>	Kopjon një ose më shumë file
<i>cut</i>	Pret rreshtat e tekstit nga një file
<i>grep</i>	Kërkon një file nga një paketë (strukturë)
<i>head</i>	Nxjerr rreshtat e parë të një file
<i>ls</i>	Lista e direktorisë
<i>make</i>	Komilon file-t për të ndërtuar një binar

mkdir	Krijon një direktori
od	Konverton inputin në oktal, binar, hexadecimal etj.
pastë	Ngjit rreshtat e tekstit në një file
pr	Formaton një file për printim
rm	Zhvendos një ose më shumë file
rmdir	Zhvendos një direktori
sort	Rendit sipas alfabetit rreshtat e një file
tail	Nxjerr rreshtat e fundit të një file
tr	Shpjegimi midis karaktereve të vendosura

Figura 10-2. Disa nga komandat e përbashkëta të programeve në UNIX të specifikuara nga POSIX

10.2.5 Struktura e Kernel-it

Në figurën 10-1 shohim strukturën e përgjithshme të një sistemi UNIX. Tani le të shohim kernelin përparrë se të shohim pjesët tjera. Paraqitura e strukturës së kernelit është pak e vështirë meqënë se kemi versione të ndryshme të UNIX, por megjithëse diagrama e figurës 10-3 përshkruan 4.4BSD, ajo mund të përdoret në shumë versione të tjera me disa ndryshime të vogla këtu dhe atje.

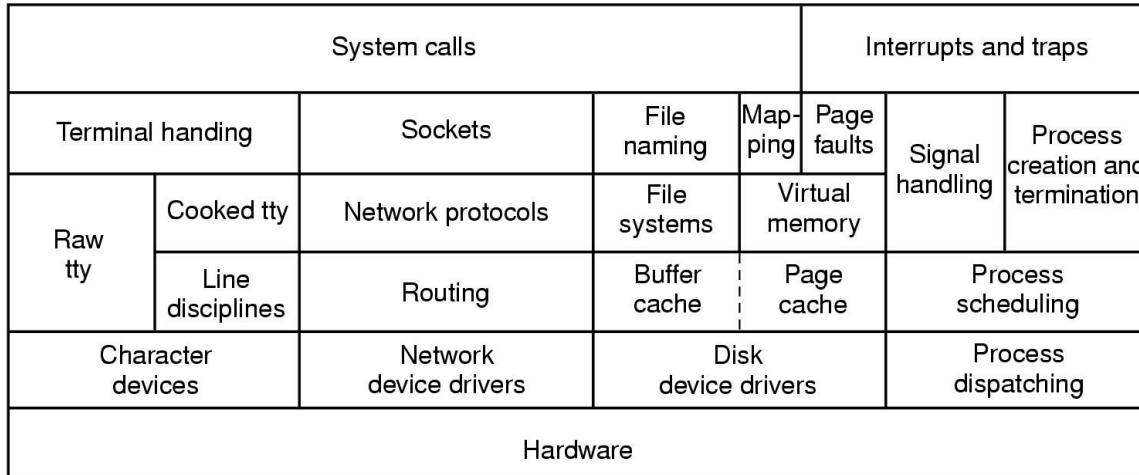


Figura 10-3. Struktura e kernelit 4.4BSD

Shtresa bazë e kernelit konsiston në driverat e paisjeve dhe dërguesin (dispatch) e proceseve. Të gjithë driverat e UNIX klasifikohen secili si driver i një paisje ose të një blloku paisjesh, me ndryshimin kryesor që kërkojnë të njihen si një bllok paisjesh dhe jo si një paisje e vetme. Teknikisht, paisjet e rrjetit janë paisje të veçanta, por ato sillen aq ndryshe saqë ekziston mundësia që ti ndajmë ato, siç është bërë në figurë. Dërgimi i proceseve (process dispatching) ndodh vetëm kur ndodh një interrupt. Kodi i nivelit të ulët këtu, ndalon procesin në ekzekutim, ruan gjendjen e tij në tabelën e proceseve të

kernelit dhe starton driverin e duhur. Dërgimi i proceseve gjithashtu ndodh kur kerneli mbaron punë dhe është koha për të filluar një proces user përsëri. Dërgimi (dispatching) i kodit bëhet në asembler dhe është plotësisht i dallueshëm nga skedulimi.

Mbi nivelin bazë, kodi është i ndryshëm në të katër kolonat në figurën 10-3. Në të majtë kemi paisjet karakter. Ato përdoren në dy mënyra. Disa programe, siç janë editorët vizualë si *emacs* dhe *vi*, duan çdo çelës sa herë që ato ndeshen. Terminali raw (tty) i I/O e bën këtë të mundur. Software tjetër, siç është shell (*sh*), është me rresht të orientuar dhe i lejon përdoruesve të modifikojnë rreshtin aktual, përpara se të shtypin ENTER dhe t'a dërgojnë atë programit. Ky software përdor mënyrën *cooked* dhe rregullat e rreshtit.

Software i network-ut është shpesh modular, me suport të ndryshëm për paisjet dhe protokollet. Shtresa mbi driverat e network-ut ka një lloj funksioni rutimi, duke bërë që një paketë e caktuar të shkojë në paisjen e duhur ose në mbajtësin e protokollit. Shumica e sistemeve UNIX përbajnjë në mënyrë të plotë funksionin e një ruteri brenda kernelit, megjithëse pëforma është më e vogël se në rastin e një ruteri hardware. Mbi kodin e ruterit ndodhet stack-u i protokollit aktual, gjithnjë duke përfshirë IP dhe TCP, por ndonjeherë edhe protokolle shtesë. Mbi gjithë network-un ndodhet ndërsaqja socket, e cila i lejon programet të krijojnë socketë për protokolle dhe network-e të posaçme, duke marrë mbrapa një fd (file descriptor) për çdo socket për ta përdorur më vonë.

Në krye të disk driverave është bufferi i cache-së së file-ve sistem dhe page cache-ja. Në sistemet e hershme UNIX, bufferi i cache-së ishte një pjesë fikse e memorjes, me gjithë pjesën e memorjes përfshirë user. Në shumë sisteme moderne UNIX, nuk ka një kufi të caktuar, dhe ndonjë faqe e memorjes mund të përdoret për çdo funksion, në varësi se kush nevojitet më shumë.

Në krye bufferit të cache-së janë file-t sistem. Shumica e sistemeve UNIX suportojnë shumë file sistem, duke përfshirë edhe *Berkley fast file system*, file-t sistem me strukturë logaritmike, si dhe file sistem të llojeve të ndryshme të sistemeve. Të gjitha këto file sistem share-ojnë të njëjtin buffer cache-je. Në krye të file-ve sistem është emërtimi i file-ve, menaxhimi i direktorisë, menaxhimi i linkeve të rënda dhe simbolike dhe karakteristika të tjera të file-ve sistem që janë të njëjta për të gjitha file-t sistem.

Në krye të page cache-së është sistemi i memorjes virtuale. E gjithë faqosja logjike është këtu, siç është edhe algoritmi i zëvendësimit të faqeve. Në krye të saj është kodi për planifikimin (mapping) e file-ve në memorjen virtuale dhe kodi për menaxhimin e page-fault-eve të nivelit të lartë. Ky është kodi që kpton se çfarë duhet bërë kur ndodh një page fault. Ai fillimisht kontrollon në se referenca në memorje është e vlefshme, dhe në se po, ku është vendosur faqja e kërkuar dhe si mund të merret ajo.

Kolona e fundit merret me menaxhimin e proceseve. Mbi dispatcher-in është skeduleri i proceseve, i cili zgjedh procesin pasardhës që do ekzekutohet. Në se proceset menaxhohen në kernel, edhe thread-et gjithashtu menaxhohen këtu, megjithëse thread-et në disa sisteme UNIX menaxhohen në hapësirën user. Mbi skedulerin është kodi për

përpunimin e sinjaleve dhe dërgimin e tyre në destinacionin e duhur, ashtu si kodi i krijimit dhe përfundimit të proceseve.

Në krye të shtresës është ndërfaqja në sistem. Në të majtë është ndërfaqja e thirrjeve sistem. Të gjitha thirrjet sistem vijnë këtu dhe drejtohen në një nga modulet më poshtë, në varësi nga natyra e thirrjes. Në të djathtë është hyrja për interruptet dhe instrukzionet trap, duke përfshirë sinjalet, page fault-et, përjashtimet e procesorit të të gjitha llojeve, dhe interruptet I/O.

10.3 PROÇESET NE UNIX

Në seksionin paraardhës, pamë UNIX nga pikpamja e tastierës, që do të thotë se si e shikonte useri në terminal. Dhamë shembuj mbi komandat e shellit si dhe programet që përdoren më shpesh. Në fund dhamë një paraqitje të shkurtër të strukturës së sistemit. Tani është koha që shikojmë më thellë në kernel dhe të vështrojmë më nga afér konceptet bazë që suportojnë UNIX, domethënë proceset, memorjen, file-t sistem dhe input/output. Këto nocione janë të rëndësishme sepse thirrjet sistem - ndërfaqja me sistemin operativ e programeve user - i manipulojnë ato. Për shembull, thirrjet sistem ekzistojnë për të krijuar procese, për të alokuar memorjen dhe për të bërë I/O. Fatkeqësisht, me kaq shumë versione ekzistuese të UNIX, përsëri ka disa ndryshime midis tyre. Në këtë kapitull, ne do të fokusohemi më shumë tek ato veti që janë të përbashkëta për të gjithë versionet, sesa në veti specifike të ndonjë versioni. Kështu që në disa seksione (veçanërisht në seksionet e implementimit), shqyrtimi mund të mos iu përshtatet njëlloj të gjitha versioneve.

10.3.1 Konceptet kryesore

Entitetet e vetme aktive në një sistem UNIX janë proceset. Proseset në UNIX janë shumë të ngjashëm me proceset që studiuam në Kapitullin 2. Çdo proces ekzekuton një program të vetëm dhe fillimisht ka vetëm një thread kontrolli. Me fjalë të tjera, ai ka një PC (Program Counter), i cili ruan gjurmën e instrukzionit pasardhës që do ekzekutohet. Shumica e versioneve në UNIX i lejojnë procesit të krijojë threade shtesë në të njëjtën kohë që ai fillon ekzekutimin.

UNIX është një sistem me multiprogramim, kështu që shumë procese të pavarura mund të ekzekutohen në të njëjtën kohë. Çdo user mund të ketë disa procese aktive në të njëjtën kohë, kështu që në një sistem të madh, mund të ketë me qindra apo edhe mijëra procese në ekzekutim. Në fakt, në shumicën e workstation-ave me një përdorues, edhe kur përdoruesi mungon shumë procese background, të quajtura **daemon**, ekzekutohen. Këto startohen automatikisht kur sistemi butohet.

Një daemon tipik është *cron daemon*. Ai zgjohet një herë në minutë për të parë në se ka ndonjë punë për të bërë. Në se ka, ai e bën këtë punë. Më pas ai kthehet përsëri në gjendjen e mëparshme derisa ti vijë koha për të bërë kontrollin tjetër.

Ky daemon është i nevojshëm sepse është i nevojshëm në UNIX skedulimi i aktiviteteve në minuta, orë, ditë ose edhe muaj në të ardhmen. Për shembull, supozoni se një përdorues ka caktuar një orar me dentistin në orën 3 të martën e ardhshme. Ai mund të bëjë një shënim në bazën e të dhënave të cron daemon duke i thënë daemonit që ta sinjalizojë atë, le të themi në orën 2:30. Kur dita dhe ora e caktuar vjen, cron daemoni shikon që ai ka punë për të bërë dhe programi i sinjalizimit startohet si një proces i ri.

Gjithashtu cron daemoni përdoret për të skanuar aktivitetet periodikë, si për shembull, bërja e backup-eve çdo ditë në orën 4 mbasdite. Tjetër rast i daemoneve janë posta elektronike (mail), menaxhimi i rreshtave që presin në rradhë për tu printuar, kontrolli në se ka mjaftueshëm faqe të lira në memorje, etj. Daemonet mund të implementohen mirë në UNIX sepse çdonjëri prej tyre është një proces i ndarë, i pavarur nga të gjitha proceset e tjera.

Proceset në UNIX krijohen në një mënyrë shumë të thjeshtë. Thirrja sistem *fork* krijon një kopje ekzakte të procesit origjinal. Prosesi origjinal quhet procesi prind, ndërsa procesi tjetër i lindur nga *fork* quhet procesi fëmijë. Secili prej proceseve prind dhe fëmijë kanë memorjen e tyre. Në se prindi më pas ndryshon ndonjë nga variablat e tij, këto ndryshime nuk shfaqen tek fëmija dhe e anasjellta.

File-t e hapur janë të share-uar midis prindit dhe fëmijës. Kështu, në se disa file ishin hapur tek prindi përpara *fork*, ato do të vazhdojnë të jenë të hapura tek prindi dhe fëmija edhe më pas. Ndryshimet e bëra ne një file nga çdonjëri (prindi ose fëmija) do të shfaqen tek tjetri. Kjo sjellje është e arsyeshme, sepse këto ndryshime shfaqen edhe në ndonjë proces që ska lidhje me to dhe që hap këtë file.

Fakti që pasqyrimi i memorjes, variablat, regjistrat dhe çdo gjë tjetër janë identike tek proceset prind dhe fëmijë, na çon në një vështirësi të vogël: Si duhet ta kuptojnë proceset se kush duhet të ekzekutojnë kodin e prindit dhe kush të fëmijës? Sekreti është që thirjet sistem *fork* kthejnë një vlerë 0 fëmijës dhe një vlerë jo 0, **PID (Process IDentifier)** e fëmijës i jepet prindit. Të dy proceset normalisht kontrollojnë vlerën e kthyer dhe veprojnë njëlloj siç tregohet në figurën 10-4.

```

pid = fork () ;                                /* në se fork arrihet, pid > 0 tek prindi*/
if (pid < 0) {
    handle_error () ;                         /* fork dështon (për shembull. memorja ose
disa                                         tabela janë plot)*/
} else if (pid > 0) {                          /* kodi i prindit shkon këtu. */
} else {                                         /* kodi i fëmijës shkon këtu. */
}

```

Figura 10-4. Krijimi i procesit në UNIX.

Proçeset emértohen nga PID-ja e tyre. Kur krijohet një proçes, prindit i jepet PID e fëmijës, siç u tha më lart. Në se fëmija do të dijë PID e tij, éshtë një thirrje sistem, *getpid*, që e mundëson këtë. PID përdoren në rrugë të ndryshme. Për shembull, kur një proçes fëmijë mbaron, prindit i éshtë dhënë PID e një fëmije që sapo mbaroi. Kjo mund të jetë e rëndësishme sepse një prind mund të ketë disa fëmijë. Meqënë se femija mund të ketë fëmijë të tjerë, një proçes origjinal mund të formojë tre fëmijë të tjerë, si dhe më pas pasardhës të tjerë.

Proçeset në UNIX mund të komunikojnë me njëri-tjetrin duke përdorur kalimin e mesazheve. Eshtë e mundur të krijohet një kanal midis dy proçeseve, ku njëri proçes mund të shkruajë një stream byte-sh për ti lexuar kanali tjetër. Këto kanale quhen **pipes**. Sinkronizimi éshtë i mundur sepse kur një proçes përpinqet të lexojë nga një pipe bosh ai bllokohet derisa të ketë të dhëna të disponueshme për të lexuar.

Shell pipelines janë implementuar me pipe. Kur shelli shikon një rresht si

```
sort <f | head
```

ai krijon dy proçese, *sort* dhe *head*, dhe vendos një pipe mes tyre, në një mënyrë të tillë sikur standart outputi i *sort* éshtë i lidhur me standart inputin e *head*. Në këtë mënyrë, të gjitha të dhënat që shkruhen nga *sort* shkojnë direkt tek *head*, në vend që të shkonin në një file. Në se pipe mbushet plot, sistemi ndalon ekzekutimin e *sort* derisa *head* të zhvendosë disa të dhëna nga pipe.

Proçeset mund të komunikojnë edhe në një rrugë të tjetër: interruptet software. Një proçes mund ti dërgojë atë që quhet **sinjal** proçesit tjetër. Proçeset mund ti tregojnë sistemit se çfarë ata duan të ndodhë kur vjen një sinjal. Zgjedhjet e sistemit janë: ti injorojë, ti kapë, ose ta lerë sinjalin ta vrasë proçesin (zgjedhja default për shumicën e sinjaleve). Në se një proçes preferon ta kapë sinjalin e dërguar, ai duhet të specifikojë procedurën e kapjes së sinjalit. Kur sinjali vjen, kontrolli i kalon menjeherë mbajtësit. Kur mbajtësi mbaron punë, kontrolli kthehet mbrapa andej nga erdhni, në mënyrë të ngjashme me interruptet hardware të I/O. Një proçes mund ti dërgojë vetëm një sinjal elementëve të tjerë të grupit të proçesit, që konsiston në prindin e tij, motrat dhe vëllezërit, dhe fëmijët (dhe pasardhësit e tjerë). Kjo mund të arrihet edhe me anë të një thirrje të vetme sistem.

Sinjalet përdoren gjithashtu edhe për qëllime të tjera. Për shembull, në se një proçes éshtë duke bërë një veprim aritmetik me numra me presje, dhe pa qëllim pjeston me 0, ai merr një sinjal SIGFPE (përjashtim për numrat me presje). Sinjalet që kërkohen nga POSIX tregohen në figurën 10-5. Shumë sisteme UNIX kanë sinjale të tjera shtesë, por programet që i përdorin ato mund të mos jenë të përshtatshme me versionet e tjera UNIX.

Sinjali	Shkaku
SIGABRT	Dërgohet për të ndërprerë një proçes
SIGALRM	Alarmi i orës pushon
SIGFPE	Një gabim me nr me presje ka ndodhur (për shembull. pjestimi me 0)
SIGHUP	Linja e telefonit që proçesi po përdortë ka qenë duke pritur
SIGILL	Useri ka shtypur tastin DEL për të ndërprerë një proçes
SIGQUIT	Useri ka shtypur butonin për të zbruzur ekranin
SIGKILL	Dërgohet për të vrarë një proçes (nuk mund të injorohet)
SIGPIPE	Proçesi ka shkruar në një pipe që ska lexues
SIGSEGV	Proçesi i referohet një adrese invalide në memorje
SIGTERM	Përdoret për të kërkuar që një proçes të përfundojë mirë
SIGUSR1	I disponueshëm për qëllimet e aplikacioneve të caktuara
SIGUSR2	I disponueshëm për qëllimet e aplikacioneve të caktuara

Figura 10-5. Sinjalet e specifikuara nga POSIX

10.3.2 Menaxhimi i thirrjeve sistem në UNIX

Le të shohim tani thirrjet sistem në UNIX shoqëruar me menaxhimin e proçesit. Më kryesoret prej tyre tregohen në figurën 10-6. Fork është një vend i mirë për të filluar diskutimin. Fork është e vetmja rrugë për të krijuar një proçes të ri në sistemet UNIX. Ajo krijon një kopje ekzakte të proçesit origjinal, duke përfshirë të gjithë file deskriptorët, regjistrat dhe çdo gjë tjetër. Pas fork, proçesi origjinal dhe kopja (prindi dhe fëmija), shkojnë në rrugët e tyre të ndara. Të gjithë variablat kanë vlerë identike në kohën e fork, por meqë prindi pasqyrohet i téri tek fëmija, më pas ndryshimet në njërin prej tyre nuk pasqyrohen tek tjetri. Thirrja fork kthen një vlerë, e cila është 0 për fëmijën dhe e barabartë me PID e fëmijës në rastin e prindit. Duke përdorur PID e kthyer, dy proçeset mund të dallohen se kush është prindi dhe kush fëmija.

Në shumicën e rasteve, pas fork, fëmijës do ti duhet të ekzekutojë një kod të ndryshëm nga prindi. Konsiderojmë rastin e shellit. Ai lexon një komandë nga terminali, ndalon proçesin e lindjes, pret fëmijën për të ekzekutuar komandën, dhe më pas lexon komandën e ardhshme kur fëmija përfundon. Për të pritur që fëmija të përfundojë, prindi ekzekuton një thirrje sistem waitpid, e cila pret derisa fëmija (fëmijët) të përfundojnë. Waitpid ka tre parametra. I pari lejon thirrësin të presë për një fëmijë specifik. Në se është -1, cilido fëmijë i vjetër do ta bëjë. Parametri i dytë është adresa e një variabli që do vendoset në gjendjen e daljes së fëmijës (përfundimi normal ose jo normal dhe dalja). Parametri i tretë përcakton në se thirrësi bllokohet ose kthehet në se asnjë fëmijë nuk ka mbaruar akoma.

Në rastin e shellit, proçesi fëmijë duhet të ekzekutojë komandën e shtypur nga përdoruesi. Ai e bën këtë duke përdorur thirrjen sistem exec, e cila bën që kopja e plotë e tij të zëvendësohet nga file i emërtuar në parametrin e tij të parë. Një shell shumë i thjeshtuar i cili ilustron përdorimin e fork, waitpid dhe exec tregohet në figurën 10-7.

Thirrja sistem	Përshkrimi
pid = fork()	Krijon një proces fëmijë identik me prindin
pid = waitpid(pid, &statloc, opts)	Pret që një proces fëmijë të përfundojë
s = execve(name, argv, envp)	Zëvendëson pasqyrimin e një procesi
exit(status)	Mbaron ekzekutimin e procesit dhe kthen gjendjen
s = sigaction(sig, &act, &oldact)	Përcakton veprimin që ndërmerret mbi sinjalat
s = sigreturn(&context)	Kthim nga një sinjal
s = sigprocmask(how, &set, &old)	Kontrollon ose ndryshon maskën e sinjalit
s = sigpending(set)	Merr bashkësinë e sinjaleve të bllokuara
s = sigsuspend(sigmask)	Rivendos maskën e sinjalit dhe pezullon procesin
s = kill(pid, sig)	Dërgon një sinjal një procesi
residual = alarm(seconds)	Vendos alarmin e orës
s = pause()	Pezullon thirrësin deri në sinjalin pasardhës

Figura 10-6. Disa thirrje sistem në lidhje me proceset. Kodi i kthyer *s* është -1 në se ndodh një gabim, *pid* është ID e procesit, dhe *residual* është koha e mbetur për alarmin e mëparshëm. Parametrat janë sipas propozimit të emrave.

```

while(TRUE) {                                /* përsëritë përgjithmonë */
    type_prompt();                            /* shfaq prompt në ekran */
    read_command(command, params);          /* lexon rreshtin e futur nga tastiera */
    pid = fork();                            /* I lirë për të krijuar një proces fëmijë */
    if(pid < 0) {
        printf("Unable to fork");           /* kushti i gabimit */
        continue;                           /* përsërit ciklin */
    }

    if(pid != 0) {
        waitpid (-1, &status, 0);          /* prindi pret për fëmijën */
    } else {
        execve(command, params, 0);         /* fëmija bën punën */
    }
}

```

Figura 10-7. Një shell shumë i thjeshtuar

Në shumicën e rasteve, exec ka tre parametra: emrin e file-s për tu ekzekutuar, një pointer në rreshtin e argumenteve, dhe një pointer në rreshtin e ambjentit. Këto do të përshkruhen shkurt. Librari procedurash të ndryshme, duke përfshirë *execl*, *execv*, *execle*, dhe *execve*, kujdesen për të lënë parametrat jashtë ose të specifikuar në rrugë të ndryshme. Secila prej këtyre procedurave kërkon të njëjtën thirrje sistem. Meqë thirrja sistem është exec, nuk ka asnjë procedurë librarie me këtë emër; një nga procedurat tjera duhet përdorur.

Le të konsiderojmë rastin e shtypes së një komande në shell si për shembull:

```
cp file1 file2
```

e përdorur për të kopjuar *file1* tek *file2*. Pasi shelli bën fork, fëmija vendoset dhe ekzekuton file-n *cp* dhe i kalon informacionin atij (shellit) rrëth file-s për tu kopjuar.

Programi main i *cp* (dhe shumë programe të tjera) përbajnjë deklarimin e funksionit

```
main (argc, argv, envp)
```

ku *argc* është një numërues i gjërave në command line, përfshirë edhe emrin e programit. Në shembullin më lart, *argc* është 3.

Parametri i dytë, *argv*, është një pointer rreshti. Elementi *i* i një rreshti është një pointer i stringës së *i*-të në command line. Në shembullin tonë, *argv[0]* mund të pointojë tek stringa "cp". Njëloj, *argv[1]* mund të pointojë tek karakteri i 5-të i stringës "file1" dhe *argv[2]* mund të pointojë tek karakteri i 5-të i stringës "file2".

Parametri i tretë i *main*, *envp*, është një pointer ambienti, një rresht stringash që përbajnjë përcaktimë të trajtës *emr = vlerë*, të përdorura për të kaluar informacionin si për shembull. nga tipi i terminalit dhe emri i direktorisë home në një program. Në figurën 10-7, asnjë ambient nuk i kalohet fëmijës, kështu që parametri i tretë i *execve* është zero në këtë rast.

Në se exec ju duket e komplikuar, mos u dëshpëroni; ajo është thirrja sistem më komplekse. Të gjitha të tjerat janë shumë më të thjeshta. Për shembull, një e thjeshtë konsiderohet exit, të cilën proceset duhet ta përdorin kur përfundojnë ekzekutimin. Ajo ka një parametër, statusin exit (0-255), siç specifikohet në thirrjen exit të fëmijës. Për shembull, në se një proces prind ekzekuton këtë rresht:

```
n = waitpid (-1, &status, 0);
```

ai do të pezullohet derisa disa procese fëmijë të përfundojnë. Në se fëmija del, le të themi me 4, siç është parametri i *exit*, prindi do të "zgjohet" me vlerën n të vendosur tek PID e fëmijës, dhe *status* të vendosur tek 0x0400 (0x tregon hexadecimal në C). Byte i fundit i *status* ka të bëjë me sinjalët; pas tij është vlera që kthehet ndaj thirrjes *exit* të fëmijës. Në se një proces del dhe prindi i tij nuk ka pritur akoma për të, procesi hyn në një lloj pezullimi të quajtur **zombie state**. Kur prindi në fund pret për të, procesi përfundon.

Disa thirrje sistem lidhen me sinjalët, të cilët përdoren në rrugë të ndryshme. Për shembull, në se një user aksidentalisht i thotë një editori teksti të shfaqë të tërë përbajtjen e një file shumë të madhe, dhe më pas realizon një gabim, duhet që ta ndërpresim editorin. Zgjidhja më e përdorshme për përdoruesin është shtypja e disa butonave (për shembull, DEL ose CTRL-C), të cilat i dërgojnë një sinjal editorit. Editori kap sinjalin dhe ndalon shfaqjen në ekran të përbajtjes së file-t.

Për të treguar këtë gatishmëri për të kapur këtë (apo ndonjë tjetër) sinjal, procesi mund të përdorë thirrjen sistem sigaction. Parametri i parë është për të zënë sinjalin (shih fig. 10-5). I dyti është pointeri në një strukturë, duke i dhënë një pointer procedurës së mbajtjes së sinjalit, si për shembull, disa bite dhe flage të tjerë. I treti pointon në një strukturë ku

sistemi kthen informacion rrreth mbajtësit aktual të sinjalit, në rast se ai duhet të ruhet më vonë.

Mbajtësi i sinjalit mund të ekzekutohet për sa kohë të dojë. Në praktikë, sidoqoftë, mbajtësit e sinjalit janë mjaft të shkurtër. Kur procedura e mbajtjes së sinjalit përfundon, ajo kthehet tek pika nga e cila ishte ndërprerë. Thirrja sistem sigaction mund të përdoret gjithashtu për të refuzuar sinjalin, ose për të ruajtur veprimin default, që është vrasja e procesit.

Shtypja e butonit DEL nuk është e vetmja rrugë për të dërguar një sinjal. Thirrja sistem kill i lejon procesit të sinjalizojë një proces tjetër që ka të bëjë me të. Zgjedhja e fjalës "kill" për thirrjen sistem nuk është më e mira, meqenë se shumica e proceseve iu dërgojnë sinjale proceseve të tjera duke menduar se ato janë kapur.

Për shumë aplikacione në kohë reale, një proces ka nevojë të ndërpritet për një interval specifik kohe për të bërë diçka, si për shembull, ritransmetimi i një pakete të humbur përmes një linje komunikimi jo të sigurtë. Për të mbajtur këtë situatë, jepet thirrja sistem alarm. Parametri specifikon një interval, në sekonda, pas të cilit, një sinjal SIGALRM i është dërguar procesit. Një proces mund të ketë vetëm një alarm në një moment të caktuar. Në se një thirrje alarm është bërë me një parametër prej 10 sekondash, dhe më pas 3 sekonda më vonë një tjetër thirrje alarm është bërë me një parametër prej 20 sekondash, vetëm një sinjal do të gjenerohet, 20 sekonda pas thirrjes së dytë. Sinjali i parë eleminohet nga thirrja e dytë alarm. Në se parametri tek alarm është 0, ndonjë sinjal alarmi i mbetur pezull eleminohet. Në se një sinjal alarm nuk është kapur, ndërmerrët veprimi default dhe procesi i sinjalizuar është vrarë. Teknikisht, sinjalet alarm mund të injorohen, por është e kotë ta bëjmë këtë gjë.

Ndonjehherë ndodh që një proces nuk ka asgjë për të bërë derisa të vijë një sinjal. Për shembull, konsideroni një program me instruksion computer-aided që po teston shpejtësinë e leximit dhe të kuptuarit. Ai shfaq disa fjalë tekst në ekran dhe më pas thërrret alarm për ta sinjalizuar atë pas 30 sekondash. Ndërsa studenti po lexon tekstin, programi s'ka asgjë për të bërë. Ai mund të vendoset në një cikël duke mos bërë asgjë, por që mund të shfrytëzojë kohën e CPU që një procesi background ose një useri mund ti nevojitet. Një zgjidhje më e mirë është përdorimi i thirrjes sistem pause, e cila i thotë UNIX që të pezullojë procesin derisa një sinjal tjetër të vijë.

Thirrjet sistem për menaxhimin e thread-eve

Versionet e para UNIX nuk kishin threade. Kjo veti u shtua disa vjet më vonë. Fillimisht ishin shumë paketa threads në përdorim, por shtimi i paketave të threadeve e bëri të vështirë shkrimin e kodeve të lëvizshme. Përfundimisht, thirrjet sistem të përdorura për menaxhimin e threadeve ishin standartizuar si pjesë e POSIX (P1003.1c).

Specifikimet e POSIX nuk ishin në gjendje të tregonin në se threadi duhet të implementohej në hapësirën user apo kernel. Avantazhi që të kesh threade në hapësirën user është që ato mund të implementohen pa qenë nevoja që të kalojmë në kernel. E meta e threadeve në hapësirën user është që në se një thread bllokohet (për shembull. një I/O, një semafor, apo një page fault), të gjithë threadet e procesit bllokohen sepse kerneli

mendon se është vetëm një thread dhe nuk e skedulon proçesin derisa threadi i bllokuar të lirohet. Kështu që thirrjet e përcaktuara në P1003.1c ishin të zgjedhura me kujdes për të qenë të implementueshme në të gjitha mënyrat. Ashtu siç programet user mbështeten me kujdes tek semantika e P1003.1c, të dyja implementimet duhet të punojnë në mënyrë korrekte. Threadet më të përdorshme tregohen në figurën 10-8. Kur përdoren threadet e kernelit, këto thirrje janë thirrje të vërteta sistem, kur përdoren threadet user, këto thirrje implementohen tërësisht në librarinë runtime të hapësirës user.

Thirrjet thread	Përshkrimi
pthread_create	Krijon një thread të ri në hapësirën e adresës së thirrësit
pthread_exit	Përfundon thirrjen e threadit
pthread_join	Pret që një thread të përfundojë
pthread_mutex_init	Krijon një mutex të ri
pthread_mutex_destroy	Shkatërron një mutex
pthread_mutex_lock	Blokon një mutex
pthread_mutex_unlock	Zhbllokon një mutex
pthread_cond_init	Krijon një variabël kushtezimi
pthread_cond_destroy	Shkatërron një variabël kushtezimi
pthread_cond_wait	Pret një variabël kushtezimi
pthread_cond_signal	Lë një thread duke pritur një variabël kushtezimi

Figura 10-8. Thirrjet thread kryesore në POSIX

Le të shikojmë shkurt thirrjet thread të treguara në figurën 10-8. Thirrja e parë, pthread_create, krijon një thread të ri. Ajo thirret nga:

```
err = pthread_create (&tid, attr, function, arg);
```

Kjo thirrje krijon një thread të ri në proçesin aktual që po ekzekuton kodi *function* me *arg* të kaluar si parametër. ID e threadit të ri vendoset në memorje në vendodhjen e pointuar nga parametrat e parë. Parametri *attr* mund të përdoret për të specifikuar disa veti për threadin e ri, si për shembull. prioriteti i skedulimit të tij.

Një thread që ka bërë punën e tij dhe do të përfundojë ekzekutimin, thërret pthread_exit. Një thread mund të presë për një thread tjeter për të përfunduar duke thirrur pthread_join. Në se threadi i pritur ka përfunduar tashmë, pthread_join përfundon menjeherë. Përndryshe ajo bllokohet.

Threadet mund të sinkronizohen duke përdorur thirrjet bllokuese **mutex**. Zakonisht një mutex ruan disa burime, siç është bufferi i share-uar nga dy threadë. Për të qenë të sigurtë që vetëm një thread në një moment kohe akseson burimin e share-uar, threadet supozohet se e bllokojnë mutex përpara komunikimit me burimin dhe e zhbllokojnë atë kur mbarojnë punë. Ashtu si të gjithë threadet i binden këtij protokolli, kushtet e shpejtësisë mund të mënjanohen. Mutex-et janë si semaforët binarë, që janë semaforë që marrin

vetëm dy vlera, 0 dhe 1. Emri "mutex" vjen nga fakti që mutex-et janë përdorur për të siguruar mutual exclusion mbi disa burime.

Mutex-et mund të krijohen dhe të shkatërrohen respektivisht me anë të thirrjeve `pthread_mutex_init` dhe `pthread_mutex_destroy`. Një mutex mund të jetë në një nga dy gjendjet: bllokuar ose zhbllokuar. Kur një threadi i duhet të bllokojë një thread të zhbllokuar (duke përdorur `pthread_mutex_lock`), bëhet bllokimi dhe threadi vazhdon. Megjithatë, kur një thread përpinqet të bllokojë një mutex të bllokuar tashmë, ai bllokohet. Kur threadi bllokues mbaron punë me burimin e share-uar, pritet zhbllokimi i mutex-it korrespondues duke thirrur `pthread_mutex_unlock`.

Mutex-et nënkuqtojnë një bllokim për një afat të shkurtër, siç është për shembull, mbrojtja e një variable të share-uar. Ato nuk nënkuqtojnë sinkronizim për një kohë të gjatë, si për shembull, pritja e një tape drive që të lirohet. Sinkronizimi për një kohë të gjatë sigurohet nga variablat e kushtezimit. Ato krijohen dhe shkatërrohen me anë të thirrjeve respektive, `pthread_cond_init` dhe `pthread_cond_destroy`.

Një variabël kushtezimi përdoret duke pasur një thread që pret për të dhe një tjetër thread sinjalizues. Për shembull, duke zbuluar që një tape drive që asaj i duhet është e zënë, një thread duhet të bëjë një `pthread_cond_wait` mbi një variabël kushtezimi që të gjithë të përshtaten me kushtin e tape drive. Kur threadi që po përdorte tape drive mbaron punë me të (ndoshta mbas disa orësh), ai përdor `pthread_cond_signal` për të lënë vetëm një thread duke pritur mbi këtë variabël kushtezimi (në se ka threade). Në se nuk ka threade duke pritur, sinjali humbet. Me fjalë të tjera, variablat e kushtezimit nuk numërojnë si semaforët. Ka edhe disa përcaktimesh të tjera mbi veprimet e threadeve, mutex-eve dhe variablave të kushtezimit.

10.3.3 Implementimi i proceseve në UNIX

Një proces në UNIX është si një iceberg; ajo çfarë shihni është pjesa mbi ujë, por është gjithashtu një pjesë e rëndësishme nën ujë. Çdo proces ka një pjesë user që ekzekuton programet user. Megjithatë, kur një nga threadet e tij bën një thirrje sistem, ai kalon në kernel mode, dhe fillon ekzekutimin në kontekstin e kernelit, me një hartë memorje të ndryshme dhe me një akses të plotë mbi të gjitha burimet e makinës. Eshtë akoma i njëjtë thread, por tani me më shumë fuqi dhe me stack-un dhe program counter-in e tij të kernelit. Këto janë të rëndësishme sepse thirrja sistem mund të bllokojë disa rrugë, si për shembull, pritja përfundimin e një veprimi të diskut. Për regjistrat ruhen, kështu që threadi mund të rifillojë më vonë në kernel mode.

Kerneli mirëmban dy struktura kryesore të dhënat lidhura me proceset, tabelën e procesit dhe strukturën e përdoruesit. Tabela e procesit është kudo gjatë gjithë kohës dhe mban informacionin e nevojshëm për të gjitha proceset, edhe për ato që aktualisht nuk janë në memorje. Struktura e përdoruesit zhvendoset (swapped ose paged) jashtë kur i shoqërohen proceset që s'janë në memorje, në mënyrë që të mos shpërdorohet memorja me informacione që nuk nevojiten.

Informacioni i tabelës së proçesit përbëhet nga kategoritë e mëposhtme:

Parametrat e skedulimit. Prioriteti i proçesit, sasia e kohës së CPU të konsumuar së fundemi, sasia e kohës së shpenzuar në gjendje pushimi. Të gjitha këto së bashku, përdoren për të përcaktuar proçesin pasardhës që do ekzekutohet.

Pasqyrimi i memorjes. Pointerat e tekstit, të dhënët, dhe segmentet e stack-ut, ose në se faqosja është përdorur, tek faqet e tabelës së tyre. Në se segmenti i tekstit është i shareuar, pointeri i tekstit pointon tek tabela tekst e share-uar. Kur proçesi nuk është në memorje, informacioni mbi vendodhjen e tij në disk është këtu.

Sinjalet. Maskimet tregojnë se cilët sinjale injorohen, cilët kapen, cilët janë përkohësisht të bllokuar, dhe cilët janë në proçeset e dërguara.

Të tjera. Gjendja aktuale e proçesit, ngjarja për të cilën pritet, në se ka, koha derisa të përfundojë alarmi i orës, PID, PID e proçesit prind dhe useri, dhe elementet identifikues.

Struktura e përdoruesit përmban informacionin që nuk nevojitet kur proçesi nuk është fizikisht në memorje dhe nuk është i ekzekutueshëm. Për shembull, megjithëse është e mundur për një proçes të dërgojë një sinjal ndërsa zhvendoset jashtë, nuk është e mundur për të të lexojë një file. Për këtë arsy, informacioni rreth sinjaleve duhet të jetë në tabelën e proçesit, kështu që ato janë në memorje gjatë gjithë kohës, edhe kur proçesi nuk është në memorje. Nga ana tjetër, informacioni rreth file deskriptorëve mund të mbahet në strukturën e përdoruesit dhe mund të paraqitet vetëm kur proçesi është në memorje dhe është i ekzekutueshëm.

Informacioni që ndodhet në strukturën e përdoruesit përfshin:

Regjistrat makinë. Kur ndodh një instruksion trap në kernel, regjistrat makinë (duke përfshirë edhe ato të nr me presje, në se përdoren) ruhen këtu.

Gjendja e thirrjes sistem. Informacioni rreth thirrjes aktuale sistem, duke përfshirë edhe parametrat dhe rezultatet.

Tabela e file deskriptor. Kur kërkohet një thirrje sistem duke përfshirë një file deskriptor, file deskriptori përdoret si një indekx në këtë tabelë për të përcaktuar vendodhjen e strukturës së të dhënave që i korrespondon kësaj file.

Llogaritësi. Pointer në një tabelë që mban gjurmën e përdoruesit dhe kohën e CPU të përdorur nga proçesi. Disa sisteme mbajnë kufizime këtu për sa i takon kohës së CPU që mund të përdorë proçesi, madhësinë e stack-ut, numrin e page frame-ve që mund të konsumojë, etj.

Stack-u i kernelit. Një stack i fiksuar për tu përdorur nga proçeset e kernelit.

Duke mbajtur në mendje përdorimin e këtyre tabelave, tani është më e lehtë të shpjegohet se si krijohen proceset në UNIX. Kur ekzekutohet një thirrje sistem fork, procesi thirrës kalon në kernel dhe shikon për një slot të lirë për tu përdorur nga fëmija, në tabelën e proceseve. Në se e gjen këtë slot, ai kopjon të gjithë informacionin nga tabela e procesit prind tek fëmija. Më pas përcakton memorjen për të dhënat e fëmijës dhe segmentin e stack-ut, dhe e dërgon kopjen e të dhënavës dhe stack-ut të prindit pikërisht këtu. Struktura e përdoruesit (që shpesh rri pranë segmentit të stack-ut), kopjohet përpëra me stack-un. Segmenti i tekstit mund të kopjohet ose share-ohet i téri meqë është vetëm i lexueshëm. Në këtë moment, fëmija është gati për tu ekzekutuar.

Kur një user shtyp një komandë, për shembull, ls, shelli krijon një proces të ri duke bërë një klon të vetes. Shelli i ri më pas therret exec për të mbuluar memorjen e tij me përbajtjen e file-s së ekzekutueshme ls. Hapat e përfshira tregohen në figurën 10-9.

Mekanizmi për krijimin e proceseve të reja aktualisht tepër i "ndershëm". Një slot i ri i tabelës së procesit dhe hapësira user krijohen për procesin fëmijë dhe mbushen larg prindit. Fëmijës i jepet një PID, strukturohet harta e memorjes së tij, dhe i jepet një akses i share-uar tek file-t e prindit. Më pas registrat e tij strukturohen dhe bëhen gati për tu ekzekutuar.

Në parim një kopje e hapësirës së adresës duhet bërë, meqënë se semantikat e fork tregojnë që s'ka memorje të share-uar mes prindit dhe fëmijës. Megjithatë, kopjimi i memorjes është i kushtueshëm, kështu që të gjitha sistemet moderne UNIX gënjejnë. Ato i japin fëmijës page table-in e tij, por iu duhet të shënjojnë tek page-t e prindit, duke i etiketuar ato si vetëm të lexueshme. Sa herë që fëmija përpinqet të shkruajë mbi një page, ai merr një dështim kalimi. Kerneli e shikon këtë dhe më pas i alokon një kopje të re të page-it tek fëmija dhe e etiketon atë si të lexueshme dhe të shkrueshme. Në këtë mënyrë, vetëm page-t që aktualisht janë shkruar duhet të kopjohen. Ky mekanizëm quhet **copy-on-write**. Ai ka të mirën që nuk kërkon dy kopje të programit në memorje, kështu që kurson memorjen RAM.

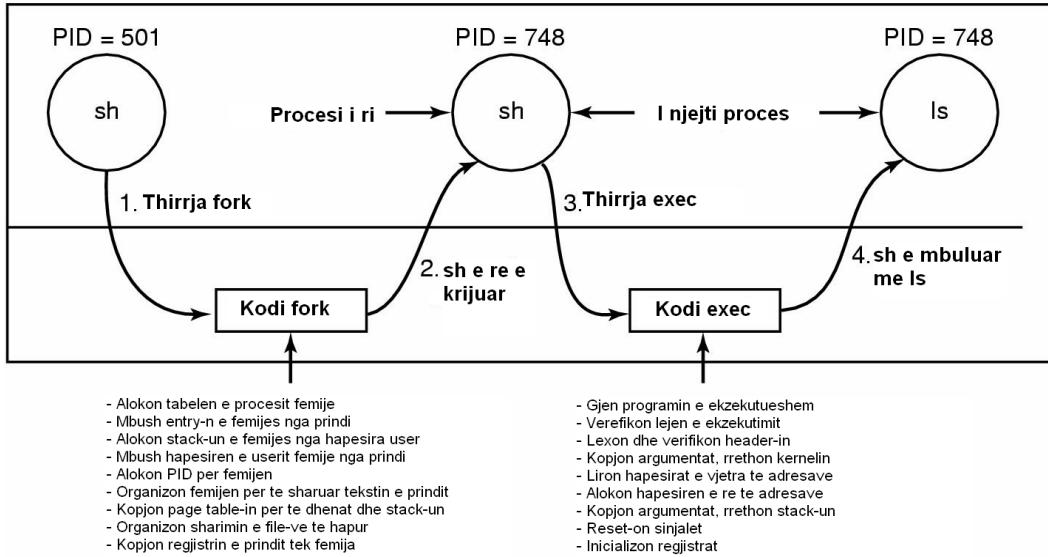


Figura 10-9. Hapat e ekzekutimit të komandës ls të shtypur në shell

Pasi procesi fëmijë fillon ekzekutimin, kodi që ekzekutohet atje (një kopje e shellit) bën një thirrje sistem exec, duke i dhënë si parametër emrin e komandës. Kerneli tani gjen dhe verifikon file-n e ekzekutueshëm, kopjon argumentat dhe rrethon stringat tek kerneli, dhe lë hapësirën e vjetër të adresave dhe page table-in.

Tani hapësira e re e adresave duhet të krijohet dhe të mbushet. Në se sistemi suporton mapped files, siç bëjnë Sistemi V, BSD, dhe shumë sisteme të tjera UNIX, page table e reja organizohen për të treguar që s'ka faqe në memorje, përvèç ndoshta të një stack page, por që hapësira e adresave kthehet mbrapa nga file-a e ekzekutueshme në disk. Kur procesi i ri fillon ekzekutimin, ai menjeherë do të marrë një page fault, e cila do ti shkaktojë faqes së parë të kodit të thirret nga file e ekzekutueshme. Në këtë mënyrë, asgjë nuk do jetë ngarkuar përpara, kështu që programet mund të startojnë shpejt dhe të dështojnë vetëm ato faqe, që duhen dhe jo më shumë. Përfundimisht, argumentat dhe stringat janë kopjuar tek stack-u i ri, sinjalat janë reset-uar dhe regjistrat janë inicializuar të gjithë me zero. Në këtë moment komanda e re mund të fillojë të ekzekutohet.

Threadet në UNIX

Implementimi i threadeve varet në se ato suportohen në kernel apo jo. Në se jo, siç është rasti i 4BSD, implementimi është i téri në hapësirën e përdoruesit. Në se suportohen, si në rastin e Sistem V dhe Solaris, kerneli ka disa punë për të bërë. Ne i diskutuam threadet në mënyrë të përgjithshme në Kapitullin 2. Këtu do të hedhim një vështrim mbi threadet e kernelit në UNIX.

Problemi kryesor tek threadet është mirmbajtja e semantikës korrekte tradicionale të UNIX. Së pari konsideroni fork. Supozoni që një proces me shumë threade (në kernel) bën një thirrje sistem fork. A duhet që të gjitha threadet e tjera të krijohen në procesin e ri? Për momentin le të përgjigjemi me po. Supozojmë që një nga threadet e tjerë është

bllokuar për të lexuar nga tastjera. A duhet që edhe threadi korespondues në procesin e ri gjithashtu të jetë i bllokuar për të lexuar nga tastjera? Në se po, cili prej tyre e merr rreshtin pasardhës? Në se jo, çfarë duhet të bëjë threadi që të jetë në procesin e ri? I njëjtë problem mbetet edhe për shumë gjëra të tjera që threadet mund të bëjnë. Në një proces me një thread të vetëm, problemi nuk duket, sepse ky thread i vetëm nuk mund të bllokohet kur thirret fork. Tani konsiderojmë rastin që threadet e tjera nuk janë krijuar tek procesi fëmijë. Supozojmë që një nga këto threade mban një mutex që threadi i vetëm në procesin e ri përpinqet ta marrë pasi të bëjë fork. Mutex nuk do të lirohet asnjeherë dhe threadi i vetëm do të ekzekutohet përgjithmonë. Gjithashtu ekzistojnë edhe disa probleme të tjera. Nuk ka zgjidhje të theshta të këtyre problemeve.

Hapësira e file-ve I/O është një tjetër problem. Supozojmë që një thread është i bllokuar për të lexuar nga një file dhe threadi tjetër mbyll file-n ose bën një Iseek për të ndryshuar pointerin aktual të file-s. Çfarë ndodh pastaj? Kush e di?

Trajtimi i sinjalit është një tjetër problem. A duhet sinjalet të drejtohen tek një thread specifik apo tek një proces në përgjithësi? Një SIGFPE (përjashto floating-point) duhet mundësishët të kapet nga threadi që e shkaktoi atë. Çfarë ndodh në se nuk e kap ai? A duhet të vritet ky thread apo të gjithë threadet? Tani supozojmë një sinjal SIGNIT të gjeneruar nga përdoruesi në tastierë. Cili thread duhet ta kapë atë? A duhet që të gjithë threadet të share-ojnë një bashkësi të përbashkët sinjalesh maskimi? Të gjitha zgjidhjet për këtë dhe problemet e tjera zakonisht bëjnë që diçka të prishet diku. Marja e drejtë e semantikave të threadeve është një punë e parëndësishme.

Threadet në Linux

Linuxi suporton threadet e kernelit në një mënyrë interesante që ia vlen ta shikojmë. Implementimi është bazuar në idetë e 4.4BSD, por threadet e kernelit nuk mundësohen në këtë sistem sepse Berkley i derdhi parate përpara se libraria e C mund të rishkruhej për të zgjidhur problemet e diskutuara më lart.

Implementimi i threadeve në Linux bëhet me anë të një thirrje sistem të veçantë, clone, e cila nuk është në asnjë version tjetër të UNIX. Ajo thirret si më poshtë:

```
pid = clone (function, stack_ptr, sharing_flags, arg);
```

Thirrja krijon një thread të ri, në procesin aktual ose në një proces të ri, në varësi të *sharing_flags*. Në se threadi i ri është në procesin aktual, ai share-on hapësirën e adresave me threadet ekzistues dhe çdo shkrim më pas i ndonjë byte në hapësirën e adresave nga ndonjë thread shfaqet menjeherë tek të gjithë threadet e procesit. Nga ana tjetër, në se hapësira e adresave nuk është share-uar, threadi i ri merr një kopje ekzakte të hapësirës së adresave, por më pas shkrimi nga threadi i ri nuk iu shfaqet të tjerëve. Këto semantika janë të njëjta si fork.

Në të dyja rastet, threadi i ri fillon ekzekutimin tek *function*, i cili është thirrur me *arg* si të vetmin parametër. Gjithashtu në të dyja rastet, threadi i ri merr stack-un e tij privat, me stack pointerin të inicializuar tek *stack_ptr*.

Parametri *sharing_flags* është një bitmap që lejon një strukturë me fine të share-imit sesa në sistemet tradicionale UNIX. Përcaktohen pesë bite, si në figurën 10-10. Çdo bit kontrollon disa aspekte të share-imit, dhe secili prej biteve mund të konsiderohet si i pavarur nga njëri-tjetri. Biti *CLONE_VM* përcakton në se memorja virtuale (për shembull, hapësira e adresave) është share-uar me threadet e vjetra apo kopjet. Në se ky bit vendoset, threadi i ri futet menjeherë tek ato ekzistues, kështu që thirrja clone efektivisht krijon një thread të ri në procesin ekzistues, në se biti fshihet, threadi i ri merr hapësirën e tij të adresave. Duke pasur hapësirën e tij të adresave do të thotë që ndikimi i instruksionit të tij STORE nuk shfaqet tek threadet ekzistues. Kjo sjellje është e ngashme me fork, me përjashtimin e mëposhtëm. Krijimi i një hapësire të re adresash efektivisht do të thotë përcaktimi i një procesi të ri.

Flagu	Domethënia kur vendoset	Dmth. kur fshihet
<i>CLONE_VM</i>	Krijon një thread të ri	Krijon një proces të ri
<i>CLONE_FS</i>	Share-on umask, root, dhe dir e punës	Nuk i share-on ato
<i>CLONE_FILES</i>	Share-on file deskriptorët	Kopjon file deskriptorët
<i>CLONE_SIGHAND</i>	Share-on tabelëne mbajtjes së sinjalit	Kopjon tabelën
<i>CLONE_PID</i>	Threadi i ri merr PID e vjetër	Threadi i ri merr PID e tij

Figura 10-10. Bitet në *sharing_flags* bitmap.

Biti *CLONE_FS* kontrollon share-imin e root, direktorive të punës dhe të umask flag. Edhe në se threadi i ri ka hapësirën e tij të adresave, në se ky bit vendoset, threadet e rinj dhe të vjetër share-ojnë direktortë e punës. Kjo do të thotë që thirrja chdir nga një thread ndryshon direktorinë e punës së një threadi tjetër, megjithëse threadi tjetër mund të ketë hapësirën e tij të adresave. Në UNIX, një thirrje chdir nga një thread gjithmonë ndryshon direktorinë e punës për threadet e tjerë në të njëjtin proces, por asnjeherë në proceset tjera. Kështu që ky bit mundëson një lloj share-imi të pamundur në UNIX.

Biti *CLONE_FILES* është i ngjashëm me bitin *CLONE_FS*. Në se ky bit vendoset, threadi i ri share-on file deskriptorët e tij me threadet e vjetër, kështu që thirrja e Iseek nga një thread shfaqet tek të tjerët, gjithnjë për threadet në një proces dhe jo në proceset të tjerë. Në mënyrë të ngjashme, *CLONE_SIGHAND* mundëson ose jo shae-imin e tabelës së mbajtjes së sinjalit midis threadeve të rinj dhe të vjetër. Në se tabela është share-uar, edhe midis threadeve në hapësira të ndryshme adresash, ndryshimi i mbajtësit të një sinjali ndikon tek mbajtësit e sinjaleve të tjerë. Së fundi, *CLONE_PID* kontrollon në se threadi i ri merr PID e tij apo share-on PID e prindit. Kjo veti nevojitet gjatë butimit të sistemit. Proseset user nuk lejohen për ta mundësuar këtë gjë.

Kjo strukturë fine e share-imit është e mundur sepse Linuxi mirëmban struktura të ndara të dhënash për gjëra të ndryshme të treguara në fillim të seksionit 10.3.3 (parametrat e skedulimit, pasqyrimi i memorjes, etj.). Tabela e procesit dhe struktura e përdoruesit vetëm pointojnë tek këto struktura të dhënash, kështu që është e lehtë për të bërë një entry

të re të tabelës së proçesit për çdo thread të klonuar dhe për më tepër duhet të pointojnë tek skedulimet e threadeve të vjetra, memorja dhe struktura të tjera të dhënash, apo kopje të tyre. Fakti që struktura fine e share-imit është e mundur nuk do të thotë që ajo është e përdorshme gjithnjë, veçanërisht në se UNIX nuk e ofron këtë funksion. Një program në Linux që i ka këto avantazhe nuk mund të transferohet në UNIX.

Skedulimi në UNIX

Tani le të shohim algoritmin e skedulimit në UNIX. Për shkak se UNIX gjithmonë ka qenë një sistem me multiprogramim, algoritmi i tij i skedulimit ishte projektuar nga fillimi për tu dhënë një përgjigje pozitive proçeseve interaktive. Eshtë një algoritem me dy nivele. Algoritmi i nivelit të ulët zgjedh proçeset pasardhëse që do ekzekutohen nga bashkësia e proçeseve në memorje dhe që janë gati për ekzekutim. Algoritmi i nivelit të lartë zhvendos proçeset ndërmjet memorjes dhe diskut, kështu që të gjithë proçeset kanë mundësinë për të qenë në memorje dhe për tu ekzekutuar.

Çdo version i UNIX ka një version pak të ndryshueshëm të algoritmit të skedulimit të nivelit të ulët, por shumica e tyre janë të mbyllur ndaj këtij që do të përshkruajmë këtu. Algoritmi i nivelit të ulët përdor shumë queue (rradhë). Çdo queue shoqërohet me një hapësirë vlerash prioriteti. Proçeset që ekzekutohen në user mode (maja e iceberg-ut) kanë vlera pozitive. Vlerat negative kanë prioritet maksimal ndërsa vlerat pozitive të mëdha kanë prioritetin më të ulët, siç tregohet në figurën 10-11. Vetëm proçeset që janë në memorje dhe që janë gati për tu ekzekutuar vendosen në queue, meqënë se zgjedhja duhet bërë nga kjo bashkësi proçesesh.

Kur skeduleri (i nivelit të ulët) ekzekutohet, ai kërkon queue duke filluar nga prioriteti i lartë (vlerat më negative) derisa të gjejë një queue që është zënë. Proçesi i parë në këtë queue zgjedhja dhe fillimi. Ai lejohet të ekzekutohet për për një kuant maksimal, zakonisht 100 msec, ose derisa të bllokohet. Në se një proçes e shfrytëzon kuantin e tij, ai vendoset në fund të queue dhe algoritmi i skedulimit ekzekutohet përsëri. Kështu që proçeset me të njëjtin prioritet share-ojnë CPU duke përdorur algoritmin round-robin.

Ndonjeherë, prioriteti i çdo proçesi rillogaritet me anë të formulës që përfshin tre komponentë:

$$priority = CPU_usage + nice + base$$

Bazuar në prioritetin e tij të ri, çdo proçes vendoset në queue e treguar në figurën 10-11, zakonisht duke e pjestuar prioritetin me një konstante për të nxjerrë numrin e queue. Le të shikojmë shkurt tre komponentët e formulës së prioritetit.

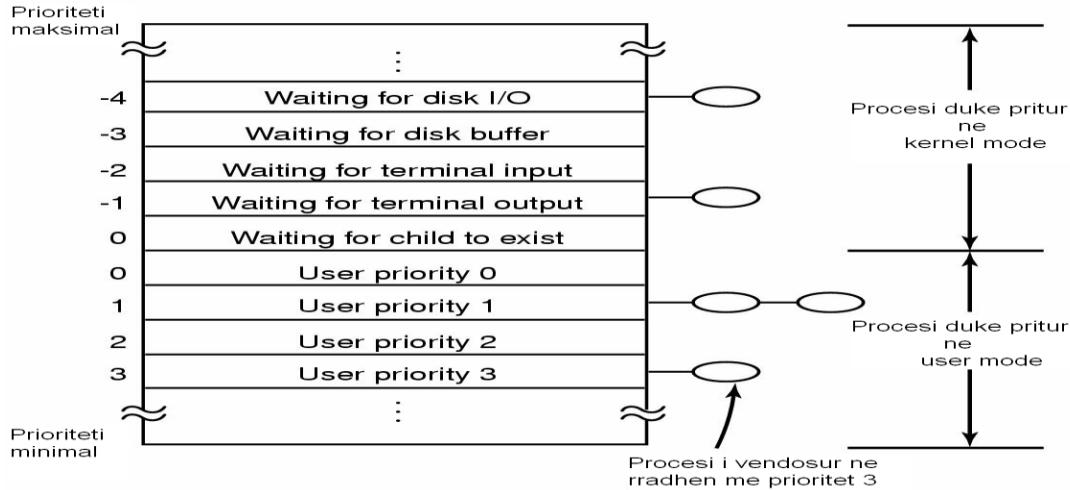


Figura 10-11. Skeduleri i UNIX bazuar në një strukturë me multiqueue

CPU_usage, paraqet numrin mesatar të tikeve të orës për sekondë që procesi ka pasur gjatë sekondave të kaluara. Në çdo tik ore, numeratori i *CPU_usage* në tabelën e procesit në ekzekutim inkrementohet me 1. Ky numeratori së fundi do ti shtohet prioritetit të procesit duke i dhënë atij një vlerë numerike të madhe, duke e vendosur kështu në një queue me prioritet të ulët.

Megjithatë, UNIX nuk e ndëshkon përgjithnjë një proces për të përdorur CPU, kështu që *CPU_usage* prishet me kalimin e kohës. Versione të ndryshme të UNIX e bëjnë këtë paksa ndryshe. Një mënyrë që ka qenë përdorur është shtimi i vlerës aktuale të *CPU_usage* numrit të tikeve të marra në të kaluarën ΔT , dhe rezultati të pjestohet me dy. Ky algoritmom vlerëson më shumë ΔT e fundit me $\frac{1}{2}$ sesa atë me $\frac{1}{4}$, e kështu me rradhë. Ky algoritmom vlerësimi është shumë i shpejtë sepse ai ka për të bërë vetëm një mbledhje dhe një zhvendosje, por dhe skema të tjera vlerësimi kanë qenë përdorur gjithashtu.

Çdo proces ka një vlerë *nice* të shoqëruar me të. Vlera default është 0, por kufiri i lejuar në përgjithësi është nga -20 në +20. Një proces mund ta vendosë *nice* në një vlerë në hapësirën 0 deri 20 me anë të thirrjes sistem nice. Vetëm administratori i sistemit mund të kërkojë një shërbim *më të mirë* se normali (domethene vlerat nga -20 në -1).

Kur një proces kalon nga user në kernel për të bërë një thirrje sistem, është plotësisht e mundur që procesi duhet të bllokohet para se të përfundojë komplet thirrja sistem dhe të kthehet në user mode. Për shembull, ai mund të ketë bërë një thirrje sistem *waitpid* dhe i duhet të presë që një nga fëmijët e tij të përfundojë (të dalë). Atij gjithashtu mund ti duhet për të pritur për inputin e terminalit ose që një disk I/O të kompletohet, duke thënë kështu vetëm pak nga mundësita e shumta. Kur bllokohet, ai zhvendoset nga struktura e queue, meqë nuk mund të ekzekutohet.

Megjithatë, kur ngjarja për të cilën ai po pret ndodh, ai vendoset në queue me një vlerë negative. Zgjedhja e queue përcaktohet nga ngjarja për të cilën ai po priste. Në figurën 10-11, disk I/O është treguar sikur ka prioritetin maksimal, kështu që një proces që sapo

ka lexuar apo shkruar në disk do ta marrë CPU për ndoshta 100 msec. Prioriteti përkatës i disk I/O, terminal I/O, etj. lidhen ngushtë me sistemin operativ, dhe mund të modifikohen duke ndryshuar disa konstante në kodin burim dhe duke rikompiluar sistemin. Këto vlera (negative) paraqiten nga *baza* (*base*) e formulës së dhënë më lart dhe janë të ndara veçmas që proceset e ristartuara për arsyet të ndryshme të jenë qartësisht të ndara në queue të ndryshme.

Ideja pas kësaj skeme është që të marrim shpejt proceset jashtë kernelit. Në se një proces po përpinqet të lexojë një file në disk, duke e bërë atë të presë nga një sekondë ndërmjet thirrjeve read do ta ngadalësojë atë jashtëzakonisht. Eshtë shumë më mirë ta lemë atë të ekzekutohet menjeherë pasi çdo kërkesë përbushet, kështu që ai mund ta bëjë herën tjetër këtë më shpejt. Në mënyrë të ngjashme, në se një proces është bllokuar duke pritur për një terminal input, ai është qartë një proces interaktiv, dhe si i tillë duhet ti jepet një prioritet i lartë apo është gati të sigurojë që procesi interaktiv mori një shërbim të mirë. Në këtë aspekt, proceset CPU bound (ato në queue pozitive) marrin një shërbim të mbetur kur të gjithë I/O bound dhe proceset interaktive janë të bllokua.

Skedulimi në Linux

Skedulimi është një nga të paktat aspekte në të cilat Linuxi përdor një algoritem të ndryshëm nga UNIX. Sapo kemi parë algoritmin e skedulimit në UNIX, kështu që tanë do të shohim algoritmin në Linux. Për të filluar, threadet e Linux-it janë threadë kerneli kështu që skedulimi bazohet tek threadet, dhe jo proceset. Linuxi dallon tre klasa threadesh për qëllime skedulimi:

Real-time FIFO.

Real-time round-robin.

Timesharing

Threadet Real-time FIFO janë me prioritet maksimal dhe nuk janë preemptible me përjashtim të threadeve Real-time FIFO të lexuar së fundi. Threadet Real-time round-robin janë të njëjta me threadet Real-time FIFO me përjashtim që ato janë preemptible nga ora. Në se shumë threadë Real-time round-robin janë gati, secili ekzekutohet për kuantin e tij, pas të cilit ato shkojnë në fund të listës së threadeve real-time round-robin. Asnjëra nga këto klasa aktualisht nuk është real-time në çdo kuptim. Këto klasa janë thjesht me prioritet më të lartë sesa threadet në klasat standarde në timesharing. Arsyet pse Linuxi i thërrret ato në kohë reale i pershtatet standartit P1003.4 (prapashtesat "real-time" në UNIX) i cili i përdor këto emra.

Çdo thread ka një prioritet skedulimi. Vlera default është 20, por kjo mund të ndryshohet duke përdorur thirrjen sistem nice (value) në një vlerë tek 20 - *value*. Meqëndë se value duhet të jetë në kufinjtë nga -20 në +19, prioritetet gjithnjë variojnë: $1 \leq priority \leq 40$. Qëllimi është që cilësia e shërbimit të jetë afërsisht proporcionale me prioriteten, me threadet me prioritet të lartë që marrin një përgjigje të shpejtë në kohë dhe threadet me prioritet të ulët më pas me një fraksion të madh të kohës së CPU.

Përveç prioritetit, çdo thread ka një kuant të shoqëruar me të. Kuanti është numri i tikeve të orës për të cilat threadi mund të vazhdojë ekzekutimin. Supozojmë se ora (clock) ka shpejtësi 100 Hz, kështu që çdo tik është 10 msec, i cili quhet **jiffy**. Skeduleri i përdor prioritetin dhe kuantitet si më poshtë. Ai fillimisht llogarit **mirësinë** e çdo threadi gati duke ndjekur rregullat e mëposhtëm:

```
if (class == real-time) goodness = 1000 + priority ;
if (class == timesharing && quantum > 0) goodness = quantum + priority ;
if (class == timesharing && quantum == 0) goodness = 0 ;
```

Të dyja klasat real-time përfshihen tek rregulla e parë. Të gjitha etiketimet e një threadi si real-time e bëjnë atë të ketë një mirësi më të lartë sesa threadet timesharing. Algoritmi ka një veti të veçantë: në se proçesi që u ekzekutua i fundit ka ende një sasi kuanti, ai merr një pikë bonus, kështu që ai fiton ndonjë detyrë. Ideja këtu është që të gjithë gjërat dojenë të njëvlershme, është më shumë eficiente që të ekzekutojmë proçesin paraardhës, meqë faqet e tij dhe blloqet e cachesë janë më të përshtatshme për tu ngarkuar.

Me dhënien e kësaj pamje, algoritmi i skedulimit është shumë i thjeshtë; kur një vendim skedulimi është bërë, zgjidhet threadi me mirësinë më të lartë. Ndërsa threadi i zgjedhur ekzekutohet, në çdo tik ore, kuanti i tij inkrementohet me 1. CPU largohet nga threadi në se një nga kushtet e mëposhtme ndodh:

Kuanti i tij vendoset në 0.

Threadi bllokoni një I/O, semafor ose diçka tjeter.

Një thread me mirësi të lartë i bllokuar para bëhet gati për ekzekutim.

Meqënë se kuanti numëron për poshtë, shpejt apo vonë çdo thread gati do ta harxhojë kuantin e tij në terren derisa të shkojnë të gjithë në 0. Megjithatë, threadet I/O bound që aktualisht janë të bllokuar mund të kenë disa kuante. Në këtë moment skeduleri reset-on kuantet e *të gjithë* threadeve, gati dhe bllokuar, duke përdorur rregullën:

$$kuanti = (kuant/2) + prioritet$$

ku kuanti i ri është në jiffies. Një thread që është shumë i kufizuar në llogaritje zakonisht do të mbarojë shpejt kuantin e tij dhe duhet të jetë 0 kur kuanti të resetohet, duke i dhënë atij një kuant të njëjtë me prioritetin e tij. Një thread I/O-bound mund të ketë një kuant të konsiderueshmë bosh dhe kështu merr një sasi më të madhe kuanti herën tjeter. Në se nice nuk është përdorur, prioriteti do jetë 20, kështu që kuanti tjeter i ardhshëm do jetë 20 jiffies ose 200 msec. Nga ana tjeter, një thread I/O bound me prioritet të lartë, mund të ketë një kuant bosh prej 20 kur kuanti resetohet, kështu që në se prioriteti i tij është 20, kuanti i tij i ri do jetë $20/2 + 20 = 30$ jiffies. Vlera asymptotike në jiffies është sa dy herë prioriteti. Si pasojë e këtij algoritmi, threadet I/O-bound marrin kuante të mëdha dhe kështu mirësia është më e madhe në threadet compute-bound. Kjo i bën threadet I/O-bound të preferueshme në skedulim.

Karakteristikë tjeter e këtij algoritmi është që kur threadet compute-bound po konkurojnë për CPU, ai që ka prioritet më të lartë merr një sasi më të madhe të saj. Për ta parë këtë

marrim dy threade compute-bound, *A*, me prioritet 20 dhe, *B*, me prioritet 5. *A* shkon e para dhe 20 tike më vonë e shfrytëzon kuantin e tij. Më pas *B* ekzekutohet për 5 kuante. Në këtë pikë kuantet reset-ohen. *A* merr 20 dhe *B* merr 5. Kjo bëhet përgjithmonë, kështu që *A* po merr 80% të CPU dhe *B* po merr 20% të saj.

10.3.4 Butimi në UNIX

Detajet ekzakte se si UNIX butohet varojnë nga një sistem në tjetrin. Më poshtë do të shohim shkurt se si butohet 4.4BSD, por idetë janë pothuajse të ngashme me të gjitha versionet e tjera. Kur kompjuteri ndizet, sektori i parë i butueshëm i diskut (master boot record) lexohet në memorje dhe ekzekutohet. Ky sektor përmban një program të vogël (512-byte) që ngarkon një program standalone të quajtur *boot* nga paisja e butueshme, zakonisht një IDE ose SCSI disk. Programi *boot* fillimisht kopjon vetveten në një adresë të lartë fikse të memorjes për të liruar pjesën e poshtme të memorjes për sistemin operativ.

Gjatë zhvendosjes, *boot* lexon direktorinë root të paisjes së butueshme. Për ta bërë këtë, ai duhet të kuptojë formatin e direktorisë dhe file-s sistem, e cila e bën këtë gjë. Më pas ai lexon në kernelin e sistemit operativ dhe kërcen tek ai. Në këtë moment, *boot* ka përfunduar punën e tij dhe kerneli po ekzekutohet.

Kodi fillestari në kernel është i shkruar në gjuhë asembler dhe është shumë i varur nga makina. Puna tipike përfshin organizimin e stack-ut të kernelit, identifikimin e tipit të CPU, llogaritja e sasisë së RAM-it prezent, ç'aktivizimi i interrupteve, aktivizimi i MMU, dhe në fund thirrja në gjuhën C e procedurës *main* për të startuar pjesën kryesore të sistemit operativ.

Kodi në C ka gjithashtu nevojë për të bërë inicializimin, por kjo është më tepër logjike sesa fizike. Ai starton jashtë duke alokuar një buffer mesazhi për të ndihmuar në eleminimin (debug) e problemeve në butim. Ashtu siç bëhet inicializimi, mesazhet shkruhen këtu për atë se çfarë po ndodh, kështu që ato mund të nxirren pas një dështimi në butim me anë të një programi të veçantë diagnostikues.

Pastaj strukturat e të dhënave në kernel janë alokuar. Shumica janë me madhësi fikse, por pakica, siç është bufferi i cachesë dhe disa struktura page table, varen nga sasia e disponueshme e RAM-it.

Në këtë moment sistemi fillon autokonfigurimin. Duke përdorur file-t e konfigurimit që tregojnë çfarë lloj paisjesh periferike mund të janë prezantë, sistemi fillon kontrollin e paisjeve për të parë se cila prej tyre është aktualisht prezent. Në se një paisje e kontrolluar i përgjigjet kontrollit, ajo i shtohet një tabele me disa paisje të caktuara. Në se përgjigja dështon, supozohet se paisja mungon dhe që tanë e tutje injorohet.

Që kur lista e paisjeve ka qenë përcaktuar, driverat e paisjeve duhet të janë vendosur. Kjo është një pikë në të cilën sistemet UNIX ndryshojnë disi. Në veçanti, 4.4BSD nuk mund të ngarkojë driverat e paisjeve dinamikisht, kështu që ndonjë paisje periferike driveri i së

cilës nuk është lidhur statikisht me kernelin nuk mund të përdoret. Në kontrast me të, disa versione të tjera UNIX, siç është Linux, mund ti ngarkojë driverat dinamikisht (ashtu si të gjitha versionet MS-DOS dhe Windows).

Argumentat pro dhe kundër për ngarkimin dinamik të driverave janë interesant dhe të rëndësishëm për tu konstatuar shkurt. Argumenti kryesor për ngarkimin dinamik është që vetëm një binar mund të dërgohet tek klientët me konfigurime të ndryshme dhe driverat duhet të ngarkohen automatikisht sipas nevojës, mundësish përmes një rrjeti. Argumenti kryesor kundër ngarkimit dinamik është siguria. Në se ti po punon në një site të sigurtë siç është baza e të dhënave të një banke apo një Web server i përbashkët, ju doni ndoshta ta bëni atë të pamundur për të futur kode të rastesishme në kernel. Administratori i sistemit mund ti mbajë burimin e sistemit operativ dhe file-t objekt në një makinë të sigurtë, ku të mbështeten të gjitha sistemet, dhe të transportojë kernelin tek makinat e tjera përmes një LAN. Në se driveri nuk mund të ngarkohet dinamikisht, kjo gjë i ndalon operatorët makinë dhe të tjerët që dinë passwordin të fusin gjëra të rrezikshme në kernel. Gjithashtu, në site të mëdha, konfigurimi hardware njihet tamam në kohën në të cilën sistemi kompilohet dhe linkohet. Ndryshimet janë jashtëzakonisht të mjaftueshme që të rilinkohet sistemi kur shtohet një paisje hardware.

Ndonjeherë të gjitha hardware-t kanë qenë konfiguruar, dhe gjëja tjetër për të bërë është për të ndjekur me kujdes procesin 0, për të organizuar stack-un e tij, dhe për ta ekzekutuar. Procesi 0 vazhdon inicializimin, duke bërë gjëra si programimi në kohë reale i orës, montimi i file-ve sistem root, dhe duke krijuar *init* (process 1) dhe page daemon (process 2).

Init kontrollon flagun e tij për të parë në se është supozuar të dalë një user apo shumë usera. Në rastin e parë, ai lejon lindjen (fork off) e një procesi që ekzekuton shellin dhe pret për këtë proces të përfundojë. Në rastin e dytë, ai lejon lindjen e një procesi që ekzekuton skriptin e shellit që inicializon sistemin, */etc/rc*, të cilat mund të bëjnë kontrolllet e lidhjes logjike të file-s sistem, montim të file-ve sistem shtesë, të startojnë procese daemon, e kështu me radhë. Më pas ai lexon */etc/ttys*, e cila liston terminalet dhe disa nga karakteristikat e tyre. Për çdo terminal aktiv, ai bën një kopje të vvetvtes, e cila bën disa administrime dhe më pas ekzekuton një program të quajtur *getty*.

Getty vendos shpejtësinë e rreshitit dhe karakteristika të tjera për çdo rresht, dhe më pas shtyp:

login:

në ekranin e terminalit, dhe përpinqet të lexojë emrin e userit nga tastjera. Kur dikush ulet në terminal dhe formon një emër tek login, *getty* përfundon duke ekzekutuar */bin/login*, programin login. *Login* më pas kërkon një password, e enkripton atë, dhe e verifikon atë përkundrejt passwordit të ruajtur në file, */etc/passwd*. Në se është korrekt, login e zëvendëson veten me userin e shellit, i cili më pas pret për komandën e parë. Në se nuk është korrekt, login vetëm pyet për një emër tjetër useri. Ky mekanizëm ilustrohet në figurën 10-12 për një sistem me tre terminale.

Në figurë, proçesi getty që po punon për terminalin 0 po pret ende për një input. Në terminalin 1, një user ka shtypur një emër tek login, kështu që getty ka mbishkruar veten me login, i cili po kërkon passwordin. Një login i suksesshëm tashmë ka ndodhur në terminalin 2, duke bërë shellin të shtypë prompt (%).

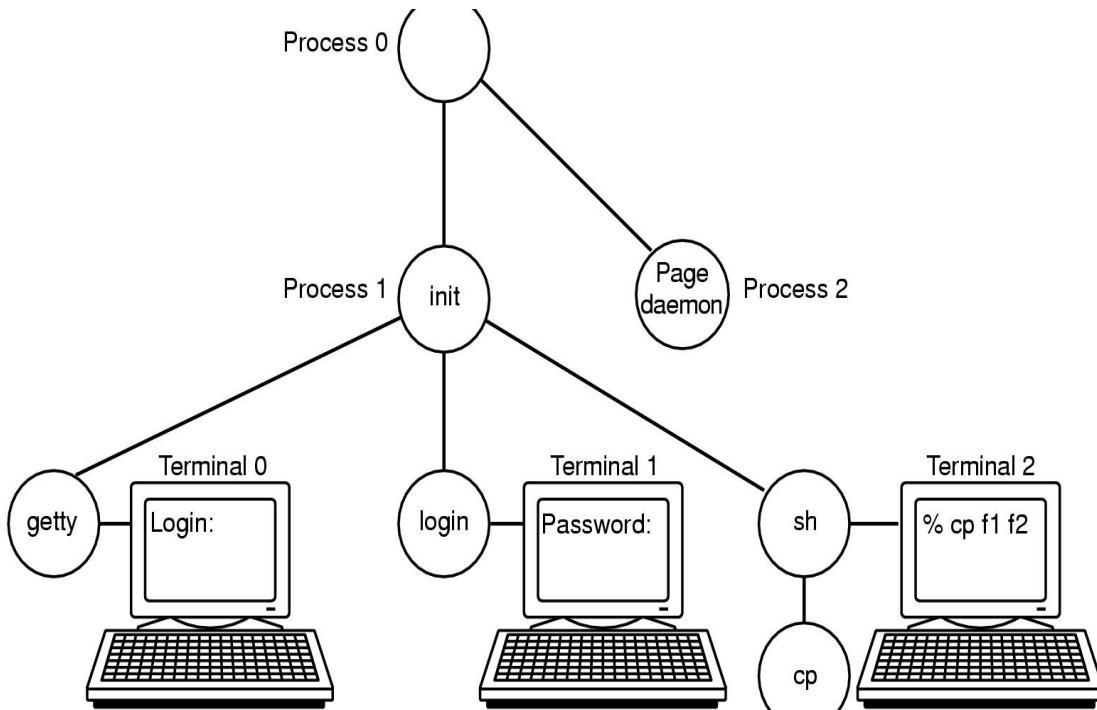


Figura 10-12. Sekuenca e proceseve për butimin e disa sistemeve LINUX.

Useri më pas shtyp:

`cp f1 f2`

e cila e bën shellin të bëjë një proces fëmijë dhe ky proces duhet të ekzekutojë programin `cp`. Shelli është i bllokuar, duke pritur fëmijën të përfundojë, kohë në të cilën shelli do të shtypë promptin tjeter dhe ta lexojë nga tastjera. Në se useri në terminalin 2 ka shtypur `cc` në vend të `cp`, programi kryesor i kompilatorit të C duhet të k ishte filluar, që të mund të kishte bërë shumë procese për të ekzekutuar hapa të ndryshëm të komplimit.