Capitotul 1 INTRODUCERE

Un sistem de operare este un program care gestioneaza hardware-ul calculatorului. Deasemenea asigura o baza pt. aplicatii si este ca un intermediar intre user si hardware-ul calculatorului.

1.1 Ce e un SO?

Este o parte importanta din aproape fiecare calculator. Un sistem poate fi impartit in 4 componente: Hardware, SO, aplicatiile shi utilizatori.
Scopul unui SO:

Sa execute programele si sa usureze rezolvarea problemelor utilizatorilor.
Sa faca sistemul mai usor de utilizat.
Folosirea harware-ului unui calculator intr-ul mod eficient.

1. Hardware – Ofera resursele de calcul (CPU, memory, I/O devices).
2. SO – controleaza si coordoneaza utilizarea hardware-ului intre aplicatiile de la fiecare user.
3. Programele aplicatie – defineste modul in care resursele sistemului sunt folosite pentru a rzovla problemele de calcul ale utilizatorilor (compilayoare, baze de date, jocuri).
4. User (oameni, alte calc).

Schema pag 4 fig 1.1

1.1.1 User view(Cum este vazut de user)

Cum este vazut calculatorul de catre user variaza cu interfata folosita.

User sta in fata PC -> un asemenea sistem este facut ca un user sa detina monopolul asupra resurselor. In acest caz, SO este proiectat sa poate fi folosit cat mai usor(ease of use), cu putina grija asupra perfomantei si nici una fata de utilizarea resurselor.

Sistem de tip mainframe(sau minicomputer) -> SO este dezvoltat pt a maximiza utilizarea resurselor.

Statie conectate la retea -> SO este realizat sa faca un compromis intreutilizare individuala si utilizarea resurselor.

1.1.2 System View(cum este vazut SO de sistem)

Din punctul de vedere al calculatorului SO este un program care este cel mai apropiat hardware-ului. Putem vedea SO ca un alocator de resurse.
O alta vedere al unui SO pune accentul pe necesitatea de a controla dispozitivele de I/O si programele user. Aici SO este un program de control care previna erorile in executia programelor si utilizarea incorecta a calculatorului.

O definitie mult mai uzuala este ca un SO este programul care ruleaza tot timpul pe calculator -> kernel.

1.1.3 Scopul sistemului(System Goal)

Un SO este mai usor de definit daca spunem ce face decat ce este. Scopul principal al unor SO este sa ofere **comoditate** pt utilizator.(este cazul PC-urilor mici)
Alte SO au scopul principal de a avea operatii **eficiente** cu sistemul calcu-lui.(cazul sistemelor mari, multiuser).
In trecut eficienta era mai improtanta decat comoditatea. Cu timpul au fost adaugate intefete grafice -> comoditate.

1.2 Sisteme mainframe

Au fost primele calculatoare.

1.2.1 Sisteme Batch

User-ul nu interactionaeza direct cu caclulatoru. User-ul pregatea un job si il introducea (submitted) in calculator. Job-ul era in forma cartelelor perfoarate; dupa un timp aparea rezultatul(output). Output-ul consta in rezultatul programului, continultul final al memoriei si continulul registrelor. SO era simplu, scopul lui era sa transfere controlul de la un job la urmatorul. SO era tot timpul rezident in memorie.
Pentru a mari rata de procesare: Operatorii grupau joburi cu nevoi asemanatoare si le rulau ca un grup.
Fig 1.2 pg 8 memoria unui sistem batch simplu
Introducerea discurilor a permis pastrarea job-urilor pe disc. Avand acces direct la mai multe job-uri, SO putea face job scheduling, putea gestiona resurslele mai eficient.

1.2.2 Sisteme multimrogramate

Cel mai improtant aspect la job scheduling-ului este abilitatea de multiprogram. Multiprogramarea mareste utilizarea CPU prin organizarea job-urilor astfel incat CPU sa albat tot timpul unul de rulat. SO pastreaza mai mult joc-uril simultan in memorie. Cand un job trebuie sa astepte SO comuta la un alt job.
Multiprogramarea este primul caz in care SO trebuie sa ia decizii pt useri. Toate joburile sunt pastrate in job pool(piscina cu joburi :))).Aici toate taskurile rezidente pe disc asteapta sa le fie alocata memorie principala. Daca mai multe joburi sunt gata sa fie adusa in mem, si nu este loc pt toate atunci SO trebuie sa aleaga pe care le aduce. Respunsabil de aceasa este **job scheduling**. Daca mai multe joburi sunt gata sa ruleze **CPU scheduling** decide care sa ruleze.
Ce terbuie sa ofere SO?

- Rutine I/O oferite de sistem
- Managementul memoriei – sistemul aloca memoriea mai multor joburi
- CPU scheduling – siistemul trebuie sa aleaga ce job sa ruleza.

1.2.3 Sisteme time-sharing

Time sharing este o extenie logica a multiprogramarii. CPU executa mai multe joburi prin comutare de la unul la altul, dar comutarea se face asa repede incat user-ul poate interactiona cu fiecare program in timp ce ruleaza. Un sistem interactiv faciliteaza comunicatia directa intre user si sistem. Timpul de raspuns este scurt.
Un SO time-sharing permite mai multor utilizatori sa foloseasca calculatorul simultan. Doar un mic timp CPU este necesar fiecarui utilizator. Ca sa se obtina timpi de raspuns rezonabili, joburile trebuie sa fie introduse in evacuate din Mem. principala de pe shi pe disk. (memorie virtuala).
Este furnizata comunicarea intre user si sistem, cand SO termina de executat o comanda, cauta urmatoarea comanda de la un alt user.

1.3 Sisteme desktop

Sunt dedicate unui singur utilizator.
Dispozitivele I/O sunt keyboard, mose, display, printer. Pricipalul scop este maximizarea confortului si sensibilitatea utilizatorului.

1.4 Sisteme multiprocesor(paralele)

Este un sistem cu mai multe CPU care comunica. Procesoarele impart bus-ul, clock-ul si cateodata memoria. Comunicarea are loc de obicei prin memoria partajata.
Este un sistem cuplat foarte starns.
Avantaje:

- Prin numarul mare de procesoare, se spera cresterii vitezei de procesare.
- Cost mai mai scazut decat mai multe sisteme uni-procesor prin paratajarea resurselor.
- Fialbilitate crescuta: Defectarea unui procesor nu va opri tot sistemul.

Multiprocesare simetrica(SMP):

- fiecare procesor ruleaza o copie identica a SO
- Mai multe procese pot rula deodata fara a deteriora performanta
- Cele mai mult SO moderne folosesc multiprocesare simetrica

Multiprocesare asimetrica(SMP):

- Fiecare procesor are un atribuit un anumit task
- Un procesor controleaza sistemul; el scehduleaza si distribuie lucrul catre procesoarele slave
- Este regasita in sisteme foarte mari.

1.5 Sisteme distribuite

Sistemele distribuite depind de retea pentru o functionaliatate corecta. Prin comunicare Sist. distribuite pot imparti taskurile si a furniza un set bogat de optinui pt useri.
Un sist distribuit distribuie calculul intre procesoare distincte.

Este un sisrtem cu o relatie de lagatura mai lejera ca la sist paralele. Fiecare procesor are memoria sa propie. Procesaorele comunica unele cu altele prin linii de comunicatie(bus-uri, linii de telefon).
Avantaje:

- Impatrirea resurselor
- Viteza de calcul marita
- Siguranta
- Comunicare

Are nevoie de infrastructura unei retele, retea locala LAN sau intinsa WAN. Poate fi un sistem peer-to-peer sau un sistem de tip client-server.

1.5.1 Sistem client-server

Sistemele centralizate de azi au rolul de sisteme server si satisfac cererile de la clienti.

- Sisteme Compute-server(server calcul) – ofera o interfata catre care clientii pot trimite ceredi de a executa un proces. Ca raspuns ele executa procesul si trimit clientului rezultatul.
- Sisteme File-server – ofera o interfata unde clientii pot creea, modificam citii shi sterge fisiere.

1.5.2 Sisteme peer-to-peer

1.6 Sisteme clustered

Aceste sisteme aduna mai multe CPU laolalta. Calculatoarele clusterate impart spatiul de stocare si sunt strans legate printr-o retea LAN. Fiecare masina poate monitoriza o alta masina. Daca masina monitorizata cedeaza, atunci ce care monitorizeaza poate sa preia memoria sa si sa reia executia aplicatiilor care rulau.
Clustering asimetric: o masina se afla in modul de standby, si monitorizeaza serverul activ. Daca acesta se defecteaza, masina dinstandby devine serve.
Clustering simetric: Doua sau mai multe masini ruleaza aplicatii si ele se monitorizeaza una pe alta.

1.7 Sisteme Real-Time

Sunt folosite cand sunt puse restrictii de timp asupra operatiilor unui procesor. Are restrictii de timp bine definite, care trebuie respectate, altfel sistemul se opreste. Sistemele in timp real pot fi hard sau soft.
Sisteme in timp real hard
Este garanta ca un task critic sa fie terminat la timp.

- Dispozitivul de stocare secundar ele limitat sau lipseste. Datele sunt stocate in memorie temporara sau in memorie read-only(ROM)
- Se genereaza conflicte cu sistemele time-sharing si nu sunt suportate de SO de uz general.

Sisteme in timp real soft

Task-urile critice au prioritate fata de celelelalte, aceasta prioritate este pastrata pana cand se termina task-ul.

- Utilizare limitata in contolul industrial si robotica.
- Folositor in aplicatii care necesita facilitati avansate ale SO.

1.8 Sisteme Handheld

Aici intra PDA-uri, celulare. Cei care dezvolta SO pt dispozitive handheld terebuie sa ia in considerare:

- memoria limitata -> gestiune a memoriei foarte buna
- viteza mica a procesoarelor -> SO care sa nu supraincarce procesorul
- display-ul foarte mic

1.9 Migrarea trasaturilor

fig 1.6 pag 20
Trasaturi mai demult disponible doarpe mainframe-uri au fost adoptate si la microcomputere.

1.10 Medii de calcul(Computing Enviroment)

1.10.1 Computing traditional

Mediu tipc pt aplicatii office. Tehnologia web impinge limitele mediului traditional.

1.10.2 Computing bazat pe Web

S-a marit importanta relelelor si viteza acestora.

1.10.3 Embedded Computing

Ruleaza SO embedded real time.

Capitloul 3 STRUCTURA UNUI SO

3.1 Componentele sistemului

Putem crea un sistem complex doar prin partitionarea lui in mai multe parti.
Componentele sistemului:

- Gestionarea proceselor
- Gestionarea memoriei principale
- Gestionarea fisierelor
- Gestionarea sistemelor I/O
- Gestionarea memoriei secundare

- Retelistica
- Protectia sistemului
- Interpretor de comenzi

3.1.1 Gestiunea Proceselor

Un proces este un program care se afla in executie. Un proces are nevoie de anumite resurse: timp CPU, memorie, fisiere, dispozitive I/O pentru a-si incheia task-ul. Executia unui proces este secventiala.
SO este responsabil de urmatoarele cand este vorba de gestiunea proceselor:

- Creearea si stergerea proceselor user si sistem
- Suspendarea si reluarea proceselor
- Asigura mecanisme pentru:
  - sincronizare
  - comunicatie
  - tratarea deadlock-urilor

3.1.2 Gestiunea memoriei principale

Memoria are un caracter volatil. Isi pierde continutul in caz de esec. De aceea este necesara o gestiune a memoriei.
SO se ocupa in cadrul gestiunii de memorie cu urmatoarele:

- Sa stie ce parte a memoriei este utilizate si de cine
- Sa decida care procese sa fie incarcate in memorie cand un spatiu devine disponibil.
- Sa aloce si dealoce memoria in functie de necesitati.

3.1.3 Gestiunea fisierelor

Este componenta cea mai vizibila a unui SO. Un fisier este o colectie, cu informatii asemanatoare, definita de utilizator.
SO este responsabil de urmatoarele:

- Creearea si stergerea fisierelor
- Creearea si stergerea directoarelor
- Oferirea sprijinului pt primitive pentru manipularea fisierelor si directoarelor.
- Maparea fisierelor in memoria secundara.
- Realizarea unui back-up a fisierelor pe medii de stocare stabile(non-volatile)

3.1.4 Gestionarea sistemelor I/O

Un sistem I/O este compus din:

- O componenta de gestionare a memoriei care inculde buffering, caching si spooling
- O interfata generala cu driver-ul dispozitivului.
- Drivere ce specifica dispozitivele hardware.

3.1.5 Gestionarea memoriei secundare

Mem principala este volatila. Dimensiunea ei este redusa si nu poate pastra toate datele si programele permanent. Sistemul trebuie sa dispuna de un mediu secundar de stocare pentu a putea pastra toate datele. Calculatoarele moderne folosesc discurile ca mediu principal de stoacre a datelor si a programelor.
In legatura cu gestiunea memoriei secundare SO este responsabil cu:

- Gestiunea spatiului liber
- Alocarea spatiului de stocare
- Disk scheduling

3.1.6 Reteaua

Un sistem distribuit este o colectie de procesoare care nu impart memorie, dispozitive periferice sau clock. Fiecare CPU are memoria lui. Procesoarele comunica intre ele cu ajutroul unei retele de comunicare. Comunicarea are loc folosind un protocol. Un sistem distribuit ofera posibilitatea ca user-ul sa acceseze diferite resurse.

3.1.7 sistem de protectie

Protectia se refera la mecanisme de control a accesului programelor, proceselor sau utilizatorilor la resursele sistemului.
Mecanismul de protectie trebuie sa ofere distinctie intre acces autorizat si neatorizat.

3.1.8 Sistem interpretor de comenzi

Unul din cele mai importante programe dintr-un SO este interpretorul de comenzi. Multe comenzi sunt date unui SO printr-o declaratie de control, care se ocupa cu: creearea proceselor, manevrarea proceselor I/O, gestiunea memoriei, accesul la sitemul de fisiere...
Programul care citeste si intepreteaza declaratiile numeste command-line interpreter sau shell(UNIX). Functia lui este simpla, sa aduca noua comanda si sa o execute.

3.2 Serviciile unui SO

Un SO faciliteaza programarea prin aceate servicii.
Executia programului: Capabiliatatea sistemului de a incarca programul in memorie si al executa.
Operatii I/O: din momnet ce programele user nu pot executa operatii de I/O in mod direc, SO trebuie sa ofere mijloace pt aputea executa operatii I/O
Manipularea sistemului de fisiere: Programul trebuie sa poate citii, scrie, creea si sterge fisiere
Comunicatie: Schimbul de informatie intre procese care se executa fie pe acelasi procesor sau pe sisteme diferite conectate prin retea. Poate fi implementat fie prin memorie partajata sau prin trimiterea de mesaje.

Detectia erorilor: Sa ofere o executie corecta prin detectia erorilor in CPU, in hardware-ul memoriei, in dsiapozitivele I/O sau in programele user.

Exista si functii aditioanale nu insa pt a ajuta user-ul ci pt a asigura operatii sinstem eficiente.
Alocarea resurselor: Alocare resurselor la mai multi useri sau la mai multe job-uri care ruleaza simultan.
Raportul resurselor: Tine evidenta care utilizatori folosesc cate si care resurse. Pentru statisctica sau intocmirea unei note de plata.
Protectie: Se asigura ca accesul ala resursele sistemului este controlat.

3.3 Apeluri de sistem

Asigura intefata dintre un proces si SO.
Sunt disponibile ca instructiuni in limbaj de asamblare. Limbajele de programare care inlocuiesc permit ca apelurile sistem sa fie facute in mod direct(ex., C, C++).
Sunt folosite 3 metode sa se transmita parementrii de la un program(care ruleaza) la SO:

- Prin registrii
- Stocarea parametriilor intr-o tabele de mamorie, si adresa tabelei este pasata ca un parametru intr-un registru.
- Parametrii se pun in stiva de catre program si se face pop de catre SO.

pag 65 poza
Tipuri de apeluri sistem: contolul proceselor, gestiunea fisierelor, managenmentul dispozitivelor, intretinerea informatiei si comunicare.

3.3.1 Controlul proceselor

Daca e face un apel de sistem sa se termine programul care ruleaza in mod anormal se pune continutul memoriei din momentul aparitiei apelului si se afiseaza un mesaj de eroare. Continultul memoriei poate fi exeminat de un debugger.
Cand un proces se termina(fie in mod normal sau abnormal) SO trebuie sa transfere contolul interpretorului de comenzi. Acesta citeste urmatoarea comanda. Intr-un sistem interactiv se continua cu urmatoarea comanda. Intr-un sistem Batch interpretorul opreste tot job-ul shi continua cu urmatorul.
MS-DOS sistem mono-tasking-> interpretorul de comenzi este incrarcat in mem cand se deschide calculatoru.(pg 68 poza 3.3 a) Interpretorul de comenzi incarca programul de rulat in mem scriind peste o mare parte a sa. Se ruleaza programul. Cand se termina din cauza unei erori sau a unui apel de sistem, codul de eroae est esalvat. Partea care a mai ramas din interpretorul de comenzi incarca restul de pe disc si face public codul de eroare pt user sau prg urmator. Cu toate ca MS-DOS = sist monotasking acesta are o metoda pt executie concurenta limitata. Un program TSR iasa cu apelul sistem terminate ans stay resident cand este activata o intrerupere. MS-DOS rezerva spatiul ocupat de programul TSR, astfel incat nu este suprescris cand interpretorul de comenzi este reluat.
FreeBSD sistem multitasking -> Cand userul se logheaza la sistem (interpretorul de comenzi) shell-ul este rulat. Pt a starta un nou proces shell-ul executa un apel sistem de tip fork iar pt a executa procesul un apel de tip exec. shell-ul poate astepta sa se execute

procesul sau poate sa il ruleze in background. Daca procesul ruleaza in backround nu poate fi accesat direct. Operatile de I/O sunt facute prin fisiere. Cand procesul a fost rulat, executa un apel de tip exit ca sa se termine.
poza pag 66

3.3.2 Gestiunea fisierelor

User-ul trebuie sa creeze, sterga, citeasca, modifice, etc fisiere. Fiecare apel sistem are nevoie de numele fisierului si poate de unele din atributele sale.

3.3.3 Gestiunea dispozitivelor(device)

Un program in timp ce ruleaza poate avea nevoie de resurse suplimentare sa poate continua(memorie, acces la fisiere, etc). Daca resursele nu sunt disponibile programul trebuie sa astepte pana acestea devin disponibile. Fisierele pot privite ca dispozitive reala sau virtuale. Daca sistemul alre mai multi useri, mai intai trebuie sa "cerem" dispozitivul pentru a asigura o utilizare exclusiva. Cand terminam cu dispozitivul trebuie eliberat.

3.3.4 Mentinerea informatiei

Multe apeluri sistem exista pt a transfera informatie intre user si SO. Apel sistem pt returnarea timpului si a datei curente, marimiea memoriei, nr de procese, bla bla.

3.3.5 Comunicarea

Doua modele de comunicare: prin mesaje si prin memorie partajata.
Comunicare prin mesaje(mesage passing): Mai intai trebuie deschisa o conexiune. Trebuie sa cunoastem numele celui cu care comunicam. Fiecare PC are un nume(host name), fiecare proces are un nume(process name). Pentru aflarea numelor avem nevoie de apeluri sistem. Procesul recepient(care primeste mesajele) terbuie sa isi dea acordul pt ca o comunicatie sa aiba loc.(apel accept). Procesele care accepta conexiuni sunt programe sistem creeate special pt asta(executa apelul wait pana cand se faca o conexiune) Sursa comunicatiei = client, iar cel care primeste = server. Clientul si serverul schimba mesaje prin apelurile read shi write message. La inchiderea conexiunii se executa apeleul close conection.
Comunicare prin memorie partajata(shared meomory): Procesele folosesc apeluri sistem pt a avea acces la regiuni de mem ala celuilalt proces. SO incearca sa previna ca un proces sa acceseze mem celuilalt proces. Comunicarea prin mem partajata are nevoie ca mai multe procese sa fie de acord sa inlature aceasta restrictie. Procesele sunt responsalbile ca sa nu scrie in acceashi locatie simultan.
pg 72 poza 3.5

3.4 Programe sistem

Ofera un mediu convenabil pt dezvoltarea si executia programelor. Programele pot fi impartite in: Gestiunea fisierelor, Informatia starii sistemului, modificare de fisiere, suport pt limbaje de programere, incarcarea si executia programelor, Comunicatie.
Cei mai multi utilizatori vad SO ca fiind definit de porgramele sistem si nu de apelurile de sistem.

3.5 Structura sistemului

Un sistem asa de mare si complex ca un SO trebuie sa fie gestionat cu grija ca sa functiuneze corect si sa fie modificat cu usurinta. Sistemul este partitionat in componente mici.

3.5.1 Structura simpla

Multe sisteme comeriale nu au o structura bine definita. Multe SO au pornit ca un sistem simplu si limitat si au fost apoi dezvlotate.
MS-DOS: a fost scris sa ofere cea mai buna functionare in cel mai mic spatiu. Nu este divizat in module si desi are o anumita structura, interfata sa si nivelele de functionare nu sunt indeajuns separate. pag 75 poza 3.6 structura MS-DOS
UNIX: Initial era limitat de functionarea hardware-ului iat UNIX avea o structura limitata. Este format din 2 parti distincte: Programele sistem si kernel-ul. Kernel-ul este si el divizat intr-o serie de interfete si drivere de dispozitive. Putem privi UNIX ca un SO structurat(layered) pag76 poza.
Totul sub interfata apelurilor sistem si peste hardware este kernel. Kernelul furnizeaza sistemul de fisiere, CPU scheduling, gestiunea memoriei, si alte functi ale SO; un numar mare de functii pt un nivel. Acest lucru face UNIX sa fie dificil de dezvoltat, deoarece schimbarile dint-o sectiune pot avea efecte adverse asupra alte zone.

3.5.2 Structura startificata

Modularizarea sistemelor poate fi facuta in mai multe moduri. Una este stratificarea unde SO este impartit intr-un numar de straturi sau nivele. Fiecare nivel este construit desaupra celuilalt. Nivelul de jos (nivelul 0) este hardware-ul iar nivelul cel mai de sus este interfata cu utilizatorul. Principalul avantaj al structurii pe nivele este modularitatea. Nivelele sunt selectate in asa fel incat fiecare foloseste doar functiile(operatiile) si serviciile nivelelor de jos. Aceasta abordare simplifica verificarea sistemului. Fiecare nivel este implementat doar cu operatiile oferite de nivelul de jos. Un nivel nu trebuie sa stiu cum sunt implementate functiile ci doar ce fac. Cea mai mare dificultate cu abordarea stratificata este definirea nivelelor, deoarece un nivel poate folosi doar nivelele de sub el. Alta problema este ca sunt mai ineficiente decat alte modele. Cand un proce executa o operatie I/O executa un apel de sistem care se regaseste in nivelul I/O, care la randul lui apeleaza alte nivele. La fiecare nivel parametrii pot fi modificat sau datele sunt pasate. Trecerea prin fiecare nivel intarzie apelul => un apel sistem dureaza mai mult decat pe un sistem fara straturi.
Sistemul OS/2 este un descendent al MS-DOS cu multitasking. sistemula fost implementat intr-un mod stratificat. pag 78 poza

3.5.3 Microkerneluri

Cu dezvoltarea sistemului UNIX kernelul a devenit tot mai mare si dificil de intretinut. S-a dezvoltat un kernel mai mic -> microkernel. Aceasta abordare scoate toate componentele neesentiale din kernel si le implementeaza in programele user si sistem. rezulta un kernel mai mic. Functia principala a kernelului este sa asigure comunicarea intre programul client si diferitele servicii care ruleaza in spatiul utilizator. Comunicarea se realizeaza prin timiterea de mesaje. Avantajul abordarii mocrokernel este usurnta cu care poate fi extins SO. SO poate fi adaptat mai usor unor noi arhitecturi. Microkernelul ofera mai multa securitate si siguranta, deoarece cele mai multe servicii sunt rulate ca procese user. Daca un serviciu se deterioaraza nu influenteaza restul SO.

3.6 Masini virtuale

O implementare stratificata este dusa la un nivel logic cu conceptul de masina virtuala. Schedulingul CPU-ului poate creea impresia ca fiecare user are propriul procesor. Spooling si sistemul de fisiere poate furniza card readere si imprimante virtuale.
Un sistem normal cu divizare de timp(time sharing) poate srevi ca consola operatorului masinii virtuale.

3.6.1 Implementare

Desi conceptul de masina virtuala este folositor se implementeaza greu. Masina care sta la baza are 2 moduri de functionare: modul user si modul monitor. Software-ul masina virtuala paote rula in mod monitor, deoarele este SO. Masina virtuala in sine poate rula doar inmod user. Un transfer de la modul user la modul monitor la o masina reala trebuie sa cauzeze un transfer la masina virtuala. Transferul poate fi facut relativ simplu. Daca un apel sistem este facut de catre un program care ruleaza pe masina virtuala are loc un transfer la monitorul masinii virtuale in masina reala. Cand monitorul masinii virtuale preia controlul poate modifica registrii si PC-ul pentru ca masina virtuala sa simuleze efectul apelului de sistem. La sfarsit poate fi reluata rularea masinii virtuale. Starea masinii virtuale a fost modificate aconform apelului. poza pag 81

3.6.2 Avantaje

Conceptul de masina virtuala furnizeaza protectie totala a resurselor sistemului dinmoment ce fiecare masina virtuala eset izolata de celelalte masini virtuale. Aceasta izolare insa nu permite o distribuite directa a resurselor.
Un sistem masina virtuala este un mod optim de a cerecta si dezvolta SO. Dezvoltarea sistemelor este facuta pe o masina virtuala si nu intrerupe operatiile de sistem normale. SO controleaza toata masina. La dezvoltare tot sistemul trebuie oprit pentru a putea face modificari si teste. O masina virtuala poate elimina o mare paret dinproblema. Programatorii au propria masina virtuala si dezvoltarea se face pe masina virtuala.

3.6.3 Java

Compilatorul Java produce bytecodes(coduri in byte) cu un aspect neutral fata de aritectura. Acestea sunt rulate apoi pe Java Virtual Machine(JVM)
JVM consta din: class loader, class verifier si un interpretor runtime. compilatoare Just-In-Time cresc perfomanta.
poza pag 85

3.7 Designul si implementarea sistemului

3.7.1 Tinta designului

La nivelul ce mai de sus modelul sistemului va fi infuentat de alegerea hardware-ului si tipul sistemului: batch, time-shared, etc.
Cerintele de design pot fi impartite in:

- scopul userilor: So sa fie usor de utilizat, usor de invatat, de incredere, sigure si rapide
- scopul sietm: SO sa fie usor de modelat, implementat si intretinut precum si flexibile, fara erori si eficeinte.

3.7.2 Mecanisme si politici

Mecanismele determina cum este implementat ceva, politicile decit ce va fi implementat. Separarea macanismelor de politici este un principiu f. important, ofera flexibilitate maxima daca deciziile referitoare la politici sunt schimbate ulterior.

3.7.3 Implementarea sistemului

Dupa ce un SO a fost modelat el trebuie implementat.
Traditional ele erau scrise in limbaj de asamblare. Acuma ele sutn scrise in limbaje de nivel inalt. Codul scris in limbaj de nivel inalt poate fi scris mai repede, este mai compact si este mai usor de inteles si debugenit. Un SO poate fi mutat pe ul alt hardware mult mai usor daca este scris intr-un limbaj de nivel inalt.

3.8 System Generation(SYSGEN)

Un SO este facut sa poate rula pe orice tip de masina, sistemul trebuie configurat pt fiecare calculator. Programul SYSGEN obtine informatia referitoare la cofigurarea sistemului hardware. Booting = pornirea calculatorului prin incarcarea kernelului. Bootstrep program = cod stocat in ROM care este in stare sa localizeze kenelul, sa il incarce in memorie si sa ii starteze executia.

Capitolul 4 PROCESE

Un sistem este compus dintr-o colectie de procese.

4.1 Conceptul de Proces

Un SO executa tipuri diferite de programe: sisteme batch executa job-uri iar sisteme time shared excuta programe user sau task-uri.

4.1.1 Procesul

Procesul este un program care se afla in executie. Excutia proceselor trebuie procesata in mod secvential. Un proces are inclus un program counter, o stiva(care patreaza datele temporare) si o sectiune de date(unde sunt pastrate varibilele globale).

4.1.2 Starea unui proces

In timp ce un proces se executa el isi schimba starea. Fiecare proces se poate afla in una din starile:

new: Procesul este creat.
running: sunt executate instuctiuni.
waiting: procesul sateapta se intervina un eveniment.
ready: procesul asteapta sa fie atribuit unui procesor.
terminated: procesul a terminat executia

poza 4.1 pg 97

4.1.3 Blocul de control al proceselor(PCB)

fiecare proces din SO este reprezentat de un bloc de control al procesului.
poza 4.2 pg 97
Acesta contine informatii asociate unui anumit proces:

- Starea unui proces
- Program counter
- Regisrtii CPU
- Informatia de scheduling a CPU-ului
- Informatie aspupra gestiunii memoriei
- Informatie resurselor
- Informatie referitoare la I/O

4.2 Schedulingul proceselor

Ideea este sa se ruleze mai multe procese simultan pt a marii utilizarea CPU.

4.2.1 Cozi de scheduling

Cand un proces este introdus in sistem el este pus in coada job(job queue). Aceasta coada contine toate procesele din sistem. Procesele rezidente in memorie care sunt ready si asteapta sa fie executate sunt pastrate intr-o lista nimita coada ready(ready queue).(pag

100 poza). Lista proceselor care asteapta dupa un dispozitiv I/O este numita coada dispozitivelor(device queue). Fiecare dispozitiv are propie coada. Reprezentarea comuna a schedulingului proceselor este o diagrama, queueing diagram(pag 102 poza). (Patratele sunt cozi. Sunt reprezentate 2 tipuri de cozi ready si I/O)
Un nou proces este pus initial in coada ready unde sta pana este selectat sa fie executat. Odata ce procesul este pasat procesorului si se executa avem cazurile:

- procesul poate avea o cerere I/O, este pasat atunci cozii I/O
- procesul poate creea un subproces si va astepta terminarea acestuia
- procesul poate fi inlaturat fortat de la CPU(interrupere) si este pus inapoi in coada ready

In primele 2 cazuri procesul isi schimba la candva starea din waiting in ready si este pus inapoi in coada ready.

4.2.2 Schedulere

Deoarece un proces migraeza intre cozile de scheduling SO trebuie sa poata selecta procese din aceste cozi in scopul schedulingului. Proecsul de selectie este realizat de scedulere.
Scheduler pe termen lung(long term): job scheduler, selecteaza care proces va fi adus in coada ready din memorie. Se executa cu o frecventa mica. Poate fi si mai incet. El controleaza gradulde multiprogramare(numarul proceselor dinmemorie). Este invocat doar cand un proces este scos din sistem.
Scheduler pe termen scurt(short term): CPU scheduler, selecteaza unul din procesele gata de executie si aloca unul din ele CPU-ului. Este executat cu o frecventa mare => trebuei sa fie f rapid.
Procesele pot fi descrise ca procese legate de I/O sau procese legate de CPU.
Procesele legate de I/O petrece mai mult timp facant o operatie de I/O decat calculand. Iar un roces legat de CPU pertrece mai putin timp executand I/O. Schedulerul pe termen lung trebuie sa faca o combinare buan intre cele 2 tipuri de procese. Daca toate procesele sunt I/O coada ready va fi mai tot timpul goala iar schedulerul pe termen scurt nu va avea de lucru. Daca procesele vor i CPU coada I/O va fi goala iar sistemul nu va fi echilibrat. pag 102 poza

4.2.3 Comutarea de context

Cand procesorul comuta de la un proces la altul starea fostului proces terbuei salvata si incarcata stara noului proces. Comutarea de proces este o "pierdere de timp", deoarece sistemul nu face lucru util in timp ce comuta. Timpii de comutare sunt dependenti de suportul hardware. Cu cat este mai complex SO cu atata terbuie depus mai mult lucru la comutare.

4.3 Operatii aspura proceselor

Procesele din sistemse executa concurent si terbuie creeate si diatruse dinamic.

4.3.1 Crearea proceselor

Procesele parinte creaza procese fiu care la arndul lor pot creea alte procese. Se formeaza astfel un arbore de procese(poza pag 104).
Exista mai multe cazuri de partajare a resurselor: parintele si fiul impart toate resursele, fii impart un subset din resursele parintelui, parintii si fii nu impart nici o resursa. Cand un proces parinte creaza altu avem 2 cazuri: Parintele isi continua executia in paralele cu fii sau parintle asteapta pana unii din fii sai au terminat. Referitor la spatiului de adrese avem 2 cazuri: procesul fiu este un duplicat al procesului parinte sau procesul fiu are incarcat in el en program.
Exemplu UNIX: un nou proces este creat cu apelul de sistem **fork**, **exec** este folosit dupa fork ca sa se inlocuiasca spatiul de memorie a procesului cu un nou program
4.3.2 Terminarea proceselor

Un proces se termina daca si-a terminat executia si ii cere SO sa fie sters(apelul **exit**). Procesul poate returna data procesului parinte(prin apelul **wait**). Toate resursele procesului sunt dealocate de SO. Terminarea unui proces mai apare shi in cazurile cand un proces cauzeaza terminarea unui alt proces. De obicei doar procesul parinte poate termina un proces fiu. Parintele poate termina(apelul **abort**) executia proceselor fiu din motivele:

- fiul a depasit utilizarea resurselor alocate lui
- taskul ascoiat fiului nu mai este necesar
- parintele si-a terminat executia si SO nu permite continuarea executiei unui fiu daca parintele a terminat. Daca unproces termina toti fii trebuie sa se termine -> terminare cascadata

4.4 Procese cooperante

Procese independente nu se pot afecta sau nu pot fi afectate de alte procese. Un proces cooperant poate afecta si poate fi afectat de celelelte procese.
Avantajele cooperari proceselor: Distribuita informatiei, marirea vitezei de computatie, modularitate, convenienta. Executia concurenta trebuei sa ofere mecanisme de comunicare si sincronizare intre procese. Un proces proucator produce informatie care este consumate de un proces consumator. Ca cele 2 procese sa ruleze simultan avem nevoie de buffere.
Unbound buffer nu pune nici o restructie de marime a bufferului. Consumatorii trebuie sa astepte noile obiecte dar producatorul poate produce tot timpul noi obiecte.
Bounded buffer: marimea buffer trebuie sa fie fixa. Consumatorul trabuie sa asepte daca bufferul este iar producatorul trebuie sa astepte daca bufferul este plic.

4.5 Comunicatia intre procese(IPC)

Este o alta metoda de a obtine rezultate identice cu procesele coopernate.

4.5.1 Sistemul de pasare a mesajelor

Trebuie sa furnizeze un mecanism de comunicare si sincroniazare intre procese. Procesele comunica fara a fi nevoie sa aiba variabile comune. IPC furnizeaza 2 operatii: send (mesaj) – mar mesajului este fixa si recieve(mesaj).
Dac a2 procese doresc sa comunice, ele trebuei sa stabileasca o conexiune intre ele si sa schimbe mesaje intre ele. Linia de comunicatie poate fi implementata fizic sau logic.
linia de comu
Acesta poate fi direct sau indirect, simetric sau asimetric.

4.5.2 Numele

Procesele care doresc sa cominica trebuie sa stie sum se refere pe celalalt.

4.5.2.1 Comunicatie directa

Fiecare proces care vrea sa comunice trebuie sa numeasca numele procesului cu acre comunica:
send (*P, message*) – send a message to process P
receive(*Q, message*) – receive a message from process Q
O linie de comunicatie are proprietatile:

Linia este stabilita in mod automat intre fiecare pereche de comunicare. procesele trebuie sa cunoasca identitatea celuilalu.

O legatura este asocatia cu fix 2 procese

Exact o legatura exista intre o prereche.

Legatura poate fi unidirectionala dar de obicei este bidirectionala.

4.5.2.2 Comunicatie indirecta

Mesajele sunt directionate si primite de la un mailbox (port). Fiecare port are un ID unic. Procesele pot comunica doar daca au un mailbox comun. Prmitivele sent si recieve sunt de felul urmator:
send(*A, message*) – send a message to mailbox A
receive(*A, message*) – receive a message from mailbox A
Un mailbox poate avea ca propietar SO sau un proces. Mailboxurile cu propietar un proces trebuie sa distingem intre proprietar si user. Cand un proces care are un mailbox temina, mailboxu dispare. Toate procesele care comunicau cu mailboxu trebuie instiintate ca acesta nu mai exista.
Proprietatile liniei de comunicatie:

Legatura este facuta doar daca procesele au un mailbox comun.

O legatura poate fi asoiata mai multor procese.

Fiecare pereche de procese poate avea mai multe legaturi de comunicatie.

Legatura poate fi uni- sau bidirectionala.

Impartirea mesajelor. Cand procesele P1, P2 si P3 impart mailboxu A iar P1 trimite un mesaj si P2 si P3 il primesc. Problema este cine intra in posesia mesajului. Solutiile la aceasta problema ar putea fi:
Doar doua procese sa fie asociate cu o legatura de comunicatie.
Doar un proces sa poata executa operatia recieve.

Permiterea sa se aleaga destinatatul in mod arbitrar de catre sistem. expeditorul este instiintat cine a fost destinatarul.

4.5.3 Sincronizare

Trimiterea mesajelor poate fi blocanta sau neblocanta adica sincrona sau asincrona.
Trimitere blocanta: procesul expeditor este blocat pana cand mesajul este primit de proces sau mailbox.
Trimitere neblocanta: Procesul trimite mesajul si isi continua executia.
Receptionare blocanta: Destinatarul se blocheaza pana cand un mesaj este disponibil.
Receptionare neblocanta: Destinatarul primeste fie un mesaj valid fie unul nul.
Sunt psibile de combinatii diferite de send si recieve. Cand si send si recieve sunt blocante avem un rendezvous intre expeditor si destinatar.

4.5.4 Buffering

La comunicatie mesajele schimbate de procesele de comunicare sunt pastrate in cozi temporare.
Avem 3 cazuri de implementare unei asemenea cozi:
Capacitate zero: Coada are lungimea de 0. In acest caz expeditorul trebuie sa blocheze pana cand este primit mesaju. Aceasta implementare este privita ca una fara buffer.
Capacitate limitata: Coada are lungime finita n. Cel mult n mesaje pot fi stocate in ea. Daca linia de comunicatie este plina expeditorul trebuie sa se blocheze pana cand se face loc in coada.
Capacitate infinita: Coada are lungime infinita. Expeditorul nu trebuie sa astepte.

4.6 Comunicatie in sisteme client server

4.6.1 Socketuri

Un socket este un capat al comunicatiei. O pereche de procese care comunica are nevoie de doua soketuri. Se foloseste in general o arhitectura client server. O conexiune trebie sa fie unica. Comunicarea se realizeaza concatenand adresa IP cu portul(socketul **161.25.19.8:162** se refera la port **1625** host **161.25.19.8).**
Trei tipuri diferite de socketuri: - Orientate pe conexiune, socketi faca ronexiune(Datagram sockets) si multicast socket(poate transmite la mai multi destinatari).
poza pag 118

4.6.2 Proceduri cu apelul la distanta(RPC)

RPC este un apel de procedura intre procesele dintr-o retea. Semantica RPC permite invocarea unei proceduri la distanta in acelasi mod ca una locala. Sistemul RPC ascunde detaliile necesare permitand ca comunicarea sa aiba loc. Acest lucru este posibil prin furnizarea unui stub pe partea clientului. Un stub separat exista pt fiecare procedura la distanta. Cand o procedura la distanta este apelata, RPC apeleaza stub-ul potrivit pasand parametrii trmisi catre preocedura. Acest stub localizeaza portul pe server si **marshalls**

parametrii. Acest proces include inpachetarea parametrilor intr-o forma in care pot fi transmisi prin retea. Stub-ul transmite apoi un mesaj serverului. Un stub similar al serverului primeste mesajul si apeleaza procedura. Daca este necesar valorile resultate sunt trimise inapoi clientului intr-un mod asemanator. Sistemele RPC definesc o reprezentare a datelor independente de masina: reprezentare externa a datelor(XDR). In partea clientului marshalling pesupune convertirea datelor dependente de masina in XDR inainte sa fie trasmise serverului. Pe partea serverului, datele in reprezentare XDR sunt transformate in reprezentare locala a serverului. poza pg 123

4.6.3 Invocarea metodelor al distanta(RMI)

poza pg 124
RMI este un mecanism Java similar cu RPC.
RMI permite programului Java sa invoce o metoda al unui obiect la distanta. Un stub este un proxy pentru obiectul la distanta si apartine de client. Cand un client invoca o metoda la distanta, stub-ul este apelat. Stub-ul client este respunsabil cu crearea unui pachet(parcel) care consta din numele metodei si parametrii metodei. Acest pachet este trimis la server unde un schelet al obiectelor la distanta il primeste. Acesta cheama metoda dorita. Rezultatul este apoi trimis intr-un pachet la client. poza pg 125

THREAD-uri

5.1 Introducere

Thread = numit si **lightweight process(LWP)**, este unitatea de baza a utilizarii CPU ; contine un ID a thread-ului, un numerator de program (PC), un set de registri si o stiva. Imparte cu thread-urile apartinatoare aceluiasi proces: segmental de cod, de date si alte resurse de operare a sist (de ex. : fisiere deschise si semnale).

Un proces normal are un singur thread de control. Daca procesul are mai multe thread-uri de control atunci el poate realiza mai multe task-uri deodata.

5.1.1. Motivatii

Multe programe care ruleaza pe un calculator modern sunt multithread. O astfel de aplicatie este implementata cu mai multe thread-uri de control. In anumite situatii unei singure aplicatii i se cere sa efectueze mai multe task-uri similare. O solutie ar fi sa ruleze ca un proces care accepta cereri. Aceasta solutie era de actualitate inainte ca thread-urile sa devina populare. Insa crearea de procese este destul de dificila, de aceea este mai simplu ca un proces care contine mai multe thread-uri sa serveasca aceluiasi scop, de ex. daca a fost

facuta o solicitare in loc sa creeze un alt proces creaza un alt thread care va raspunde cererii.

Thread-urile joaca un rol semnificativ in sistemele RPC (remote procedure call). In mod normal serverele RPC sunt multithread. Daca serverul primeste un mesaj, ii raspunde folosind un alt thread. Acest lucru permite serverului sa raspunda in acelasi timp la mai multe cereri.

5.1.2 Beneficii

Reactie : cu ajutorul multithreading o aplicatie permite unui program sa functioneze chiar daca o parte din ea este blocata sau executa operatie lunga, crescand astfel reactia.

Impartirea resurselor : in mod prestabilit thread-urile isi impar memoria si resursele procesului de care apartin. Beneficiul: o aplicatie poate avea mai multe thread-uri ale activitatii la aceiasi adresa de memorie

Economie : alocarea memoriei si a resurselor pentru crearea unui proces este costisitoare. De aceea e mai economic sa creezi si sa comuti contextual thread-uri deoarece ele impart resursele procesului.

Utilizarea arhitecturii multiprocesor : beneficiile multithreading cresc semnificativ atunci cand avem o arhitectura multiprocessor, unde fiecare thread poate rula in parallel pe procesoare diferite.

5.1.3 Thread-uri user si Kernel

Thread-uri user: sunt suportate deasupra thread-urile Kernel si sunt implemetate de library thread la nivel user. Libraria ofera posibilitatea crearii, aranjarii si administrarii thread-urilor fara ajutorul Kernel, de aceea sunt usor de folosit si administrat

Thread-uri Kernel: sunt suportate direct de catre sistemul de operare: realizeaza creari, administrari si aranjari ale thread-urilor directin spatial Kernel; din aceasta cauza thread-urile Kernel necesita mai mult timp pentru creare si administrare decat thread-urile user.

5.2 Modele Multithreading

Multe modele suporta atat thread-uri user cat si Kernel.

5.2.1 Modelul Many-to-One

Acest model mapeaza (insumeaza) thread-urile user la un singur thread Kernel. Administrarea thread-ului se face in spatiul user deci este efficient, dar intreg procesul se va bloca daca un thread face un apel de bllocare a sistemului. Deoarece doar un user thread poate accesa thread-ul Kernel la un moment, thread-urile multiple nu pot rula pe multiprocesoare.

5.2.2 modelul One-to-One

Mapeaza pentru fiecare thread user un thread Kernel. Asigura o mai multa concurenta decat modelul de dinainte deoarece daca un thread user solicita un apel de blocare a sistemului alte thread-uri pot functiona in continuare. Singura deficienta este ca creand un thread user trebuie creat si thread-ul Kernel corespunzator, acest lucru ingreutatind performantele aplicatiei.

5.2.3 Modelul Many-to-Many

Multiplexeaza mai multe thread-uri de nivel user cu o cantitate mai mica sau egala de thread-uri de nivel Kernel. Acest model permite programatorului sa creeze cate thread-uri vrea, iar concurenta nu este castigata deoarece Kernel poate aranja doar cate un thread pe rand. Acest model permite cresterea concurentei dar programatorul trebuie sa aiba grija sa nu creeze multe thread-uri intr-o aplicatie.

5.3 Restrictii de threading

In acest cap. vom discuta despre restrictiile ce apar in programele multithreading.

5.3.1 fork and exec System call

Spre deosebire de cap 4 semantica fork si exec System call se modifica. Daca un thread dintr-un program face o cerere fork, se dubleaza thread-urile sau este noul proces single-threaded? Unele sisteme UNIX au 2 versiuni de fork unul care dubleaza toate thread-urile si unul care dubleaza doar thread-ul care a cerut fork. Apelul exec functioneaza in acelasi mod descries in cap 4. inseamna ca daca un thread cere exec, programul specificxat ca si parametru inlocuieste intreg procesul inclusive toate thread-urile si LWP-urile.

Utilizare celor doua versiuni de fork depinde de aplicatie. Daca exec este cerut imediat dupa fork atunci nu mai este necesar dublarea fiecarui thread, deoarece programul dat ca parametru va inlocui procesul, cel mai bine este dublarea thread-ului care a facut cererea.

5.3.2 Anularea

Anularea threadurilor reprezinta terminarea unui thread inaite ca acesta sa se fi executat complet. Thread-ul anulat este deseori referit ca thread-ul tinta.
Anulare asincrona: un thread anuleaza imediat threadul tinta
Anulare cu intarziere: Threadul tinta poate sa verifice daca ar trebui sa termina, oferind modalitatea threadului tinta se se anuleze in mod ordonat.
Dificultate la anulare: Cand resurse sunt alocate unui thread anulat sau daca thredul a fost anulat in timp ce acualiza date ce le impartea cu alte threaduri. Aceasta problema apare la anulare asincrona. SO pune stapanire pe resurse dar de multe ori nu pe toate. Astfel anularea asincrona nu va elibera resursele necesare. La anularea cu intarziere anularea va avea loc doar cand threadul tinta verifica acest lucru. Anularea are loc la un moment cand ea este sigura. Pthreadurile numeste astfel de puncte puncte de anulare.

5.3.3 *Manipularea semnalelor*

La UNIX un semnal este folosit sa instiinteze un proces ca a avut loc un anumit event. toate semnalele urmaresc acelasi model:

- Un semnal este generat la aparitia unui anumit event.
- Semnalul generat este transmis la proces.
- Dupa ce a fost transmis, semnalul trebuie tratat.

Semnale sincrone: cand avem divison by 0, acces ilegal la mem...Semnalele sincrone sunt trimise aceluiasi proces care a executat operatia care a generat semnalul.
Semnale asincrone: sunt generate de eventuri externe unui proces care ruleaza. De obicei semnalul asincron este trimit unui alt proces.
Fiecare semnal trebuie sa fie tratat de unul din handlere: hanlerul default sau handlerul definit de user. Fiecare semnal are un handler de semnal default care este rulat de kernel cand tratam semnalul. Actiunea default poate fi suprascrisa de un handler de semnal definit de utilizator.
Transmiterea semlalelor este mai dificila la programele multithreading. In general exista urmatoarele optiuni unde sa se trimita semnalul:
- trimiterea semnalului la threadul la care se aplica semnalul
- trimiterea semnalului la toate threadurile din proces
- trimiterea semnalului la anumite threaduri din proces
- desemnarea unui thread special care va primi toate semnalele din proces

Versiuni multithreading ale lui UNIX permit threadului sa specifice ce semnale va accepta si care le va bloca.
Desi win 200 nu are suport pt semnale, ele pot fi emulate folosind apelluri asincrone de procedura(APC). APC pemite unui thread user sa specifice functia care va fi apelata cand threadul este anuntat de un event pariticular.

5.3.4 *Grupuri de threaduri*

Cand un server primeste o cerere el creeaza un thread care sa rezolve cererea.
Dezavantaje: Timpul pt creearea unui thread este destul de mare, threadul va vi sters indata ce iti va termina lucrul, prin existanta simultana a unui numar nelimitat de threaduri resursele sistemului se epuizeaza.
Solutia: Folosirea unui grup de threaduri(thread pool)
La inceperea procesului sunt create un nr de threaduri care sunt puse in acest grup. Cand serverul primeste o cerere el "trezeste" un thread. Cand thread-ul termina el este pus din nou in grup in starea de asteptare.
Avantajele thread pool: o cerere poate fi tratata mult mai repede, numarul de threaduri este limitat-> nu apare o supraincarcare a sistemului.

5.3.5 *Date specifice threadului*

Threadurile unui proces impart datele procesului. Este posibil ca threadul sa aiba nevoie de o copie proprie de date. Acest tip de date se numesc date specifice threadului.

5.4 *Pthreaduri*

Pthread se refera la standardul POSIX (IEEE 1003.1c) si defineste un API care se refera la creearea sincronizarea threadurilor. API specifica comportamentul thredului din librarie, implementarea depinde de dezvoltarea librariei. Pthreads sunt deseori regasite in SO UNIX.

5.5 *Thread-uri SOLARIS 2*

Capitolul 6 (CPU SCHEDULING)

6.1 Concepte de bază

Ideea de bază este că fiecare proces este planificat înainte de a fi executat pentru ca procesorul să nu stea degeaba când procesul în execuţie trece în starea de *wait*.

6.1.1 CPU – I/O Burst Cycle

Procesele au următoarea propritetate: executia lor constă într-un ciclu de executie CPU si apoi un I/O *wait*. Procesele încep cu o izbucnire (burst) de CPU urmată de o izbucnire(burst) de I/O wait după care cer iară CPU şi tot aşa. (a se urmări şi diagramele de la pag. 152-153)

6.1.2 CPU Scheduler

Atunci când procesorul devine nefolosit (idle), SO trebuie să aleagă un proces din coada de *ready*. Acest lucru este făcut de **short-term scheduler** (planificator de termen scurt, scrie de ăsta si prin cap 3 parcă). Coada de ready nu este neapărat de tip FIFO ci depinde de algoritmul fiecărui planificator.

6.1.3 Preemtive Scheduling

Deciziile de planificare pentru execuţia CPU sunt luate în următoarele circumstanţe:

1.Când un proces trece din *running* în *waiting* (cerere de I/O)
2.Când un proces trece din *running* în *ready* (apare o întrerupere)
3.Când un proces trece din *waiting* în *ready* (terminarea unei cereri de I/O)
4.Când se termină un proces

În cazurile 1 şi 4 nu există termen de planificare (trebuie ales un proces din coada de ready). Când planificarea are loc în aceste cazuri se spune ca ea este nepreemtivă (**nonpreemtive**), altfel planificarea este preemtivă (**preemtive)**. Sub planificare nepreemtivă, odată ce CPU a fost alocat unui proces, acesta deţine CPU până când se termină sau pâna când trece in *waiting*. Planificarea preemtivă are un dezavantaj (cost) şi anume: în momentul în care un proces este în mijlocul unui update de date şi este scos de un alt proces (prin preemţiune) care vrea să citească acele date care nu sunt completate trebuiesc folosite mecanisme care fac coordonarea acceselor la datele comune (shared). Preemţiunea are efect şi în design-ul kernel-ului (în timpul unei procesări de apel de sistem, kernel-ul poate fi ocupat cu un proces (ex schimbarea datelor importante de kernel, cum ar fi cozile I/O). Dacă un proces este preemtat în mijlocul acestor schimbări şi kernel-ul trebuie să modifice aceeaşi structură ar fi haos).
Cocluzie: Un sistem este preemtiv atunci când SO poate decide să oprească un proces (face planificare împotriva voinţei procesului).

6.1.4 Dispatcher (expeditorul??)

El este modulul care predă controlul CPU unui proces selectat de short-term scheduler (traducerea e mai sus) şi trebuie să: schimbe contextul, schimbe utilizatorul (user mode), sară la locaţia adecvată a programului utilizator pentru a-l restarta. Timpul care îi ia dispatcher-ului să oprească un proces şi să pornească altul se numeşte **dispatch latency**.

6.2 Scheduling Criteria

Caracteristicile folosite la comparaţia diferiţilor algoritmi de planificare pot face diferenţe substanţiale în determinarea celui mai bun algoritm. Criteriile includ urmatoarele:

Utilizarea CPU (acesta trebuie să fie cât mai ocupat posibil; într-un sistem real utilizarea lui este de la 40% până la 90%)

Throughput (nr-ul proceselor completate per unitate de timp)

Turnaround time (timpul total de execuţie al unui proces; suma timpilor de aşteptare pentru a intra în memorie, *waiting* în coada de ready, execuţie CPU şi I/O)

Waiting time (timpul de aşteptare în coada de ready)

Response time (timpul de cerere a unui răspsuns până când primul răspuns este primit).

Este mai important a se cunoaşte timpii de răspuns a cererilor (sau micşorarea variaţiilor timpilor de răspuns) decât micşorarea timpului mediu de răspuns deoarce astfel se pot face predicţionările mai corect.

6.3 Scheduling Alorithms

Trebuie ales ce proces să intre în execuţie din coada de *ready*.

6.3.1 Planificarea Primul venit, Primul servit (FCFS – first come, first served)

Este de departe cel mai simplu algoritm; se foloseşte o coada de ready de tip FIFO.

Este dezavantajoasă ca timp mediu de aşteptare deroarece unele procese pot dura foarte mult iar altele foare puţin. Explicaţia, mai greu de reprodus, este la pagina 157.

Algoritmul FCFS este nepreemtiv, odata ce a fost CPU alocat unui proces, acesta îl ţine ocupat până când se termină sau are cerere I/O.

6.3.2 Cea mai scurtă treabă, primul la execuţie (SJF – Shortest job-first)

Algortimul asociază fiecărui proces lungimea următoarei izbucniri CPU (CPU burst) şi îl alege totdeauna pe cel cu cea mai mică izbucnire. Dacă 2 procese au aceeaşi izbucnire (sau mai bine zis durată de execuţie CPU) se foloseşte algoritmul FCFS. Deoarece este dificil de ştiut durata cererii următoare de CPU este frecvent folosit la planificarea pe termen lung (long-term scheduling). Acest algoritm poate fi preemtiv sau nepreemtiv. Dacă este preemtiv, se mai numeşte şi **shortest-remaining-time-first scheduling**. (explicaţia cu caclcule este le pag 159-160).

6.3.3 Planificarea prioritară (Priority Scheduling)

Algoritmul precedent este un caz special al organizării prioritare. O prioritate este asociată fiecărui proces şi i se alocă CPU procesului cu prioritatea cea mai mare. Priorităţile pot fi definite intern (timpi limită, cereri memorie, rata medie I/O si CPU burst, etc) si extern sistemului de operare (importanţa procesului,…).

- poate fi preemtiv sau nepreemtiv

Problema mare a acestui algoritm este blocarea infinită (**indefinite blocking**). Procesele cu prioritate mică pot ajunge să nu fie servite niciodată. O soluţie la această problemă ar fi folosirea vârstei (**aging**) şi anume din timp în timp se creşte prioritatea procesului din coada de ready cu un număr de unităţi.

6.3.4 Round-Robin Scheduling (RR)

Este folosită în special la sistemele cu time sharing;

Este similară cu FCFS dar este adăugată preemţiune la schimbarea între procese

Este definită o cantitate mică de timp numită **time quantum** (sau **time slice**), fiecare cuantă având între 10 şi 100 de milisecunde. Coada *ready* este tratată ca circulară de tip FIFO iar planificatorul de CPU trece pe la fiecare proces din coadă şi îi alocă procesor un interval de o cuantă. (calcule si explicaţii sunt la pag 163-164).

6.3.5 Multilevel Queue Scheduling

Algoritmul separă caoda de *ready* în mai multe cozi în funcţie de clasele proceselor (sau proprietăţile lor) (ex: foreground şi backgroud processes care au cerinţe de timpi de răspuns diferite). În plus, trebuie făcută planificare între cozi cel mai des implementată fiind fixed-priority preemtive scheduling. Mai există posibilitatea ca fiecare coadă să primească o anumită cantitate de timp pentru execuţie iar avansarea în cozi se face după FIFO.

6.3.6 Multilevel Feedback Queue Scheduling

Diferenţa între acest algoritm şi cel precedent este că procesele pot migra dintr-o coadă în alta. Ideea este de a separa procesele cu durată de execuţie CPU diferită. Dacă un proces utilizează prea mult timp CPU, va fi mutat într-o coadă cu prioritate mai mică şi invers. Acest algoritm este definit de următorii parametrii: numărul cozilor, algoritmul de planificare al fiecărei cozi, metoda folosită pentru promovarea unui proces intr-o coadă mai prioritară şi invers, metoda folosită pentru intrarea unui nou proces într-o anumită coadă.

6.4 Planificarea multi-procesor (Multiple-Processor Scheduling)

Se presupune că procesoarele sunt identice (omogene) în funcţionalitate şi se presupune **uniform memory acces(UMA)**. Toate procesele se introduc într-o coadă de unde sunt distribuite la procesoarele disponibile (libere), acest lucu este pentru e evita **load sharing**. Trebuie asigurat să nu aleagă două procesoare acelaşi proces şi ca procesele să nu se piardă din coadă. Acest lucru se poate face prin folosirea unui porcesor ca planificator => ia naştere o structură master-slave (devine multiprocesare asimetrică).

6.5 Planificare în timp real (Real-Time Scheduling)

Există sisteme **hard real-time** şi sisteme **soft real-time**. Primele sunt solicitate la terminarea unei sarcini critice într-o cantitate de timp garantată. În general, un proces vine cu o declaraţie a timpului necesar pentru a îndeplini un I/O. Planificatorul poate admite procesul, garantând că procesul se va termina la timp, sau poate să-l respingă. Acest lucru se cunoaşte ca rezervarea resurselor. Aceste garantări presupun ca planificatorul să cunoască exact cât durează fiecare funcţie a SO ceea ce este imposibil în sistemele cu memorie virtuală.

Evaluarea **soft real-time** este mai puţin restrictivă; necesită ca procesele critice să primească prioritate peste cele mai puţin norocoase. Implementarea unei funcţionalităţi necesită atenţie la design-ul planificatorului şi la aspecte legate de SO. SO trebuie să aibe planificare prioritară şi procesele în timp real trebuie să aibă prioritatea cea mai mare iar aceasta nu trebuie să scadă în timp.

Pentru a menţine latenţa de dispatch la nivel jos, chemările sistem trebe să fie preemtibile. Acest lucru poate fi făcut prin crearea de puncte de preemţie în cadrul kernel-ului sau prin facerea a tot kernel-ului preemtibil. La ultima variantă poate aparea fenomenul de **priority inversion** atunci când un proces cu proioritate mai mare vrea să citească sau să modifice date din kernel care sunt accesate de către un proces cu prioritate mai mică. Problema poate fi rezolvată cu **priority-inheritance protocol**, în care procesele cu prioritate mai mică care acceseaza datele unuia cu prioritate mai mare moştenesc prioritatea mare până când sunt gata cu resursele.

6.6 Evaluarea algoritmilor

Criteriile pentru selectarea unui algoritm pot fi următoarele:

- maximizarea utilizării CPU sub constrângerea că timpul maxim de răspuns este de 1 secundă.
- maximizarea throughput a.î. timpul de execuţie al unui proces este în medie dir. prop. cu timpul total de execuţie

6.6.1 Modelul determinist (Deterministic Modeling)

Evaluarea analitică foloseşte un algoritm dat şi încărcarea unui sistem (workload) pentru a produce o formulă sau un număr care evaluează performanţa algorimului pentru acea încărcare. Un tip de astfel de evaluare analitică este modelul determinist. Această metodă ia o încărcătuă (workload) particular predeterminată şi defineşte performanţele fiecărui algoritm. (calcule la pag 173-174)

Este simplă şi rapidă. Produce numere exacte permiţând ca algoritmii să fie comparaţi. Dezavantajul esta că metoda are nevoie cifre exacte ca intrare şi de aceea nu prea este folositoare.

6.6.2 Modele tip coadă (Queueing Models)

Deoarece procesele care rulează pe multe sisteme variază de la o zi la alta nu se poate folosi modelul determinist. Ce poate fi determinat este distribuţia izbucnirilor (burst) CPU şi I/O. Sistemul calculatorului este descris ca o reţea de servere. Fiecare server are o coadă de procese în aşteptare. CPU este un server cu coada lui de ready precum şi sistemul I/O cu cozile dispozitivelor. Cunoscând ratele de sosire şi ratele de servire, se pot calcula utilizarea de calcul, media lungimilor cozilor, media de aşteptare, etc. Acest lucru se numeşte **queueing-network analysis**.

Ex: n= media aritmeticaă a lungimilor cozilor
W=media aritm. a timpului de aşteptare în coadă
λ=media ratei de sosire a noilor procese în coadă

Dacă sistemul este într-o stare sigură (steady state), atunci numărul proceselor care părăsesc coada trebuie să fie egal cu numărul proceselor care ajung. => $n = \lambda \times W$ (această ecuaţie este cunoscută sub numele de **Little's formula**).

Dezavantaje: matematica algoritmilor sau distribuţiilor poate fi dificilă, sosirea si servirea poate fi definită des nerealistă, trebuiesc făcute anumite presupuneri care nu sunt precise.

6.6.3 Simulări

Pentru a obţine evaluări mai precise a algoritmilor de planificare se pot folosi simulări. Se folosesc simulatoare care folosesc variabile reprezentând ceasul care sunt incrementate pe măsură ce simulatorul îşi schimbă starea. În timp ce se execută simularea, statistici care indică performanţa algoritmului sunt adunate şi printate.

Totuşi, datoarită relaţiilor între evenimentele succesive dintr-un sistem în timp real, o simulare distribution-driven poate fi imprecisă. Pentru a corecta această problemă se pot folosi **trace tapes** (fig. de la pag 176). Se poate crea un trace tape prin monitorizarea sistemului real, înregistrând secvenţa evenimentelor actuale.

6.6.4 Implementarea

Până şi simularea are acurateţe limitată. Cea mai precisă metodă de evaluare ar fi implementarea practică a algoritmului într-un SO. Dezavantajul acestei abordări este costul. Acesta nu constă doar din scrisul codului algoritmului şi modficarea SO pentru a-l suporta ci şi reacţia utilizatorilor la un SO în continuă schimbare. Mai există dezavantaj şi în faptul că odată fiecare algoritm de evaluare se schimbă şi mediul în care este folosit algoritmul. Astfel este nevoie de scrierea a noi programe şi apariţia a probleme diferite.

6.7 Process Scheduling Models

În această secţiune sunt tratate planificarea proceselor în Solaris 2, Windows 2000 şi Linux. Librăria de thread-uri organizează user-level threads ca să ruleze pe un LWP (lightwight process) disponibil, schemă cunoscută ca **process local scheduling**, în care planificarea thread-urilor se face local în aplicaţie. Invers, kernel-ul foloseşte **system global schedulin** pentru a decide care thread de kernel să fie planificat.

6.7.1 Exemlu: Solaris 2

Planificarea proceselor se face după prioritate. Are patru clase de planificare care sunt, în ordinea priorităţii: real time, system, time sharing şi interactive. Fiecare clasă include priorităţi diferite şi algoritmi de planificare diferiţi (a se vedea fig. de la pag 179).
Un proces porneşte cu un LWP şi poate crea oricâte LWP are neboie. Fiecare LWP moşteneşte clasa de planificare şi prioritatea de la procesul părinte. Default, clasa de planificare a unui proces este time sharing.

6.7.2 Exemplu: Windows 2000

Windows 2000 planifică thread-uri folosind o planificare preemtivă bazată pe priorităţi. Planificatorul lui win2000 asigură că thread-ul cu cea mai mare prioritate va fi întotdeauna executat. Porţiunea kernel-ului care se ocupă cu planificarea se numeşte *dispatcher*. Un thread selectat pentru rulare se va executa până când va fi preemtat de un thread cu prioritate mai mare, până când se termină, până când îşi termină cuanta sau până când cere I/O.

Dispatcher-ul foloseşte un plan de priorităţi pe 32 de nivele pentru a determina ordinea execuţiei thread-urilor. Priorităţile sunt împărţite în 2 clase: **variable class** (thread-uri cu priorităţi de la 1 la 15) şi **real-time class** (thread-uri cu priorităţi de la 16 la 31). Când nu este găsit nici un thread în *ready* dispatcher-ul va executa un thread special numit **idle thread**. Există o relaţie între priorităţile numerice a kernel-ului lui win2000 şi Win32 API (carte pag 180-181).

6.7.3 Exemplu: Linux

Linux are doi algoritmi de planificare a proceselor separaţi. Unul este algoritmul time-sharing (pemtru planificare preemtivă corectă între procese multiple) iar celălalt este proiectat pentru cereri în timp real unde priorităţile absolute sunt mai importante decât corectitudinea. Linux permite ca doar procesele care rulează în modul utilizator să fie preemtate. Un proces care rulează în modul kernel nu poate fi preemtat chiar dacă este disponibil un proces în timp real cu prioritate mai mare.

Prima clasă de planificare este pentru procesele time-sharing. Linux foloseşte un algoritm prioritizat, bazat pe credite (**credit-based)**. Fiecare proces procesează un anumit număr de credite planificate; când trebuie aleasă o nouă clasă pentru execuţie, procesul cu cele mai multe credite este selectat. La fiecare unitate de timp un proces in execuţie pierde un credit până ajunge la 0 şi un alt proces este selectat. Dacă nici un proces rulabil nu are credite, SO face o operaţie de recreditare fiecărui proces din sistem după următoarea regulă: credits= credits/2 + priority.

Planificarea în timp real a lui Linux implementează 2 clase: FCFS şi RR. În ambele cazuri, fiecare proces are adiţional la clasa de planificare o prioritate.

Capitolul 8 DeadLocks (Puncte Moarte)

DeadLock- seminificatie:

Intr-un mediu multiprogram, mai multe procese concureaza pt un numar finit de resurse. Daca resursele nu sunt disponibile in acel moment procesul respectiv intra in stare de asteptare. Dar s-ar putea ca acest proces sa stea mereu in stare de asteptare pt ca resursele de care are el nevoie sunt folosite de alte procese. Aceasta situatie se numeste **DeadLock**

8.1 DeadLock-uri in cadrul unui sistem

Un sistem este alcatuit dintr-un nr finit de resurse, care sunt impartite in mai multe tipuri, fiecare tip continand mai multe instante identice. Exemple de tipuri de resurse: spatiu de memorie, cicluri procesor, fisiere, dispozitive I/O(imprimante, tape drivers). Daca sistemul are 2 CPU atunci resursa de tip CPU are 2 instante, la fel resursa de tip imprimanta poate avea 5 instante.

Daca un proces solicita o instanta a unei resurse, alocarea *oricarei* dintre instante ar trebui sa satisfaca cererea. Daca nu se intampla asa atunci instantele nu sunt identice, si clasele pentru tipuri de resurse nu au fost definite corect.

Un proces trebuie sa ceara o resursa inainte de a o folosi si trebuie sa elibereze resursa respectiva dupa ce nu mai are nevoie de ea. Un proces poate face cereri pentru atatea resurse de cate are nevoie, dar in mod evident cererea sa nu poate depasi numarul total de resurse din sistem.

Intr-un mod normal de operare un proces utilizeaza o resursa in urmat secventa:

1. **Request:** daca cererea nu poate fi satisfacuta imediat, procesul intra in starea de asteptare pana cand resursa este eliberata.
2. **Use:** procesul poate folosi resursa.
3. **Release:** procesul elibereaza resursa.

Cererea si Eliberare unei resurse(Requset & Release) sunt apeluri sistem(vezi capitolul 3). Exemple: *requeste* and *release* device, *open* and *close* file, *allocate* and *free* memory-system calls. Cererea si eleiberarea altor resurse poate fi facuta prin intermediul operatiilor *wait* and *signal* cu ajutorul semafoarelor(cap anterioare). Sistemul are o tabela unde unde este memorata starea fiecarei resurse(free or allocated). Daca resursa solicitata este alocata deja atunci procesul in cauza este adaugat intr-o coada de asteptare(*queue*).

Un set de procese se afla in stare de *deadlock* cand fiecare proces din acel set asteapta dupa un eveniment ce poate fi cauzat doar de catre un proces din cadrul aceluiasi set.

Exemplu: avem un sistem cu 3 tape drivers. Fiecare unitate este manipulata de catre un proces. Daca fiecare proces face o cerere pentru alta unitate atunci cele 3 procese se afla in deadlock.

8.2 Caracterizarea DeadLock-urilor

8.2.1 Conditii necesare:

Un deadlock poate aparea daca sunt indeplinite simultan cele 4 conditii de mai jos:

1. **Excluziune Mutuala:** cel putin o resursa trebuie tinuta intr-o stare "non-sharabila", adica nu poate fi partajata. Un singur proces o poate folosi la un moment dat.
2. **Ocupare si Asteptare(Hold and Wait):** un proces trebuie sa ocupe cel putin o resursa si sa astepte dupa alte resurse care sun momentan folosite de alte procese.
3. **Fara Preemptiuni:** resursele nu pot fi eliberate numai in mod voluntar de catre procesul care le tinea ocupate, numai dupa ce acesta si-a teminat "treaba".
4. **Asteptare circulara:** trebuie sa existe un set de *n* procese {P0, P1, …, Pn} care sa astepte in urmatorul mod: P0 asteapta dupa dupa o resursa care este ocupata de P1, P1 asteapta dupa o resurasa care este ocupata de P2, …, Pn-1 asteapta dupa Pn. Pn la randul lui asteapta dupa P0.

8.2.2 Graful Alocarii Resurselor:

Este un graf prin care deadlock-urile pot fi descrise mai precis. El contine noduri, notate cu V si muchii, notate cu E. Nodurile V sunt impartite in doua categorii: P={P1, P2 ,…, Pn} **care reprezinta toate procesele active din sistem,** si R={R1, R2, …, Rn} **care reprezinta toate tipurile de resurse din sistem.**

Daca avem o muchie Pi->Rj asta inseamna ca procesul Pi solicita o resursa de tipul Rj, si acum asteapta dupa acee resursa. Rj->Pi inseamna ca o resursa de tip Rj a fost alocata unui proces Pi. Pi->Rj: **request edge(muchie de cerere),** Rj->Pi: **assignement edge(muchie de atribure)**.

Grafic fiecare proces este reprezentat printr-un cerc, iar o resursa printr-un patrat. Fiecare instanta a unei resurse este desenata ca un punct in interiorul patratului. Muchiile de cerere pointeaza doar spre punctul resursei corespunzatoare. Cand un proces Pi cere o resursa de tipul Rj este desenata in graf o muchie de cerere. Cand aceasta cerere poate fi satisfacuta muchia de cerere desenata este instantatneu transformata intr-o muchie de atribuire.

O situatie concreta este prezentata la pg 247 fig 8.1.

Daca graful de alocare a resurselor nu contine nici un ciclu atunci nici un proces din sistem nu se afla in deadlock.

Daca fiecare resursa are exact o instanta atunci un ciclu implica aparitita unui deadlock. Daca ciclul implica doar un set de resurse de un anumit tip, fiecare din ele avand doar o instanta atunci a aparut un deadlock. Fiecare proces ce face parte din acest ciclu este in deadlock. Daca fiecare tip de resursa are mai multe instante atunci nu este neaparata sigura aparitia unui deadlock. Aceste concepte sun ilustrate mai bine in fig 8.2 pg 248.

8.3 Metode de rezolvare a Deadlock-urilor

Pentru a rezolva problema putem adopta 3 strategii:

- Putem folosi un protocol pt a preveni sau evita deadlock-urile, astfel ne asiguram ca sistemul nu va ajunge niciodata in aceasta stare.
- Putem permite ca sistemul sa intre in dealock, sa il detectam si apoi sa rezolvam situatia.
- Putem sa ignoram problema, si sa presupunem ca nici un deadlock nu a avut loc in sistem. Acesta solutie este folosita de majoritatea sistemelor de operare.

In continuare ne vom ocupa pe rand de fiecare dintre aceste strtegii.

8.4 Prevenirea DeadLock-urilor

Pentru a putea prevenii aparitia acestor situatii trebuie sa ne asiguram ca cel putin una dintre conditiile prezentate in cap 8.2.1 nu este indeplinita:

8.4.1 Excluziune Mutuala

Aceasta conditie trebuie indeplinita numai pentru resurselor nepartajabile. Resursele partajabile nu necesita excluziune mutuala(exemplu fisierele read-only: ele pot fi folosite simultan de mai multe procese). In general nu putem preveni aparitia unui deadlock asigurand excluziune mutuala pt ca unele resurse sunt prin definitie nepartajabile.

8.4.2 Hold and Wait

Pentru a ne asigura ca nu este indeplinita aceasta conditie trebuie ca un proces sa nu ceara o resursa in timp ce tine ocupata o alta. In acest caz putem folosi un protocol care presupune ca un proces sa ceara si sa primesca resurse inainte sa isi inceapa executia.

Un alt protocol viabil ar fi ca sa permitem unui proces sa ceara resurse numai daca acesta nu mai foloseste si altele. Aceste protocoale au 2 mari dezavantaje: utilizare resurselor nu se face eficient, pt ca procesele tin ocupate resurse dar nu le folosesc; este posibil ca un proces sa astepte la infinit dupa o resursa ce este ocupata de alt proces.

8.4.3 Fara Preemtiuni

Pentru a fi siguri ca nu se indeplin aceasta cond folosim protocolul urmator: daca un proces retine anumite resurse si cere alte resurse care nu ii pot fi alocate in acel moment, atunci toate resursele detinute sunt preemptionate(sunt eliberate). Resursele preemptionate sunt adaugate kla lista de resurse pe care le asteapta procesul. Acesta va fi restartat numai dupa ce va putea folosi vechile resurse precum si cele noi pt care a facut cerere.

Deasemenea daca un proces cere niste resurse care nu sunt disponibile verificam daca acele resurse sunt alocate unui proces care asteapta dupa alte resurse. Daca este asa eliberam resursele de procesul vechi(le preemtionam) si le alocam procesului nou(cel care a facut cererea).

8.4.4 Asteptare Circulara

Fie R={R1, R2,…, Rn} setul de resurse al sistemului. Fiecarei resurse ii asignam un numar si obligam procesele sa ceara resurse doar in ordinea crescatoare a numerelor. De exemplu F(disk drive)=1, F(imprimanta)=5. Astfel procesul care vrea sa foloseasca diskul si imprimanta trebuie sa ceara mai intai accesul la disk pt ca numarul asignat acestuia este mai mic. Cand un proces cere o resursa ce are asignat, sa zicem numarul *i,* atunci el trebuie sa elibereze toate resursele ce au numere>=*i*.

Daca sunt indeplinite aceste doua conditii atunci **asteptarea circulara** nu va avea loc niciodata.

8.5 Evitarea DeadBlock-urilor

Daca aplicam algoritmii descrisi la 8.4 vom putea evita deadlock-urile dar pot aparea si efecte secundare cum ar fi utilizarea scazuta a resurselor sistemului.

O alternativa la aceasta metoda ar fi sa cunoastem despre cum vor fi alocate resurele. Daca cunoastem exact secventa de cereri si eliberari ale resurselor(de exemplu avem doua procese: primul are nevoie mai intai de floppy si apoi de imprimanta si al

doilea exact invers), atunci putem decide la fiecare cerere daca procesul va intra in stare de asteptare sau nu.

Cel mai simplu algoritm de evitare presupune ca fiecare proces sa decare un **numar maxim** de resurse de care are nevoie.

8.5.1 Stari Sigure

O stare a unui sistem este sigura daca sistemul poate aloca resurse fiecarui proces intr-o anumita ordine astfel incat sa evite un deadlock. Cu alte cuvinte trebuie sa existe o **secventa sigura** de procese. Orice proces Pi din secventa poate primi resursele cerute, iar un proces Pj, j<i ii va ceda resursele detinute de el. Daca nu exista o asemene secventa atunci starea nu este sigura(unsafe). Nu toate starile nesigure sunt deadlock.

8.5.2 Algoritmul grafului de alocare a resurselor

Daca avem un sistem cu cate o singura instanta din fiecare tip de resursa, atunci putem folosi o varianta a grafului de alocare a resurselor pentru a evita deadlock-urile. In plus fata de graful descris anterior mai adaugam un nou tip de muchie, **muchia de cerere posibila(**Pi->Rj**),** care indica ca procesul Pi s-ar putea sa aiba nevoie pe viitor de resursa Rj. Se reprezinta printr-o linie punctata.

Cand cand aceasta cerere chiar are loc atunci muchia este covertita in muchie de cerere, iar cand resursa este eliberata este convertita inapoi in muchie de cerere posibila. Daca nu exista nici un ciclu atunci sistemul se afla intr-o stare sigura.

8.5.3 Algoritmul bancherului(sau algoritmul bancar)

Acest algoritm se poate aplica si pt mai multe instante aleunei resurse dar nu este la fel de eficient ca cel prezentat mai sus. Orice proces trebuie sa declare mai intai numarul maxim de instante de resurse de care s-ar pute sa aiba nevoie.

Consideram *n* numarul de procese din sistem, si *m* numarul de tipuri de resurse. Pentru a implementa acest algoritm trebuie sa declaram mai multe structuri de date:

- **Available**: un vector de dimensiune=*m* care indica numarul de instante disponibile dintr-o anumita resursa.
- **Max**: o matrice $n \times m$ care indica cererea maxima de instante ale unei resurse facuta de un proces. Daca Max[i,j]=k, procesul Pi va cere cel mult k instante ale resursei Rj.
- **Allocation**: o matrice $n \times m$ care indica numarul de instante de fiecare tip alocate in prezent unui proces. Daca Allocation[i,j]=k, prcesul Pi are in prezen alocate k instanet ale resursei Rj.
- **Need**: o matrice $n \times m$ care indica numarul de resurse de care mai are nevoie un proces. Daca Need[i,j]=k, procesul Pi mai are nevoie de inca k instante din resursa Rj pt a-si termina misiunea.

Notatii: X is Y vectori de lungime *n.* Spunem ca $X \leq Y$ daca $X[i] \leq Y[i]$, unde i=1,2,…,n. Facem aceasta notatie pt a putea trata fiecare linie din matricile Allocation si Need ca pe niste vectori corespunzatori unui proces. Allocation(i) indica resursele alocate procesului Pi.

8.5.3.1 Algoritmul de siguranta- algoritm care ne spune daca sistemul este sau nu in stare sigura

1. Fie Work[m] si Finish[n] doi vectori. Se initializeaza Work:=Available, Finish[i]=false i=1,n

2. Se cauta un i astfel incat:
    a. Finish[i]=false
    b. Need(i) ≤Work
3. Work:= Work + Allocation(i)
   Finish[i]:=true
   Go to step 2
4. If Finish[i]=true pentru toti i atunci sistemul e in stare sigura.

Acest algoritm are nevoi de m × n^2 operatii.

8.5.3.2 Algoritmul cererii de resurse

Fie Request[i] vectorul de cereri pt procesul Pi. Daca Request[j]=k, atunci procesul Pi cere k instante ale resursei Rj. Cand Pi face o cerere de resurse se executa urmatorii pasi:

1. If Request(i) ≤Need(i), go to step 2 Else se activeaza un event de eroare pt ca procesul a depasi numarul maxim de resurse de care a spus ca are nevoie.
2. If Request(i) ≤Available, go to step 3 Else pi trebuie sa astepte pt ca resursele nu sunt disponibile.
3. Available:= Available – Request;
   Allocation:=Allocation + request;
   Need(i):=Need(i) – Request(i);

Daca se intra intr-o stare sigura atunci Pi primeste resursele, daca nu Pi trebuie sa astepte dupa Request(i) si este restaurata vechea stare.

8.5.3.3 Un exemplu ilustrativ pg 259

8.6 Detectia DeadLock-urilor

Avem nevoie de:

- Un algoritm care sa examineze starea unui sistem pt a detemina daca a aparut un deadlock
- Un algoritm care sa revina din deadlock(recover)

8.6.1 Cate o singura instanta din fiecare resursa

Daca fiecare resursa are exact o instanta atunci putem defini un algoritm de detectie care foloseste o varianta a grafului de alocare a resurselor numita **graf de asteptare(wait-for graph)**. Obtinem acest graf din grful de alocare din care nodurile pentru resurse si colapsam muchiile corespunzatoare. Daca in graf gasim un ciclu atunci a aparut un deadlock. Exemplu fig 8.7 pg 261.

8.6.2 Mai multe instante pentru fiecare resursa

Se folosesc aceleasi structuri de date prezentate la algoritmul bancherului(8.5.3).

1. Fie Work[m] si Finish[n] doi vectori. Se initializeaza Work:=Available
   For i=1,2,…,n
   if Allocation(i)≠0
   Finish[i]=false
   else
   Finish[i]=true
2. Se cauta un i astfel incat:
    a. Finish[i]=false
    b. Request(i) ≤Work

Daca nu exista un astfel de i go to step 4

3. Work:= Work + Allocation(i)
   Finish[i]:=true
   Go to step 2
4. If Finish[i]=false pentru anumiti i atunci sistemul e in stare deadlock. Mai exact daca Finish[i]=false atunci Pi este deadlock.

Acest algoritm are nevoi de m × n^2 operatii.

8.6.3 Eficienta algoritmului de detectie

Apelul acestui algoritm se face in functie de 2 factori:

1. Cat de des a aparut un deadlock
2. Cate procese vor fi afectate de deadlock in momentul cand acesta va aparea

Intr-un caz ideal am putea apela acest algoritm la fiecare cerere de resursa, dar ar rezulta o mare pierdere de putere de calcul. O solutie mai ieftina ar fi sa il apelam la anumite intervale de timp.

8.7 Rvevenirea din starea de DeadLock(Recovery)

Cand s-a detectat un DeadLock avem mai multe posibilitati:

- Informam operatorul, deci va intra in sarcina acestuia sa rezolve prblema manual
- Sistemul isi revine automat(automatically recover)

Sunt 2 solutii de a revenii dintr-un Deadlock:

1. Renuntam la unul sau mai multe procese pt a intrerupe asteptarea circulara.
2. Preemtionam resurse de la procesele care cauzeaza deadlockuri(le luam resursele)

8.7.1 Terminarea proceselor

Metode de abandonare a proceslor:

- **Abandonarea tuturor proceselor ce cauzeaza deadlock**: are dezaavantajul ca intrerupand multe procese pierdem rezulatetel pe care acestea trebuiau sa le returneze, deci irosim timp.
- **Abandonarea tot a cate unui proces pe ran pana cand dealock-ul este eliminat**: de fiecare data trebuie apelat si un algoritm de detectie pt a vedea daca mai avem deadlock

Factori pe care ii luam in considerare cand alegem procesul pe care il intrerupem:

1. Ce prioritate are procesul
2. De cat timp este pornit acel proces si cat timp mai are pana se termina
3. Cat de multe resurse a folosit acel proces si ce fel de resurse(simple sau preemtionate)
4. De cate resurse mai are nevoie procesul pentru a termina
5. Procesul mai interactioneaza sau nu cu altele

8.7.2 Preemtionarea Resurselor

Prin preemtionarea resurselor unui proces, ii luam acestuia resursele si le atribuim unui alt proces pana cand ciclul deadlock este intrerupt. Trebui sa rezolvam urmatoarele probleme:

- **Selectionare unei victime**: trebuie sa avem in vedere un cost minim(cat mai putine resurse ocupate de acel proces si un timp de cand se afla in executie cat mai mic)

- **Rollbak(refacerea contextului)**: dupa ce am preemtionat resursele de la un process, la revenirea acestuia trebuie sa refacem stare in care se afla. Cea mai simpla solutie este sa il abandonam de tot si pe urma sa il restartam, dar nu este asa de eficient.
- **Infometare(Starvation)**: orice proces nu trebuie ales ca victima de prea multe ori pentru a se evita "infometarea" lui.

CAP 9

Managementul memoriei

Metodele de management al memoriei nu tin numai de sistemul de operare si de partea hardware. Multi din algoritmii de management al memoriei necesita support hardware.

9.1.1Legatura dintre adrese

In mod normal un program se afla pe disc sub forma unui fisier executabil. Programul trebuie adus in memorie pus intr-un process si apoi executat. In functie de managementul memoriei procesul se poate fi mutat de pe disc in memorie in timpul executiei lui. Colectia de procese de pe disc care asteapta sa fie aduse in memorie pentru a fi executate formeaza coada de intrare.
Datorita faptului ca spatiul un program se poate afla (de obicei) in orice zona a memoriei fizice este nevoie de o mapare a adreselor facandu-se o legatura intre doua spatii de adrese (adresele logice sunt transformate in adrese fizice).

Legatura dintre instructiuni (si date) si adresa de memorie se poate face astfel:

1. *In timpul compilarii*: se genereaza codul absolut (adresele din program coincide cu cele fizice). Daca programul este mutat de la locatia initiala el trebuie sa fie recompilat.
2. *In timpul incarcarii (load time):* Compiler-ul genereaza un cod care isi poate muta pozitia (relocatable code). Daca adresa de start se schimba vom reincarca codul adugand noua valoarea.
3. *In timpul executiei:* Legatura dintre adrese se face in timpul executiei.

9.1.2Spatiul de adrese logice vs spatial de adrese fizice

CPU genereaza de obicei adrese logice. Schimarea adreselor logice in adrese fizice se face cu ajutorul MMU (memory management unit).

O schema simpla de mapare a adreselor se face cu ajutorul unui registru de relocare. In acest registru se memoreaza o valoare (ex 14000) care se adauga la fiecare la fiecare adresa logica (ex 14000 +456 unde 456 adresa logica).

9.1.3Incarcarea dinamica

In schema de mai sus ca un process sa fie executat el trebuie sa se afle in memoria fizica. Deci procesele vor fi limitate de dimensiunea fizica. Pentru a obtine o mai buna utilizare a spatiului de memorie se poate folosi incarcarea dinamica (dynamic loading). In acest caz un subprogram (rutina) nu este incarcata in memorie pana cand nu este chemata. Aceasta metoda se foloseste atunci cand avem programe cu cod mult.

9.1.4Legaturi dinamice si Impartirea librariilor

Pentru economisirea spatiului se folosesc legaturi catre librarii. In cod se face doar legatura catre o librarie. Aceasta metoda permite modificarea doar a librariilor (repararea de bug-uri) fara ca legaturile sa se modifice (se pot face update-uri la program folosind aceasta metoda).
Impartirea librariilor(shared libraries) se refera la modificarile mari aduse la o librarie care ar putea afecta executia programelor. In acest caz se retine pe langa legatura si versiunea librariei. Programele care au fost compilate cu o versiune mai veche vor continua sa ruleze pe versiunea veche fara ca programul sa sufere modificari.

9.1.5Extinderi

Pentru a permite unui proces sa fie mai mare decat spatiaul de memorie alocat lui se foloseste extinderea (overlay). Ideea este de a tine in memorie doar acele instructiuni si date care sunt necesare in orice moment. Cand o alta instructiune trebuie incarcata ea este pusa peste spatiul ocupat de o instructiune de care nu mai este nevoie.

9.2 Transferul de memorie(swapping)

Un process poate fi scos afara din memorie intr-un spatiu de stocare (backing store) si adus inapoi in memorie pentru continuarea executiei. Acest transfer se foloseste la algoritmul round-robin si algoritmi bazati pe prioritate (eu cred ca la toti algoritmii preemtivi-precauti). In mod normal un process care a fost scos din memorie va fi pus in acelasi loc de unde a fost scos datorita legaturii dintre adrese. Aceasta restrictie este inlaturata daca se foloseste maparea a adreselor in timpul executiei. Spatiul de stocare (backing store) este de obicei un disc foarte rapid care are un spatiu suficent pentru a retine mai multe procese.
Solutia transferului de memorie (swap) este folosita in putine sisteme datorita timpului mare dintre transferuri si timpului mic de executie. (dureaza mult transferal din memorie in backing store si invers).

9.3 Alocarea contigua a memoriei

Memoria este de obicei impartita in doua zone: una alocata sistemului de operare si una alocata aplicatilor user. Prin alocarea contigua fiecare process contine o singura sectiune contigua de memorie.

9.3.1Protectia memoriei

Diferentiera dintre o zona de memorie a sistemului de operare si o zona de memorie dedicata user-ului se poate face cu ajutorul unui registru de relocare si un registru de limita.Registrul de relocare contine adresa fizica cea mai mica iar registrul limita contine dimensiunea maxima a adresei logice.

9.3.2 Alocarea memoriei

Cea mai simpla metoda de alocare a memoriei este de a divide memoria in partitii de dimensiune fixa. Fiecare partitie contine exact un proces. Atunci cand o partitie este goala un process este selectat din din coada de intrare si incarcat in acea partitie. Sistemul de operare retine intr-o tabela care parte a memoriei e ocupata si care nu.
Initial toata memoria este libera pentru procesele user. Cand un process este incarcat si are nevoie de memorie se cauta un spatiu suficient de mare pentru process. I se aloca doar atat cat este nevoie restul memoriei fiind disponibila pentru pentru celelalte procese. La terminarea procesului spatiul ocupat de process poate fi reutilizat.
Sistemul de operare aloca memorie proceselor pan ace nu se mai gaseste spatiu pentru process. In acest caz sistemul de operare poate sa astepte pana se gaseste sufficient spatiu pentru process sau sa aloce spatiu pentru un process care are nevoie de mai putina memorie.
In cazul in care sunt mai multe locuri in care se poate pune procesul in memorie se pot folosi trei strategii:

a) First fit: se pune in primul spatiu gasit
b) Best fit: se pune in spatiul cel mai mic (suficent pentru process)
c) Worst fit: se pune in spatiul cel mai mare

Acesti algoritmi sufera insa de o fragmentare externa datorata incarcarii si stergerii din memorie a proceselor. O fragmentare externa exista atunci cand spatiu liber din memorie este sufficient pentru incarcarea unui process dar zona nu este contigua.

9.3.3Fragmentarea

Fragmentarea poate fi interna sau externa. Fragmentarea interna este diferenta dintre spatiul ocupat de o partitie si spatiul ocupat de procesul care trebuieadus in memorie. Fragmentarea externa este spatiul liber total dar necontigu. Reducerea fragmentarii externe se face prin compactare. Compactarea se poate face doar daca alocarea este dinamica. Scopul compactarii este de a pune toate locurile libere intr-un singur bloc mare.

9.4 Paginarea

Permite un spatiu de adrese logice necontinuue. Prin paginare memoria fizica e impartita in blocuri de dimensiune fixa numite frame (dimensiune fiind o putere a lui 2) iar memoria logica e impartita in blocuri de aceeasi dimensiune numite pagini. Partea hardware care ajuta la paginare e formata dintr-o tabela de paginare (care care retine adresa de baza a fiecarui frame din memoria fizica). CPU-ul emite o adresa formata din doua parti : numarul paginii si offsetul pagini. Cu ajutorul numarului paginii se ia din tabela adresa fizica de baza care impreuna cu offsetul formeaza adresa fizica. Lungimea frame-lor este definita prin hard. Pentru a rula un program cu dimensiunea de n pagini vom avea neovie de n frame libere. Informatia despre numarul total de frame libere , framele libere, framele ocupate se gasesc in tabela de frame.

9.4.2Implementarea hardware

Implementarea hard a tabelei de paginare poate fi facuta cu ajutorul unui set de registre dedicate. Aceste registre lucreaza la o viteza mare pentru a putea face translatia cat mai completa. Folosirea registrelor este rezonabila daca tabela de paginare are o dimensiune mica(pana la 256 de intrari). Pentru masinile cu tabela de paginare foarte mare , tabela se tine in memoria principala si se foloseste un registru numit PTBR(page table base register) care pointeaza la aceasta zona de memorie. Aceasta metoda este si ea dezavantajoasa deoarece se face doua accese la memorie.

Solutia standard pentru implementarea tabelei de paginare este folosirea TLB-ului (translation look aside buffer). TLB-ul este o memorie asociativa foarte rapida. Fiecare intrare in TLB contine o cheie si o valoare. Cand se acceseaza TLB-ul se compara cheia cautata cu cheile din TLB (principiul memoriilor asociative). Daca se gaseste atunci se returneaza valoarea de la acea cheie(numarul de intrari este de obicei 64-1024).

Daca numarul paginii nu se afla in TLB (TLB miss) atunci trbuie sa se faca o referinta la memorie la tabela de paginare. La miss in TLB se adauga cheia si valoarea pentru ca data viitoare san u mai fie miss. Daca TLB-ul este plin atunci se foloseste algoritmul LRU pentru eliminare. Unele informatii din TLB nu se pot elimina fiind folosite de kernel. Aceste intrari se numes wired down.

Unele TLB-uri stocheaza un identificator al spatialului de adrese (ASID) in fiecare intrare a tabelei. Cand TLB-ul este folosit atunci pe langa verificarea cheii se verifica si ASID-ul procesului current sa fie egal cu cel din TLB altfel miss in TLB.

Timpul de acces la memorie este : hit rate TLB*timp TLB+miss rate TLB*timp acces memorie.

9.4.3Protectia

Protectia se face prin asocierea unui bit fiecarui frame(poate fi read-write sau read only). Inca un bit se adauga la intrarea in TLB care indica daca informatia din TLB este valida sau nu. O informatie din TLB este valida daca adresa logica pentru care se cere maparea face parte din spatiul de memorie a procesului.

9.4.4Structura tabelei de paginare

9.4.4.1Paginarea ierarhizata

Spatiul de adrese logice este impartim in mai multe tabele de paginare. Un exemplu simplu este un algoritm de paginare pe doua nivele. La aceasta metoda adresa logica (formata din numarul paginii si offset) numarul paginii este impartit in doua parti (ex daca este o adresa pe 32 biti 20 biti pt nr paginii si 12 pt offset. Cei 20 biti se impart in 10b si 10b). Prima parte pointeaza spre o tabela de paginare externa (outer page table) iar a doua parte este deplasamentul paginei din interiorul tabelei de paginare.(se foloseste la Pentium 2). Metoda de translatie a adresei:

9.4.4.2 Hashed Page Table(Tabele de dispersie)

Fiecare intrare in tabela de dispersie contine o legatura catre o lista de elemente care contine adrese din acelasi spatiu de adresare. Algortimul functionaeaza astfel: din adresa logica se ia doar numarul paginii *q* (nu si offsetul) si se trece printr-o functie de dispersie. Valoarea obtinuta este indexul din tabela de dispersie. Fiecare intrare din tabela de dispersie avand atasata o lista se face o cautare in acea lista dupa valoarea paginii *q*. Fiecare element din lista contine: valoarea paginii(adresa logica), valoarea frame-ului(adresa fizica), adresa next element.

9.4.4.3Inverted page table

O tabla de paginare inversata are proprietatea ca fiecare pagina sau frame are o singura intrare. In tabela de paginare inversata se face indexarea cu ajutorul PID (process ID). CPU-ul in acest caz returneaza o adresa logica de forma <PID, nr pagina,offset>.

Pentru a micsora timpul de cautare se poate folosi si o tabela de dispersie.

9.4.4.5Pagini partajate (shared page)

Se pot folosi in cazul in care procesele nu au un cod care se schimba in timpul executiei. In memeoria fizica sunt retinute o copie a codului program si pentru fiecare user datele introduse. Avantajul consta in faptul ca nu se ocupa spatiu in memorie (nu se copiaza codul programului pt fiecare user in parte).

9.5 Segmentarea

Un spatiu de adrese logice este impartit in mai multe segmente. Fiecare segment are un nume si o lungime(segment de date segment de cod..).

9.5.2Hardware

Segmentarea se face cu ajutorul unei tabele de segmentare. Adresa logica (emisa de CPU) contine un index si un offset. Indexul pointeaza la tabela de segmente in ea aflandu-se un tuplu de forma <limita, base>.

9.5.3Partajarea segmentelor

Segmentarea ajuta foarte mult in partajare. Cu ajutorul segmentelor se pot partaja doar datele unui process nu si codul acestuia.

9.5.4Fragmentarea

Are loc o fragmentare externa ca la paginare.

9.6 Paginarea si segmentarea

Se folosesc ambele idei. Andresa logica este impartita in: page number si page offset. Cu page number se pointeaza la o tabela de segmentare care ne dah base page –ul pt tabela de paginare. Din tabela de paginare se ia adresa fizica.

Capitol 10 MEMORIA VIRTUALA

Mem virtuala este o tehnica care permite executia proceselor care nu pot fi completate in memorie.

10.1 Background

Mem virtuala este separatia dintre mem logica de mem fizica. Separarea permite ca mem vrtuala multa sa fi disponiblila la numai o mem fizica mica.

Avantaje:

- doar o parte a programului terbuie sa se afle in mem cand se executa.
- spatiul logic de adrese poate fi mai mare decat spatiul fizic de adrese
- spatiul de adrese poate fi impartit de mai multe procese(prin page sharing)
- Permite o creeare de procese mult mai eficeinta

Mem virtuala poate fi implementata prin: demand paging sau demand segmentation.

10.2 Demand paging

Este similar cu un sistem de pagini cu schimbare. Procesele se afla in mem secundara si sunt aduse in mem doar cand vor fi executate.

10.2.1 Concepte de baza

poza pg 320

Cand un proces trebuie schimbat, paginatorul ghiceste care pagini vor fi folosite pana cand procesul va fi schimbat iara. Paginatorul aduce doar panginil necesare in mem => Mai putine I/O, mai putina mem folosita, raspuns mai rapid, mai multi useri.

Avem nevoie de o forma de ajutor hardware sa stim daca pagina este in mem sau pe disc -> bit valid/invalid. Daca bitul este setat pe valid -> pagina este valida si inmem. bit = invalid -> pagina este fie invalida sau se afla pe disc. (poza pag 321). Initial bitul valid/invalid va fi pus pe 0 pentru toate paginile. Daca procesul nu va accesa pagina nu conteaza daca pagina este valida sau nu. Daca ghicim paginile corecte procesul va rula ca si cum am afi adus toate paginile.

Daca procesul acceseaza o pagina cere nu se afla in memorie(accesul la o pagina marcata ca invalid) => page fault.

Tratarea unui page fault:

- cautam in tabela intern a procesului sa vedem dac este o referinta valida la memorie sau nu
- daca referinta esre invalida terminam procesul, daca este valida si pagina nu a fost adusa inca se aduce pagina
- gasim un frame gol
- executam o operatie de disc de citire a paginii in frame-ul nou alocat
- dupa citire modificam tabela interna a procesului indicand ca pagina este in mem
- restartam instr care a fost intrerupta din cauza page fault

poza pag 322

Paginare la cerere = se aduce o pagina in mem doar daca a fost accesata.

O instr trebuie restartata dupa fiecare page fault. Putem rezulva acest lucru prin aducerea inca odata i instructiunii. Dificultatea este cand o instr modifica mai multe locatii diferite.

Solutii:

- se incearca detectarea unui paga fault inainte ca sa se scrie in locatii, inainte ca ele sa fie midificate.
- folosirae unor registrii temporari cara pastraeza valorile locatiilor scrise. La un page fault toate valorile vechi sunt scrise in mem.

10.2.2 performata paginarii la cerere
Rata de page fault este: $0 \leq p \leq 1.0$

- daca p = 0 nu avem page fault
- daca p = 1, fiecare accesare este un page fault

timpul efectiv de acces(EAT):
EAT = (1 – *p*) x memory access + *p* (page fault overhead + [swap page out] + swap page in + restart overhead)
Rata de page fault trebuie pastrata cat mai mica in paginarea la cerere.

10.3 Creearea proceselor
10.3.1 Copy on write
Aceasta tehnica prminte ca Procesul parinte si procesul fiu sa imparta initial aceleasi pagini. Aceste pagini sunt denumite pagini copy on write. Daca unul din procese scrie intr-o pagina partajata, doar atunci se face o copie a paginii partajate. Tehnica permite o creare eficienta a proceselor deoarece doar pagina modificata este copiata. Paginile copiate trebuie puse in unele libere. Trebuie determinat de unde luam acestese pagini. multe SO au un pool(grupa) de pagini libere pentru asemenea cereri.

10.3.2 Fisiere mapate in Mem
I/O de fiseere mapate in mem permite ca I/O de fisiere sa fie tratat ca un acces de mem prin maparea unui bloc de disc intr-o pagina din mem. Un fisier este citit intial folosind paginarea la cerere. O portiune de marimiea unui pagini din fisier este apoi citita din sistemul de fisiere intr-o pagina fizica. Citiri/scrieri secventiala la/dinfisier sunt tratate ca accese obisnuite la mem. Simplifica accesul la fisiere prin tratarea fisierelor ca I/O prin memorie decat prin apelurile de sist read() write(). Deasemenea permite ca mai multe procese sa mapeze acelasi fisier prin partajarea paginilor in mem.poza pag 331

10.4 Inlocuirea paginilor
Previne supraincarcarea memoriei prin modificarea rutinei de tratare page fault sa includa si inlocuirea panginilor. Supraincarcarea memoriei apare cand toata mem este folosita(ne e nici un frame liber pentru a pune noua pagina).

10.4.1 Schema de baza
Daca nici un frame nu este liber gasim unul care nu este folosit si il eliberam. Continutul frame-ului il scriem in spatiul de schimb(swap) si schimbam tabela de pagini indicand ca pagina nu mai este in mem. Frame-ul care va fi scos dim memoria are numele de frame victima. Cand nu este nici un frame liber -> doua transferuri de pagina. Acest lucru poate fi prevenit prin folosire unui bit ditry. Bitul este setat cand se scrie in pagina. Doar pagini care au fost modificate sunt scrise pe disc.
Algoritmii de inlocuire trebuie sa aiba cea mai mica rate de page fault. vezi ex pag 335

10.4.2 Algoritmul de inlocuire FIFO
Cel mai simplu algoritm de inlocuire. Un algoritm fifo asociaza fiecare pagina cu timpul cand a fost adusa in mem. Cand o pagina trebuie inlocuita se alege cea care este de cel mai mult timp in mem. Se poate implement acu ajutorul unei cozi fifo. ex pag 336
Anomalia lui Belady: pt anumiti algoritmi se pot obtine rez mai slabe cu cresterea fremeurilor.

10.4.3 Inlocuire optimala a paginilor
Un algoriym optimal are cea mai mica rata de page faul-uri si nu sufera de anomalia lui Belady. El este denumit OPT sau MIN:

- inlocuieste pagina care nu va fi folosit pt cea mai mare perioada de timp.

Dificil de implementat, deoarece trebuie sa stim ce pagini vor fi referite in viitor.
Ex pag 338

10.4.4 Algoritmul LRU
Pagina care va fi inlocuita este cea care nu a fost folosita de cel mai mult timp. LRU asociaza fiecarei pagina timpul cand a fost utilizata.
Implementare:

- cu numaratoare: fiecare intrare din tabele de pagini are asociat un camp timpul cand a fost folosita pagina. cand se acceseaza pagina acest contor primeste valoarea ceasului logic al CPU. Aceasta schema implica insa cautarea paginii LRU in tabele.
- cu stiva: cand o pagina este referita ea este pusa in varful stivei. Astfel in vf stivei este pagina accesata de curand iar in coada este pagina LRU. Aceasta implementare nu necesita cautari dar mutarea paginii din mijlocul stivei in varf este consumatoare de timp.

10.4.5 Algoritm LRU aproximativ
Introducerea unui bit de accesare(reference bit). Bitul paginii este initial pus pe 0. Cand pagina eset accesata el devine 1. Inlocuirea se face cu pagina care are bitul 0. Nu stim insa ordinea in care au fost accesate paginile.

10.4.5.1 Biti aditionali de accesare
In locul unui bit avem 8 biti. La intervale regulate acesti cate un bit este inregistrat si ceilalti shiftati. ex 11000100 a fost accesat mai recent ca 011110111.

10.4.5.2 Second_chance algoritm
Daca pagina care tebuie inlocuita are bitul de acces 1 atunci el este pus pe 0, se lasa pagina in mem si alta este inlocuita.

10.4.5.3 Algoritm imbunatatit second-chance
se folosesc bit dirty si modificat

- (0, 0) – nu a fost accesata nici modificate recent – pagina cea mai buna de inlocuit
- (0, 1) – nu a fost folsita de curand dar este modificata – nu e f bine la inlocuit deoarece trebuie scrisa inapoi
- (1, 0) – folosita dar nemodificata – poate fi folosita incurand
- (1, 1) – recenta si modificata

10.4.6 Algoritmi de inlocuire bazati pe numarare
Tin o evidenta de cate ori au fost accesate paginile:
- LFU: inlocuieste pagina care a fost accesata de cele mai putine ori
- MFU: se bazeaza pe argumentul ca paagina LFU abia a fost adusa si trebuie sa fie folosita

10.4.7 Algoritmi cu Page-buffering
Cand pagina este adusa din mem ea este introdusa intr-un buffer pana cand pagina va fi evacuata din mem. In asa fel nu mai trebuie sa asteptam cu executia pana aceasa va fi evacuata.

10.5 Frame-uri de alocare
Fiecare proces are nevoie de unmainar min de pagini.

10.5.2 Algoritmi de alocare
Alocare fixa:
- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- alocare proportionala – alocarea are loc in functie de marimea procesului

Alocare prioritara:
- se foloseste o schema de alocare care foloseste prioritati si nu marime
- Daca un price Pi genereaza un page fault: se selecteaza una din framele lui sa se inlocuiasca sau una de la un proce cu o prioritate mai mica

10.5.3 Alocare globala VS alocare locala
- Inlocuire locala: fiecare proces selecteaza doar din setul de frame-uri alocate lui
- Inlocuire globala: Procesul selecteaza un frame de inlocuit din setul total de frame-uri; un proces poate lua de la celelalt

10.6 Thrashing
Thrashing este fenomen cand un proces este ocupat cu schimbarea paginilor.

10.6.1 Cauza
Daca un proces nu are destule pagini, rata de page fault este mare. Aceasta conduce la Utilizare slaba a CPU, SO crede ca trebuie sa creasac nivelul de mutiprogramare, alt proces este adaugat sistemului. Fig pag 349

Ca sa marim utilizarea CPU si sa inlaturam thrashing trebuie sa micsoram gradul de multiprogramare. Putem limita efectele Thrashingului prin folosirea unui algoritm de inlocuire local. Cand un proces incepe thrashing nu poate luaa frame-uri de la un alt proces si sa il faca si pe el sa thrashing.

10.6.2 modelul Set de lucru(working set)
Modelul este bazat pe presupunerea de localitate. Folosete parametrul Δ pentru a defini fereastra setului de lucru. Setul de pagini din pagina Δ cea mai recenta se numeste set de lucru. Daca pagina se folosesta activ se afla in setul de lucru. Acuraterea setului de lucru depinde ce Δ. Daca e prea mic nu acopera toata localitatea si daca e pre mare poate acoperi mai multe loaclitati.
SO monitorizeaza setul de lucru al fiecarui porces si aloca setului un numar suficient de frame-uri. Daca mai sunt frameuri libere poate porni alt proces. Daca nr de seturi de lucru se mareste SO poate decide sa suspende un proces. Aceasta strategie previna Thrashingul si mentine multiprogramarea la cel mai ridicat nivel. Dificultate: Pastrarea evidentei asupra setului de lucru. Setul de lucru este o fereasrtra care se misca
.
10.8 Alte considerente
10.8.1 Prepaging
Aducerea in mem la un moment toate paginile care vor fi folosite

10.8.2 Marimea paginii
O marime mica creste numarul paginilor. Deoarece in tabela de pagini fiecare pagina are o copie se doreste un nr mic de pagini -> marv mare a paginii.
Memoria este mai bine utilizata daca folosim pagini mici. Avem rezolutie mai buna, permitandu-ne sa izolam doar portiunea de memorie de care avem nevoie. Cu o pagine mica numarul tota de I/O este redus din cauza ca locatlitatea este imbunatatita.

10.8.3 TLB reach
TLB Reach = se refera la cantitatea de mem ce poate fi accesata din TLB.
TLB Reach = (TLB Size) X (Page Size)
In cazul ideal setul de lucru al fiecarui proces este tinut in TLB. Altfel numarul de page fault va fi mare.
Cresterea marimii TLB-ului:

- Un numar de pagini mai mari: Poate duce insa la cresterea fregmentarii din cauza ca nu toate aplicatiile au nevoie de pagini mari
- Asigurarea marimilor multiple ale paginii: Permite aplicatiilor care au nevoie de pagini mai mare sa le foloseasca fara sa mareasac fragmentarea.

10.8.5 Structura programului

- **int A[][] = new int[1024][1024];**
- Each row is stored in one page

- Program 1
```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```
1024 x 1024 page faults

- Program 2
```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

1024 page faults

10.8.6 I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory.

- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Cap 12

12.1 Avantaje discuri: - pot fi rescrise
- se poate accesa direct orice bloc.

-transferul DISC-Mem : se transfera blocuri (unul sau mai multe sectoare). Marimea blocurilor variaza intre 32B-4096B, usual 512B.

Sistemul de fisere ofera un acces convenabil la disc. Implementariea unui system de fisere are la baza 2 concepte:

-definirea fisierului in raport cu utilizatorul adica definirea atributelor fisierului si a operatiilor premise precum si directorul din care va face parte.

- crearea unui algoritm care face legatura intre logica sist. de fisiere si nivelul fizic al discului.

Sist de fisiere este compul dinmai multe nivele. Fiecare nivel se foloseste de serviciile nivelele inferioare si adauga noi servicii care vor fi folosite de nivelele superioare.

Aplicatie Program
|
Logica sist de fisere
|
Modul ogranizare fisiere
|
Sist fisiere de baza
|
I/0 Control
|
Dispozitiv (device) nivel fizic

I/O Control:Este format din drivere pt dispozitive si handlerului de intrerupere. Un dispozitiv poate fi privit ca un translator el primeste mesaje de tipul "citeste bloc 113" si transforma aceste mesaje in altele specifice hardware. Un dispozitiv scrie diferite biti specifici in controlerul de I/O pt a allege o anumita locatie si o anumita actiuni care este ceruta.
Sistemul de fisiere de baza: trimtie comenzi de scriere/citire a blocurilor fizice de pe disc catre dispozitivele corespunzatoare . (mode de adresare a unui bloc: disc,cilindru, pista sector).
Modul de ogranizare fisiere: face traslatarea din adresa logica a blocului de fisier in adresa fizica . Fiecare bloc de fisier logic primeste un numar de la 1 la N acesta nu este obligatoriu identic la nivel fizic. Tot aici se afla si maangerul pentru spatial liber care tine evidenta blocurilor nealocate.
Logica sist de fisiere: se ocupa cu informatiile "metadata". Acestea contin structura fisierului mai putin data stocata in el . Acest nivel se ocupa de structura directoarelor si ofera nivelului inferior toate informatiile de care are nevoie pentru un anumit fisier. El mentine structura fisierelor cu ajutorul file control blocks (FCB).FCB contine informatii despre fisier(proprietar, permisii, locatie). Tot aici este implementat si sistemul de protectie si securitate.

12.2 Implemenatare sistem de fisere

On disk:
Bloc de boot control: are informatii necesare pentru bootare unui sistem operare (doar pe partitiile care contin un SO).
Blocul de control al partitiei: detalii cu privire la partitie : nr de blocuri, dimensiune, dimensiune bloc, blocuri libere, FCB, FCB libere, etc.
O structura de directoare care organizeaza fisierele
FCB descries mai sus. (figura 12.2 pag 415)

In memory: este folosita pentru managementul unui sist de fisiere si pentru imbunatatirea performantelor folosind cacheing.
Tabela de partitie: informatii cu privire la fiecare partitie incarcata
Structura de directoare: retine informatii despre directoarele accesate recent.
Tabela de fisiere deschise la nivelul sist global (TFG): contine FCB pt fiecare fisier deschis si alte informatii.
Tabela de fisiere deschise la nivel de process (TFP): contine un pointer catre randul corespunzator din tabela de fisere globala si alte informatii.

Creaare fisier: Se creaza un nou FCB, se citeste directorul corespunzator se fac modificarile si apoi se scrie inapoi pe disc.
Deschidere fisier: Mai intai se verifica daca fiserul nu este deja deschis cautandu-se in tabela TFG, daca este deja deschis atunci se creaza o noua intrare in TFP care va pointa la randul corespunzator din TFG. Daca fisierul nu este deschis atunci atunci se cauta fisierul in structura de directoare(parti din structura de directoare sunt stocate in memorie), dupa ce se gaseste fisierul FCB-ul lui este copiat in TFG si se aloca un rand in TFP cu un pointer catre TFG.Toate operatiile urmatoare vor fi realizate prin intermediul acestui pointer.. (figura 12.3 pag 417)

12.2.2 Patititi si Mounting

Un disc poate fi impartiti in mai multe partitii sau o partitie poate contine mai multe discuri.

Partitie “raw”- nu contine un system de fisiere.

“cooked”- contine un system de fisiere.

Informatiile pentru bootare pot fi stocate pe partitii diferite. De obicei aceste informatii sunt niste secvente de blocuri care se incarca in memorie. Executia lor incepe de la o anumita locatie predefinita. Aceste imagini de boot pot contine mai multe instructiuni decat cele necesare pentru a starta un SO. Sunt cazuri in care pe un PC avem 2 sau mai multe sisteme de operare si trebuie sa alegem pe care il vor proni. Un boot loader care intelege conceptual de multiple sisteme de fisiere si de operare poate fi pus in spatial special rezervat pt boot, dupa ce va fi incarcat el va alege SO.

Parttitia root contine kernelul SO si alte informatii utile. Este incarcata la momentul sectiunii de boot. Alte partitii sunt incarcate fie in acelasi moment sau manual mai tarziu acest lucru depinde de SO. SO verifica daca partitia incarcata are un sistem valid de fisiere.

12.2.3 Sistem de fisere virtual

Sistemele de operare moderne suporta mai multe sisteme de fisiere in acelasi timp. Acest deziderat este realizat prin implementarea sistemului de fisiere in 3 nivele majore. (figura 12.4 pag 419)

Nivelul Virtual File System (VFS) are 2 functii:

-separa sistemului de fisiere generic de implementarea prin interfata VFS. Mai multe implementari ale VFS pot sa existe in paralel astfel permitandu-se accesul transparent la diferite tipuri de sisteme de fisiere locale.

-VFS are la baza o strucutura de reprezentare a fisierelor numita **vnode**, aceasta contine un cod numeric unic pentru fiecare fisier din retea in acest fel se poate implemeta un sistemde fisiere la nivelul retelei. Kernelul mentine un vnode pentru fiecare nod active (fisier sau director).

VFS distinge fisierele locale de cele aflate pe alte hosturi in retea. Iar fisierele locale sunt la randul lor separate dupa modul de implemetare a sistemului de fisiere. Astfel ca VFS sties a aleaga handl-ul corespunzator

12.3 Implemantare Director

12.3.1 Lista Liniara

Este cea mai simpla impletare. Se foloseste o lista liniara pentru definirea intrarile in director. Accesul se face tot liniar, timpul fiind destul de mare. Pentru a crea un fisier trebuie cautat tot directorul ca sa vedem ca nu exista si apoi sa il adaugam la final. Ca sa stergem un fisier el trebuie cautat si apoi sters. Spatiul ramas liber in urma stergerii poate fi marcat fie printr-un nume special “all-blank”, prin implementarea unui bit de folosit/nefolosit sau se poate crea o lista care sa retina locatiile libere. O alta posibilitate este sa copiem in locatia libera ultimul fisier din lista si sa scadem marimrea directorului cu 1.

O reducere a timpul de cautare se poate face prin mentinrea listei sortate.

12.3.2 Hash Table

Tot o lista liniara mentine intrarile in director. Dar pe langa aceasta se implementeaza si un hash table. Hash table adresat cu numele fisierului (aplica o functie de hash) returneaza un pointer catre locatia acestuia din lista liniara. Timpul de cautare este redus. Inserarea si stergerea se produc implementeaza usor. Singurul lucru de care trebue tinut cont sunt coliziunile adica 2 nume de fisiere sa pointeze catre aceasi locatie.
Pentru a preveni fenomenul de hash overflow (ex tabela are doar 64 de locatii si sunt 65 de fisiere in director) se poate ca pointerul returnat sa nu adreseze o locatie ci o lista inlantuita.

12.4 Metoda de alocare a spatiului

12.4.1 Alocatie contigua

Fiecare fiser ocupa o zona contigua pe disc. Acest lucru inseamna ca accesul la un fisier va implica o singura miscare a capului de citire (blocul b+1 este accesibil fara miscarea capului dupa ce acesta s-a pozitionat pentru citirea blocului b).
Problema in acest caz este gasirea unui loc liber suficient de mare pt stocarea fisierului pentru aceasta se folosesc strategiile first fit sau best fit. Totusi acesti algoritmi nu pot maximize folosirea eficienta a spatiului total disponibil pe disc.=> fragmentare externa mare.
La momentul creeri unui fisier nu se stie de cat spatiu va avea nevoie acel fisier, aceasta este o alta problema a alocatiei contigue. Aceasta problema este rezolvata intr-unele SO prin alocarea unui alt spatiu contiguu in cazul depasirii spatiului alocat.
Avantaje: fragmentarea interna potrivita, acces direct, acces secvential
Dezavantaje:fragmentarea externa.

Fragmentarea interna=spatial pierdut in interiorul spatiului alocat.

12.4.2 Alocare inlantuita

In aceasta implementare fiecare fisier este o lista de blocuri de pe disc. Fiecare bloc poate fi oriunde pe disc(figura 12.6). Directorul contine un pointer catre primul fisier primul si ultimul bloc al fisierului. Fiecare bloc contine un pointer catre urmatorul bloc.
La crearea unui fisier se aloca intrarii in drector un pointer la primul bloc al fisierului, initial acest pointer este initializat cu nil (end of list pointer value) adica un fisier gol .La scrierea in fisier un bloc liber va fi gasit si este pus un legatura cu sfarsitul fisierului.
Dezvantajul major este ca poate fi folosit efectiv doar in cazul fisierelor secventiale. Nu putem accesa al n-lea bloc din structura unui fisier decat parcurgand cele n-1 blocuri anterioare. Un alt dezanvataj sunt cei 4 B necesari pentru stocarea pointerului in fiecare bloc. Acest inconvenient se rezolva prin alocarea de clustere(grupari de blocori) in loc de blocuri, pointerii folositi vor ocupa atunci un loc mai mic (acest lucru duce la cresterea fagramentarii interne: adica in cazul in cazul in care un cluster nu este plin atunci se pierde loc mai mult decat in cazul in care doar un bloc nu ar fi plin).
Un al dezvantaj al aceastei implementari o constituie fiabilitatea. Daca un pointer este pierdut atunci se pierde calea sper urmatorul bloc/cluster al fisierului.
Tabela FAT: Este pastrata o sectiune pe disc de la inceputul fiecare partitii pt tabela. Tabela este indexata cu numarul fiecarui bloc. Fiecare intrare contine urmatorul bloc al fisierului. Blocurile libere contin valoarea 0 iar blocul final al fisierului contine o valoare speciala end of file.

Tabela fat imbunateste accesul direct.
Avantaje alocare inlantuita:fragmentare externe inexistenta.
Dezavantaje: accesul direct foarte slab, fragmentarea interna mai ales in cazul clusterelor.

12.4.3 Alocare indexata

In index block vor fi stocati toti pointerii rezolvandu-se astfel problema accesarii directe avute de alocarea inlantuita. Fiecare fisier are propriul index block. Intrararea n in system block pointeaza catre blockul n al fisierului. Directorul contine adresa index blockului . figura pag 427.
Aceasta implementare rezolva accesul direct sin u sufera de framgmentare externa.
Se pune problema cat de mare sa fie index blockul.
Linked scheme: marimea index block=marime disc bloc. In cazul fisierelor care necesita un index mai mare se vor face legaturi intre index blocuri.
Multilevel index: exista mai multe nivele de index blocuri. Primul nivel contine o lista de index blocuri de nivel second. Care contin lista de index de nivel inferior si asa mai departe. Cel mai inferior nivel va contine pointerii catre blocurile fisierului.
Schema combinata: aceasta varianta presupunea retinerea unui numar de pointer catre index bloc in nodul fisierlui. O parte din acesti pointeri vor pointa direct catre index blocuri care mapeaza blocurile fisierului, iar o alta parte vor implementa principiul de multiveleve index.

12.5 Managementul spatiului liber

Bit vector: fiecarui bloc ii corespunde o intrare in acest bit vector. Daca blocul este liber atunci va avea valoarea 1 daca blocul este ocupat va avea valoarea 0.
Avantajul este simplitate si eficienta metodei. Dar rezultatele nu sunt bune decat daca vectorul este stocat in memorie si copiat la intervale de timp pe disc pt consisetenta. Ori vectorul poate avea o dimensiune semnificativa in cazul discurilor de capacitate mare. Soluatia este folosita doar in cazul microcomputerelor.
Liste inlantuite: se retine doar adrese primului bloc liber acesta va adresa urmatorul bloc liber si asa mai departe.
Schema nu este eficienta accesul la blocurile libere fiind unul secvential. De remarca e ca metoda FAT are implementat acest mecanism nefiind necesara o redefinire a tabelei.
Grouping: Lista inlantuita va mentine adresele a n blocuri libere in fiecare nod. Avantajul este ca adresele a mai multor blocuri libere pot fi gasite mai repede.
Counting: In aceasta metoda lista va avea in fiecare nod adresa primului bloc liber si nurmarul de blocuri libere care ii urmeaza. Metoda se bazeaza pe principiul de alocare contigua sau pe alocarea care foloseste clustere sustinand ca o stergere din memorie va elibera de cele mai multe ori mai multe blocuri consecutive. Prin pastrarea acestor date lista inlantuita va scadea in dimensiune.

+12.6 Eficienta si performanta facultative

12.7. Recovery

Fisierele si directoarele sunt tinute atat in memorie cat si pe disc din acest motiv trebuie asigurata consistenta datelor. Este de presupus ca datele din memorie vor fi mai recente decat cele de pe disc. Cu toate acestea in cazul unei caderi a sistemului datele din memorie vor fi pierdute iar modificarile la nivelul discului vor fi incomplete ducand la

eventuale stari de inconsitenta a informatiei. In aceste cazuri la repornirea sistemului va fi lansat un program special, acesta va compara data din structura director cu data din blocurile de pe disc si va incercat sa elimine orice inconsitenta.

12.7.2 Backup and Restore

Se utilizeaza un sistem de salvare ciclic folosind n copii ale datelor. In prima zi se va salva copia sistemului intreg in a doua zi doar datele care s-au modificat intre timp si asa mai departe. Cand ciclul ajunge la final el se va relua printr-o noua salvare completa a sistemului.

12.8. Log structured File System

Toate schimbarile la nivel de metadata sunt scrise secvential in log. Fiecare set de operatii este o tranzactie. Dup ace schimbarile sunt scrise in log ele sunt considerate comise. Intrarile din log vor schimba sistemul de fisiere actual si pe masura ce se fac schimbarile un pointer indica care actiune s-a facut si care urmeaza sa se faca. Cand o tranzactie intreaga este comisa ea este stearsa din log. Logul este defapt un buffer circular. In cazul in care sistemul cade starea de consistenta se va pastra iar modificarile vor putea fi implementate la nivelul sist de fisiere datorita logului. Singura problema este in cazul in care sistemul cade in mijlocul unui tranzactii, in acest caz toate modificarile facute trebuie sa fie inlaturate.

12.9 NFS

NFS este un exemplu bun de un sistem de fisiere dintr-un sistem de retea client server.

12.9.1 Overview

NFS vede n set de statii de lucru conectate ca un set de masini independente cu sistem de fisiere independente. Scopul este sa se permita un o anumita partajare intre sistemele de fisiere in mod transparent. Sa se asigure independenta dintre masini, partajarea unui sistem de fisiere afecteaza doar masina client si nu alta masina. Un director la distanta este instalat peste sistemul local de fisiere. Directorul montat arata ca un subarbore integral al sistemului local de fisiere si inlocuieste subarborele care coboara din directorul local.
Specificarile directorului la distanta la operatia de montare(suprapunere) este netransparenta, numele gazdei directorului la distanta trebuie furnizat. Fisierele din directorul la distanta pot fi accesate intr-un mod transparent. Orice sistem de fisiere sau director poate fi montat la distanta peste un director local. Montarile cascadate sunt si ele permisa in unele implementari NFS.
NFS este realizat sa opereze intr-un mediu heterogen de mai multe masini, SO si arhitecturi de retele. Specificatiile NFS sunt independente de acestea. Independenta este atunsa prin folosirea primitivelor RPC(proceduri la distanta) construite pe baza protocolului XDR(reprezentarii extrne a datelor). Daca sistemul consta din masini heterogene si sisteme de fisiere interfatate corect, tipuri diferite de sisteme de fisiere pot fi montate(mounted) si local si la distanta. Specificariile NFS afce diferenta dintre servciciile furnizate de un mecanism de montare si de serviciile de accesa al fisierelor la distanta. poze pg 442 si 443

12.9.2 Protocolul de montare(Mount Protocol)

Proctocolul de instalare stabileste conectiunea logica initiala intre server si client. Operartiile de montare includ numele directorului la distanta si numele srverului pe care se afla. Cererea de montare este mapata intr-un RPC corespunzator si inaintat serverului care se ocupa de montat care ruleaza pe serverul corespunzator. Serverul gestioneaza o lista de export(export list) care specifica sistemele locale de fisiere care sunt exportate pt a fi montate impreuna cu numele serverelor ca pot sa le monteze. Cand un server primeste o cerere de montare care se coformeaza cu lista de export, acesta returneaza clientului un handle al fisierului care are rolul de cheie pentru accese viitoare la fisiere din sistemul montat. Handle-ul fisierului contine toate informatile de care are nevoie serverul ca sa face distinctie intre fisierele individuale pe care le are. Handle-ul fisierelor consta dintr-un identificator de sitem fisier si un numar care identifica directorul montat in sistemul de fisiere exportat. Serverul mentine o lista cu clientii si directoarele montate corespunzatoare. Operatia de montare schimba doar vederea din partea user-ului si nu afecteaza partea serverului.

12.9.3 Protocololul NFS

Asigura un set de apeluri de proceduri pentru operatii cu fisiere la distanta. Procedurile suporta operatile urmatoare:

- cautarea unui fisier in cadrul directorului
- citirea unui set de intrari in director
- manipularea linkurilor si a directoarelor
- accesarea atrobutelor fisierelor
- citirea si scrierea fisierelor

Serverele NFS nu au stare, fiecare cerere trebuie sa furnizeze un set complet de argumente. Serverele nu gestioneaza informatia despre clientii lui de la un acces la altul. Datele modificate trebuie salvate pe discul serverului inainte ca rezultatele sa fie returnate clientului. Protocolul NFS nu furnizeaza un mecanism de control al concurentei.

Interfata de fisiere UNIX este bazata pe apelurile open, read, write, close si pe descriptori de fisier. Layer-ul sistemului de fisiere virtual(VFS) distinge intre fisiere locale si la distanta; fisierele locale se disting dupa tipul sistemului de fisiere. VSF activeaza operatii specifice sistemului de fisiere care se ocupa de cererile locale conform tipului sistemului de fisiere. VFS apeleaza proceduri ale protocolului NFS pentru cereri al distanta. O cerere RPC este facuta nivelului serviciu NFS al serverului la distanta. Poza pg 446

12.9.4 Path-Name Translation

Translatia numelui caii se face prin impartirea caii in componente si executarii unui apel separat lookup call pentru fiecare pereche de componente si directoare vnode. Pt ca aceasta cautare sa fie rapida, un cahce implementat la client retine nodurile pentru numele directoarelor la distanta. Acest cahe mareste viteza de referire la fisiere cu numele caii initiale identic. Cache-ul este indepartat cand atributele returnate de la sever nu se potrivesc cu cele ala nodurilor din cache.

12.9.5 Operatii la distanta

Exista o corespondenta aproape de 1 la 1 intre apelurile de sistem uzuale UNIX si protocoalele NFS de RPC (cu exceptia deschiderii si inchiderii fisierelor). Nu exista corespondenta directa intre operatii la distanta si un RPC. Blocurile sunt adunate si introduse intr-un cache local. Doau tipuri de cache:
File-blocks cache: cand un fisier este deschis, kernelul controleaza daca atributele din cache trebuie aduse sau revalidate. Blocurile cu fisiere cahced sunt folosite doar odata daca atributele corespunzatoare din cache sunt actuale.
File-attribute cache: Cache-ul cu atribute este actualizat de cate ori apar noi atribute de la server.
Clientii nu elibereaza blocurile intarziate de scriere decat cand serverul confirma ca data a fost scrisa pe disc.

Cap 17
Coordonarea in sistemele distribuite
17.1 Ordonarea evenimentelor

In sistemele distribuite nu avem un ceas comun sau memorie comuna.
17.1.1 Happened Before Relation

Din moment ce consideram doar procese secventiale rezulta ca toate evenimentele dintr-un process sunt ordonate.

A->B inseamna ca A se executa inainte de B. De exemplu un mesaj de raspuns este conditionat de un mesaj cerere. + figura pag 597.

17.1. 2 Implementare

Fiecare proces are asociat un ceas logic Pi are asociat LCi. Acest ceas este un simplu contor care se poate doar incrementa. El va desemna un numar unic fiecarui eveniment din interiorul procesului.

Ordonarea la nivel global se face in felul urmator. Daca un Pj primeste un mesaj A de la un Pi si LCi(A) > LCj atunci LCj =LCj+1 in caz contrar nu se va intampla nimic.

17.2. Excluderea mutuala

17.2.1 Principiul centralizat

Exista un singur process ales care va coordona intrarea celorlalante procese in sectiunea critica. El va decide care dintre procese va intra in aceasta sectiune dupa urmatorul algoritm:

Un process Pi care vrea sa intre in sectiunea critica va trimite o cerere catre procesul central Pc, acest va verifica daca alt process este in sectiunea critica exita 2 cazuri: 1. nici un proces nu este in SC => va trimite un reply acordand permisiunea procesului Pi. La terminarea sectiunii critice Pi va trimtie un mesaj de elibera catre Pc.

2. in caz contrrar Pi este pus intr-o coada de asteptare

La receptia unui mesaj de elibarare Pc va scoate din coada primul process si ii va trimite un mesaj de permisiune.

Daca Pc cade atunci un alt proces este ales Pc.

17.2.2 Principiul descentralizat

Nu exista un process central. Cand procesul Pi vrea sa intre in sectiunea critica el va trimite o cerere (Pi, TS) catre toate procesele existente. Daca va primi un reply de la toate procesele atunci inseamna ca nici unul nu este in sectiunea critica si deci are permisiunea sa intre in ea. In cazul in care un process este in sectiunea critica el nu va trimtie reply si va pune intr-o coada cererea procesului Pi.

Mai exista cazul in care 2 procese vor sa intre in acelasi timp in SC. Atunci ele compara marca de timp TS, cea cu marca de timp cea mai mica va avea prioritate.

La intrarea unui nou process numele acestuia trebuie trimis tuturor proceselor existente iar el trebuie sa primeasca numele proceselor existente.

17.2.3 Token Passing

Are la baza o topologie logica de tip inel . Un token va trece pe rand pe la fiecare proces. Un proces care are token-ul va putea sa intre in SC, daca nu vrea sa intre in SC va ceda tokenul urmatorului proces. Exista 2 cazuri defavorabile daca tokenul este pierdut atunci trebuie creat altul nou , sau daca un process cade atunci trebuie redesenata topologia .

17.3 Atomicitate

Atomicitatea=o unitate de program care trebuie executata in totalitate sau neexecutat deloc.

Un coodronator de tranzactie intr-un sistem distribuit trebuie sa aiba grija sa pastreze atomicitatea tranzactiei..

17.3.1 Two Phase Commit Protocol (2PC)

T- tranzactie initiate pe procesorul Si cu coordonatorul de process Ci.

Prima faza: Ci adauga in logul sau <pregateste T>. Ci transmite la toate procesoarele care au executat pe T , mesajul pregateste (T) . La receptia acestui mesaj managerul de tranzactie al statiei va decide daca este dispus sa comita sau nu T. Daca nu vrea sa comita T atunci va adauga in log <no T> si va trimite mesajul abort(T), daca estepregatit sa comita T atunci va adaugain log <ready T> si va trimtie mesajul ready(T).

Faza 2: Ci primeste mesajele raspuns. Daca cel putin un mesaj este abort(T) atunci va concluzia ca T este <abort T> si va pune in log randul acesta. In caz contrar va concluzia comiterea lui T si va adauga in log <commit T>. Dup ace a luat decizia Ci trimite mesajul abort(T) sau commit(T). Statile vor stoca acest mesaj in logurile lor.

17.3.2 Erori in 2PC

17.3.2.1 Eroare la nivelul unei statii participante

Logul statie contine <commit T> => statia executa redo(T). (nu a terminat sa efectueze tranzactia complet si o va lua de la capat)

Logul contine <abort T> => undo T

Logul contine <ready T> Statia nu stie daca T este comisa sau nu. Asa ca va cere aceasta informatia la Ci printr-un query. Daca Ci a picat si el el va cere aceasta informatia tututor statiilor din retea, daca nici un site nu are aceasta informatii el va astepta nestiind ce decizie sa ia nu va elimina aceasta linie din log. Ci va avea intotdeauna in logul lui decizia luata cu privire la T deci la revenirea lui Si statia va stii care a fost decizia.

17.3.2.2 Eroare la nivelul lui Si

Daca o statie active contine <commit T> atunci T trebuie sa fie comisa tututor statiilor

Daca o statie active contine <abort T> atunci T trebuie abort la nivelul tututor statiilor.

Daca niste statii active nu contn <ready T> atunci T trebuie abort. tututor statiilor

Daca toate statiile contin <ready T> atunci nici o decizie nu poate fi luata. Statiile vor trebui sa astepte revenirea lui Si. Intre timp este posibil ca anumite resurse de la nivelul statiilor sa fie blocate in asteptarea unei decizii.

17.4. Controlul Concurentei

17.4.1 Locking Protocols

Se refera la date si algoritmi partial modificati din cap 7.9

17.4.2 Marci de timp

Fiecare tranzactie are atribuit un numar unic astefel se poate decide oridinea de executie.

17.4.2.1 Generalizarea marci de timp unice

Fiecare statie genereaza o marca de timp locala unica. Marca globala este obtinuta prin concatenerea la aceasta marca a indentificatorului statiei care este de asemenea unic (!!!acesta se pune la final pe cei mai putini semnificativi biti). Ajustarea marcilor de timp locale se poate face dupa principiul ceasurilor logice discutate mai sus.

17.4.2.2 Schema de ordonare cu marci de timp

Combina schma centraliza a marcilor de timp cu 2PC pentru a realize un protocol care asigura serializarea si inlatura roll back cascading.

se refera la date si algoritmi partial modificari din cap 7.9

17.5 Deadlock handling

17.5.1 Deadlock prevention

Resource ordering-prevention deadlock: fiecare resursa primeste un numar unic. UN process poate cere o reursa cu numarul i doar daca nu retine o resursa cu un numar mai mare ca i.

Banker algoritm: alege un singur process din sistem care va implementa acest algoritm.

1. Fiecare process are un numar de prioritate.
2. Numerele de prioritate sunt utilzate ca sa se decida daca un process va asteapta dupa altul
3. Elimina deadlockurile.

Pentru controlul Preemption (?) fiecare process va avea o prioritate. Se stabileste ca Pi va astepta dup Pj doar daca prioritatea sa este mai mare decat a lui Pj in caz contrar Pi va fi roll back. Desi aceasta schema previne deadlockurile ea poate duce la fenomenul de infometare. Pt eliminarea acestui se combina aceasta tehnica cu marcile de timp.

1. wait –die scheme. Daca Pi cere o resursa detinuta momentan de Pj el va fi pus sa astepte doar daca marca lui de timp este mai mica decat a lui Pj in caz contrar va muri(roll back).
2. wound-wait scheme. Este invers ca la 1 daca Pi are marca de timp mai mare va fi lasat sa astepte.

Fenomenul de informatare se poate elimina prin conditia ca atunci cand un process moare san u ii se aloce alta marca de timp

17.5.2 Deadlock detection

Fiecare statie mentine un graf local wait-for. Nodurile din graf corespund tuturor proceselor locale si nonlocale care mentin sau cer o anumita resursa a statiei. Fig pag 612. Daca graful are un ciclu atunci avem un deadlock. Se demonstreaza ca nu este neaparat necesar ca un graf local sa aiba un ciclu pt a exista un deadlock. Detectia completa a unui deadlock este facuta doar in momentul unirii tuturor grafurile locale intrun graf global.

17.5.2.1 Medota centralizata

Un singur process va retine graful global acest process se numeste process coordinator de deadlock detection. Din moment ce exista intarzieri de comunicatie acest graf nu este unul in timp real. Daca se detecteaza un deadlock atunci statiile vor fi anuntatte si se va intra in starea de deadlock.

Moduri de obtinerea a grafului: update cand o noua muchie este inserata.

Periodic cand se intampla un numar de schimbari in graf

Oricand processor coordinator are nevoie sa invoce un ciclu de detectie.

Primele 2 modalitati pot induce deadlockuri false datorita intarizerilor din retea. Ultima modalitate este implementata in asa fel incat aceste deadlockuri false sunt eliminate. (explicatie pag 615)

17.5.2.2 Metoda descentralizata

Se mentin mai multe grafuri partiale la nivelul fiecare statii. Se pleaca de la idea ca daca exista un deadlock atunci el va determina cel putin un ciclu intr-unul din grafurile existente. (pag 616)

17.6 Algoritmul de alegere

In multe implementari din acest capitol un process trebuie ales pentru a coordona o activitate. Se pleaca de la principiul ca fiecare process un numar de prioritate unic (Pi are prioritate i)Procesu coordinator va fi cel cu numarul cel mai mare.

17.6.1 Bully Agoritm

Pi trimite un mesaj de alegere tuturor proceselor cu prioritate mai mare ca a sa si asteapta un raspun intr-un interval de timp T. Daca nu primeste nici un raspuns se allege pe el process coordinator si trimite un mesaj de instiintare celorlalte procese. Daca primeste un raspuns in acest interval e va astepta un interval de timp T1 pentru a primii un mesaj de la procesul coordinator. Daca nu primeste acest mesaj atunci el va relua algoritmul de alegere presupand ca procesul cu prioritate mai mare a cazut.

17.6.2 Ring algoritm

Se foloseste de listele active. Aceste liste contin numerele de prioritate ale tuturor proceselor active in momentul in care algortimul se termina. Fiecare process are propria lista.

Algoritm :

Daca Pi determina caderea procesului coordinator creaza o noua lista initial goala. Apoi trimite mesajul elect(i) la vecinul di dreapta.

Daca Pi primeste un mesaj de la vecinul din stanga elect(j) avem 3 posibilitai:

1. Este primul mesaj primit. Atunci creazao noua lista adauga numerele I si j si trimtie la vecinul din dreapta elect (j) si elect(i).
2. Daca i!=j atunci inseamna ca vecinul din stanga nu are numarul lui Pi el va adauga I in lista si va trimtie mai departe lista.
3. daca i=j atunci s-a ajuns la capatul cercului. Si Pi poate sa determine cel mai mare numar din lista si sa identifice procesul coordinator.

17.7 Ajungerea la intelegere

Comunicatie ureliable

Dup ace Pi trimite un mesaj lui Pj el astepata un timp t sa primeasca un mesaj de confirmare de primire. Daca nu va primi acest mesaj el va retrimite mesajul initial pana cand va primi confirmarea sau va fi notificat de sistem ca statia Sj este cazuta. Daca Pj asteapta si el o confirmare de la Pi ca a primit replay se poate concluziona ca cele 2 procese ajung la o intelegere

17.7.2 Faulty

Presupunem ca mediul de comunicatie este viabil dar procesele pot sa cada. Avem n procese din care doar m sunt supuse esecului. Problema se pune cum determinam pe cele m care pot da rateuri .

Fiecarui process i se asociaza o valoare Vi private. Algoritmu va crea pentru fiecare process nonfualty un vector Xi= {Ai1, Ai2, .. Ain} cu prorpietatile

1. daca Pj nonfaulty atunci Ai,j =Vj
2. daca Pi nonfaulty atunci Xi=Xj.

+ explicatie pentru m=1 si n=4 pag 623.