OS klausimai

1. Operacinės sistemos sąvoka
2. Operacinių sistemų kategorijos
3. Multiprogramavimo sąvoka
4. Virtualios mašinos sąvoka
5. Lygiagretūs procesai. Notacija “and”
6. Lygiagretūs procesai. Notacija “FORK, JOINT, QUIT”
7. Kritinė sekcija
8. Kritinės sekcijos apsauga
9. Dekerio algoritmas
10. Semaforai
11. Procesų “gamintojas” ir “naudotojas” sąveika
12. Įvedimo-išvedimo spuleris. Bendra charakteristika
13. Įvedimo-išvedimo spulerio pagrindinis procesas
14. Įvedimo-išvedimo spulerio įvedimo ir skaitymo procesai
15. Įvedimo-išvedimo spulerio išvedimo ir rašymo procesai
16. Dvejetainių ir bendrų semaforų sąryšis
17. Operacijų su semaforais realizacija
18. Operaciju su semaforais realizacija be užimancio laukimo
19. Procesų ir resursų sąvoka
20. Proceso deskriptorius
21. Resurso deskriptorius
22. Primityvas KURTIP
23. Primityvas NAIKINTIP
24. Primityvas STABDYTIP
25. Primityvas AKTYVUOTIP
26. Primityvas KEISTIPP
27. Primityvas KURTIR
28. Primityvas NAIKINTIR
29. Primityvas PRAŠYTIR
30. Primityvas ATLAISVINTIR
31. Procesų būsenų schema
32. Procesų planuotojas
33. Pasiruošusio proceso su aukščiausiu prioritetu nustatymas ir neaktyvaus procesoriaus radimas
34. Procesų planuotojo procesoriaus perskirstymo etapas
35. OS architektūros raida
36. Mikrobranduolio architektūra
37. Mikrobranduolio architektūros charakteristika
38. Gijos. Bendra charakteristika
39. Vartotojo ir branduolio gijos
40. Virtualios atminties sąvoka
41. Komandos vykdymo schema architektūroje su dinaminiu adresų nustatymu
42. Statinis ir dinaminis atminties skirstymas
43. Puslapinė organizacija
44. Polisegmentinė virtuali atmintis
45. Atminties skirstymo puslapiais strategijos
46. Failų sistemos sąvoka
47. Failinės atminties įrenginių charakteristika
48. Įvedimo/išvedimo įrenginių greičių charakteristika
49. Failų sistemos hierarchinis modelis
50. RAID0-RAID6, RAID 10, RAID 50, RAID 60 organizacijos

Turinys

Operacinės sistemos sąvoka

OS – tai organizuota programų visuma, kuri veikia kaip interfeisas tarp kompiuterio aparatūros ir vartotojo. OS sudaro tai kas vykdoma supervizoriaus rėžime. Ji aprūpina vartotojus priemonių rinkiniu, projektavimo ir programavimo palengvinimui, programų saugojimui ir vykdymui, ir tuo pat metu valdo resursų pasiskirstymą, kad būtų užtikrintas efektyvus darbas. OS – tai programa kuri modeliuoja kelių virtualių mašinų darbą, vienoje realioje mašinoje (projektuoja VM į RM). OS branduolyje yra priemonės, kurių pagalba realizuojamas sinchronizuotas procesorių, OA ir periferinių įrenginių darbas.

OS turi tenkinti tokius reikalavimus:

1. *Patikimumas* – sistema turėtų būti mažų mažiausiai tokia patikima, kaip aparatūra. Klaidos atveju, programiniame arba aparatūriniame lygmenyje, sistema turi rasti klaidą ir pabandyti ją ištaisyti arba minimizuoti nuostolius.
2. ***Apsauga* – apsaugoti vartotoją kitų vartotojų atžvilgiu.**

Operacinių sistemų kategorijos

Yra trys grynosios OS kategorijos. Skirstymas į jas remiasi šiais kriterijais:

1. Vartotojo (užduoties autoriaus) sąveika su užduotimi jos vykdymo metu;
2. Sistemos reakcijos laikas į užklausą užduočiai vykdyti.

Grynosios OS kategorijos:

- Paketinio apdorojimo OS. Tai sistema, kurioje užduotys pateikiamos apdirbimui paketų pavidale įvedimo įrenginiuose. Vartotojas neturi ryšio su užduotimi jos vykdymo metu. Sistemos reakcijos laikas matuojamas valandomis. Tokios OS yra efektyviausios mašinos resursų naudojimo prasme, bet labai neefektyvios žmogaus resursų atžvilgiu.
- Laiko skirstymo OS. Užtikrina pastovų vartotojo ryšį su užduotimi. Ji leidžia vienu metu aptarnauti keletą vartotojų. Kiekvienam vartotojo procesui „kompiuteris“ suteikiamas nedideliam laiko kvantui, kuris matuojamas milisekundėmis. Jei procesas neužsibaigė tol, kol baigėsi jo kvantas, tai jis pertraukiamas ir pastatomas į laukiančiųjų eilę, užleidžiant „kompiuterį“ kitam procesui.
- Realaus laiko OS. Jos paskirtis – valdyti greitaeigius procesorius (pvz.: skrydžio valdymas). Sistema turi pastovų ryšį su užduotimi užduoties vykdymo metu. Jos reikalauja papildomų resursų(prioritetinių). Čia labai griežti reikalavimai procesų trukmei. Būtina spėti sureaguoti į visus pakitimus, kad nei vieno proceso nei vienas signalas nebūtų praleistas. Reakcijos laikas matuojamas mikrosekundėmis.

Visos šios sistemos pasižymi multiprogramavimu – galimybe vienu metu vykdyti kelias užduotis.

Multiprogramavimo sąvoka

Multiprogramavimas atsirado kaip idėja, kuri turėjo reaguoti į skirtingus procesoriaus bei periferijos greičius. Multiprograminė operacinė sistema (MOS) – viena operacinių sistemų rūšių. Šio tipo operacinė sistema užtikrina kelių užduočių lygiagretų vykdymą, t. y. leidžia operatyvioje atmintyje būti kelioms vartotojo programoms, skirstydama procesoriaus laiką, atminties vietą ir kitus resursus

aktyvioms vartotojo užduotims. MOS privalumai yra akivaizdūs. Vartotojui vienu metu paprastai neužtenka vienos aktyvios programos. Tai ypač akivaizdu, kai programa vykdo ilgus skaičiavimus ir tik kartais prašo įvesti duomenis. Tuo metu vartotojas yra priverstas stebėti užduoties vykdymą ir tampa pasyviu.

Tam, kad būtų galima realizuoti MOS, kompiuterio architektūrai keliami tam tikri reikalavimai:

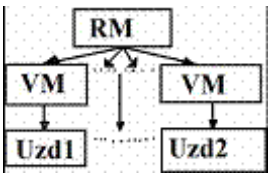
- *Pertraukimų mechanizmas* (jei jo nebūtų, liktų interpretavimo mechanizmas).
- *Privilegijuotas režimas*, t. y. esant privilegijuotam režimui uždrausti neprivilegijuotų komandų vykdymą. Priešingu atveju būtų labai ilgas darbas. MOS turi pasižymėti savybe, kad vienu metu dirbantys procesai neturi įtakoti vieni kitų (ar tai sisteminiai, ar vartotojo).
- *Atminties apsauga*. Jei vykdant komandą suformuojamas adresas, išeinantis iš komandoms skirtos adresų erdvės – suformuojamas pertraukimas.
- *Relokacija* (papildoma savybė). Tai programos patalpinimas į bet kokią atminties vietą, t.y. programos vykdymas gali būti pratęstas ją patalpinus į kitą atminties vietą. Tai efektyvumo klausimas.

MOS yra populiariausias šio laikmečio operacinių sistemų tipas. MOS – kai vienam vartotojui suteikiama galimybė vienu metu daryti kelis darbus.

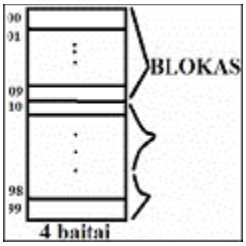
Virtualios mašinos sąvoka

Reali mašina - tai kompiuteris. Užduotis susideda iš programos, startinių duomenų ir vykdymo parametrų. Rašyti programą realiai mašinai būtų sudėtinga ir nepatogu. Todėl vienas iš operacinės sistemos tikslų yra paslėpti realią mašiną ir pateikti mums virtualią. Užduoties programą vykdo ne reali, o virtuali mašina. Virtuali mašina – tai tarsi virtuali realios mašinos kopija. Virtuali reiškia netikra. Mes tarsi surenkame reikalingas realios mašinos komponentes, tokias kaip procesorius, atmintis, įvedimo/išvedimo įrenginiai, suteikiame jiems paprastesnę nei reali vartotojo sąsają ir visa tai pavadiname virtualia mašina. Vienas iš virtualios mašinos (VM) privalumų yra programų rašymo palengvinimas, todėl realios mašinos komponentės, turinčios sudėtingą arba nepatogią vartotojo sąsają, virtualioje mašinoje yra supaprastintos. Virtuali mašina dirba su operacinės sistemos pateiktais virtualiais resursais, kurie daugelį savybių perima iš savo realių analogų ir pateikia kur kas paprastesnę vartotojo sąsają. Tai lengvina programavimą.

Kiekviena užduotis turi savo virtualią mašiną, kurios, iš tikrųjų, ir konkuruoja dėl realaus procesoriaus. Vienas esminių virtualios mašinos privalumų yra tas, kad užduotis, kurią vykdo virtuali mašina, elgiasi lyg būtų vienintelė užduotis visoje mašinoje. Tai yra didelė parama programuotojui. Dabar jam tenka rūpintis tik pačios programos rašymu. Pav. Virtualios mašinos pateikimas užduotims multiprograminės operacinės sistemos atveju.



VM specifikacija. Tarkime, yra 100 žodžių atmintis (0-99). Kiekvienas žodis yra 4 baitų □□□□. Žodžiai adresuojami nuo 0 iki 99. Tegul atmintis yra suskirstyta blokais po 10 žodžių.

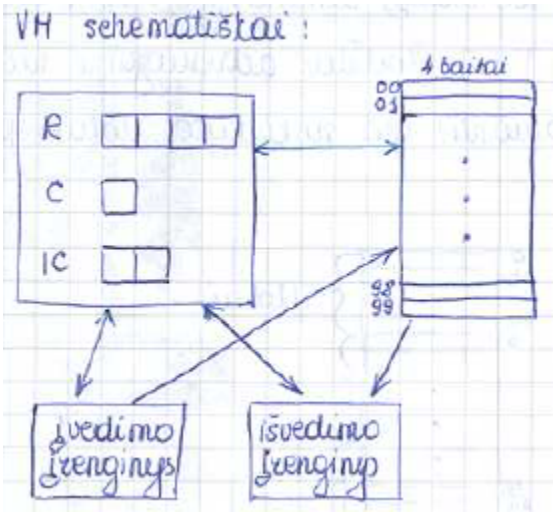


Procesorius turi 3 registrus: a) R – bendrasis registras □□□□ – 4 baitai; b) C – loginis trigeris, priima reikšmes true (T) arba false (F), kad būtų atliktas sąlyginis valdymo perdavimas □ – 1 baitas; c) IC – komandų skaitliukas □□ – 2 baitai.

Atminties žodis interpretuojamas kaip komanda arba duomenys. Operacijos kodas užima 2 vyresniuosius baitus, o adresas – 2 jaunesniuosius.

O P R	a dr .

- komandos struktūra. VM turi nuoseklaus įvedimo bei išvedimo įrenginius.



Įvesties/išvesties įrenginiai valdomi procesoriaus.

Virtualios mašinos procesoriaus komandos su paaiškinimais:

Komanda	Paaiškinimas
AD	sudėties komanda – x_1x_2, R:=R+[a], a - adresas, a:=10* x_1 + x_2, x_1,x_2 є {0,..,9}; ADx_1x_2;
LR	registro pakrovimas iš atminties - x_1,x_2 => R: = [a]; LRx_1x_2;
SR	įsimenama registro reikšmė - x_1,x_2 => a: = R; SRx_1x_2;
CR	palyginimo komanda - x_1,x_2 => if R = [a] then C: = 'T' else C: = 'F';
BT	sąlyginis valdymo perdavimas - x_1,x_2 => if C: = 'T' then IC: =a; BTx_1x_2;
GD	apsikeitimas su išore vyksta blokais (x_1 – bloko nr.) - x_1,x_2 => Read([β+i], i = 0,..,9), β= 10* x_1; GDx_1x_2;
PD	išvedami duomenys - x_1,x_2 => Print([β+i], i = 0,..,9); PDx_1x_2;
H	sustojimo komanda => HALT. VM pradeda darbą, kai registro IC reikšmė yra 00 (įvykdo komandą, kuri patalpinta nuliniame žodyje).

Lygiagretūs procesai. Notacija „and“

Nuoseklus procesai veikia vienu metu – lygiagrečiai. Procesai neturi jokių tarpusavio sąryšių. Proceso aplinką sudaro resursai, kuriuos procesas naudoja, ir kuriuos sukuria.

Prasminis ryšys tarp procesų išreiškiamas per proceso resursus.

OS gali būti apibūdinta kaip procesų rinkinys, kur procesai:

- veikia beveik nepriklausomai (lygiagrečiai),
- bendrauja per pranešimus ir signalus (resursus),
- konkuruoja dėl resursų.

Skaičiavimo sistemose minimas aparatūrinis ir loginis lygiagretumas (paralelizmas).

Aparatūrinis lygiagretumas – reiškia lygiagretų, vienalaikį aparatūros darbą (pvz. išorinių įrenginių kontrolė, kur kiekvieną iš jų kontroliuoja kitas procesas).

Loginiame lygiagretume nesvarbu lygiagretumas. Apie jį kalbama tada, kai teoriškai darbas gali būti vykdomas lygiagrečiai.

Aparatūrinis paralelizmas įvedamas efektyvumo sumetimais, kad greičiau vyktų darbas. Procesoriuje nesant aparatūriniam paralelizmui vis vien svarbu vienintelį proc. darbo laiką skirstyti keliems procesams. Todėl įvedama lygiagrečiai vykdomo proceso abstrakcija.

Neformaliai procesas – tai darbas, kurį atlieka procesorius, vykdydamas darbą su duomenimis.

Loginis paralelizmas pasižymi tuo, kad kiekvienas procesas turi savo procesorių ir savo programą. Realiai, skirtingi procesai gali turėti tą patį procesorių ar tą pačią programą.

Procesas yra pora (procesorius, programa).

Procesai – tai būsenų seka s0,s1,...,sn, kur kiekviena būsena saugo visų proceso programos kintamųjų reikšmes. Pagal proceso būseną galima pratęsti proceso darbą. Proceso būsena turi turėti kitos vykdomos programos adresą. Proceso būsena gali būti pakeista paties proceso arba kitų procesų.

Valdymo ir informacinis ryšys tarp procesų realizuojamas per bendrus kintamuosius. Nagrinėjant tik pačius procesus, gaunami nuo procesoriaus nepriklausomi sprendimai.

(s1, s2 – sakinai)

s1, s2 – procesai vyksta nuosekliai

s1 and s2 – lygiagrečiai

pvz. (a+b)*(c+d)-(e/f)

```
begin   t1:= a+b and t2:= c+d;   t4:=t1*t2 end and t3:= e/f; t5:=t4-t3;
```

Transliatorius turėtų išskirti lygiagrečius veiksmus ir sugeneruoti aukščiau užrašytą programą.

Lygiagretūs procesai. Notacija „FORK, JOIN, QUIT“

Jei procesas p įvykdo komandą FORK W (išsišakojimas), tai sukuriama p proceso kopija q, vykdoma nuo žymės W.

Jei procesas p įvykdo komandą QUIT, tai jis pasibaigia.

Procesų aplinkybių komanda JOIN T, W:

```
T := T – 1; IF T = 0 THEN GOTO W;
```

(Svarbu! Šios komandos yra nedalomos/nepertraukiamos!)

Pvz.

```
N := 2;     FORK P3;     M := 2;     FORK P2; /* iš viso dabar dirba 3 procesai
(pirmas tas, kuris viską inicializavo) */     T1 := A+B;   JOIN M, P4; QUIT; P2:
T2 := C+D;   JOIN M, P4; QUIT; P4: T4 := T1*T2; JOIN N, P5;  QUIT; P3: T3 := E/F;
JOIN N, P5;  QUIT; P5: T5 := T4-T3;
```

Kritinė sekcija

Tarkime turime du procesus p1 ir p2, kurie atlieka tą patį veiksmą x:=x+1(x-bendras kintamasis). jie asinchroniškai didina x reikšmę vienetu. p1 vykdo procesorius c1 su registru r1, o p2 – c2 su bendru registru r2. t_o momentu x = v

t_o----------->t_n laiko ašis

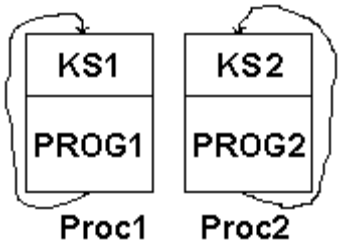
```
a) p1:r1:=x; r1:=r1+1; x:=r1; …
p2:    …     r2:=x; r2:=r2+1; x:=r2;… [x=v+1]

b)p1: r1:=x; r1:=r1+1; x:=r1; …
p2:              …            r2:=x; r2:=r2+1; x:=r2;… [x=v+2]
```

Gavome: a) x = v+1 ir b) x = v+2, o taip būti negali. Tai dviejų procesų problema. Ir pirmu atveju gali būti panaši situacija, kaip antru, dėl pertraukimų.

Programos dalis, dirbanti su bendrais procesų resursais, vadinama kritine sekcija. Negalima leisti kad du procesai vienu metu įeitų į kritinę sekciją. Todėl reikia užtikrinti kad kritinėje sekcijoje tuo pačiu metu būtų tik vienas procesas. Geras būdas kritinei sekcijai tvarkyti – semaforų naudojimas.

Tarkime yra keletas ciklinių procesų:



N lygiagrečiai dirbantys procesai Proc1, Proc2, ... ProcN:

```
Begin   Proc1: begin L1: KS1; PROG1; GOTO L1; end;   and   Proc2: begin L2: KS2;
PROG2; GOTO L2; end;   and   ...   and   ProcN: begin LN: KSN; PROGN; GOTO LN; end;
End;
```

Kritinės sekcijos apsauga

Bet kuriuo laiko momentu turi būti vykdoma tik vieno proceso kritinė sekcija. Reikalavimai:

- atminties blokavimas (vienu laiko momentu kreipiasi į tą pačią atmintį tik vienas procesas);
- visos kritinės sekcijos yra lygiareikšmės;
- vykdymo greičiai bet kokie;
- programa gali būti nutraukta tik už kritinės sekcijos ribų.

Tannenbaumo knygoje "Operating systems design and implementation" kritinės sekcijos apsaugai pateikti reikalavimai truputi kitokie: 1. Du procesai vienu metu negali būti kritinėje sekcijoje 2. Sprendžiant problemą negalima daryti prielaidų apie procesų vykdymo greičius 3. Procesai vykdydami komandas nekritinėje sekcijoje neturėtų blokuoti kitų procesų 4. Procesai tam, kad patektų į kritinę sekciją, neturėtų laukti be galo ilgai.

Tegu 2 procesai turi bendrą kintamąjį (EILĖ), kuris nurodo, kurio proc. eilė kreiptis į atm. P1: jei EILĖ =2 tada blokuokis, kitu atveju: KS1; EILĖ = 2; PROG1; į pradžią. P2 – atvirkščiai. Sprendimas negeras, nes blokuojamas kritinės sekcijos vykdymas, kai kitas procesas nėra kritinėje sekcijoje (Tannenbaumo knygos 3 reikalavimas). Jei vykdymo laikas P1>>P2 tai P2 ilgai lauks, kol P1 įvykdys savo PROG1, kad galėtų vykdyti savo KS2:

```
BEGIN INTEGER EILĖ:=2;   P1: BEGIN L1: IF EILĖ=2 THEN GOTO L1;                KS1;
EILĖ:=2;                 PROG1;                 GOTO L1;       END   P2: BEGIN L2:
IF EILĖ=1 THEN GOTO L2;                KS2; EILĖ:1;                 PROG2;
GOTO L2;       END;
```

Įveskime du bendrus kintamuosius: Ci = FALSE, kai vykdoma KSi. P1: jei vykdoma KS2, tai blokuokis, kitu atveju: C1=FALSE; KS1; C1=TRUE; PROG1, į pradžią. P2 – atvirkščiai. Bet šiuo atveju egzistuoja tikimybė, kad abu procesai pradės vykdyti KS vienu metu:

```
BEGIN BOOLEAN C1:= TRUE; C2:= TRUE;   P1: BEGIN L1: IF NOT C2 THEN GOTO L1;
C1: FALSE; KS1; C1:=TRUE;                PROG1;                GOTO L1;       END
and   P2: BEGIN L2: IF NOT C1 THEN GOTO L2;                C2:=FALSE; KS2;
C2:=TRUE;                PROG2;                GOTO L2;       END; END
```

Problemą išsprendžia Dekerio algoritmas.

Paprasčiau problemą išspręsti naudojant Semaforus:

Tegul M – kritinę sekciją apsaugantis semaforas, n – procesų skaičius.

```
BEGIN SEMAPHORE M;   M:=1 //pradinė reikšmė   P1: BEGIN…END   and   …   P_i: BEGIN
L_i: P(M); KS_i; V(M); PROG_i; GOTO L_i; END   …   and   P_n: BEGIN…END END;
```

Dekerio algoritmas

Procesas atžymi savo norą įeiti į KS loginiu kintamuoju C_i=false. Išėjus iš kritinės sekcijos C_i=true. Įeiti į KS procesas gali tik tada, kai kitas procesas nėra KS‘je arba nėra pareiškęs noro ją vykdyti. Sveikas kintamasis EILE naudojamas tada, kai du procesai susiduria KS‘je (pvz.: noras vykdyti KS, įėjimas į KS). Šis kintamasis parodo, kurio proceso eilė vykdyti KS. Proc. kuris neturi eilės vykdyti KS atsisako savo noro.

```
BEGIN
```

```
INTEGER EILE;     BOOLEAN C1, C2;     C1:=C2:=true; EILE:=1; P1: BEGIN A1:
C1:=false;  L1:   IF not C2        THEN          BEGIN           IF EILE=1 THEN
GOTO L1;            C1:=true;           B1: IF EILE =2 THEN GOTO B1;           GOTO
A1;         END;       KS1;       EILE:=2;        C1:=true;       PROG1;       GOTO
A1;     END    AND  P2: BEGIN A2:   C2:=false; L2:   IF not C1        THEN
BEGIN           IF EILE=2 THEN GOTO L2;           C2:=true;           B2: IF EILE=1
THEN GOTO B2;           GOTO A2;         END;       KS2;       EILE:=1;
C2:=true;       PROG2;       GOTO A2     END END.
```

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Toks sprendimas per sudėtingas, kad juo remiantis būtų galima toliau organizuoti darbą. Netinka, nes nėra tinkamų primityvų.

Postuluoti primityvus yra maža. Svarbu – realizacija.

Reikia apibrėžti aparatą, tinkantį lygiagrečiam vykdymui – semaforus.

Semaforai

Semaforas S tai sveikas neneigiamas skaičius, su kuriuo atliekamos operacijos P(S) ir V(S), kur P ir V nauji primityvai. Operacijos pasižymi savybėmis:

1. P(S), V(S) – nedalomos operacijos, t.y. jų valdymo negalima pertraukti ir jų vykdymo metu negalima kreiptis į semaforą S;
2. V(S): S:S+1; (didinama semaforo reikšmė)
3. P(S): S:S-1; (sumažinama jei S>0)
4. Jei S=0, tai procesas P, kuris vykdo operaciją P(S), laukia, kol sumažinimas vienetu bus galimas. Šiuo atveju P(S) yra pertraukiamas

5. Jei keletas procesų vienu metu iškviečia V(S) ir/ar P(S) su vienu semaforu, tai užklausimai vykdomi nuosekliai, kokia nors iš anksto nežinoma tvarka.
6. Jei keletas procesų laukia operacijos P(S) įvykdymo tam pačiam semaforui, tai reikšmei tapus teigiamai (kai kažkuris kitas procesas įvykdė V(S) operaciją), kažkuris iš laukiančių procesų bus pradėtas vykdyti.

Pagal prasmę operacija P atitinka perėjimo iškvietimą, o V – kito proceso aktyvaciją.

Jei semaforas įgyja tik dvi reikšmes 0 ir 1, tai jis vadinamas dvejetainiu, jei bet kokias - bendriniu.

Pvz. Semaforus galima naudoti procesų sinchronizacijai. Turime du procesus, norime, kad antras pradėtų vykdyti savo programą tuomet, kai pirmas pasiųs jam atitinkamą signalą.

Procesų „gamintojas“ ir „naudotojas“ sąveika

Procesas gamintojas sukuria informaciją ir patalpina ją į buferį, o lygiagrečiai veikiantis kitas procesas naudotojas paima informaciją iš buferio ir apdoroja.

Problema atsiranda, kai gamintojas nori patalpinti informaciją į buferį, bet jis jau yra pilnas. Sprendimas yra pasiųsti gamintoją miegoti ir pažadinti tada, kai naudotojas paims bent vieną informacijos vienetą iš buferio. Analogiška problema iškyla, kai vartotojas nori paimti informaciją iš buferio, kuris yra tuščias.

Tegul buferio atmintis susideda iš N buferių.

Semaforas T – tuščių buferių skaičius.

Semaforas U – užimtų buferių skaičius.

B – semaforas, saugantis kritinę sekciją, atitinkančią veiksmus su buferio sekcijomis.

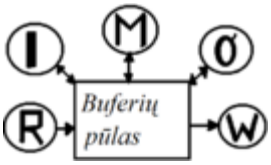
```
BEGIN SEMAPHORE T,U,B;       T:=N; U:=0; B:=1; // B=1 - kritinė sekcija nevykdoma,
B=0 - vykdoma
 GAM:  BEGIN   LG:    įrašo gaminimas;         P(T);          P(B);
užrašymas į buferį;          V(B);         V(U);           GOTO LG;       END

       AND

NAUD:  BEGIN   LN:    P(U);          P(B);          paėmimas iš buferio;
V(B);         V(T);          įrašo apdorojimas;         GOTO LN;       END
END;
```

Įvedimo-išvedimo spuleris. Bendra charakteristika

Spuleris (angl. *spooler*) – OS dalis atliekanti I/O virtualizaciją, kuomet yra sukaupiama įvedama informacija, ji apdorojama ir išvedama kaip rezultatas. Spuleris 5 procesų visuma. Visi jie yra procesoriuje vykdomi procesai.



R–Read; I–Input; M–Main; O–Out; W–Write;

Buferių pūlas – sąrašas (faktiškai 3 sąrašai – laisvų, įvedimo ir išvedimo buferių). Iš pradžių buferių pūlas apjungtas į laisvų buferių sąrašą, o įvedimo ir išvedimo sąrašai tušti. Apdorojimo proceso užduotis paruošti informaciją išvedimui.

Darbui su buferiais reikia apsirašyti tokius parametrus: laisvų buferių skaičių; įved/išved buferių skaičių; KS apsaugos semaforą; įved/išved buferių KS apsaugos semaforus.

nl – laisvų, nin – įvedimo, nout – išvedimo buferių skaičius; ml – laisvų, min – įvedimo, mout – išvedimo buferių KS apsaugos semaforas.

Spūlerio veikimas:

```
BEGIN  nl := n;  nin := nout := 0;  SR := FR := SW := FW := 0;  M and I and O and R and W; // Main dirba lygiagrečiai su I,O,R,WEND;
```

Įvedimo-išvedimo spuleris. Pagrindinis procesas

1. Ima buferį iš įvedimo buferių sąrašo;
2. Apdoroja jame esančią informaciją;
3. Ima buferį iš laisvų buferių sąrašo.
4. Jį užpildo išvedama informacija;
5. Buferį įjungia į išvedimo buferių sąrašą.

Pagrindinio proceso M programa:

```
M:  BEGINLM:   P(nin); // M turi imti įvedimo buferį: Jei tokio nėra, tai M blokuojasi      P(min); // jei kitas procesas vykdo įvedimą, tai M blokuojasi Paimti pirmą buf. iš įvedimo buf. sąrašo;      V(min); // nuimama įvedimo KS apsauga      Apdoroti buferio turinį;      P(nl);      P(ml);      Paimti pirmą buf. iš laisvų buf. sąrašo;      V(ml);      Užpildyti paimtą buf. išvedama info; P(mout); // išvedimo buferio KS apsauga      Įjungti buf. į išved. buf. sąrašą; // imama iš sąrašo pradžios, bet rašoma į galą.      V(mout); !!!      V(nout); P(ml); // buvo paimtas įved. buferis. Jį reikia atlaisvinti. Tai darbas su KS; Įjungti buferį į laisvų buf. sąr;      V(ml);      V(nl);      GOTO LM;    END;
```

Įvedimo/išvedimo spuleris. Įvedimo ir skaitymo procesas.

Įvedimo procesas I gali vykti tik tada, kai yra tuščių buferių. Atitinkamai M gali vykti, kai yra įvedimo sąraše elementai ir laisvi buferiai. Jei I ima paskutinį laisvą buferį, tai M užsiblokuoja, nes jam nėra laisvų buferių.

Atliekant įvedimo veiksmą, darbas turi būti sinchronizuojamas su įvedimo įrenginiu. Įvedimo įrenginio darbo pradžios ir pabaigos situaciją (realiai – pertraukimo situacija) modeliuosime semaforais. SR – skaitymo įrenginio startas; FR – skait. įreng. finišas.'

Procesas R vykdo P(SR), o I – V(SR). Jei SR=0, tai V(SR)=1 ir procesas R galės baigti darą. I blokuojasi nuo P(FR), o procesą R paleidžia V(FR).

I ir R galima sinchronizuoti bendram darbui. I turi aprūpinti skaitymo įrenginį tuščiais buferiais, inicijuoti R ir prijungti į pūlą užpildytus per įvedimą buferius, kai R baigia darbą.

Įvedimo proceso valdymas

```
I:  BEGIN
LI:   P(nl);      P(ml);      IF liko paskutinis laisvas buf. THEN
      BEGIN
        V(nl);        V(ml);        GOTO LI;      END;      Paimti buf. iš laisvų
buf. sąrašo;      V(ml);      V(SR);      P(FR)
      P(min);      Prijungti įvedimo buf. prie įvedimo buf. sąrašo;      V(min);
V(nin);      GOTO LI;    END.
```

Skaitymo proceso valdymas

```
R:  BEGIN  LR:   P(SR);       Perskaityti į nurodytą buf.;       V(FR);       GOTO
LR;     END.
```

Proceso I veiksmų seka yra tokia:

- įvedimo procesas paima buferį iš laisvų buferių sąrašo;
- paleidžia skaitymo procesą R ir perduoda jam laisvą buferį;
- užpildytą buferį įjungia į įvedimo buferių sąrašą.

Įvedimo procesas I gali vykti tik tada, kai yra tuščių buferių ir lieka bent vienas laisvas, nes jo reikia procesui M.

Įvedimo-išvedimo spuleris. Išvedimo ir rašymo procesai

Atliekant išvedimo veiksmą, darbas turi būti sinchronizuojamas su išvedimo įrenginiu. Išvedimo įrenginio darbo pradžios ir pabaigos situaciją (realiai – pertraukimo situacija) modeliuosime semaforais. SW – išvedimo įrenginio startas; FW – išvedimo. įreng. finišas.'

Išvedimo proceso valdymas

```
O:  BEGIN LO:   P(out);       P(mout);       Paimti pirmą buferį iš išvedamų buf.
sąr.;       V(mout);       V(SW);       P(FW);       P(ml);       Prijungti
atlaisvintą buferį prie laisvų buf.sąr.;       V(ml);       V(nl);       GOTO LO;
END.
```

Rašymo proceso valdymas

```
W:  BEGIN LW:   P(SW);       Užrašyti iš nurodyto buf.;       V(FW);       GOTO LW;
END.
```

Dvejetainių ir bendrų semaforų sąryšis

Semaforinių primityvų relizacijai, reikia bendruosius semaforus išreikšti dvejetainiais. Jei semaforas gali įgyti tik dvi reikšmes – jis dvejetainis. Bet kuris bendras semaforas gali būti išreikštas dvejetainiu semaforu.

Tegu S – bendrasis semaforas. Tada jis gali būti pakeistas kintamuoju NS ir dviem dvejetainiais semaforais M ir D (M – kritinę sekciją apsaugantis semaforas).

P(S) – nedaloma operacija, M – kritinę sekciją apsaugantis dvejetainis semaforas.

Pradiniai M:=1; NS:= S; D:= 0;

```
P(S) ~ BEGIN          P(M); // KS apsauga nuo dvejetainio semaforo M
NS := NS-1;          IF NS <= -1 THEN // turi įvykti blokavimasis
BEGIN            V(M); // nuimama apsauga, nes kitu atveju, kitas procesas
kreipdamasis negalėtų atlikti P(S) veiksmo.
P(D); // turi iššaukti blokavimą;
END          ELSE V(M);        END.
```

P(S) atvejai:

- S > 0 – nuimama KS apsauga ir neiššaukiamas laukimas;
- S = 0 – turi įvykti perėjimas į laukimą, tada bus pasiekiamas procesas V(S).

NS < 0 – parodo laukiančių procesų skaičių.

```
V(S) ~ BEGIN          P(M);          NS := NS+1;          IF NS <= 0 THEN V(D); //
yra laukiančių procesų, kurie užsikodavę semaforu D
  V(M);        END.
```

RS; PRAŠYTIR(RS,D,A) ATLAISVINTI(RS,D);

Semaforo mechanizmas reiškiamas taip:

```
P(S) ~ PRAŠYTIR(S,Ω,Ω);
 V(S) ~ ATLAISVINTIR(S,Ω);
```

Operacijų su semaforais realizacija

Paprastai komandų sistema neturi operacijų P ir V. Norint realizuoti semaforinius primityvus, reikia kad kompiuterio architektūra leistų nedalomu veiksmu *patikrinti ir pakeisti* žodžio reikšmę x - PP(x).

```
BOOLEAN PROCEDURE PP(BOOLEAN var x);
 BEGIN
   PP:=x;
   x:= TRUE;
 END;
```

Dabar pasinaudosime bendrų ir dvejetainių semaforų ryšiais. Turėdami bendrą semaforą S, P(S) ir V(S) norėtumėme išreikšti komandomis: priskirti ir patikrinti.

- NS - semaforo skaitinė reikšmė
- MS - KS apsauga
- DS - semaforo blokavimas

```
P(S) ~ BEGIN
    L1:  IF PP(MS) THEN GOTO L1; // vyksta laukimas
         NS:= NS - 1;
         IF NS <= -1 THEN
         BEGIN
           MS:= FALSE
           L2: IF PP(DS) THEN GOTO L2;
         END
         ELSE MS:= FALSE;
       END;
```

```
V(S) ~ BEGIN
       L3:  IF PP(MS) THEN GOTO L3; // vyksta laukimas
            NS:=NS+1;
            IF NS<=0 THEN DS:= FALSE;
            MS:= FALSE;
          END;
```

Operacijų su semaforais realizacija be užimančio laukimo

Kad būtų padidintas sistemos efektyvumas reikia, kad P procesą blokuotų, o V jį aktyvuotų.

Blokuoti procesą reiškią – nuo jo vykdymo atlaisvinamas procesorius ir proceso P būsena įjungiama į blokuotų pagal semaforą S procesų sąrašą.

Su V(S) atvirkščiai.

P(S) ir V(S) programinės realizacijos

Operacija P(S), kai S – bendrasis semaforas, ekvivalenti tokiai programinei konstrukcijai:

```
P(S) ~ BEGIN
          uždrausti pertraukimus;
          L: IF PP(x) THEN GOTO L;
          S:=S-1;
          IF S<=-1 THEN
          BEGIN
            užblokuoti iškviečiantį procesą; // (2),(1)
            Paimti procesą iš sąrašo LIST_A; // (2)
            X:= FALSE;                       // (2)
            Perduoti valdymą;                // (2)
          END
          ELSE X:=FALSE;
          Leisti pertraukimus;
        END;
V(S) ~ BEGIN
          Uždrausti pertraukimus;
     L:   IF PP(x) THEN GOTO L;
          S:=S+1;
          IF S<=0 THEN // (3)
          BEGIN
            Paimti procesą iš LIST_BS -> LIST_A;
            IF procesorius laisvas THEN vykdyti procesą iš LIST_A;
          END;
          X:= FALSE;
          Leisti pertraukimus;
        END.
```

(1) Tai reiškia įjungimą į blokuotų procesų sąrašą $LIST_{BS}$, kur s reiškia nuo semaforo S.
(2) Faktiškai tai yra planuotojo darbas.
(3) Suaktyviname vieną iš blokuotų semaforo S procesų.

Procesų ir resursų sąvoka

Kiekvienai naudotojo užduočiai j sukuriamas procesas Pj, kuris tvarko užduoties naudojamų resursų aprašymą. Aprašymas susideda iš statinės ir dinaminės informacijos. Statinė – max laikas. Dinaminė –

duotu laiko momentu naudojami resursai. Kiekvienas procesas gali kurti kitus procesus. Gauname procesų medį.

Proceso deskriptorius (PD)

PD – (proceso) veiklumo stadiją apibūdinantis proceso aprašas. Apraše ir yra laikomi visi procesui reikalingi parametrai: virtualus procesorius, registrų reikšmės ir jam reikalingi kintamieji. Procesų aprašai dinaminiai objektai – jie gali būti sukurti/sunaikinti sistemos veikimu metu. Realiai procesą, kaip ir resursą OS-je atstovauja Deskriptorius. PD – tai tam tikra struktūra(ne masyvas) – jei kalbame apie visų procesų deskriptorius, tai turime struktūrų masyvą, kur *i* proceso vidinis vardas – nurodytų struktūros numerį masyve. PD – susideda iš komponenčių, kurioms priskiriame vardus:

1. *Id[i]* – proceso išorinis vardas, reikalingas statiniams ryšiams tarp procesų nurotyti.
2. Mašina – čia turime omeny procesą vykdančio procesoriaus apibūdinimą.
    1. *CPU[i]* – apibūdina centrinio procesoriaus būseną ir teises vykdant procesą. Kai proceso vykdymas nutraukiamas, proceso būsena išsaugoma.
    2. *P[i]* – identifikuoja procesorių
    3. *OA[i]* – operatyvi atmintis turima i-tojo proceso (nuoroda į sąrašą)
    4. *R[i] – i-jo* proceso turimi resursai – informacija, kokius resursus yra gavęs procesas. Nuoroda į sąrašą, kuriame yra išvardinta resursai.
    5. *SR[i]* – Sukurtų resursų sąrašas (resursų deskriptorių).
3. informacija apie proceso būseną:
    1. ST[i] – proceso statusas (RUN, READY, READYS, BLOCK, BLOCKS)
    2. SD[i] – nuoroda į sąrašą kuriame yra procesas (PPS – pasiruošusių proc.sąr., LPS – laukiančių proc.sar.)
4. proc.sąryšis su kitais procesais
    1. T[i] – i-tojo proceso tėvas (tėvinio proc.vidinis vardas)
    2. S[i] – nuoroda į i-tojo proceso vaikinių proc.vidinių vardų sąr.
5. PR[i] – prioritetas.

Resurso deskriptorius

Resurso deskriptorius (Resurso valdymo blokas) yra fiksuoto formato duomenų struktūra, sauganti informaciją apie resurso einamąjį stovį. Remiantis informacija resurso deskriptoriuje nurodomas jo užimtumo laipsnis, laisvas kiekis, nuoroda į pačius resurso elementus ir kt. Šia informacija naudojasi duotojo resurso paskirstytojas. Resurso inicializavimas reiškia deskriptoriaus sukūrimą. Darbas su deskriptoriais galimas tik per specialias operacijas - OS branduolio primityvus.

r-resurso vidinis vardas (indeksas resursų deskriptorių masyve)

1. identifikacija.
    1. Rid[r] – išorinis vardas
    2. PNR[r] – ar tai pakartotinio panaudojimo resursas (jei taip - jį reikia gąžinti)
    3. K[r] – resursą sukūrusio proceso vidinis vardas
2. PA[r] – nuoroda į prieinamumo aprašymo sąrašo pradžią.
3. LPS[r] – nuoroda į resurso laukiančių procesų sąrašo prad.
4. PASK[r] – resurso paskirstytojo programos adresas.

Primityvas „kurti procesą“ (KURTIP)

Sukuriamas naujas deskriptorius masyve. Procedūra įvykdoma iššaukiančio proceso aplinkoje.

PROCEDURE KURTIP(n,s0,M0,R0,k0);

n – išorinis vardas

S0 – procesoriaus pradine būsena;

M0 – OA pradinė būsena (kiek išskirta OA resursu);

R0 – kiti išskiriami resursai;

K0 – proceso prioritetas

* – duotu metu dirbantis procesas

```
PROCEDURE KURTIP(n,s0,M0,R0,k0);
BEGIN
  i:=NVV; // NVV - naujas vidinis vardas
  Id[i]:=n;
  CPU[i]:=s0;
  OA[i]:=M0;
  R[i]:=R0; // kitų turimų proceso resursų komponente (nuoroda į sąrašą);
  PR[i]:=k0;
  ST[i]:=READYS; // laikysime, kad procesas sukuriamas su statusu (pvz.
pasiruošes)

  SD[i]:=PPS;
  T[i]:=*;
  S[i]:=Λ; // Sūnų sąrašas iš pradžių jis tuščias
  Įrašyti(s[*],i); // * įgavo naują sūnų i;
  Įjungti(PPS,i);
END;
```

Primityvas „naikinti procesą“ (NAIKINTIP)

Procesas naikina visus savo palikuonis (sūninius procesus), bet negali sunaikinti savęs. Tada jis nusiunčia pranešimą savo tėvui, kuris jį ir sunaikina.

Pakartotino naudojimo resursus reikia atlaisvinti.

```
PROCEDURE NAIKINTIP(n);
BEGIN
  L:= FALSE; // jei L = true – yra vykdomų palikuonių, tai iškviečiamas
planuotojas;
  i:=VV(n);
  P:=T[i];
  NUTRAUKTI(i); // panaikina visus proceso palikuonis
  Pašalinti(S[T[i]], i);
  IF L THEN PLANUOTOJAS;
END;
PROCEDURE NUTRAUKTI(i);
BEGIN
  IF ST[i] = RUN THEN
  BEGIN
    STOP(i);
    L:=true;
  END;
  Pašalinti(SD[i], i); // pašalinti iš pasiruošusių arba laukiančiųjų procesų
sąr.
  FOR ALL s ϵ S[i] DO NUTRAUKTI(s); // reikia nutraukti i-tojo proceso sūnų
vykdymą
  FOR ALL r ϵ OA[i] v R[i] DO // r – resusrsai. Atlaisviname pakartotinio
panaudojimo resursus
    IF PNR THEN Įrašyti(PA[r], Pašalinti(OA[i] VR[i])); //Jeigu buvo pakartotinai
panaudojamas resursas, jį įjungia į Prieinamumo sąrašą
  FOR ALL r ϵ SR[i] DO NDR(r); //naikinami resursų deskriptoriai
  NPD(i); //naikiname proceso deskriptorių
END;
```

Primityvas „Stabdyti procesą“ (STABDYTIP)

Tėvinis procesas gali stabdyti vaikinį nestabdant vaikinio palikuonių.

n – išorinis vardas
a – adresas į proceso deskriptoriaus būseną (jo kopiją)

```
Procedure STABDYTIP(n,a);
begin
  i:=VV(n); // pagal IV nustatomas proceso VV
  S:=ST[i]; // einamasis proceso statusas
  if S=RUN then STOP(i); // (1)
  a:=copydesc[i];
  if S=BLOCK or BLOCKS then ST[i]:=BLOCKS
                       else ST[i]:=READYS;
  if S=RUN then PLANUOTOJAS; // (2)
End;
```

(1) STOP(i) įvykdo veiksmus:

1. Pertraukia procesorių P[i]
2. Įsimena pertraukto procesoriaus P[i] būseną proceso deskriptoriaus komponentėje CPU[i]
3. PROC[P[i]]:= Λ - pažymima, kad procesorius P[i] yra laisvas

(2) Jei buvo pristabdytas vykdomas procesas, tai reikia iškviesti planuotoją, nes galbūt kiti procesai laukia procesoriaus. Planuotojo vykdymui naujas procesas nesukuriamas.

Svarbu, kad iki planavimo veiksmo viskas jau būtų padaryta, nes planuotojas atims procesorių iš šito paties proceso, kuriame planuotojas buvo iškviestas.

Primityvas „aktyvuoti procesą“ (AKTYVUOTIP)

Nuima pristabdymo buseną. Galbūt iškviečia planuotoją, jei būsena yra READY.

```
PROCEDURE AKTYVUOTI(n);
BEGIN
  i:=VV(n);
  IF ST[i]=READYS THEN ST[i]:=READY
                  ELSE ST[i]:=BLOCK; // buvo BLOCKS
  IF ST[i]=READY THEN PLANUOTOJAS;
END;
```

Primityvas „keisti proceso prioritetą“ (KEISTIPP)

Proceso įterpimas į sąrašą vyksta atsižvelgiant į proceso prioritetą, todėl proceso prioriteto pakeitimas vyksta taip: pašalinamas iš sąrašo ir po to įterpiamas pagal naują prioritetą.

n - IV
k - naujas prioritetas

```
PROCEDURE KEISTIPP(n,k);
BEGIN
  I:=VV(n);
  M:=PR[i];
  Pašalinti(SD[i],i);
  PR[i]:=k;
  Įrašyti(SD[i],i);
  If M<k and ST[i]=READY THEN PLANUOTOJAS;
END;
```

Primityvas „kurti resursą“ (KURTIR)

Resurso deskriptoriaus sukūrimas. Resursus kuria tik procesas.

Parametrai:

- RS – resurso IV (Išorinis Vardas)
- PNR - ar res. yra iš naujo panaudojamas
- PA_0 – res.prieinamumo aprašymas
- LPS_0 – to resurso laukiančių proc. sąr.
- $PASK_0$ – res. paskirstutojo programos adresas
- * - Resurso kūrėjo VV. Tai tuo metu vykstantis procesas, kuriame ir panaudotas šis primityvas.

```
Procedure KURTIR (RSo, PNRo, PAo, LPSo, PASKo);
Begin
  r:=NRVV; // naujas resurso vidinis vardas
```

```
  Rid[r]:=RSo;
  PNR[r]:=PNRo;
  k[r]:=*;
  PA[r]:=PAo;
  LPS[r]:=LPSo;
  PASK[r]:=PASKo;
  Įrašyti (SR[*], r); // (1)
End;
```

(1) Į šiuometu vykdomo proceso sukurtų resursų sąrašą įtraukiamas naujai sukurtas resursas.

Primityvas „naikinti resursą“ (NAIKINTIR)

Sunaikinti resursa gali jo tevas arba pirmtakas. Sunaikinamas resurso deskriptorius, prieš tai suaktyvinami jo laukiantys procesai.

R.P - proceso visinis vardas

```
PROCEDURE NAIKINTIR(RS);
Begin
  r:=RVV(RS);
  //atblokuojami resurso laukiantys proc.jiems pasiunčiant fiktyvų res.//
  R:=Pašalinti(LPS[r]); // turėsim nuorodą į pašalinamą elementą ir nuorodą į
procesą
  While R<>1 do Begin

    IF ST[R.P]=BLOCK then ST[R.P]:=READY /* dirbtinai atblokuojam procesą
                     else ST[R.P]:=READYS;
    Įrašyti(PPS,R.P); // gal turėtų būti Įjungti(PPS, R.P)?
    SD[R.P]:=PPS;
    R.A:='PRANEŠIMAS';
    R:=Pašalinti(LPS[r]);
  End;
  NRD(r); // Naikinti resurso deskriptoriu
  PLANUOTOJAS;
End
```

Primityvas “prašyti resurso” (PRAŠYTIR)

Procesas, kuriam reikia resurso, iškviečia šį primityvą, nurodydamas išorinį vardą ir adresą. Toks procesas pereina į blokavimosi būseną. Blokavimasis įvyksta tik prašant resusrso. Procesas įjungiamas į laukiančių to resusro procesų sąrašą.

- RS - resurso IV
- D – kokios resusrso dalies prašoma
- A – atsakymo srities adresas, į kur pranešti
- * - šiuo metu veikiantis procesas

```
PROCEDURE PRASYTIR(RS, D, A);
BEGIN
  r := RVV(RS);
  IJUNGTI(LPS[r], ( *, D, A)); // procesas įjungiamas į laukiančių šio resurso
procesų sąrašą
  PASK(r, K, L); // Resurso paskirstytojo programa
```

```
                  // K - kiek procesų aptarnauti
                  // L - aptarnautų procesų vidinių vardų masyvas
  B := true; // Ar masyve L yra einamas procesas?
  FOR J :=1 STEP 1 UNTIL K DO
    IF L[J] <> * THEN
    BEGIN
      i:=L[J];
      Įrašyti(PPS, i);
      SD[i]:=PPS;
      IF ST[i]=BLOCK THEN ST[i]:=READY
                     ELSE ST[i]:=READYS;
    END ELSE
      B:=false; // Procesas * iš karto gavo resursą ir nebuvo įrašytas į
laukiančių procesų sąr.

  IF B THEN
  BEGIN
    ST[*]:=BLOCK;
    SD[*]:=LPS[r];
    PROC[P[i]]:= Λ; // Procesorius laisvas
    PASALINTI(PPS, *)
  END;
  PLANUOTOJAS;
END;
```

Primityvas „atlaisvinti resursą“ (ATLAISVINTIR)

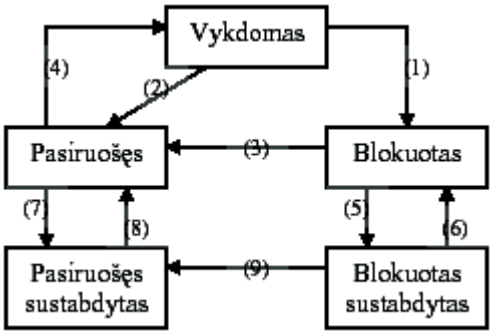
Tai atitinka situaciją, kai procesas gauna pakartotino naudojimo resursą ir kai jam nereikia, jis jį atlaisvina ir įjungia į laisvųjų resursų sąrašą bei papildo resurso prieinamumo aprašymo sąr.

```
PROCEDURE ATLAISVINTIR(RS,D);
BEGIN
  r:=RVV(RS);
  Ijungti(PA[r],D); // D - atlaisvinamos resurso dalies apibūdinimas
  PASK(r,k,L);
  IF k>0 THEN FOR J:= 1 STEP 1 UNTIL k DO
  BEGIN
    i:=L(J);
    Ijugti(PPS,i);
    SD[i]:=PPS;
    IF ST[i]= BLOCKS THEN ST[i]:= READYS
                     ELSE ST[i]:= READY
  END;
  IF k<>0 THEN PLANUOTOJAS;
END;
```

Procesų būsenų schema

Procesas gali gauti procesorių tik tada, kai jam netrūksta jokio resurso. Procesas gavęs procesorių tampa vykdomu. Procesas, esantis šioje busenoje, turi procesoriu, kol sistemoje neįvyksta pertraukimas arba einamasis procesas nepaprašo kokio nors resurso (pavyzdžiui, prašydamas ivedimo iš klaviaturos). Procesas blokuojasi priverstinai (nes jis vis tiek negali tęsti savo darbo be reikiamo resurso). Tačiau, jei procesas nereikalauja jokio resurso, iš jo gali būti atimamas procesorius, pavyzdžiui, vien tik dėl to, kad pernelyg ilgai dirbo. Tai visiškai skirtinga busena nei blokavimasis del resurso (neturimas omeny resursas - procesorius). Taigi, galime išskirti jau žinomas procesu busenas: vykdomas – jau turi

procesorių; blokuotas – prašo resurso(bet ne procesoriaus); pasiruošęs – vienintelis trūkstamas resursas yra procesorius; sustabdytas – kito proceso sustabdytas procesas.

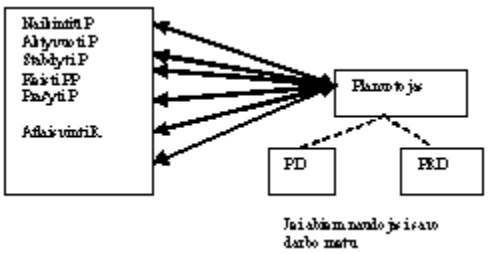


(1) Vykdomas procesas blokuojasi jam prašant ir negavus resurso.
(2) Vykdomas procesas tampa pasiruošusiu atėmus iš jo procesorių dėl kokios nors priežasties (išskyrus resurso negavimą).
(3) Blokuotas procesas tampa pasiruošusiu, kai yra suteikiamas reikalingas resursas.
(4) Pasiruošę procesai varžosi dėl procesoriaus. Gavęs procesorių procesas tampa vykdomu.
(5) Procesas gali tapti sustabdytu blokuotu, jei einamasis procesas ji sustabdo, kai jis jau ir taip yra blokuotas.
(6) Procesas tampa blokuotu iš blokuoto sustabdyto, jei einamasis procesas nuima buseną sustabdytas.
(7) Procesas gali tapti pasiruošusiu sustabdytu, jei einamasis procesas ji sustabdo, kai jis yra pasiruošęs.
(8) Procesas tampa pasiruošusiu iš pasiruošusio sustabdyto, jei einamasis procesas nuima buseną sustabdytas.
(9) Procesas tampa pasiruošusiu sustabdytu iš blokuoto sustabdyto, jei procesui yra suteikiamas jam reikalingas resursas.

Procesų planuotojas

Planuotojas užtikrina, kad visi procesai būtų vykdomi maksimaliu prioritetu (Jei prioritetai vienodi, tai vykdomas tas kuris sąraše pirmesnis).

Planuotojo veikimo schema: (Planuotojas iškviečiamas iš branduolio primityvų)



Planuotojo darbas 3 etapais: 1) Surasti procesą su aukščiausiu prioritetu. 2) Surasti neaktyvų procesorių ir jį išskirti.(skirti pasiruošusiam procesui) 3) Jei procesorių nėra, peržiūrėti vykdomus procesus, ir, jei jų prioritetai mažesni, atiduot procesorių pasiruošusiems (Perskirstymas)

Pasiruošusio proceso su aukščiausiu prioritetu nustatymas ir neaktyvaus procesoriaus radimas

Kol neperžiūrėti visi pasiruošusių procesų sąrašai ir prioritetas nelygus nuliui darom: jei peržiūrimas sąrašas tuščias, tai prioritetas mažinamas vienetu ir imamas to prioriteto pasiruošusių proc.sąr.

Pradedame nuo 1 ir sukame ciklą kol pasiekiame procesorių skaičių. Tikriname ar procesoriui einamasis procesorius turi vadą. Jei turi (t.y. yra užimtas) sukame ciklo skaitliuką. Radus procesą be vardo, t.y. neaktyvų, jam priskiriamas procesas. Jei tai procesorius, kuris laisvas dėl to, kad negavęs resurso, tai įsimenama kuriam procesui reikės jį atiduoti.

Procesų planuotojo procesoriaus perskirstymo etapas

Perskirstant procesorių reikia iš proceso su mažesniu prioritetu atimti procesorių. Pirmiausiai prasukame visus procesus ir surandame, kuris turi mažiausią prioritetą ir mažesnį negu mūsų proceso įsimename to proceso vidinį vardą. Iš to procesoriaus atimame procesorių ir jį atiduodame mūsų norimam procesui

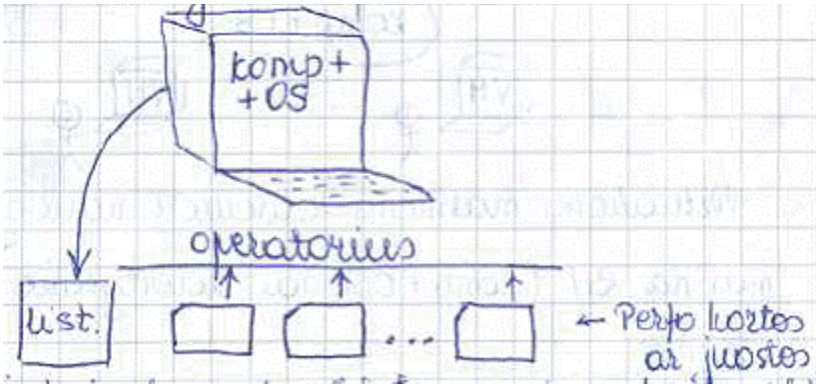
OS (kompiuterių) architektūros raida

I

Žmogus betarpiškai valdo kompiuterį. Darbas vyksta mašinine kalba. Kompiuterio panaudojimas neefektyvus (labai brangi technika). Kompiuterių resursų nėra.

II

Sprendžiama efektyvumo problema: žmogus nutolinamas nuo kompiuterio, paliekant jam kompiuterio teikiamas galimybes. Programuotojai suformuluoja savo užduotis kaip programų paketus, kurie perduodami operatoriui, o šis perduoda juos kompiuteriui.

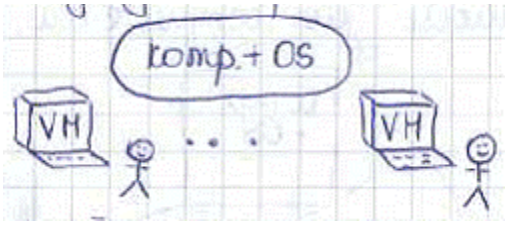


Kompiuteris buvo papildytas OS tam, kad galima būtų bendrauti su vartotoju. Tokia OS priima visą paketą užduočių ir grąžina rezultatus (listingus) programuotojui. Reikėjo pateikti visus būtinus

duomenis. Ši paketinio apdorojimo OS buvo neefektyvi žmogaus darbo atžvilgiu (pvz.: kai programos buvo užrašomos ant popieriaus, perduodamos į skaičiavimo centrą, kur programuotojai suvesdavo jas į kompiuterį, ten kompiliuojamos ir listingai buvo grąžinami rašiusiam programą asmeniui).

III

Virtualių mašinų etapas. Tikslas: sugrąžinti žmogui kompiuterį (I etapas) paliekant efektyvų technikos panaudojimas (II etapas). Čia su kompiuteriu žmogus bendrauja jau suprantama kalba.

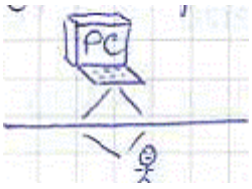


Virtualioms mašinoms realizuoti naudojama reali mašina RM (kompiuteris + OS). Tai kolektyvinio naudojimo OS. OS iš II-o ir III-io etapų yra multiprogramines OS. Skirtumas: kai II-me etape persijungiama nuo vienos užduoties vykdymo prie kitos, tai pertraukimas įvyksta dėl vidinių kompiuterio priežasčių, o III-me etape pertraukimas įvyksta dėl tos pačios priežasties bei dėl laiko kvanto pasibaigimo. Jei užduotis sudėtinga (pvz. transliavimas), tai žmogui suprantami yra pertraukimai. Tampa patogu dirbti su dideliais resursais. Šis etapas labai brangus (pastarosios temperatūros palaikymas, energijos sąnaudos ir t. t.).

IV

Situacija pasikeičia atsiradus PK (personaliniams kompiuteriams). Grįžtama prie I-mo etapo: jokių multiprogramininių OS.

Rimti uždaviniai atliekami kaip ankščiau (III etapas), o smulkūs – kaip I-me etape. Žmogui norėjosi vienu metu daryti daug darbų. Iškilo poreikis turėti keletą virtualių mašinų vienam žmogui. Tam tikslui grįžtama prie multiprogramininių OS. Tai šiandieninė būsena.



Mikrobranduolio architektūra

Mikrobranduolys yra OS nedidelė atminties dalis, įgalinanti OS modulinį plėtimą. Nėra vieningos mikrobranduolio sandaros. Problema yra driver'iai, juos reikia padaryt efektyvius. Į driver'į galima žiūrėti kaip į virtualų įrenginį, kuris palengvina įrenginio valdymą, pateikdamas patogesnį interfeisą.

Kitas klausimas, kur vyksta procesai, ar branduolio erdvėj, ar už jo ribų. Pirmos OS buvo monolitinės, kur viena procedūra galėjo iškviesti bet kokią kitą procedūrą. Tai tapo kliūtimi augant OS. Buvo įvesta OS sluoksninė architektūra. Sudėtingumas nuo to nedingo. Kiekvienas sluoksnis gana didelis. Pakeitimai vienam sluoksnyje iššaukia pakeitimus ir gretimuose sluoksiniuose. Sunku kūrti versijas pagal konkrečią konfigūraciją. Sunku spręsti saugumo problemas dėl gretimų sluoksnių sąveikos.

Mikrobranduolio architektūros charakteristika

Mikrobranduolys yra OS nedidelė atminties dalis, įgalinanti OS modulinį plėtimą. Nėra vieningos mikrobranduolio sandaros. Problema yra driver'iai, juos reikia padaryt efektyvius. Į driver'į galima žiūrėti kaip į virtualų įrenginį, kuris palengvina įrenginio valdymą, pateikdamas patogesnį interfeisą. Kitas klausimas, kur vyksta procesai, ar branduolio erdvėj, ar už jo ribų.

Pirmos OS buvo monolitinės, kur viena procedūra galėjo iškviesti bet kokią kitą procedūrą. Tai tapo kliūtimi augant OS. Buvo įvesta OS sluoksninė architektūra. Sudėtingumas nuo to nedingo. Kiekvienas sluoksnis gana didelis. Pakeitimai vienam sluoksnyje iššaukia pakeitimus ir gretimuose sluoksniuose. Sunku kurti versijas pagal konkrečią konfigūraciją. Sunku spręsti saugumo problemas dėl gretimų sluoksnių sąveikos.

Mikrobranduolio sistemos atveju visi servisai perkelti į vartotojo sritį. Jie sąveikauja tarpusavyje ir su branduoliu. Tai horizontali architektūra. Jie sąveikauja per pranešimus, perduodamus per branduolį. Branduolio funkcija tampa pranešimo perdavimas ir priėjimas prie aparatūros.

Mikrobranduolio architektūros pranašumai: 1) Vieningas interfeisas 2) Išplečiamumas 3) Lankstumas 4) Pernešamumas (Portability) 5) Patikimumas 6) Tinkamumas realizuoti paskirtytas (išskirstytas) sistemas.

Neigiama savybė – nepakankamas našumas, kalta pranešimų sistema, Jį reikia pernešti, perduoti, gavus atkoduoti. Atsiranda daug perjungimų tarp vartotojo ir supervizoriaus režimų.

Vartotojo ir branduolio gijos

Vartotojo gijos

Šių gijų perjungimui nereikia branduolio įsikišimo. Gijų biblioteka nepriklauso nuo OS įsikišimo, tai taikomojo lygmens programa. O vartotojo gijų trūkumai – jei gija blokuoja procesą, blokuojamos ir kitos proceso gijos. Vartotojo gijos negali išnaudoti daugiaprocesorinės sistemos.

Branduolio gijos

Jų valdymas atliekamas branduolyje. Jei taikomojoje programoje sukurtos gijos priklauso vienam procesui ir turi tą patį kontekstą, tai branduolys valdo visas gijas ir įveikia visus minėtus trūkumus, iššauktus vartotojo gijų. Branduolio gijų skaičius ribotas.

Jei persijungimai vyksta vartotojų gijų ribose, tai užtenka vartotojo gijų, jei kviečiamas sisteminis procesas, tai naudojamos branduolio gijos.

Virtualios atminties sąvoka

OA – tai ta atmintis, į kurią procesorius gali kreiptis tiesiogiai imant komandas ar duomenys programos vykdymo metu. Tokia schema turi daug nepatogumų, kai programoje nurodomi absoliutūs adresai.

Transliatoriai buvo perdaryti taip, kad gamintų objektinę programą, nesusietą su patalpinimo vieta, t.y. kilnojamus objektinius modelius (nustatant programos adresus pagal išskirtą atminties vietą). Nuo atminties skirstymo programos vykdymo metu pereita prie atminties skirstymo prieš programos vykdymą. Prieš vykdymą programas reikia susieti ir patalpinti į atmintį.

Kai visi darbai atliekami prieš programos vykdymą, kalbama apie statinį adresų nustatymą – statinį atminties skirstymą. Dinaminis atminties skirstymas – kai adresai nustatomi betarpiškai kreipiantis į atmintį. Statiškai nustatant adresus būtina prieš programos vykdymą žinoti išskirtos atminties vietą. Dinaminis – kai fizinis adresas nustatomas tiesiogiai prieš kreipimąsi į tą adresą.

Sudarant programas tenka abstrahuotis nuo atminties žodžių ir nauduoti tam tikrus lokalius adresus, kurie vėliau kokiu tai būdu susiejami su fiziniais adresais. Tokia lokalių adresų visuma – virtuali atmintis. Fizinių adresų visuma – reali atmintis. Virtuali atmintis su fizine gali būti susiejama statiškai arba dinamiškai.

Komandos vykdymo schema architektūroje su dinaminiu adresų nustatymu

Schematiškai pavaizduosime komandų vykdymą, kuomet nėra dinaminio atminties skirtymo:

H[O:N] – atmintis;

K – komandų skaitliukas;

R – registras.

L: W:=H[K];

OK:=W op kodas;

ADR:+ W operando adr;

K:=K+1;

Add: IF OK=1 THEN R:=R+H[ADR];

ELSE

SR: IF OK=2 THEN H[ADR]:=R;

ELSE

BR: IF OK=3 THEN K:=ADR;

GOTO L

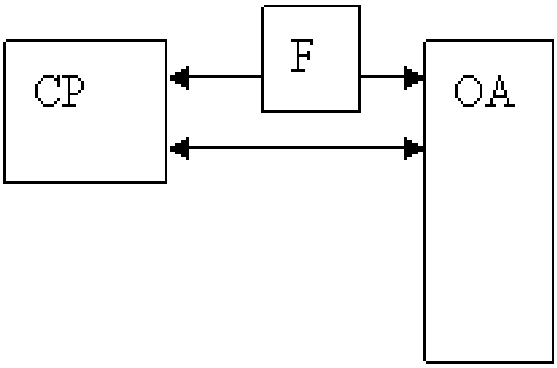
Čia K, ADR vadinami efektyviais adresais.

Jei yra dinaminė adresų nustatymo aparatūra, tai efektyvus adresai yra atvaizduojami į absoliučius (fizinius) adresus: F(ea)=aa.

H[k]~H[F(k)]

H[ADR]~H[F(ADR)].

Į F galima žiūrėtu kaip į aparatūrinį bloką, jungiantį procesorių su OA. Schematiškai:

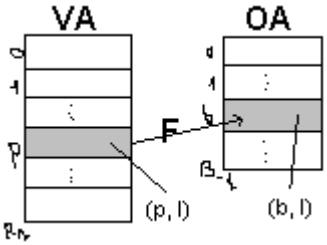


Statinis ir dinaminis atminties skirstymas

Kai visi darbai atliekami prieš programos vykdymą, kalbama apie statinį adresų nustatymą – statinį atminties skirstymą. Dinaminis atminties skirstymas – kai adresai nustatomi betarpiškai kreipiantis į atmintį. Statiškai nustatant adresus būtina prieš programos vykdymą žinoti išskirtos atminties vietą. Dinaminis – kai fizinis adresas nustatomas tiesiogiai prieš kreipimąsi į tą adresą.

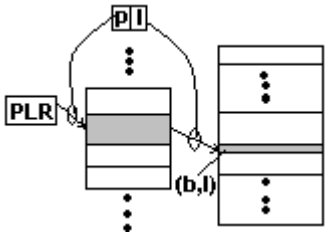
Sudarant programas tenka abstrahuotis nuo atminties žodžių ir nauduoti tam tikrus lokalius adresus, kurie vėliau kokiu tai būdu susiejami su fiziniais adresais. Tokia lokalių adresų visuma – virtuali atmintis. Fizinių adresų visuma – reali atmintis. Virtuali atmintis su fizine gali būti susiejama statiškai arba dinamiškai.

Puslapinė organizacija



Puslapinė organizacija – tai konkretus vartotojo atminties (VA) organizavimo būdas. Operatyvi atmintis (OA) yra suskaidoma į vienodo ilgio ištisinius blokus $b_0b_1...b_{B-1}$. OA absoliutus adresas (AA) nurodomas kaip pora (b, l), kur b – bloko numeris, l – žodžio numeris bloko b viduje. Atitinkamai VA adresinė erdvė suskaidoma į vienodo dydžio ištisinius puslapius $p_0p_1...p_{P-1}$. VA adresas nustatomas pora (p, l). Puslapio dydis lygus bloko dydžiui. Bendra suminė VA yra daug didesnė už OA. VA adresas lygus efektyviam adresui (EA), kurį reikia atvaizduoti į AA: F(EA)=AA, šiuo atveju F(p, l)=(b, l).

Puslapinės organizacijos schema: a) VA turime [p|l]; b) aparatūroje turi būti numatytas [PLR] – puslapių lentelės registras(rodo aktyvaus proceso puslapių lentelę); c) puslapių lentelės saugomos atmintyje.



Puslapių lentelės registro dydis atitinka VA puslapių skaičių. Kiekvienam procesui sudaroma puslapių lentelė.

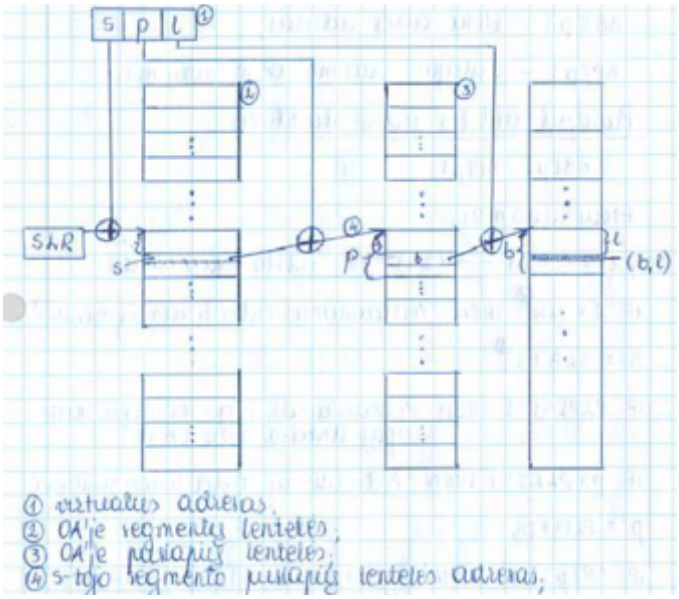
Vieno segmento VA atvaizdavimas: F(p, l) = M[PLR+p]*2^k+l, čia M[PLR+p] – bloko numeris; 2^k – bloko dydis; i – poslinkis.'

Polisegmentinė virtuali atmintis

Puslapinė organizacija su segmentacija. SLR rodo į aktyvaus proceso SL. Kiekvienas Segmentas yra išreiškiamas puslapiu ir turi atitinkamą puslapių lentelę.

(s,w) yra išreiškiamas trejetu (s,p,l)

Kiekvienas segmentas suskaidomas puslapiais, tokio pat dydžio kaip bloko dydis.



SLR – Segmentų lentelės registras, saugo aktyvaus proceso segmentų puslapių lentelės adresą.

??PSL – Saugo bloko numerį į kurį tas puslapis atvaizduojamas.

SLR: SLB komponentė – segmentų lentelės bazė – nurodo adresą OA‘je.

SLD – kiek segmentų VA‘je (segmentų lentelės dydis)

SL: PLB(S) – puslapių lentelės bazė

PLD(S) – puslapių lentelės dydis

PLP(S) – puslapių lentelė buvimo OA požymis

PL: BB[p] – bloko bazė

BP[p] – buvimo OA požymis

```
IF S>SLD THEN „Pertraukimas. Neleistinas segmentas“
S:=SLB+S;
IF PLP[S] = 1                    THEN “Pertraukimas. OA nera s-ojo segmento
psl.lenteles”
IF p>PLD[S] THEN “Pertraukimas. Neleistinas psl.segmente”
p:=PLB+p;
IF BP[p]=1 THEN “Pertraukimas. Psl.nera atminty”
F := BB[p] + l
```

Atminties skirstymo puslapiais strategijos

Esant dinaminiam atminties skirstymui VA dydis gali būti didesnis nei OA dydis. Tada reikia priverstinai atlaisvinti vietą. Puslapių skirstymo uždavinys turi atsakyti į klausimus:

a) kokiu momentu sukeisti puslapius;

b) kokį puslapį patalpinti į atmintį (iš išorės į OA);

c) vietoje kokio puslapio.

Puslapių skirstymo strategija, pagal kurią vieta puslapiui skiriama betarpiškai prieš į jį kreipiantis, vadinama strategija pagal pareikalavimą (SPP). Ji užfiksuoja atsakymus į pirmuosius du klausimus.

Įrodyta, kad bet kokiai puslapių skirstymo strategijai egzistuoja neblogesnė strategija pagal pareikalavimą. Vadinasi, optimalių strategijų paieškoje galima apsiriboti SPP klase.

SPP, kurioje išstumiami tie puslapiai, kreipiamaisi į kuriuos bus vėliausiai puslapių trasoje, vadinama Bellady strategija. Plačiau: http://en.wikipedia.org/wiki/Belady%27s_anomaly

Pvz.: tarkime, turime du blokus b1, b2 ir puslapių seką p1, p2, p3, p4, p1, p2, p3. Sprendimas:

B_1 | p_1 p_1 p_1 p_1 p_1 p_1

B_2 | p_2 p_3 p_4 p_2 p_3

Šitokia strategija yra optimali. Tačiau iškyla taikymo problemos, nes prieš programos vykdymą puslapių trasa nėra žinoma.

Praktikoje naudojamos tam tikros strategijos, kai pakeičiamas: 1)atsitiktinis, 2)ilgiausiai būnantis OA’je puslapis, 3)į kurį buvo kreiptąsi seniausiai (paskiausiai).

Vidutiniškai du kartus mažiau puslapių pakeitimų, jei naudojama trečioji strategija: naujas puslapis įrašomas vietoj seniausiai naudoto.

Atminties išskyrimas ištisinėmis sritimis

GETAREA(adress, size); FREEAREA(address, size); FREEGARBAGE;

Kintamo dydžio ištisinės atminties sričių grąžinimas

Procedure FREEAREA(adress, size)' Begin **pointer L;M;U;Q;**

M:=address – 1;

U:=M+M.size;

L:=M-1;

IF U.tag = '-' **THEN**

Begin

M.size:=M.size +U.size;

U.BLINK.FLINK:=U.FLINK;

U.FLINK.BLINK:=U.BLINK;

End;

Q:=M+M.size-1;

IF L.tag='-' **THEN** //Jei prieš tai yra laisva

Begin

L:=M-L.size;

L.size:=L.size+M.size;

Q.size:=L.size;

Q.tag:='-';

End;

ELSE Begin //Jei užimta

Q.size:=M.size;

M.tag:=Q.tag:='-';

M.FLINK:=FREE.FLINK; //M įjungiamas į laisvų sričių sąrašą.

M.BLINK:=address(FREE);

FREE.FLINK.BLINK:=M;

FREE.FLINK:=M;

End;

Failų sistemos sąvoka

FS yra OS dalis, kuri valdo įv/išv, įv/išv metu naudojamus resursus ir operuoja informacija failuose. Į F galima žiūrėti kaip į virtualų įv/išv įrenginį tokį, kuris turi patogią programuotojui struktūrą. Programuotojui patogu operuoti failine informacija loginiame lygyje. Į failą galima žiūrėti kaip į: (F, e), kur F - failo vardas, e – elemento identifikatorius faile.

Galima kalbėti apie virtualią failinę atmintį. FS virtualios failinės atminties paskirtis – aprūpinti vartotoją tiesine erdve jų failų patalpinimui. Pagrindinės funkcijos: 1. užklausimų VFA‘iai transformavimas į RFA; 2. informacijos perdavimas tarp RFA ir OA. **FS** - tai OS dalis, kuri yra atsakinga už failinės informacijos sukūrimą, skaitymą, naikinimą, skaitymą, rašymą, modifikavimą, bei perkėlimą. Failų sistema kontroliuoja failų naudojamus resursus ir priėjimą prie jų. Programuotojam nesunku naudotis failų informacija, jai ji yra loginiame lygmenyje.

Failinės atminties įrenginių charakteristikos

Fizinės failinės atminties įrenginiai apibūdinami tam tikromis charakteristikomis:

Talpumas – maksimalus informacijos kiekis, kurios gali būti saugomos;

Įrašo dydis – minimalus informacijos kiekis, į kuri galima adresuoti įrenginyje. Įrašai įrenginiuose gali būti kintamo arba pastovaus ilgio.

Priėjimo būdai: a) tiesioginis – operuojama aparatūra; b) nuoseklus – kai priėjimui prie įrašo reikalingas visų tarpinių įrašų peržiurejimas (atmintis magnetinėje juostoje).

FA informacijos nešejas – tomas. Tomo charakteristika – jo pakeičiamumas – įgalina iš esmės padidinti vartotojo naudojamos VA apimtį. Pvz. Diskelis..

Duomenu perdavimo greičiai. Jis matuojamas baitais arba bitais per sekundę perduodant informaciją tarp OA ir įrenginio. KB – kbaitai, kb – kbitai

Užlaikymas. Įrenginiui gaves eilinę įv/iš komandą (jei tai juostinis įrenginys), jam reikia įsibegėti nuo praeito skaitymo iki naujo segmento pradžios. Jei diskas – užlaikymas – tai apsisukimas nuo pradžios iki tos vietos, kur yra informacija.

Nustatymo laikas. Tai galvučių perstūmimo laikas(diskai).

Priklausomai nuo įrenginių charakteristikų ir nuo jų panaudojimo sąlygų, vieni ar kiti įrenginiai yra naudojami tam tikrais atvejais.

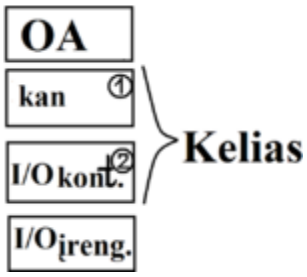
Įvedimo/išvedimo įrenginių greičių charakteristika

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Įvedimo/išvedimo įrenginių procesų schema

~~51.1. Įvedimo-išvedimo procesai~~

I/O valdymui kiekvienam I/O įrenginiui *i* sukuriamas jį aptarnaujantis procesas *Pi*. Jis atstovauja įrenginį sistemoje. Jo paskirtis inicijuoti I/O operacijas. Perduoti pranešimus apie pertraukimus ir pranešti apie I/O veiksmų pabaigą.



(1)Specializuotas procesorius su savo komandų sistema(registrais, valdymo įrenginiu). Perduoda duomenis tarp OA ir I/O kontrolerio (2)kontroleris – specializuotas kontreleris nukreiptas i tam tikrą tipą.

I/O įrenginį atstovaujantis procesas *Pi.*

Begin

L: PRAŠYTIR(III(1), Ω(2), (P(3), IIprogr.(4)));

PRAŠYTIR (KK(5), III(6), Kelias);

Komutavimo ir įrenginių pranešimo komandos;

I/O inicijavimas;

PRAŠYTIR (IIP(7), Kelias(8), Pran)(9);

Pertraukimo dekodavimas;

Papildomos I/O komandos;

Atsakymo suformulavimas;

ATLAISVINTIR(KK, Kelias);

ATLAISVINTIR(III, Pran, (P,Ats));

GOTO L:

End

Paaiškinimai: (1)Resursas yra I/O įrenginys (2)Nėra specifikuojama kokios resurso dalies reikia (3)Prašantis proceso P (4)Ką turėtų atlikti I/O įrenginį aptarnaujantis procesorius(skaitymą, rašymą, failo atidarymą, ..) (5)Kanalo kontroleris (6)Reikia kelio iki tokio įrenginio (7)I/O pertraukimas (8)Bet kuri iš kelio sudedamųjų dalių (9)Sėkmės atveju jos nėra būtinos.

VM nustato I/O komandą. Įvyksta pertraukimas.

Pertraukimo apdorojimo metu nustatomas procesas, kuris turi aptarnauti konkretų pertraukimą, schematiškai tai aprašome:

Pi: Įsiminti(CPU[*]);

ST[*]:=READY; // pertraukimas nepervedant proceso į blokavimo būseną

PROC[P[*]]:=Ω; // procesoriaus atlaisvinimas

Nustatyti Pt;

ATLAISVINTIR (PERT, (Pt, PERTR_INF));

Taimerio pertraukimo apdorotojas: jei viršytas laiko limitas, tai nutraukia proceso vykdymą, o jei viršytas procesoriaus kvanto limitas, tai perkelią į to paties prioritetų procesų sąrašą.

Užklausimo failinei sistemai realizavimo pavyzdys

Darbui su failine atmintimi naudojama bendros paskirties failų sistema. Užklausimas skaitymui iš failo gali atrodyti taip:

Loginiame lygmenyje vartotojo pateikta komanda: READ(A) A- nuskaitoma OA sritis.

1. READ1(FN, A) // tarkim perskaito 80B įrašą

FN – išorinis failo vardas

A – adresas

Nuskaitys į adresą A[0]....A[79]

2. Tarkim r yra nuoroda į sekanti įrašą, kurį reikia nuskaityti:

READ2(FN, A, r, 80);

r := r + 80; r – sekantis įrašas; 80 - skaitomos informacijos ilgis baitais.

Skirtingi vartotojai gali parinkti vienodus vardus, todėl reikalingas unikalus vidinis failo vardas ir deskriptorius. Pagal failo vidinį vardą galimą nustatyti failo deskriptorių, o failų deskriptoriai kaip ir patys failai saugomi išorinėje atmintyje. Failų atidarymo metu, failų deskriptoriai iš išorinės atminties perkeliami į vidinę atmintį, aktyvių failų katalogą.

READ3(d, A, r, 80)

Siekiant optimizuoti I/O realizavimo veiksmus naudojama buferizacija, t.y. skaitoma iš išorinės atminties tam tikrais blokais. Tarkim, kad buferių blokas yra 10 buferių, o mūsų buferio dydis 80.

READ4(įreginys, įrašo adr., Buf[k], 800)

Įrašo adr. := įrašo adr + 800

Turi būti sugeneruotos apsikeitimo su išoriniu įrenginiu komandos priklausomai nuo architektūros. Įvykdžius komandą grąžinamas pranešimas f-jai READ(A).

Failų deskriptorius. Aktyvių failų katalogas

Failo deskriptorius:

1. Failo vardas: FN;

2. Failo padėtis : įrenginio adresas, failo pradžios adresas įrenginyje;

3. Failo organizacija: nuosekli;

4. Failo ilgis: L;

5. Failo tipas: {pastovus, laikinas};

6. Failo savininkas: U;

7. Failo naudotojai: {U};

8. Failo apsauga: READ; (skirtas tik skaitymui)

Aktyvių failų kataloge (AFK) laikomi aktyvių failų deskriptoriai (aktyvūs failai - „atidaryti failai“). Neaktyvūs („neatidaryti“) failai neturi deskriptoriaus ir jų nėra AFK. Ieškant failo deskriptoriaus, pirmiausiai ieškoma AFK, tik paskui (jeigu nėra AKF) – sisteminiam kataloge (šiuo atveju deskriptoriaus nėra, todėl ieškoma pagal išorinį vardą, o tik tada yra sukuriamas deskriptorius.). AFK yra operatyvioje atmintyje.

Failų sistemos hierarchinis modelis

Failų sistemos funkcijas patogu apibūdinti pagal jų lygį, pradedant nuo aparatūrinio iki vartotojo serviso klausimų. Failų sistemos suskirstymas į loginius lygius: DBVS(5)-Priėjimo metodai(4)-Virtuali (loginė) failų sistema (3)-Reali (bazinė) failų sistema(2)-Įvesties/išvesties sistema(1).

1) koordinuoja fizinių įrenginių darbą. Šio lygio procesai atlieka informacijos blokų apsikeitimą tarp OA ir išorinės atminties pagal užduotą adresą.

2) transformuoja failo vidinį unikalų identifikatorių į failo deskriptorių.

3) pagal vartotojo suteiktą IV nustato jo vidinį unikalų vardą. Naudojamas vidinis failų katalogas. Virtualus lygis nepriklauso nuo fizinių įrenginių.

4) realizuoja dėsnį. Pagal kurį apdorojami failo įrašai. Tokį dėsnį užduoda vartotojas pagal programos prasmę. Pvz.: nuoseklus priėjimas arba kai įrašai apdorojami pagal lauko (rakto) reikšmes didėjimo ar mažėjimo tvarka.

5) realizuoja failo loginės struktūros vaizdavimą į fizinę struktūrą.

Failų sistemos įvedimo-išvedimo posistemė

Bazinė failų valdymo sistema

Tai įvedimo/išvedimo sistema, kurioje aparatūros specifika izoliuoja įvedimą/išvedimą nuo likusios OS dalies. Tai leidžia įvedima/išvedimą nagrinėti persiunčiamų informacinių blokų terminais.

Prieš pasiunčiant informacijos bloką reikia nustatyti operatyvios atminties adresą ir fizinį bloko adresą išoriniame įrenginyje. Tokį adresą ir suformuoja bazinė failų valdymo sistema pagal failo deskriptorių. Be to bazinė sistema valdo išorinės atminties failus ir apdoroja tomų failų deskriptorius.

Darbui su deskriptoriais bazinė sistema turi komandas(funkcijas):

SUKURTI(failo vardas, sritis), kur sritis – iš kokių blokų sukurti failą. Ji vykdoma inicializuojant procesą, aptarnaujantį tomą. Veliau laisvos atminties failas skaidomas į dalinius failus.

DALINTI(failo vardas, sritis). Kai failas yra naikinamas, jo užimama atmintis turi būti atlaisvinta ir sugražinta į laisvos atminties failą.

IŠPLĖSTI(failo vardas, sritis). Komanda skirta išorinės atminties padidinimui.

ATLAISVINTI(failo vardas, sritis). Visa arba dalis išorinės atminties grąžinama į laisvos atminties failą.

Bazinė sistema turi turėti ryšį su operatoriumi arba vartotoju tam, kad pranešti apie galimybę siūsti pranešimus apie tomų pakeitimus.

Tomų deskriptoriaus sudėtis:

Tomo vardas, unikalus tomo ID, proceso arba loginio įrenginio vardas,atitinkamas požymis, nuoroda į tomo turinio lentelę, informacija apie tomo apsaugą, tomo sukurimo ir galiojimo data.

Tomų valdymo bazinė sistema turi funkcijas:

REGISTRUOTI(tomo vardas) – sukuriamas naujas tomo deskriptorius,

PAŠALINTI(tomo vardas),

PRITVIRTINTI(tomo vardas) – tomo priskyrimas įrenginiui,

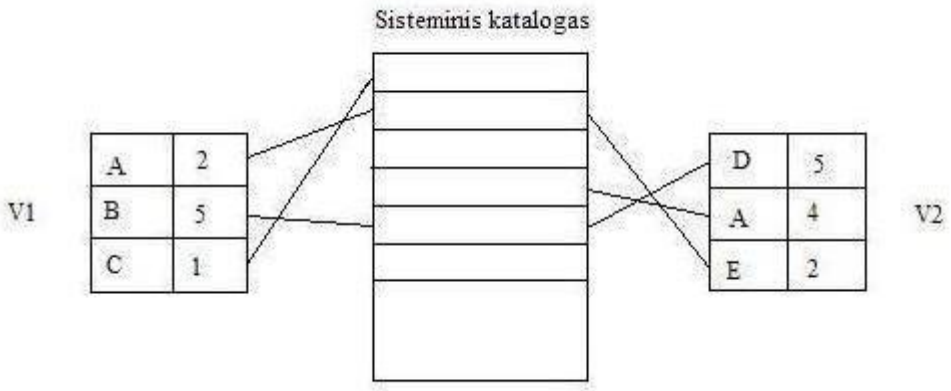
ATLAISVINTI(tomo vardas) – atjungimas nuo įrenginio.

Loginė failų valdymo sistema

Ji atvaizduoja lokalius vartotojo failų vardus į unikalius failų identifikatorius. Loginė failų valdymo sistema pateikia išoriniam interfeisui komandsa, kurias realizuoja bazinė sistema, kuriose jau nurodomas unikalus F identifikatorius.

Loginė sistema reikalauja iš vartotojo pranešimo apie darbo su failais pradžią – komanda ATIDARYTI(funkciškai ją atlieka bazinė sistema).

Vartotojo žinynas – tai informacija apie failus. Jis susieja failo išorinį vardą su jo unikaliu vardu, nurodančiu į bendrą sisteminį žinyną.



Vartotojo žinyno panaudojimas leidžia laisvai parinkti failo vardus ir užtikrina efektyvų kreipimąsi į failus.

Loginė sistema išoriniam interfeisui pateikia komandas (jas realizuoja bazinė sistema):

SUKURTI(failo vardas);

SUNAIKINTI(failo vardas);

ATIDARYTI(failo vardas);

UŽDARYTI(failo vardas);

SKAITYTI(failo vardas, bloko nr., OA, bloko RAŠYTI(failo vardas, bloko nr., OA, bloko sk.);

Komandų rezultatai priklauso nuo konkrečios situacijos. Vartotojo procesas tas situacijas gali išskirti ir apdoroti arba naudoti kaip klaidą.

Priėjimo metodai

Loginės sistemos komandos naudoja bloko nr. Bet vartotojui nepatogu operuoti terminais. Tam įvedamas dar vienas failų sistemos lygmuo – priėjimo metodai.

Įrašų apdorojimui gali būti naudojami raktai. Dalinis raktas – duomenų laukas, kurio reikšmė duotu momentu gali atitikti vieną iš daugelio įrašą. Raktai gali būti naudojami įrašų identifikavimui.

Įrašai su vienodais galimais raktais apjungiami į sąrašą, todėl pakanka surasti tik pirmą sąrašo elementą.

Priėjimo prie įrašų metodai turi dvi charakteristikas: a) baziniai arba su eilutėmis; b) tiesioginiai arba nuoseklūs. Tiesioginiai – leidžia kreiptis į įrašus individualiai, o nuoseklūs – fizinė įrašų tvarka atitinka jų loginę tvarką.

Nuoseklaus ir tiesioginio metodų suderinimui naudojamas indeksacijos metodas.

RAID

RAID 0 consists of striping, without mirroring or parity. The capacity of a RAID 0 volume is the sum of the capacities of the disks in the set, the same as with a spanned volume. There is no added redundancy for handling disk failures, just as with a spanned volume. Thus, failure of one disk causes the loss of the entire RAID 0 volume, with reduced possibilities of data recovery when compared with a broken spanned volume. Striping distributes the contents of files roughly equally among all disks in the set, which makes concurrent read or write operations on the multiple disks almost inevitable and results in performance improvements. The concurrent operations make the throughput of most read and write operations equal to the throughput of one disk multiplied by the number of disks. Increased throughput is the big benefit of RAID 0 versus spanned volume, at the cost of increased vulnerability to drive failures.

RAID 1 consists of data mirroring, without parity or striping. Data is written identically to two drives, thereby producing a "mirrored set" of drives. Thus, any read request can be serviced by any drive in the set. If a request is broadcast to every drive in the set, it can be serviced by the drive that accesses the data first (depending on its seek time and rotational latency), improving performance. Sustained read throughput, if the controller or software is optimized for it, approaches the sum of throughputs of every drive in the set, just as for RAID 0. Actual read throughput of most RAID 1 implementations is slower

than the fastest drive. Write throughput is always slower because every drive must be updated, and the slowest drive limits the write performance. The array continues to operate as long as at least one drive is functioning.

RAID 2 consists of bit-level striping with dedicated Hamming-code parity. All disk spindle rotation is synchronized and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive. This level is of historical significance only; although it was used on some early machines (for example, the Thinking Machines CM-2), as of 2014 it is not used by any commercially available system.

RAID 3 consists of byte-level striping with dedicated parity. All disk spindle rotation is synchronized and data is striped such that each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive. Although implementations exist, RAID 3 is not commonly used in practice.

RAID 4 consists of block-level striping with dedicated parity. This level was previously used by NetApp, but has now been largely replaced by a proprietary implementation of RAID 4 with two parity disks, called RAID-DP. The main advantage of RAID 4 over RAID 2 and 3 is I/O parallelism: in RAID 2 and 3, a single read/write I/O operation requires reading the whole group of data drives, while in RAID 4 one I/O read/write operation does not have to spread across all data drives. As a result, more I/O operations can be executed in parallel, improving the performance of small transfers.

RAID 5 consists of block-level striping with distributed parity. Unlike RAID 4, parity information is distributed among the drives, requiring all drives but one to be present to operate. Upon failure of a single drive, subsequent reads can be calculated from the distributed parity such that no data is lost. RAID 5 requires at least three disks. RAID 5 implementations are susceptible to system failures because of trends regarding array rebuild time and the chance of drive failure during rebuild (see "Increasing rebuild time and failure probability" section, below). Rebuilding an array requires reading all data from all disks, opening a chance for a second drive failure and the loss of the entire array. In August 2012, Dell posted an advisory against the use of RAID 5 in any configuration on Dell EqualLogic arrays and RAID 50 with "Class 2 7200 RPM drives of 1 TB and higher capacity" for business-critical data.

RAID 6 consists of block-level striping with double distributed parity. Double parity provides fault tolerance up to two failed drives. This makes larger RAID groups more practical, especially for high-availability systems, as large-capacity drives take longer to restore. RAID 6 requires a minimum of four disks. As with RAID 5, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced. With a RAID 6 array, using drives from multiple sources and manufacturers, it is possible to mitigate most of the problems associated with RAID 5. The larger the drive capacities and the larger the array size, the more important it becomes to choose RAID 6 instead of RAID 5. RAID 10 also minimizes these problems.

Nested (hybrid) RAID

The final array is known as the top array. When the top array is RAID 0 (such as in RAID 1+0 and RAID 5+0), most vendors omit the "+" (yielding RAID 10 and RAID 50, respectively).

RAID 0+1: creates two stripes and mirrors them. If a single drive failure occurs then one of the stripes has failed, at this point you are running effectively as RAID 0 with no redundancy, significantly higher

risk is introduced during a rebuild than RAID 1+0 as all the data from all the drives in the remaining stripe has to be read rather than just from 1 drive increasing the chance of an unrecoverable read error (URE) and significantly extending the rebuild window.

RAID 1+0: creates a striped set from a series of mirrored drives. The array can sustain multiple drive losses so long as no mirror loses all its drives.

JBOD RAID N+N: With JBOD (Just a Bunch Of Disks), it is possible to concatenate disks, but also volumes such as RAID sets. With larger drive capacities, write and rebuilding time may increase dramatically (especially, as described above, with RAID 5 and RAID 6). By splitting larger RAID sets into smaller subsets and concatenating them with JBOD, write and rebuilding time may be reduced. If a hardware RAID controller is not capable of nesting JBOD with RAID, then JBOD can be achieved with software RAID in combination with RAID set volumes offered by the hardware RAID controller. There is another advantage in the form of disaster recovery, if a small RAID subset fails, then the data on the other RAID subsets is not lost, reducing restore time.