

Cap. 1 COMPUTER SYSTEM OVERVIEW

1. Enumerați și definiți pe scurt cele patru elemente principale ale unui calculator.
Procesorul, memoria principală, modulele I/O magistralele sistem.
2. Definiți cele două categorii principale de regiștrii procesor.
Regiștrii vizibili utilizatorului, regiștrii de control și stare.
3. Care sunt, în termeni generali, cele patru acțiuni distincte pe care le pot specifica instrucțiunile procesor.
Procesor-memorie, procesor – I/O, procesarea datelor, control,
4. Ce este o întrerupere?
Un mecanism prin care alte module (I/O sau memoria) pot întrerupe secvența normală de lucru a procesorului.
5. Cum se lucrează cu întreruperile multiple?
Se pot lua în considerare două abordări. Prima este de a dezactiva întreruperile atunci când se tratează o întrerupere. A doua abordare este de a defini priorități pentru întreruperi și de a permite unei întreruperi mai prioritare să întrerupă handler-ul de tratare a întreruperii mai puțin prioritare.
6. Ce caracteristici disting diferențele elemente ale ierarhiei de memorie?
Costul, capacitatea și timpul de acces.
7. Ce este memoria cache?
Este o memorie de capacitate mai mică și mai rapidă decât memoria principală care este interpusă între procesor și memoria principală.
8. Care este diferența între un sistem multimicroprocesor și unul multinucleu?
Sistemele multimicroprocesor conțin mai multe procesoare fizice distincte iar sistemele multinucleu au mai multe procesoare pe același chip.
9. Care este diferența între localizarea spațială și cea temporală?
Localitatea spațială se referă la tendința ca un program să fie executat în locații de memorie alăturate (cluster). Spațialitatea temporală se referă la tendința ca un procesor să acceseze locații de memorie care au fost accesate recent.
10. În general, care sunt strategiile pentru a exploata localitatea spațială și cea temporală?
Localitatea spațială este exploată utilizând blocuri cache mari și prin încorporarea de mecanisme de preextragere. Localitatea temporală este exploatață prin păstrarea datelor și instrucțiunilor recent utilizate în memoria cache și prin exploatarea ierarhiei memorie cache.

11. Presupunem că procesorul ipotetic din figura 1.3 are două instrucțiuni I/O:

- 0011 Load AC from I/O (încarcă AC de la I/O)
- 0111 Store AC to I/O (Memorează AC la I/O)

În acest caz adresa pe 12 biți identifică un anumit dispozitiv extern. Prezentați execuția programului (utilizând formatul din figura 1.4) pentru următorul program:

1. Load AC from device 5 (Încarcă AC de la dispozitivul 5).
2. Add contents of memory location 940 (adună conținutul locației de memorie 940).
3. Store AC to device 6 (memorează AC în dispozitivul 6).

Presupuneți că valoarea preluată de la dispozitivul 5 este 3 și că locație 940 conține valoarea 2.

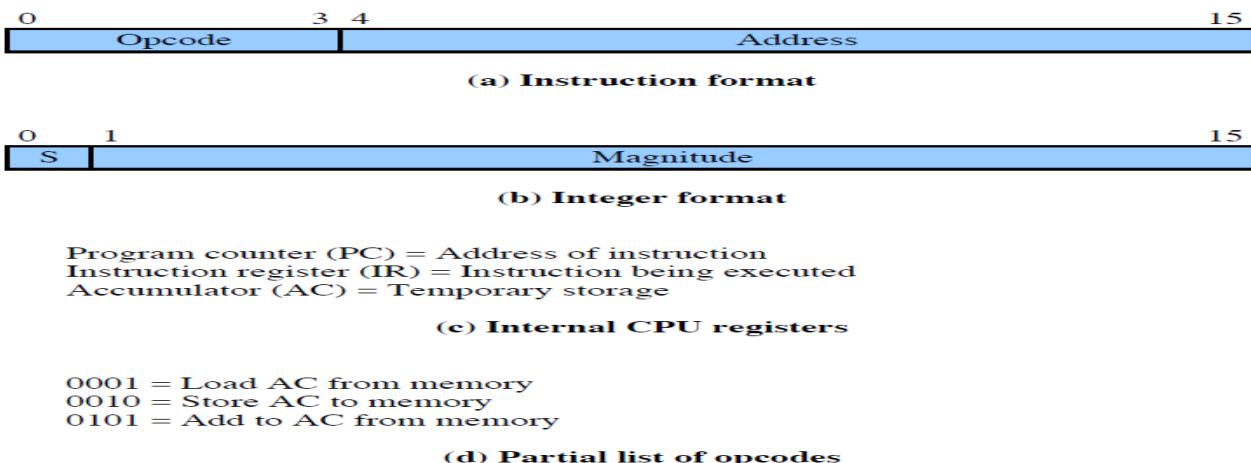
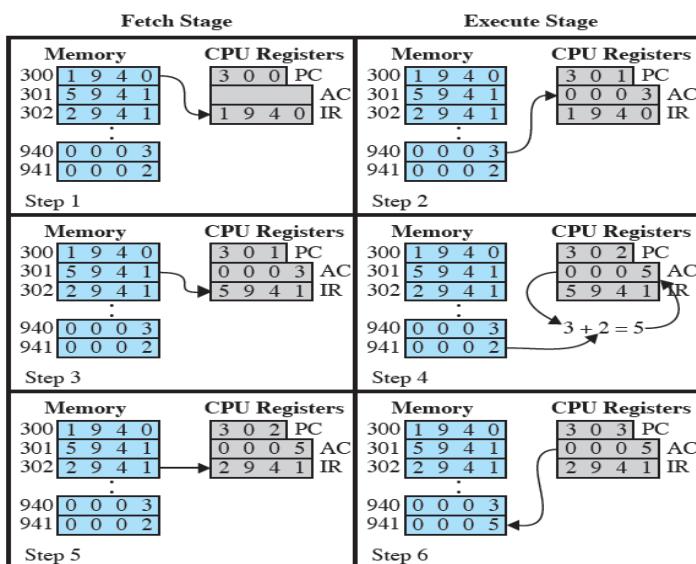


Figure 1.3 Characteristics of a Hypothetical Machine



**Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)**

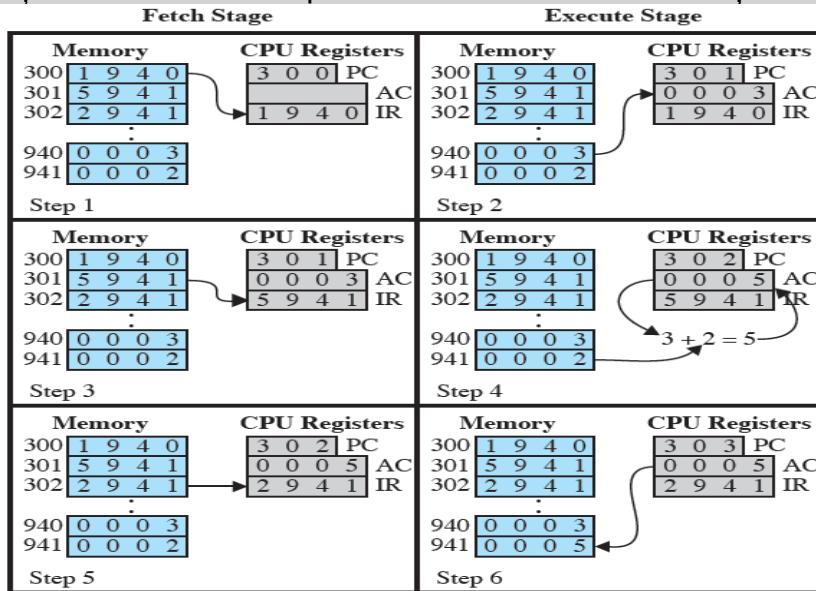
Memory (contents in hex): 300: 3005; 301: 5940; 302: 7006

Step 1: 3005 ® IR; **Step 2:** 3 ® AC

Step 3: 5940 ® IR; **Step 4:** 3 + 2 = 5 ® AC

Step 5: 7006 ® IR; **Step 6:** AC ® Device 6

12. Execuția programului din figura 1.4. este descrisă în text (carte) utilizând 6 pași.
Exandați această descriere pentru a arăta utilizarea MAR și MBR.



**Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)**

1a. PC conține adresa 300, adresa primei instrucțiuni. Această valoare este încărcată în MAR; 1b. Valoarea de la locația 300 (care este instrucțiunea cu valoarea 1940H) este încărcată în MBR, iar PC este incrementat. Acești doi pași pot fi realizati în paralel; 1c. Valoarea din MBR este încărcată în IR; 2a. Portiunea de adresă din IR (940) este încărcată în MAR; 2b. Valoarea din locația 940 este încărcată în MBR; Valoarea din MBR este încărcată în AC; 3a. Valoarea din PC (301) este încărcată în MAR; 3b. Valoarea de la locația 301 (unde este instrucțiunea cu valoarea 5941) este încărcată în MBR, și PC este incrementat; 3c. Valoarea din MBR este încărcată în IR; 4a. Portiunea de adresă din IR (941) este încărcată în MAR; 4b. Valoarea din locația 941 este încărcată în MBR; 4c. Vechea valoare din AC și valoarea din MBR sunt adunate și rezultatul se depune în AC; 5a. Valoarea din PC (302) este încărcată în MAR; 5b. Valoarea de la locația 302 (unde este instrucțiunea cu valoarea 2941) este încărcată în MBR, și PC este incrementat; 5c. Valoarea din MBR este încărcată în IR; 6a. Portiunea de adresă din IR (941) este încărcată în MAR; 6b. Valoarea din AC este încărcată în MBR; 6c. Valoarea din MBR este stocată la locația 941.

13. Să considerăm un procesor ipotetic pe 32 de biți care are instrucțiuni pe 32 de biți compuse două câmpuri. Primul octet conține opcodul instrucțiunii și ceilalți trei un operand imediat sau un operand de tip adresă. Care este maximul de memorie direct adresabilă (în octeți)?

$$2^{24} = 16 \text{ Mocți}$$

14. Să considerăm un procesor ipotetic pe 32 de biți care are instrucțiuni pe 32 de biți compuse două câmpuri. Primul octet conține opcodul instrucțiunii și ceilalți trei un operand imediat sau un operand de tip adresă. Discutați impactul asupra vitezei sistemului dacă magistrala microprocesorului are:

1. o magistrală locală de adrese pe 32 de biți și o magistrală locală de date pe 16 biți, sau
2. o magistrală locală de adrese pe 16 de biți și o magistrală locală de date pe 16 biți.

Pentru primul caz adresa este completă dar sunt necesari doi cicli pentru a extrage cele 32 de biți ai instrucțiunii. Pentru cel de-al doilea caz adresa trebuie demultiplexată (se depune, de exemplu, partea inferioară a adresei și apoi partea superioară a adresei). Își aici sunt necesari doi cicli pentru a extrage operandul.

15. Să considerăm un procesor ipotetic pe 32 de biți care are instrucțiuni pe 32 de biți compuse două câmpuri. Primul octet conține opcodul instrucțiunii și ceilalți trei un operand imediat sau un operand de tip adresă. Câte biți sunt necesari pentru program counter și pentru instruction register.

24 și 32 sau 8 dacă se memorează doar opcodul instrucțiunii.

16. Considerați un microprocesor ipotetic care generează adrese pe 16 biți (presupuneți că PC-ul are 16 biți) având o magistrală de date pe 16 biți. Care este spațiul maxim de memorie pe care procesorul îl poate accesa direct, dacă este conectat la o memorie pe 16 biți?

$$2^{16} = 64\text{K bytes cu transfer pe 16 biți.}$$

17. Considerați un microprocesor ipotetic care generează adrese pe 16 biți (presupuneți că PC-ul are 16 biți) având o magistrală de date pe 16 biți. Care este maximul de memorie pe care procesorul îl poate accesa direct, dacă este conectat la o memorie pe 8 biți?

$$2^{16} = 64\text{K bytes cu transfer pe 8 biți.}$$

18. Considerați un microprocesor ipotetic care generează adrese pe 16 biți (presupuneți că PC-ul are 16 biți) având o magistrală de date pe 16 biți. Care este trăsătura arhitecturală care permite acestui procesor să acceseze un spațiu I/O separat? Generarea de semnale și instrucțiuni I/O separate.

19. Dacă o instrucțiune de intrare sau de ieșire poate specifica o adresă I/O pe 8 biți, câte porturi pe 8 biți poate suporta procesorul? Dar câte porturi pe 16 biți?

256 de porturi de intrare, 256 de porturi de ieșire pe 8 biți și tot atâtea pe 16 biți.

20. Considerați un microprocesor pe 32 de biți cu o magistrală externă de date pe 16 biți, cu un ceas de 8 MHz. Presupuneți că acest microprocesor are un ciclu de magistrală a cărui durată minimă este de 4 cicli de ceas. Care este viteza maximă de transfer a datelor pe magistrală , în octeți/s, pe care o poate susține acest microprocesor ? Pentru a crește performanțele este mai bine să mărim magistrala externă la 32 de biți sau să dublăm ceasul extern al microprocesorului? Formulați orice altă presupunere și explicați. Hint: Determinați numărul de octeți care pot fi transferați pe ciclu de magistrală.

4 Mocteți/ secundă. Dublarea frecvenței poate duce la o nouă tehnologie pentru chip. Dublarea magistralei externe însemnă dublarea resurselor pe chip și modificarea controlului logic al magistralei. În primul caz trebuie dublată și viteza memorie procesorului. În al doilea caz trebuie dublată lungimea cuvântului de memorie.

21. Considerați un calculator care conține module I/O pentru a controla o tastatură și o imprimantă Teletype. Următorii registri sunt conținuți de UCP și sunt conectați direct la magistrala sistem:

INPR: Input Register, 8 bits

OUTR: Output Register, 8 bits

FGI: Input Flag, 1 bit

FGO: Output Flag, 1 bit

IEN: Interrupt Enable, 1 bit

Apăsarea unei taste de la Teletype și ieșirea pe imprimantă sunt controlate de modulul I/O. Teletype este capabilă să codifice un simbol alfanumeric pe un cuvânt de 8 biți și să decodifica un cuvânt pe 8 biți într-un simbol alfanumeric. Input flag este setat când se preia un cuvânt pe 8 biți de la Teletype în input register. Output flag este setat când se tipărește un cuvânt. Descrieți modul cum UCP, utilizând primii 4 registrii prezentați în problemă , poate realiza operații I/O cu Teletype.

Intrarea de la Teletype este memorată în INPR. INPR acceptă date de la Teletype atunci când FGI = 0. Când sosesc date aceste sunt memorate în INPR și FGI este setat pe 1. UCP citește periodic FGI. Dacă FGI este 1 UCP citește conținutul INPR în AC și setează FGI pe 0. Dacă UCP are date de transmis, UCP testează FGO. Dacă FGO este 0, UCP trebuie să aștepte. Dacă FGO=1, UCP transferă conținutul AC în OUTR și setează FGO pe 0. Teletype setează FGI pe 1 după ce caracterul este tipărit.

22. Considerați un calculator care conține module I/O pentru a controla o tastatură și o imprimantă Teletype. Următorii registri sunt conținuți de UCP și sunt conectați direct la magistrala sistem:

INPR: Input Register, 8 bits

OUTR: Output Register, 8 bits

FGI: Input Flag, 1 bit

FGO: Output Flag, 1 bit

IEN: Interrupt Enable, 1 bit

Apăsarea unei taste de la Teletype și ieșirea pe imprimantă sunt controlate de modulul I/O. Teletype este capabilă să codifice un simbol alfanumeric pe un cuvânt de 8 biți și să decodifica un cuvânt pe 8 biți într-un simbol alfanumeric. Input flag este setat când se preia un cuvânt pe 8 biți de la Teletype în input register. Output flag este setat când se tipărește un cuvânt. Descrieți cum se poate implementa funcția mult mai eficient utilizând IEN.

Prin utilizarea întreruperilor. UCP poate seta registrul IEN.

23. Virtual, în toate sistemele care includ module DMA, accesul DMA la memorie este mai prioritar decât accesul procesorului la memoria principală. De ce?

Poate fi antrenat într-un transfer continuu care nu poate fi întrerupt (citirea unui hard disk).

24. Un modul DMA transferă caractere în memoria principală de la un dispozitiv extern care transmite cu viteza de 9600 bps. Procesorul poate extrage instrucțiuni cu o viteză de 1 milion de instrucțiuni pe secundă. Cât de mult va fi încetinit procesorul de activitatea DMA?

Încetinește procesorul cu 12 %.

25. Un calculator constă din UCP și un dispozitiv I/O D, conectate la memoria principală M via o magistrală partajată cu o magistrală de date având lărgimea de 1 cuvânt. UCP poate executa maximum 106 instrucțiuni pe secundă. O instrucțiune medie necesită cicli procesor, dintre care trei utilizează magistrala de memorie. O operație de citire sau de scriere utilizează un ciclu procesor. Presupuneți că UCP execută continuu în fundal programe care necesită 95% din viteza sa de execuție a instrucțiunilor dar nici o instrucțiune I/O. Presupuneți că un ciclu procesor este egal cu un ciclu de magistrală. Presupuneți acum că trebuie transferat un bloc mare de date între M și D. Dacă se utilizează transferul I/O programat și fiecare cuvânt transferat prin I/O necesită ca UCP să execute două instrucțiuni, estimați viteza maximă de transfer I/O posibilă prin D, în cuvinte pe secundă.

25000 cuvinte/secundă.

26. Un calculator constă din UCP și un dispozitiv I/O D, conectate la memoria principală M via o magistrală partajată cu o magistrală de date având lărgimea de 1 cuvânt. UCP poate executa maximum 106 instrucțiuni pe secundă. O instrucțiune medie necesită cicli procesor, dintre care trei utilizează magistrala de memorie. O operație de citire sau de scriere utilizează un ciclu procesor. Presupuneți că UCP execută continuu în fundal programe care necesită 95% din viteza sa de execuție a instrucțiunilor dar nici o instrucțiune I/O. Presupuneți că un ciclu procesor este egal cu un ciclu de magistrală. Presupuneți acum că trebuie transferat un bloc mare de date între M și D. Estimați viteza dacă se utilizează DMA.

2.15×10^6

27. Considerați următorul cod:

for (i=0; i< 20; i++)

```
for (j=0; j< 10; j++)
    a[i] = a[i] * j
```

Dați un exemplu de localitate spațială în cod.

Referință la prima instrucțiune este imediat urmată de o referință la a doua.

28. Considerați următorul cod:

```
for (i=0; i< 20; i++)
    for (j=0; j< 10; j++)
        a[i] = a[i] * j
```

Dați un exemplu de localitate temporală în cod.

Zece accese la $a[i]$ = în cadrul unei bucle care apare la intervale scurte de timp.

29. Generalizați ecuațiile (1.1) și (1.2) din Appendix 1A la n – niveluri ierarhice de memorie.

$$a. \quad T_s = \sum_{i=2}^N \sum_{j=1}^{i-1} (B_j H_i) + T_1$$

30. Considerați un sistem de memorie cu următorii parametrii:

T_c 100 ns C_c 0.01 cents/bit

T_m 1,200 ns C_m 0.001 cents/bit

Care este costul a unui MB de memorie principală?

\$80

31. Considerați un sistem de memorie cu următorii parametrii:

T_c 100 ns C_c 0.01 cents/bit

T_m 1,200 ns C_m 0.001 cents/bit

Care este costul a unui MB de memorie principală? Utilizând tehnologia memoriei cache?

\$800

32. Considerați un sistem de memorie cu următorii parametrii:

T_c 100 ns C_c 0.01 cents/bit

T_m 1,200 ns C_m 0.001 cents/bit

Dacă timpul efectiv de acces este cu 10% mai mare decât timpul de acces la memoria cache care este rata de potrivire H?

$H = 1190/1200$

33. Un calculator are memorie cache, principală și un disk utilizat pentru memoria virtuală. Dacă un cuvânt adresat este în cache timpul de acces este de 20ns. Dacă este în memoria principală dar nu în cache timpul de acces este de 60ns pentru a aduce cuvântul în cache după care se reia accesul. Dacă cuvântul nu este în memoria principală, sunt necesare 12 ms pentru a extrage cuvântul de pe disk, urmate de 60 ns pentru al copia în cache, iar apoi este adresat din nou. Rata de potrivire pentru cache este de 0,9 iar pentru memoria principală este de 0,6. Care

este timpul mediu de acces în ns necesar pentru a accesa cuvântul adresat în acest sistem?

480026

34. Presupunem că o stivă este utilizată de procesor pentru a gestiona apelurile de procedură și revenirile. Poate fi eliminat PC-ul (program counter) prin utilizarea vârfului stivei ca program counter?

Da, dacă stiva este utilizată pentru a păstra doar adresa de revenire.

Cap. 3 Procese

1. Ce este un trazor (trace) de instrucțiuni?

Un trazor de instrucțiuni (urmărirea instrucțiunilor) pentru un program este secvența de instrucțiuni care se execută pentru acel proces.

2. Care sunt evenimentele cele mai cunoscute care conduc la crearea unui proces?

Crearea unui nou job, logarea interactivă, crearea de către OS ca urmare a execuției unui serviciu, generarea ca urmare a acțiunii unui proces existent.

3. Pentru modelul de procesare din Fig. 3.6 , definiți pe scurt fiecare stare.

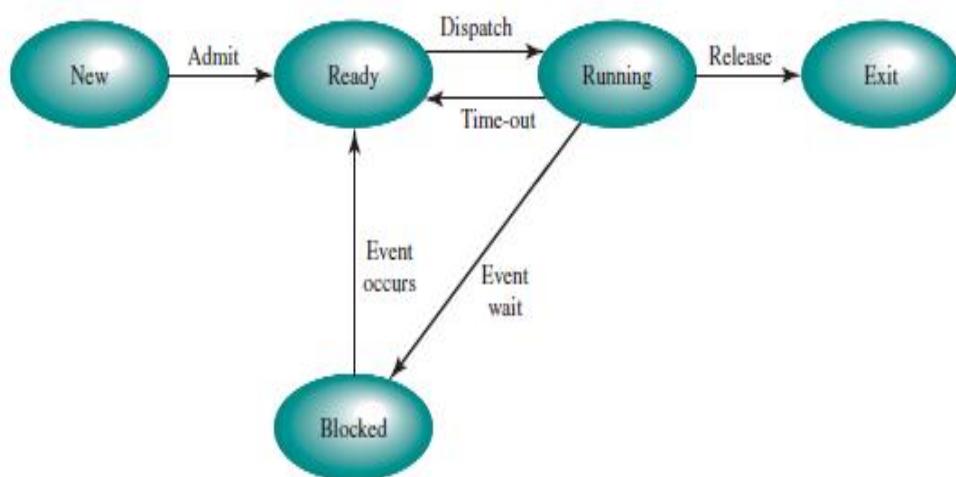


Figure 3.6 Five-State Process Model

Running: procesul este curent în execuție. Ready: un proces gata de execuția când va exista oportunitatea. Blocked: Un proces care nu se poate executa până nu apare un eveniment cum ar fi completarea unei operații I/O. New: un proces care a fost tocmai creat dar care nu a fost admis în aria proceselor executabile de către sistem. Exit: un proces care a fost eliminat din aria proceselor executabile de către SO doarece s-a terminat sau a fost abandonat din diverse motive.

4. Ce înseamnă întreruperea (suspendarea) unui proces - What does it mean to preempt a process?

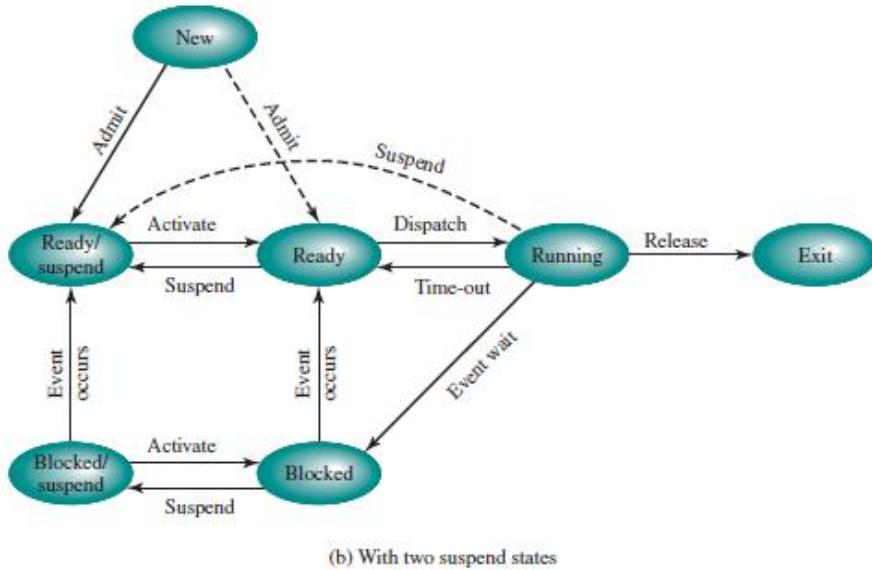
Suspendarea unui proces apare când procesul curent este întrerupt de către procesor cu scopul de a executa alt proces.

5. Ce se înțelege prin swapping și care este scopul acestuia.

Prin swapping se înțelege mutare de părți sau a întregului proces din memoria principală pe disk. Când nu există nici un proces din memoria principală gata de execuție SO comută unul din procesele blocate pe disk într-o coadă de procese

suspendate, astfel încât alt proces poate fi adus în memoria principală pentru a fi executat.

6. De ce sunt două stări blocate în figura Fig. 3.9b?



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

Există două concepte independente: când un proces așteaptă un eveniment (blocat sau nu), și când un proces a fost comutat (swapped) din memoria principală (suspendat sau nu). Pentru a ține cont de aceste 2×2 combinații, avem nevoie de două stări Ready și două stări Blocked.

7. Enunțați patru caracteristici ale unui proces suspendat.

1. Procesul nu este disponibil imediat pentru execuție. **2.** Procesul poate sau nu să aștepte un eveniment. Dacă da, această condiție de blocare este independentă de condiția de suspendare, și apariția condiției de blocare nu face ca procesul să fie executat. **3.** Procesul a fost plasat într-o stare de suspendare de un agent; fie de către el însuși, de un proces părinte, ori de SO cu scopul de a preveni execuția sa. **4.** Procesul nu poate fi scos din această stare până ce agentul nu ordonă explicit acest lucru.

8. Pentru ce tipuri de entități SO menține tabele cu informații pentru management?

Pentru memorie, I/O, fișiere și procese.

9. Enunțați trei categorii generale de informații dintr-un bloc de control al procesului.

Identificarea procesului, informația de stare a procesorului și informația de control a procesului.

10. De ce sunt necesare două moduri de lucru (utilizator și kernel)?

Modul utilizator are restricții asupra instrucțiunilor care pot fi executate și a zonei de memorie pe care o poate accesa. Astfel se protejează SO de alterare sau stricări. În modul kernel, SO nu are aceste restricții, pentru așa putea îndeplini sarcinile sale.

11. Care sunt pașii realizati de un SO pentru a crea un nou proces?

1. Asignarea unui identificator unic noului proces. **2.** Alocarea spațiului pentru proces. **3.** Inițializarea blocului de control al procesului. **4.** Setarea legăturilor corespunzătoare. **5.** Crearea sau expandarea altor structuri de date.

12. Care este diferența între o intrerupere și o capcană (trap)?

Întreruperea are loc ca urmare a unui tip de eveniment care este extern și independent de procesul curent în execuție, cum ar fi completarea unei operații I/O. O capcană se referă la o eroare sau condiție de excepție generată de procesul curent în execuție, cum ar fi încercarea de a accesa un fișier ilegal.

13. Dați trei exemple de intreruperi.

De ceas, I/O, externă pe un pin al procesorului.

14. Care este diferența între comutarea între moduri și comutarea între procese?

Comutarea modului apare fără schimbarea stării procesului care este în starea Running. O comutare de proces implică scoaterea procesului curent în execuție în afara stării Running, în favoarea altui proces. Comutarea proceselor implică salvarea și eventual restaurarea informațiilor de stare.

15. Tabelul următor cu tranziții de stare este un model simplificat al managementului proceselor, unde etichetele reprezintă tranzițiile între stările READY, RUN, BLOCKED, și NONRESIDENT.

	READY	RUN	BLOCKED	NONRESIDENT
READY	-	1	-	5
RUN	2	-	3	-
BLOCKED	4	-	-	6

Dați un exemplu de un eveniment care poate cauza oricare dintre tranzițiile din tabel. Desenați o diagramă dacă acest lucru vă ajută.

RUN la READY poate fi cauzată de expirarea cuantei de timp. READY la NONRESIDENT apare dacă memorie este plină și un proces este temporar comutat în afara memoriei (swapped out). READY la RUN apare dacă un proces este alocat UCP-ului de către dispecer. RUN la BLOCKED poate să apară dacă un proces lansează o operație I/O sau altă cerere către kernel. BLOCKED la READY apare dacă s-a

completat un eveniment așteptat (completarea unei operații I/O). BLOCKED la NONRESIDENT – la fel ca la READY la NONRESIDENT.

16. Presupuneți că la momentul 5 nu se utilizează nici o resursă a sistemului cu excepția procesorului și a memoriei. Considerați acum următoarele evenimente:

- La momentul 5: P1 execută o comandă de citire de pe unitatea 3 de disk.
- La momentul 15: timpul pentru P5 expiră.
- La momentul 18: P7 execută o comandă de scriere pe unitatea 2 de disk.
- La momentul 20: P3 execută o comandă de citire de pe unitatea 2 de disk.
- La momentul 24: P5 execută o comandă de scriere pe unitatea 3 de disk.
- La momentul 28: P5 este comutat din memorie (swapped out).
- La momentul 33: Apare o întrerupere de la unitatea 2 de disk: citirea lui P3 este completă.
- La momentul 36: Apare o întrerupere de la unitatea 3 de disk: citirea lui P1 este completă.
- La momentul 38: P8 se termină.
- La momentul 40: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P5 este completă.
- La momentul 44: P5 este adus înapoi în memorie (swapped back in).
- La momentul 48: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P7 este completă.

Pentru momentul 22 identificați în ce stare este fiecare proces. Dacă procesul este blocat, identificați ce eveniment l-a blocat.

P1: blocat pentru I/O. P3: blocat pentru I/O. P5: ready/running. P7: blocat pentru I/O. P8: ready/running.

17. Presupuneți că la momentul 5 nu se utilizează nici o resursă a sistemului cu excepția procesorului și a memoriei. Considerați acum următoarele evenimente:

- La momentul 5: P1 execută o comandă de citire de pe unitatea 3 de disk.
- La momentul 15: timpul pentru P5 expiră.
- La momentul 18: P7 execută o comandă de scriere pe unitatea 2 de disk.
- La momentul 20: P3 execută o comandă de citire de pe unitatea 2 de disk.
- La momentul 24: P5 execută o comandă de scriere pe unitatea 3 de disk.
- La momentul 28: P5 este comutat din memorie (swapped out).
- La momentul 33: Apare o întrerupere de la unitatea 2 de disk: citirea lui P3 este completă.
- La momentul 36: Apare o întrerupere de la unitatea 3 de disk: citirea lui P1 este completă.
- La momentul 38: P8 se termină.
- La momentul 40: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P5 este completă.
- La momentul 44: P5 este adus înapoi în memorie (swapped back in).
- La momentul 48: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P7 este completă.

Pentru momentul 37 identificați în ce stare este fiecare proces. Dacă procesul este blocat, identificați ce eveniment l-a blocat.

P1: ready/running. P3: ready/running. P5: blocat cu suspendare. P7: blocat pentru I/O. P8: exit.

18. Presupuneți că la momentul 5 nu se utilizează nici o resursă a sistemului cu excepția procesorului și a memoriei. Considerați acum următoarele evenimente:

- La momentul 5: P1 execută o comandă de citire de pe unitatea 3 de disk.
- La momentul 15: timpul pentru P5 expiră.
- La momentul 18: P7 execută o comandă de scriere pe unitatea 2 de disk.
- La momentul 20: P3 execută o comandă de citire de pe unitatea 2 de disk.
- La momentul 24: P5 execută o comandă de scriere pe unitatea 3 de disk.
- La momentul 28: P5 este comutat din memorie (swapped out).
- La momentul 33: Apare o întrerupere de la unitatea 2 de disk: citirea lui P3 este completă.
- La momentul 36: Apare o întrerupere de la unitatea 3 de disk: citirea lui P1 este completă.
- La momentul 38: P8 se termină.
- La momentul 40: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P5 este completă.
- La momentul 44: P5 este adus înapoi în memorie (swapped back in).
- La momentul 48: Apare o întrerupere de la unitatea 3 de disk: scrierea lui P7 este completă.

Pentru momentul 47 identificați în ce stare este fiecare proces. Dacă procesul este blocat, identificați ce eveniment l-a blocat.

P1: ready/running. P3: ready/running. P5: ready cu suspendare. P7: blocat pentru I/O. P8: exit.

19. Pentru modelul de proces cu 7 stări din Figura 3.9b desenați o diagramă a cozilor similară cu aceea din Figura 3.8b .

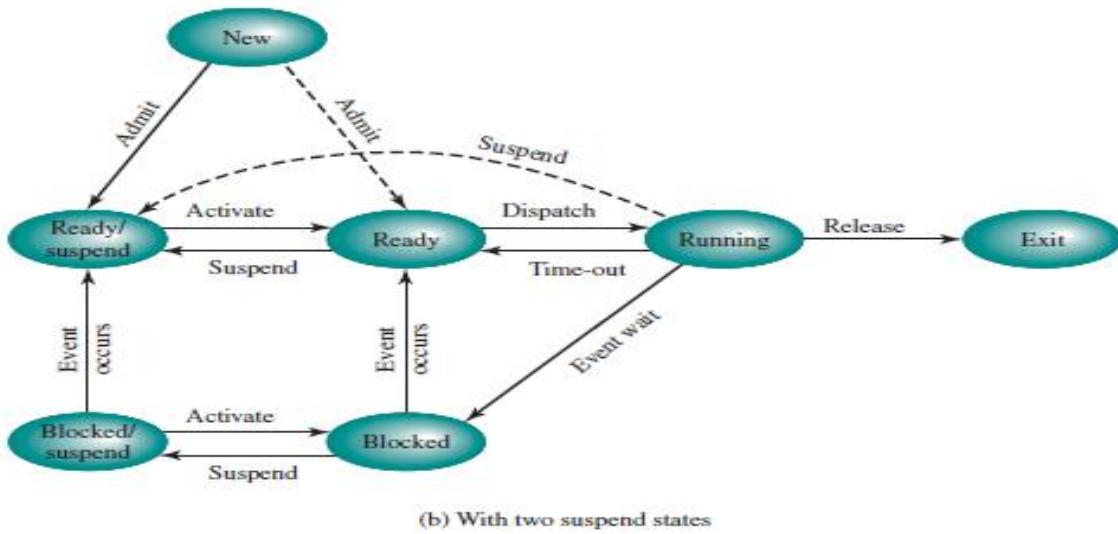


Figure 3.9 Process State Transition Diagram with Suspend States

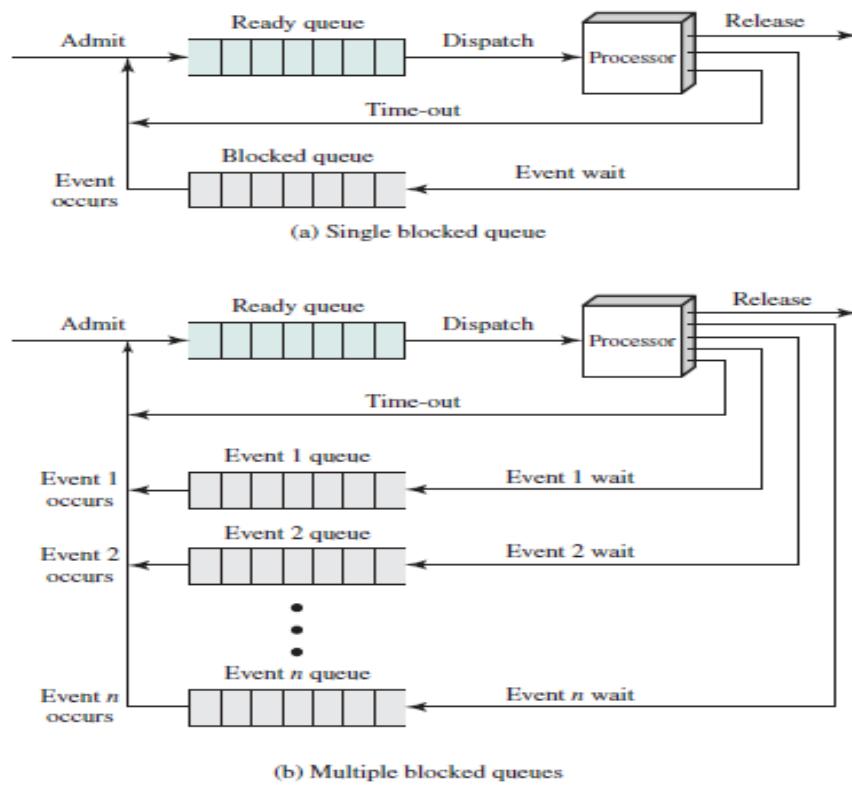


Figure 3.8 Queueing Model for Figure 3.6

Raspuns:

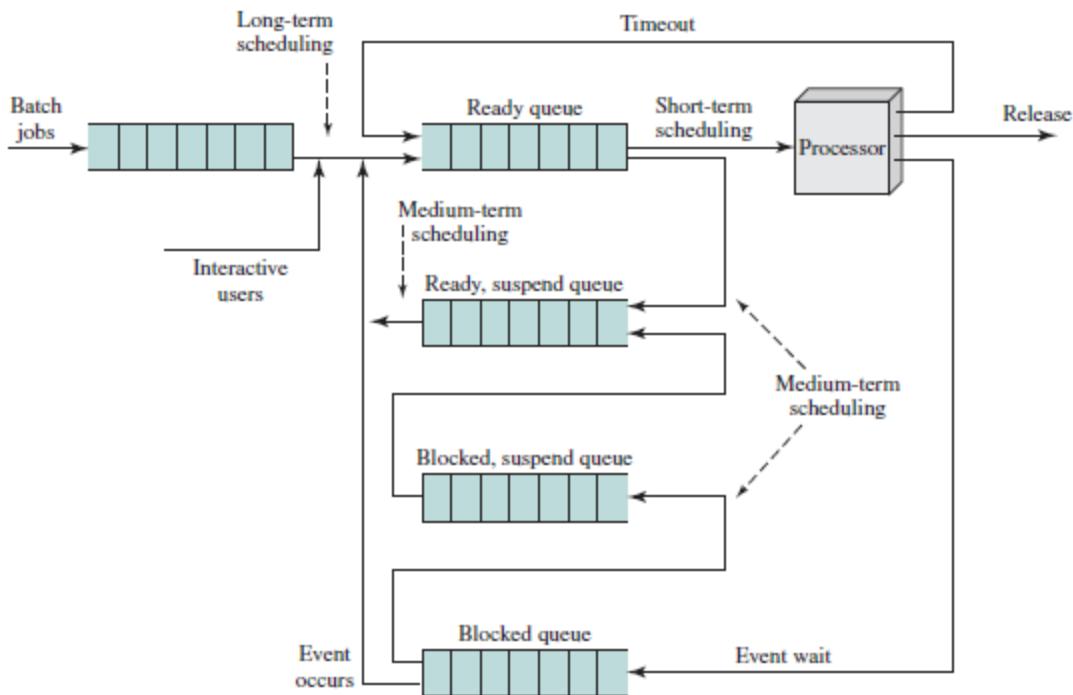
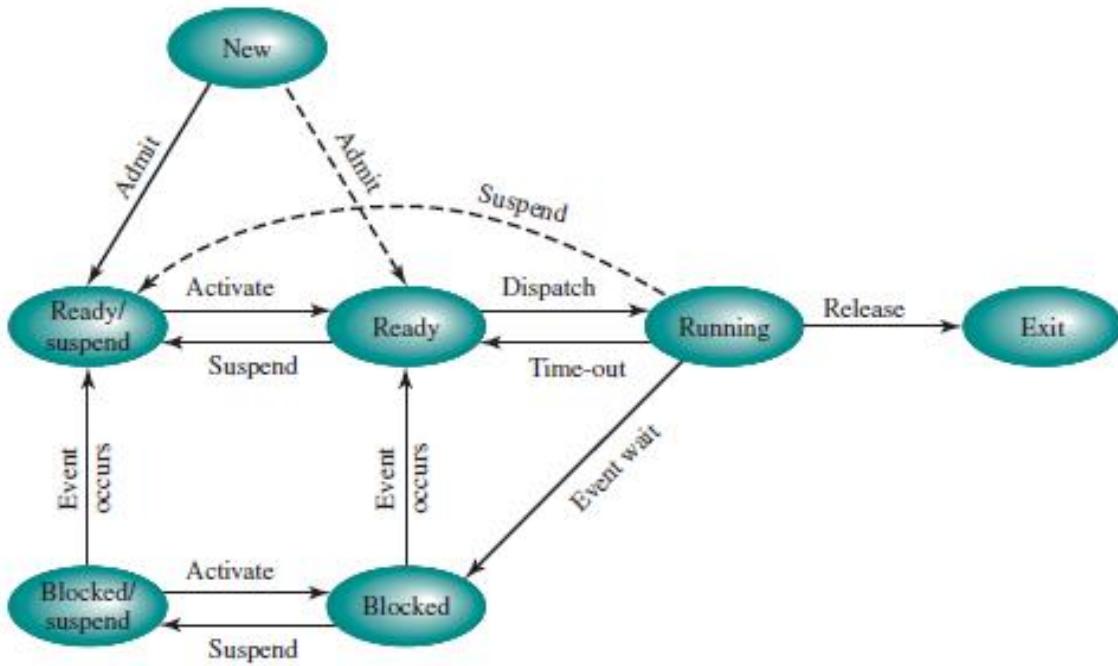


Figure 9.3 Queueing Diagram for Scheduling

20. Considerați diagrama de tranziție a stărilor din Figura 3.9b . Presupuneți că este timpul ca SO să dispecereze un proces și că sunt trei procese atât în starea Ready cât și în starea Ready/Suspend, și că cel puțin un proces din starea Ready/Suspend are o prioritate mai mare de planificare decât oricare alt proces din starea Ready. Există două politici extreme după cum urmează: (1) dispecerează întotdeauna un proces din starea Ready, pentru a minimiza timpul de comutare (swapping), și (2) întotdeauna alege procesul cu prioritatea cea mai mare chiar dacă aceasta ar însemna comutare (swapping) când aceasta nu este necesară . Sugerați o politică intermedieră care să pună în balanț prioritatea și performanță.



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

Se penalizează procesele în starea ready, suspend cu o cantitate fixă, cum ar fi cu unul sau două niveluri de prioritate, astfel încât un proces ready, suspend este ales ca următorul proces numai dacă are o prioritate mai mare decât cel mai priorită proces din starea Ready dintre mai multe niveluri de prioritate.

21. Tabelul 3.13 prezintă stările procesorului pentru sistemul de operare VAX/VMS.

Puteți da o justificare existenței atâtorexistării stări distincte de așteptare? (wait states)?

Table 3.13 VAX/VMS Process States

Process State	Process Condition
Currently Executing	Running process.
Computable (resident)	Ready and resident in main memory.
Computable (outswapped)	Ready, but swapped out of main memory.
Page Fault Wait	Process has referenced a page not in main memory and must wait for the page to be read in.
Collided Page Wait	Process has referenced a shared page that is the cause of an existing page fault wait in another process, or a private page that is in the process of being read in or written out.
Common Event Wait	Waiting for shared event flag (event flags are single-bit interprocess signaling mechanisms).
Free Page Wait	Waiting for a free page in main memory to be added to the collection of pages in main memory devoted to this process (the working set of the process).
Hibernate Wait (resident)	Process puts itself in a wait state.
Hibernate Wait (outswapped)	Hibernating process is swapped out of main memory.
Local Event Wait (resident)	Process in main memory and waiting for local event flag (usually I/O completion).
Local Event Wait (outswapped)	Process in local event wait is swapped out of main memory.
Suspended Wait (resident)	Process is put into a wait state by another process.
Suspended Wait (outswapped)	Suspended process is swapped out of main memory.
Resource Wait	Process waiting for miscellaneous system resource.

Pentru fiecare stare se asociază o coadă separată. Diferențierea proceselor care așteaptă în cozi reduce munca necesară pentru a localiza un proces care așteaptă atunci când acesta este afectat de un eveniment. De exemplu când se completează o eroare de pagină, planificatorul știe că procesul care așteaptă poate fi găsit în coada Page Fault Wait.

22. Tabelul 3.13 prezintă stările procesorului pentru sistemul de operare VAX/VMS.

De ce următoarele stări : Page Fault Wait, Collided Page Wait, Common Event Wait, Free Page Wait, și Resource Wait, nu au versiuni de tip rezidență și comutare în afară (resident and swapped-out versions).

Table 3.13 VAX/VMS Process States

Process State	Process Condition
Currently Executing	Running process.
Computable (resident)	Ready and resident in main memory.
Computable (outswapped)	Ready, but swapped out of main memory.
Page Fault Wait	Process has referenced a page not in main memory and must wait for the page to be read in.
Collided Page Wait	Process has referenced a shared page that is the cause of an existing page fault wait in another process, or a private page that is in the process of being read in or written out.
Common Event Wait	Waiting for shared event flag (event flags are single-bit interprocess signaling mechanisms).
Free Page Wait	Waiting for a free page in main memory to be added to the collection of pages in main memory devoted to this process (the working set of the process).
Hibernate Wait (resident)	Process puts itself in a wait state.
Hibernate Wait (outswapped)	Hibernating process is swapped out of main memory.
Local Event Wait (resident)	Process in main memory and waiting for local event flag (usually I/O completion).
Local Event Wait (outswapped)	Process in local event wait is swapped out of main memory.
Suspended Wait (resident)	Process is put into a wait state by another process.
Suspended Wait (outswapped)	Suspended process is swapped out of main memory.
Resource Wait	Process waiting for miscellaneous system resource.

În fiecare caz, va fi mai puțin eficient de a permite comutarea procesului când acesta este în această stare. De exemplu, așteptarea pentru o eroare de pagină, nu are sens să conduce la o comutare a procesului, așteptând aducerea unei pagini astfel încât acesta să poată fi executat.

23. Tabelul 3.13 prezintă stările procesorului pentru sistemul de operare VAX/VMS. Desenați diagrama cu tranzitiiile stărilor și indicați acțiunea sau evenimentul care cauzează fiecare tranzitie.

Table 3.13 VAX/VMS Process States

Process State	Process Condition
Currently Executing	Running process.
Computable (resident)	Ready and resident in main memory.
Computable (outswapped)	Ready, but swapped out of main memory.
Page Fault Wait	Process has referenced a page not in main memory and must wait for the page to be read in.
Collided Page Wait	Process has referenced a shared page that is the cause of an existing page fault wait in another process, or a private page that is in the process of being read in or written out.
Common Event Wait	Waiting for shared event flag (event flags are single-bit interprocess signaling mechanisms).
Free Page Wait	Waiting for a free page in main memory to be added to the collection of pages in main memory devoted to this process (the working set of the process).
Hibernate Wait (resident)	Process puts itself in a wait state.
Hibernate Wait (outswapped)	Hibernating process is swapped out of main memory.
Local Event Wait (resident)	Process in main memory and waiting for local event flag (usually I/O completion).
Local Event Wait (outswapped)	Process in local event wait is swapped out of main memory.
Suspended Wait (resident)	Process is put into a wait state by another process.
Suspended Wait (outswapped)	Suspended process is swapped out of main memory.
Resource Wait	Process waiting for miscellaneous system resource.

Raspuns:

Current State	Next State				
	Currently Executing	Computable (resident)	Computable (outswapped)	Variety of wait states (resident)	Variety of wait states (outswapped)
Currently Executing		Rescheduled		Wait	
Computable (resident)	Scheduled		Outswap		
Computable (outswapped)		Inswap			
Variety of wait states (resident)		Event satisfied	Outswap		
Variety of wait states (outswapped)			Event satisfied		

24.3.7 SO VAX/VMS utilizează patru moduri de acces la procesor pentru a facilita protecția și partajarea resurselor sistemului de către proceze. Modul de acces determină:

- **Nivelul de prioritate al execuției instrucțiunilor:** care instrucțiuni le poate executa procesorul
- **Nivelul de prioritate privind accesul la memorie:** la care locație din memoria virtuală are acces instrucțiunea curentă.

Cele patru moduri sunt:

- Kernel: Execută kernelul SO VMS, care include gestiunea memoriei, a întreruperilor și operațiile I/O.
- Executive: Execută cele mai multe dintre apelurile SO, inclusiv rutinele de gestiune a înregistrărilor pe disk și bandă și a fișierelor.
- Supervisor: Execută alte servicii ale SO, cum ar fi răspunsul la o comandă utilizator.
- User: Execută programe utilizator, plus utilități cum ar fi compilatoare, editoare, editoare de legături (linkers) și depanatoare.

Un proces care se execută pe nivelul cel mai puțin priorită adesea trebuie să apeleze proceduri care se execută pe nivelurile mai prioritare: de exemplu, un program utilizator cere un serviciu al SO. Aceste apel se realizează utilizând o instrucțiune de schimbare a modului (change-mode (CHM) instruction), care cauzează o întrerupere care transferă controlul unei rutine specifice noului mod de acces. O revenire se face executând o instrucțiune de tip REI (return from exception or interrupt). Unele SO au două moduri, kernel și user. Care sunt avantajele și dezavantajele furnizării a patru moduri în loc de două?

Avantajul celor patru moduri este acela că există mai multă flexibilitate pentru a controla accesul la memorie, permitând un control mai fin al protecției acesteia. Dezavantajul constă în complexitate și procesarea suplimentară.

25.3.7 SO VAX/VMS utilizează patru moduri de acces la procesor pentru a facilita protecția și partajarea resurselor sistemului de către proceze. Modul de acces determină:

- **Nivelul de prioritate al execuției instrucțiunilor:** care instrucțiuni le poate executa procesorul
- **Nivelul de prioritate privind accesul la memorie:** la care locație din memoria virtuală are acces instrucțiunea curentă.

Cele patru moduri sunt:

- Kernel: Execută kernelul SO VMS, care include gestiunea memoriei, a întreruperilor și operațiile I/O.
- Executive: Execută cele mai multe dintre apelurile SO, inclusiv rutinele de gestiune a înregistrărilor pe disk și bandă și a fișierelor.
- Supervisor: Execută alte servicii ale SO, cum ar fi răspunsul la o comandă utilizator.
- User: Execută programe utilizator, plus utilități cum ar fi compilatoare, editoare, editoare de legături (linkers) și depanatoare.

Un proces care se execută pe nivelul cel mai puțin priorită adesea trebuie să apeleze proceduri care se execută pe nivelurile mai prioritare: de exemplu, un program utilizator cere un serviciu al SO. Aceste apel se realizează utilizând o instrucțiune de schimbare a modului (change-mode (CHM) instruction), care cauzează o întrerupere care transferă controlul unei rutine specifice noului mod de acces. O revenire se face executând o instrucțiune de tip REI (return from exception or interrupt). Puteți să susțineți ideea de a folosi chiar mai mult de patru moduri?

În principiu, mai multe noduri mai multă flexibilitate, dar este dificil de justificat mărirea numărului acestora.

26. Schema VMS discutată în problema precedentă este adesea denumită structură de protecție în inel, aşa cum se prezintă în figura Figura 3.18. Într-adevăr schema simplă kernel/user aşa cum este descrisă în secțiunea 3.3 este o structură în inel cu două niveluri. [SILB04] a evidențiat o problemă referitoare la această abordare. Principalul dezavantaj al structurii în inel (ierarhice) este aceea că nu ne permite să punem în practică principiul nevoie de a cunoaște (need-to-know principle). Astfel, dacă un obiect trebuie să fie accesibil în domeniul D_j dar nu trebuie să fie accesibil în domeniul D_i , atunci trebuie să avem $j < i$. Dar aceasta înseamnă că fiecare segment accesibil în D_i este de asemenea accesibil D_j . Explicați mai clar care este problema referită în citarea anterioară.

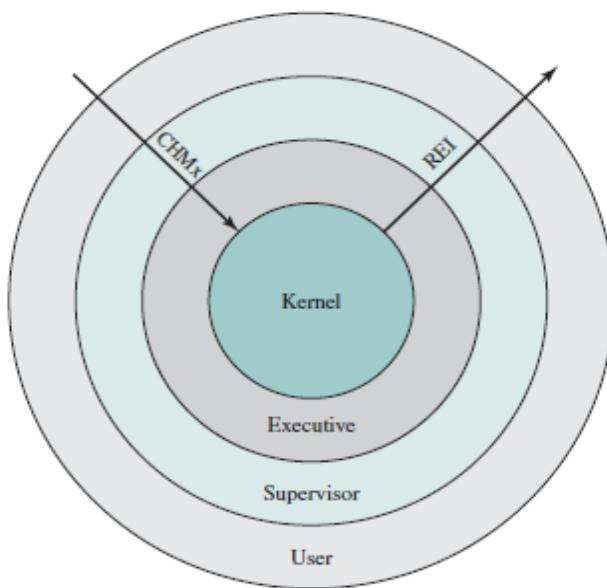


Figure 3.18 VAX/VMS Access Modes

Dacă $j < i$, un proces care rulează în D_i este oprit ca să acceseze obiecte din D_j . Astfel, dacă D_j conține informații care sunt mai privilegiate sau care trebuie să fie mai securizate decât informațiile din D_i , atunci această restricție este corespunzătoare. Totuși, această politică de securitate poate fi evitată în felul următor. Un proces rulând în D_j poate citi date din D_j și apoi le poate copia în D_i . În consecință, un proces rulând în D_i poate accesa această informație.

27. Figura 3.8b sugerează că un proces poate fi la un moment dat într-o singură coadă de evenimente. Este posibil ca să doriți să permiteți ca un proces să aștepte mai mult de un eveniment la un moment dat? Dați un exemplu.

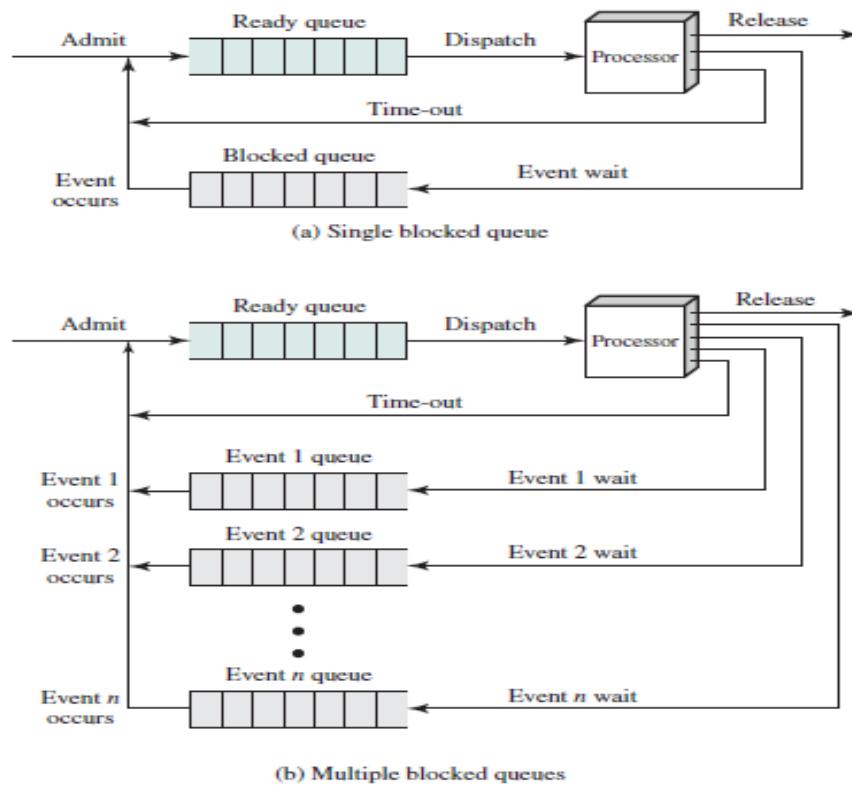


Figure 3.8 Queueing Model for Figure 3.6

Aplicația poate procesa date recepționate de la un alt proces și poate memora rezultatele pe disk. Dacă există date care așteaptă a fi luate de la alt proces, aplicația poate să aleagă să ia acele date și să le proceseze. Dacă scrierea anterioară pe disk s-a completat, și există date procesate pentru a fi scrise, aplicația poate să treacă la scrierea pe disk. Poate fi un punct unde procesul așteaptă atât pentru datele de intrare de la procesul de intrare, cât și pentru disponibilitatea discului.

28. Figura 3.8b (vezi slide-urile precedente) sugerează că un proces poate fi la un moment dat într-o singură coadă de evenimente. Este posibil ca să doriți să permiteți ca un proces să aștepte mai mult de un eveniment la un moment dat. Dați un exemplu. În acest caz va trebui să modificați structura cozilor din figură pentru a suporta această nouă facilitate?

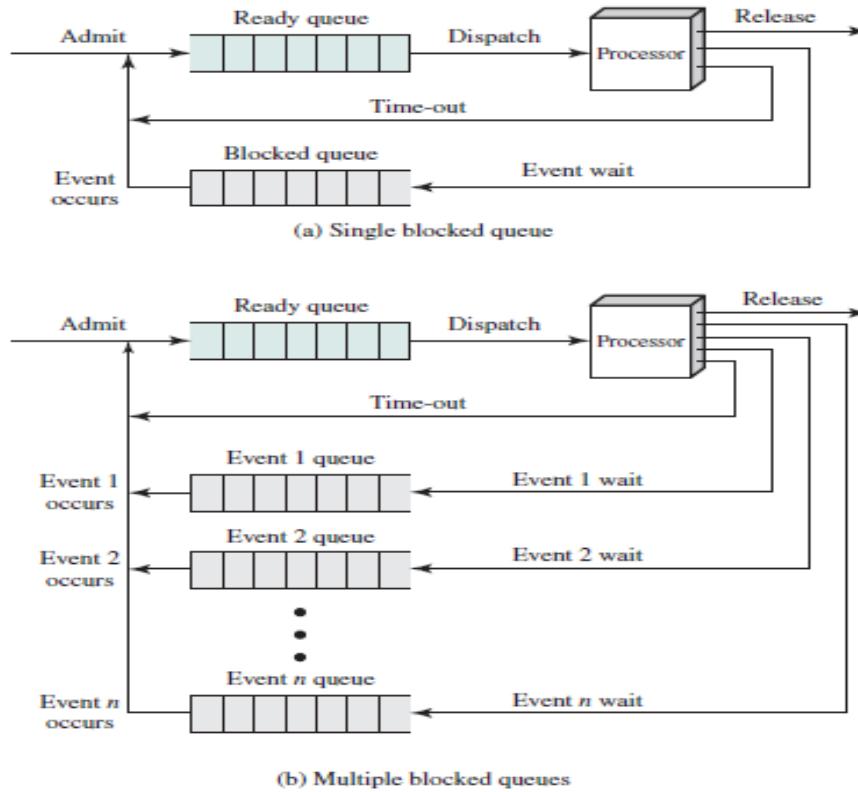


Figure 3.8 Queueing Model for Figure 3.6

Există mai multe moduri în care se poate trata acest lucru. Poate fi utilizat un tip special de coadă, sau procesul poate fi pus în două cozi separate. În ambele cazuri, SO trebuie să aibă posibilitatea de a procesa detaliiile pentru alerta procesul de apariția ambelor evenimente, unul după altul.

29. În unele calculatoare mai timpurii, o întrerupere cauza ca valorile din registrii să fie memorate în locații fixe asociate cu semnalul de întrerupere dat. În ce circumstanțe este o tehnică practică? Explicați de ce este în general un inconvenient.

Această tehnică este bazată pe presupunerea că un proces A întrerupt va continua să ruleze după răspunsul la o întrerupere. Dar, în general, o întrerupere poate cauza ca monitorul de bază să suspende procesul A în favoarea unui alt proces B. Este acum necesar să se copieze starea de execuție a procesului A de la locația asociată cu întreruperea la locația de descriere a procesului asociat cu A.

30. În secțiunea 3.4, s-a arătat că UNIX nu este potrivit pentru aplicații de timp real deoarece un proces care se execută în modul kernel nu poate fi suspendat. Detaliați.

Deoarece există circumstanțe sub care un proces nu poate fi suspendat (se execută în modul kernel), este imposibil pentru SO să răspundă rapid la cerințele de timp real.

31. Se executa urmatorul program în C:

```
main ()  
{
```

```
int pid;  
pid = fork ();  
printf ("%d \n", pid);  
}
```

Care sunt posibilele ieșiri presupunând că fork se execută cu succes?

Raspuns:

- 0
- <child pid>
- sau
- <child pid>
- 0

Cap. 2 COMPUTER SYSTEM OVERVIEW-2

1. Care sunt trei dintre obiectivele proiectării unui sistem de operare?

Confortabilitate: un sistem de operare face un computer mult mai ușor de utilizat. Eficiență: un sistem de operare permite ca resursele sistemului de calcul să fie utilizate într-o manieră eficientă. Abilitatea de a evoluă: un sistem de operare trebuie construit într-un astfel de mod încât să permită dezvoltarea efectivă, testarea și introducerea unor noi funcții sistem fără a afecta serviciile deja oferite.

2. Ce este nucleul (kernel-ul) unui sistem de operare?

Nucleul (kernel-ul) este o porțiune a sistemului de operare care include software-ul cel mai solicitat din SO. Este menținut permanent în memoria principală. Rulează în modul privilegiat și răspunde la apelurile de la procese și la întreruperile generate de dispozitive.

3. Ce este multiprogramarea?

Este un mod de operare care permite execuția întrețesută a două sau mai multe programe de calculator de către un singur procesor.

4. Ce este un proces?

Este un program în execuție. Este controlat și planificat de către SO.

5. Care este contextul de execuție al unui proces utilizat de către sistemul de operare OS?

Reprezintă datele interne prin care SO este capabil să supraveze și să controleze procesul. Sunt informații interne separate de proces, deoarece SO are informații la care nu pot avea acces procesele.

6. Enumerați și explicați pe scurt cinci responsabilități ale gestiunii sistemului de memorare ale unui SO tipic.

Izolarea proceselor: acestea nu trebuie să interfere între ele în zona de date și instrucțiuni. Alocarea automată și managementul: programele trebuie să fie alocate și gestionate dinamic fără ca programatorul să aibă grija acestor operații. Suport pentru programarea modulară: programatorii trebuie să poată defini module de program, să creeze, și să altereze dinamic mărimea acestor module. Protecția și controlul accesului: asigură protecția memoriei la diferite tipuri de accese și controlul accesului între mai mulți utilizatori. Memorarea pe termen lung: Multe aplicații solicită stocarea informației pentru dure de timp.

7. Explicați diferența dintre o adresă reală și o adresă virtuală.

O adresă virtuală se referă la o locație de memorie virtuală. O astfel de locație este pe disc și în același timp în memoria principală. O adresă reală este o adresă în memoria principală.

8. Descrieți tehnica de planificare de tip round robin.

Este un algoritm de planificare care planifică (activează) procesele într-o ordine ciclică fixă utilizând o coadă circulară.

9. Explicați diferența dintre un nucleu monolitic și un microkernel.

Un kernel monolitic este un kernel extins care conține aproape întreg SO. Un microkernel este un nucleu de mărime mică, privilegiat care furnizează planificarea proceselor, gestiunea memoriei și serviciile de comunicație.

10. Ce este multithreading (fire de execuție multiple)?

Este o tehnică prin care un proces care execută o aplicație, este divizat în fire de execuție care pot rula concurrent.

11. Enumerați facilitățile cheie ale proiectării unui sistem de operare SMP (Multiprocesor Simetric).

Execuția proceselor sau firelor de execuție concurrent și simultan; sincronizarea; gestiunea memoriei; fiabilitatea și toleranța la defecte.

12. Presupuneți că aveți un calculator cu multiprogramare, unde fiecare job are caracteristici identice. Pe o perioadă de calcul, T, pentru un job se consumă o jumătate de perioadă cu operații I/O, și o jumătate de perioadă cu activitate procesor. Fiecare job rulează un timp de N perioade. Presupuneți că se utilizează o planificare simplă de tip round-robin, și că operațiile I/O se pot suprapune cu operațiile procesorului. Definiți următoarele cantități:

- Turnaround time = timpul actual de completare a unui job,
- Throughput = numărul mediu de job-uri completeate per perioada T
- Processor utilization = procentul de timp cât procesorul este activ.

Calculați aceste cantități pentru unul, două și patru job-uri simultane, presupunând că perioada T este distribuită astfel: I/O prima jumătate, procesorul a doua jumătate.

Number of jobs	TAT	Throughput	Processor utilization
1	NT	1/N	50%
2	NT	2/N	100%
4	(2N - 1)T	4/(2N - 1)	100%

13. Presupuneți că aveți un calculator cu multiprogramare, unde fiecare job are caracteristici identice. Pe o perioadă de calcul, T, pentru un job se consumă o jumătate de perioadă cu operații I/O, și o jumătate de perioadă cu activitate

procesor. Fiecare job rulează un timp de N perioade. Presupuneți că se utilizează o planificare simplă de tip round-robin, și că operațiile I/O se pot suprapune cu operațiile procesorului. Definiți următoarele cantități:

- Turnaround time = timpul actual de completare a unui job,
- Throughput = numărul mediu de job-uri completate per perioada T
- Processor utilization = procentul de timp cât procesorul este activ.

Calculați aceste cantități pentru unul, două și patru job-uri simultane, presupunând că perioada T este distribuită în unul din următoarele moduri:
a. I/O prima jumătate, procesorul a doua jumătate;

Number of jobs	TAT	Throughput	Processor utilization
1	NT	1/N	50%
2	NT	2/N	100%
4	(2N – 1)T	4/(2N – 1)	100%

14. Un program orientat pe operații I/O este acela care dacă rulează singur va petrece mai mult timp așteptând completarea operațiilor I/O decât utilizând procesorul. Un program orientat pe utilizarea procesorului este unul opus primului tip. Presupuneți că un planificator pe termen scurt favorizează acele programe care au utilizat puțin procesorul în trecutul apropiat. Explicați de ce acest algoritm favorizează programele cu operații I/O și totuși nu refuză permanent să aloce timp procesor programelor orientate utilizarea procesorului.

Programele orientate pe operații I/O utilizează puțin timp procesor și sunt favorizate de algoritm. Totuși dacă un proces orientat pe calcule este refuzat pe o perioadă suficient de lungă, același algoritm va oferi procesorul acestui proces deoarece nu a utilizat procesorul în trecutul apropiat. Ca urmare procesele orientate pe calcule nu vor fi permanent refuzate.

15. Comparați politicile de planificare pe care le puteți utiliza când încercați să optimizați sistemele cu partajarea timpului cu cele pe care le utilizați să optimizați un sistem cu multiprogramarea loturilor de job-uri.

Partajarea timpului favorizează timpul turnaround (de așteptare pentru a fi executat). Planificarea bazată pe cuante de timp este preferată deoarece oferă tuturor proceselor accesul la procesor pentru o scurtă perioadă de timp. În sistemele care lucrează cu loturi (batch) preocupările se referă la viteza fluxului de prelucrare, mai puține comutări de context și timpul mai mare disponibil pentru calcul. Ca urmare, politicile care minimizează comutările de context sunt favorite.

16. Care este scopul apelurilor sistem, și cum se raportează apelurile sistem la SO și la conceptul modului de operare dual (modul kernel și modul utilizator)?

Un apel de sistem este utilizat de un program pentru a invoca o funcție furnizată de sistemul de operare. Tipic, un apel de sistem determină un transfer către un program sistem care se execută în modul nucleu.

17. În mainframe-ul IBM OS, OS/390, unul din modulele principale ale nucleului este managerul resurselor sistemului (System Resource Manager – SRM). Acest modul este responsabil cu alocarea resurselor între spațiile de adrese (procese). SRM oferă pentru OS/390 un grad de SO sofisticat între SO-uri. Nici un alt SO de mainframe, și cu siguranță nici un alt tip SO, nu are aceste funcții realizate de SRM. Conceptul de resurse include procesorul, memoria reală, și canalele I/O. SRM acumulează statistică privind utilizarea procesorului, canalelor și diferitelor structuri de date cheie. Scopul său este de a furniza performanțe optime bazate pe monitorizare și analiză. Instalarea setează patru obiective variate privind performanța, și acestea ajută ca ghid pentru SRM, care modifică în mod dinamic caracteristicile de instalare și performanțele job-urilor pe baza utilizării sistemului. În schimb, SRM furnizează rapoarte care permit unui operator antrenat să rafineze configurarea și setarea parametrilor pentru a îmbunătății serviciile pentru utilizatori. Această problemă se referă la activitatea SRM. Memoria reală este divizată în blocuri de dimensiune egală denumite cadre (frames), care pot fi în număr mare. Fiecare cadru poate păstra un bloc de memorie virtuală denumit pagină. SRM reține controlul de aproximativ 20 de ori pe secundă, și inspectează fiecare și oricare cadru de pagină. Dacă pagina nu a fost referită (accesată) sau modificată se incrementează cu 1 un contor. În timp SRM mediază aceste numere pentru a determina numărul mediu de secunde cât o pagină este neaținsă. Care poate fi scopul acestei operații și ce acțiune poate lua SRM?

Operatorul de sistem poate analiza această cantitate pentru a determina stresul din sistem. Prin reducerea numărului de joburi active permise în sistem, această poate fi ținută la valori mari. În mod tipic această medie trebuie ținută peste 2 minute. Aceasta poate însemna mult, dar nu este.

18. Un sistem multiprocesor cu 8 procesoare are atașate 20 de benzi magnetice. Există un număr mare de sarcini atașate sistemului care necesită un maximum de patru benzi pentru ași completa execuția. Presupuneți că fiecare job pornește execuția pentru trei benzi pentru o perioadă lungă înainte de a cere o patră bandă pentru o perioadă scurtă, înainte de ași termina operarea. Presupuneți, de asemenea, o alimentare fără sfârșit de astfel de job-uri. Presupuneți că planificatorul din SO nu pornește un job fără să aibă patru benzi disponibile. Când se pornește un job, cele patru benzi sunt asignate imediat și nu sunt eliberate până când nu se termină job-ul. Care este numărul maxim de job-uri care pot fi în simultan în execuție? Care este numărul maxim și minim de benzi care pot fi în așteptare ca rezultat al acestei politici?

5 procese active simultan; cel mult 5 driver-e pot fi în idle la un moment dat; În cel mai bun caz nici un drive nu va fi în idle.

19. Un sistem multiprocesor cu 8 procesoare are atașate 20 de benzi magnetice. A multiprocessor with eight processors has 20 attached tape drives. Există un număr mare de sarcini atașate sistemului care necesită un maximum de patru benzi pentru ași completa execuția. Presupuneți că fiecare job pornește execuția pentru trei benzi

pentru o perioadă lungă înainte de a cere a patra bandă pentru o perioadă scurtă, înainte de astfel de job-uri. Sugerați o politică alternativă pentru a îmbunătății utilizarea benzii și în același timp evitând congestia? Care este numărul maxim de job-uri care poate fi în execuție simultan. Care sunt limitele (bounds) privind numărul de benzi în așteptare?

Fiecare proces va avea alocate trei benzi. A patra se va aloca la cerere. În această politică pot fi active cel mult 6 procese. Numărul minim de benzi în așteptare este 0 și numărul maxim este 2.

Cap. 4 Fire de execuție

1. Tabelul 3.5 prezintă elementele tipice care se regăsesc într-un bloc de control al procesului pentru un SO fără fire de execuție. Dintre acestea, care pot apartine unui bloc de control al firului de execuție și care pot să aparțină unui bloc de control al procesului într-un sistem cu fire multiple de execuție.

Table 3.5 Typical Elements of a Process Control Block

Process Identification
Identifiers Numeric identifiers that may be stored with the process control block include <ul style="list-style-type: none">• Identifier of this process• Identifier of the process that created this process (parent process)• User identifier
Processor State Information
User-Visible Registers A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.
Control and Status Registers These are a variety of processor registers that are employed to control the operation of the processor. These include <ul style="list-style-type: none">• Program counter: Contains the address of the next instruction to be fetched• Condition codes: Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)• Status information: Includes interrupt enabled/disabled flags, execution mode
Stack Pointers Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.
Process Control Information
Scheduling and State Information This is information that is needed by the operating system to perform its scheduling function. Typical items of information: <ul style="list-style-type: none">• Process state: Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).• Priority: One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).• Scheduling-related information: This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.• Event: Identity of event the process is awaiting before it can be resumed.
Data Structuring A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.
Interprocess Communication Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.
Process Privileges Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.
Memory Management This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
Resource Ownership and Utilization Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Răspunsul depinde de sistem, dar în general, resursele sunt proprietatea proceselor și fiecare fir de execuție are propria stare de execuție. Câteva comentarii despre fiecare categorie din tabelul 3.5 ar putea fi următoarele: Identificarea: procesele trebuie identificate dar fiecare fir din proces trebuie să aibă propriul ID. Processor State Information: acestea se referă în general la procese. Process control information: informația de planificare și de stare trebuie să fie în majoritatea sa la nivelul fir de execuție; structurile de date pot să apară la ambele niveluri; pot fi suportate atât comunicațiile interproces și cât și cele interfir; Privilegiile pot să la ambele niveluri: informațiile despre resurse vor fi în general la nivelul proces.

2. Enumerați motivele pentru care o comutare de mod între firele de execuție poate fi mai rapidă decât o comutare de moduri între procese.

Este implicată mai puțină informație de stare.

3. Care sunt cele două caracteristici separate și potențial independente înglobate în conceptul de proces?

Proprietarul resurselor și planificarea/execuția.

4. Dați patru exemple generale de utilizare a firelor de execuție într-un sistem cu multiprocesare și un singur utilizator.

Lucrul pe ecran (foreground) și în fundal (background), procesarea asincronă, creșterea vitezei de execuție prin procesarea paralelă a datelor; structura modulară a programelor.

5. Care sunt resursele tipice partajate de toate firele de execuție ale unui proces?

Spațiul de adrese, resursele de tip fișier și privilegiul de execuție.

6. Enumerați trei avantaje ale utilizării ULT-urilor în raport cu KLT-urile.

1. Comutarea firelor nu necesită utilizarea privilegiilor kernelului deoarece toate structurile de date ale managementului firelor sunt în spațiul de adrese al utilizatorului al unui singur proces. Ca urmare, procesul nu va comuta în modul kernel pentru a realiza gestiunea firului. Aceasta salvează supracontrolul care ar fi fost dat de cele două comutări (user – kernel, kernel - user). 2. Planificarea poate fi specifică aplicației. O aplicație poate beneficia mai mult de pe urma unei planificări round-robin, în timp ce alta ar putea beneficia mai mult de pe urma unei planificări bazate pe priorități. Algoritmul de planificare poate fi croit pe aplicație fără a perturba planificatorul sistemului de operare. 3. ULT poate rula pe orice sistem de operare. Nu se cere nici o schimbare kernelului existent pentru a suporta ULT-uri. Libraria pentru fire este un set de utilități de la nivelul aplicație partajate de toate aplicațiile.

7. Enumerați două dezavantaje ale ULT-urilor în raport cu KLT-urile.

1. Într-un sistem de operare tipic, multe dintre apelurile sistem sunt cu blocare. Astfel, atunci când un ULT apelează o funcție sistem se blochează nu numai acel fir ci și toate celelalte fire ale procesului. 2. În strategia ULT pură, o aplicație multi-fir nu poate avea avantajele multiprocesării. Kernelul asignează un proces unui singur procesor la un moment dat. Ca urmare, numai un singur fir din proces se execută la un moment dat de timp.

8. Definiți jacketing-ul.

Jacketing-ul convertește un apel sistem cu blocare într-un apel sistem fără blocare prin utilizarea unor rutine I/O la nivelul aplicație care testează starea dispozitivelor I/O.

9. Au fost punctate două avantaje ale utilizării firelor de execuție multiple în cadrul unui proces care sunt (1) este necesară mai puțină muncă din partea procesorului pentru a crea un nou fir în cadrul unui proces decât a crea un proces nou și (2) comunicațiile între firele de execuție din cadrul aceluiași proces sunt simplificate. Comutarea între modurile de lucru între două fire de execuție în cadrul aceluiași proces implică mai puțină muncă din partea procesorului decât comutarea modului între două fire de execuție aparținând unor proceze diferite?

Da, deoarece trebuie salvată mai multă informație de stare pentru a comuta de la un proces la altul.

10. În discuția despre ULT-uri versus KLT-uri s-a subliniat că dezavantajul ULT-urilor apare atunci când ULT execută un apel sistem, fapt care conduce nu numai la blocarea ULT-ului care a realizat apelul, dar și la blocarea tuturor firelor de execuție din procesul care este blocat. De ce se întâmplă acest lucru?

Deoarece, cu ULT-urile, structurile firelor unui proces nu sunt vizibile sistemului de operare, care planifică doar pe baza proceselor.

11. Considerați un mediu în care există o mapare unu-la-unu între firele de la nivelul utilizator și firele de la nivelul kernel care permite ca unul sau mai multe fire din cadrul unui proces să genereze un apel de funcție sistem cu blocare, în timp ce alte fire continuă să ruleze. Explicați de ce acest model poate face ca programele multi-fir să ruleze mai repede decât modelul corespondent cu un singur fir din calculatoarele uniprocesor.

Ideia aici este că un procesor consumă o cantitate considerabilă de timp așteptând completarea operațiilor I/O. Într-un program multi-fir, un KLT poate realiza un apel sistem cu blocare, în timp ce alt KLT poate să-și continue execuția. Pe sisteme uniprocesor, un proces care se blochează

pentru toate aceste apeluri poate continua să ruleze alt fir dintre firele sale de execuție.

12. Dacă un proces dorește să se termine și există încă fire ale procesului care se execută, vor continua acestea să ruleze?

Nu. Un proces care se termină, ia totul cu el – KLT-urile, structurile procesului, spațiul de memorie – inclusiv firele.

13. Multe limbi de programare, cum ar fi C și C++, sunt inadecvate pentru programarea multi-fir. Aceasta poate avea impact asupra compilatoarelor și a corectitudinii codului astfel că se va ilustra în continuare această problemă. Să considerăm următoarele decalajii și definiții de funcții:

```
int global_positives = 0;  
typedef struct list {  
    struct list *next;  
    double val;  
} * list;  
void count_positives(list l)  
{  
    list p;  
    for (p = l; p; p = p -> next)  
        if (p -> val > 0.0)  
            ++global_positives;  
}
```

Să considerăm acum cazul în care firul A realizează
count_positives (<listă ce conține numai valori negative>);
iar firul B realizează
++global_positives;

Ce face funcția ?

a. Funcția numără elementele pozitive din listă.

14. Multe limbaje curente de programare, cum ar fi C și C++, sunt inadecvate pentru programarea multi-fir. Aceasta poate avea impact asupra compilatoarelor și a corectitudinii codului aşa cum se va ilustra în continuare această problemă. Să considerăm următoarele declarații și definiții de funcții:

```
int global_positives = 0;
typedef struct list {
    struct list *next;
    double val;
} * list;
void count_positives(list l)
{
    list p;
    for (p = l; p; p = p -> next)
        if (p-> val > 0.0)
            ++global_positives;
}
```

Să considerăm acum cazul în care firul A realizează
count_positives (<listă ce conține numai valori negative>);

iar firul B realizează

```
    ++global_positives;
```

Limbajul C este pregătit doar pentru execuția unui singur fir de execuție.
Creează utilizarea a două fire de execuție probleme sau probleme potențiale?

Ar trebui să funcționeze corect deoarece count_positive, în acest caz specific nu actualizează global_positives, și ca urmare cele două fire operează pe date globale distincte care nu necesită nici o blocare.

15. Unele compilatoare optimizate existente (inclusiv gcc, care poate fi relativ conservativ) vor “optimiza” `count_positives` la ceva similar cu

```
void count_positives(list l)
{
    list p;
    register int r;
    r = global_positives;
    for (p = l; p; p = p -> next)
        if (p -> val > 0.0) ++r;
    global_positives = r;
}
```

Ce problemă sau problemă potențială apare la această versiune de program compilat dacă firele A și B sunt executate concurențial?

Transformarea este consistentă cu specificațiile limbajului C, care se adresează execuției cu un singur fir de execuție. Prin regulile pthreads, un compilator care nu ia în considerație operarea multi-fir transformă un program perfect legitim într-unul cu semantică nedefinită.

16. Considerați următorul cod care utilizează API Pthreads din POSIX:

```

thread2.c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;
void *thread_function(void *arg) {
    int i,j;
    for ( i=0; i<20; i++ ) {
        j=myglobal;
        j=j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal=j;
    }
    return NULL;
}
int main(void) {
    pthread_t mythread;
    int i;
    if ( pthread_create( &mythread, NULL, thread_function,
        NULL) ) {
        printf("error creating thread.");
        abort();
    }
    for ( i=0; i<20; i++) {
        myglobal=myglobal+1;
        printf("o");
        fflush(stdout);
        sleep(1);
    }
    if ( pthread_join ( mythread, NULL ) ) {
        printf("error joining thread.");
        abort();
    }
    printf("\nmyglobal equals %d\n",myglobal);
    exit(0);
}

```

În main() s-a declarat prima dată o variabilă denumită mythread, care are tipul pthread_t. Aceasta este ID-ul esențial pentru un fir de execuție. Apoi, linia if creează un fir de execuție asociat cu mythread. Apelul pthread_create() returnează zero dacă apelul s-a realizat cu succes și o valoare diferită de zero dacă apelul a eșuat. Al treilea argument al funcției pthread_create() este numele funcției pe care noul fir o va executa la pornire. Când se revine din thread_function(), firul se termină. Între timp, programul principal însuși definește un fir, astfel încât sunt două fire în execuție. Funcția pthread_join validează firul main să aștepte până ce noul fir își completează execuția.

Ce realizează acest program?

Acest program crează un nou fir. Atât firul main cât și noul fir apoi incrementează variabila globală myglobal de 20 de ori.

17. Considerați următorul cod care utilizează API Pthreads din POSIX:

```
thread2.c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;
void *thread_function(void *arg) {
    int i,j;
    for ( i=0; i<20; i++ ) {
        j=myglobal;
        j=j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal=j;
    }
}
```

```
        return NULL;
}
int main(void) {
    pthread_t mythread;
    int i;
    if ( pthread_create( &mythread, NULL, thread_function,
        NULL) ) {
        printf("error creating thread.");
        abort();
    }
    for ( i=0; i<20; i++) {
        myglobal=myglobal+1;
        printf("o");
        fflush(stdout);
        sleep(1);
    }
    if ( pthread_join ( mythread, NULL ) ) {
        printf("error joining thread.");
        abort();
    }
    printf("\nmyglobal equals %d\n",myglobal);
    exit(0);
}
```

În main () s-a declarat prima dată o variabilă denumită mythread, care are tipul pthread_t. Acesta este ID-ul esențial pentru un fir de execuție. Apoi, linia if creează un fir de execuție asociat cu mythread. Apelul pthread_create() returnează zero dacă apelul s-a realizat cu succes și o valoare diferită de zero dacă apelul a eşuat. Al treilea argument al funcției pthread_create() este numele funcției pe care noul fir o va executa la pornire. Când se revine din thread_function(), firul se termină. Între timp, programul principal însuși definește un fir, astfel încât sunt două fire în execuție. Funcția pthread_join validează firul main să astepte până ce noul fir își completează execuția.

Ieșirea execuției programului este:

```
$ ./thread2
```

myglobal equals 21

Este această iesire ceea ce se astepta? Dacă nu, ce a mers gresit?

Total neașteptat! Imaginați-vă ce s-ar întâmpla dacă firul main incrementează `myglobal` doar după ce noul fir copie valoarea `myglobal`

în j. Când `thread_function()` scrie valoarea lui `j` înapoi în `myglobal`, acesta va suprascrie modificarea realizată de firul `main`.

18. În Solaris 9 și 10, există o mapare unu-la-unu între ULT și LWP.
În Solaris 8, un singur LWP suportă unul sau mai multe ULT. Care este beneficiul posibil permitând o mapare mulți-la-unul a ULT-urilor la LWP?

Unele programe au un paralelism logic care poate fi exploatat pentru a simplifica și structura codul dar nu au nevoie de paralelism hardware. Ca exemplu, o aplicație care necesită ferestre multiple, dintre care doar una este activă la un moment dat, poate fi avantajos implementată ca un set de ULT pe un singur LWP. Avantajul restricționării unor astfel de aplicații la ULT este eficiența. ULT – urile pot fi create, distruse, blocate, activate, etc. fără a invoca kernelul. Dacă fiecare ULT era cunoscut nucleului, nucleul trebuia să aloce structuri de date kernel pentru fiecare fir și să execute comutarea firelor. Comutarea firelor la nivelul kernelului este mult mai scumpă decât comutarea la nivelul utilizator.

19. În Solaris 9 și 10, există o mapare unu-la-unu între ULT și LWP.
În Solaris 8, un singur LWP suportă unul sau mai multe ULT. În Solaris 8, starea de execuție a firelor ULT este distinctă de aceea a LWP-ului său. Explicați de ce?

Execuția firelor la nivelul utilizator este gestionată de o librărie pentru fire de execuție unde LWP este gestionat de kernel.

20. În Solaris 9 și 10, există o mapare unu-la-unu între ULT și LWP.
În Solaris 8, un singur LWP suportă unul sau mai multe ULT. Figura 4.17 prezintă diagrama tranzițiilor de stare pentru un ULT și LWP său

asociat, în Solaris 8 și 9. Explicați modul de lucru al celor două diagrame și relațiile dintre ele.

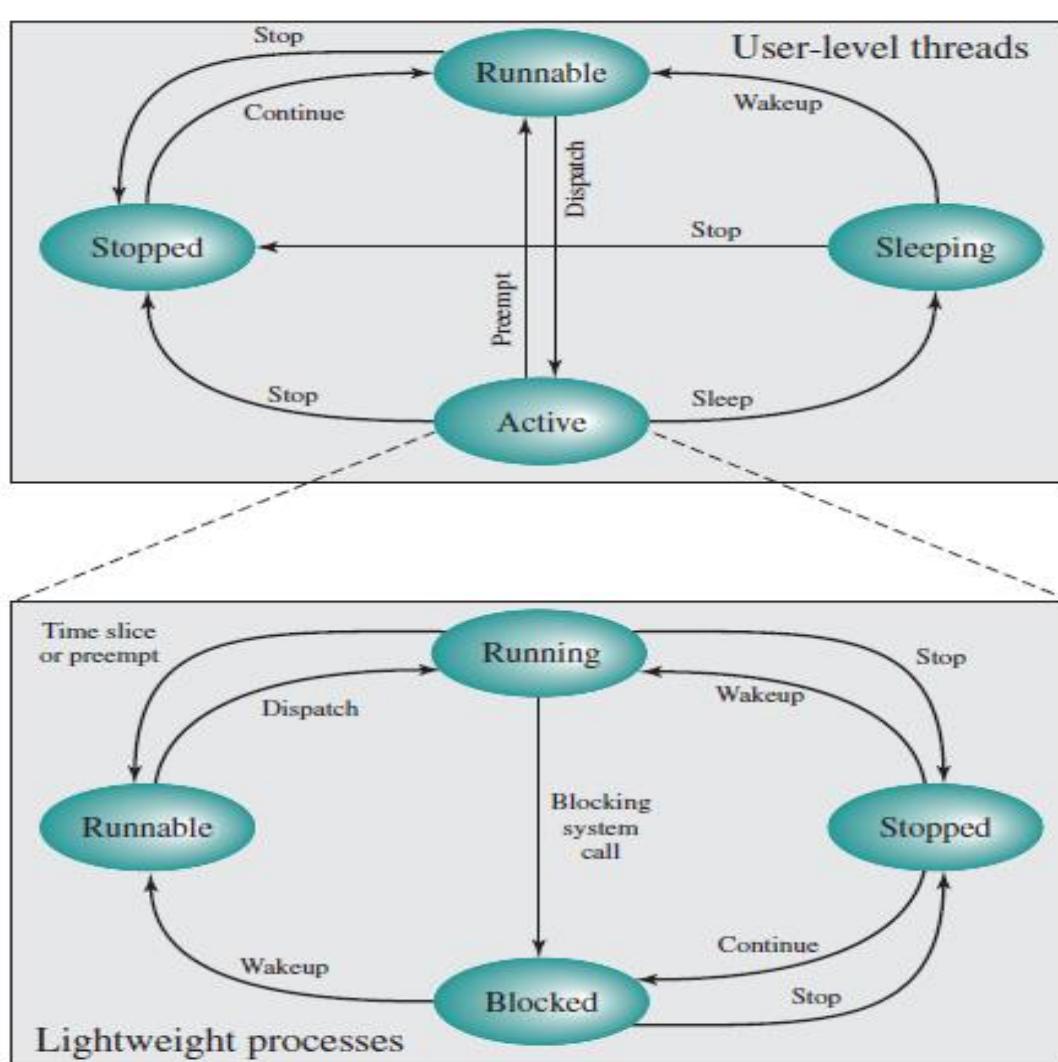


Figure 4.17 Solaris User-Level Thread and LWP States

Un fir liber poate fi în una din patru stări: în execuție, activ, adormit sau oprit. Aceste stări sunt gestionate de librăria pentru fire. Un ULT în starea activ este asignat curent la LWP și se execută în timp ce firul de la nivelul kernelului se execută. Putem diagrama de stare a LWP ca o descriere detaliată a stării active a ULT, deoarece un fir are o singură asignare LWP când este în starea activă. Diagrama de stare LWP este rezonabil de auto-explicativă. Un fir activ este în execuție numai când WP său este în starea în execuție. Când un fir activ execută un apel

sistem cu blocare, LWP intră în starea blocat. Totuși, ULT rămâne atașat la acel LWP și, cât timp librăria de fire este implicată, acel ULT rămâne activ.

4.12 Explicați motivul existenței stării Uninterruptible în Linux.

O stare Uninterruptible este un alt tip de stare blocată. Diferența între această stare și starea Interruptible constă în aceea că în starea Uninterruptible, un proces așteaptă direct pe o condiție hardware, și ca urmare nu va gestiona nici un semnal. Aceasta se utilizează în situații în care procesul trebuie să aștepte fără întrerupere sau când un eveniment este așteptat să apară cât de curând. De exemplu, această stare poate fi utilizată când un proces deschide un fișier de tip dispozitiv și device driver-ul corespunzător pornește să testeze dispozitivul hardware. Device driver-ul nu trebuie să întrerupă până când testarea dispozitivului nu este gata, sau dispozitivul hardware poate fi lăsat într-o stare impredictibilă.

Cap. 5 Concurență: excluderea mutuală și sincronizarea.

1. Enumerați patru caracteristici care fac relevant conceptul de concurență.

Comunicația între procese, partajarea și competiția pentru resurse, sincronizarea activităților proceselor multiple, și alocarea timpului procesor pentru procese.

2. Care sunt cele trei contexte în care apare concurență?

Aplicații multiple, aplicații structurate, structura sistemului de operare.

3. Care este cerința de bază pentru execuția proceselor concurente?

Abilitatea de a aplica excluderea mutuală.

4. Enumerați trei grade de conștientizare între procese și definițiile pe scurt pe fiecare.

Procese care nu sunt conștiente unele de altele: Acestea sunt procese independente care nu intenționează să lucreze împreună. Procese indirect conștiente unele de altele: acestea sunt procese care nu sunt conștiente unele de altele prin ID-ul lor, dar partajează accesul la obiecte comune, cum ar fi bufferele de I/O. Procese conștiente direct unele de altele: aceste procese sunt capabile să comunice unele cu altele utilizând ID-ul procesului, și au fost proiectate să lucreze împreună pentru anumite activități.

5. Care este diferența între procesele competitive și procesele cooperative?

Procesele competitive realizează accesul la aceeași resursă în același timp, cum ar fi discul, fișierele, sau imprimanta. Procesele cooperative fie partajează accesul la obiecte comune cum ar fi bufferele în memorie sau sunt capabile să comunice și să coopereze unele cu altele cu scopul de a obține performanțe pentru unele aplicații sau activități.

6. Enumerați trei probleme de control asociate cu procesele competitive și definițiile pe scurt pe fiecare.

Excluderea mutuală: procesele competitive pot accesa o resursă, numai unul la un moment dat de timp; mecanismele de excludere mutuală trebuie să asigure această politică unul la un moment dat. Deadlock (blocajul): dacă procesele competitive doresc accesul exclusiv la mai mult de o resursă atunci poate să apară blocajul dacă fiecare proces câștigă controlul unei resurse și așteaptă la altă resursă. Starvation (înfometarea): un proces dintr-un set de procese poate aștepta un timp nedefinit să acceseze o resursă deoarece alții membri ai setului monopolizează acea resursă.

7. Enumerați cerințele pentru excluderea mutuală.

1. Excluderea mutuală trebuie aplicată: numai un proces la un moment dat poate intra în secțiunea sa critică, dintre toate procesele care au aceeași secțiune critică pentru aceeași resursă sau obiect partajat. 2. Un proces care se oprește în secțiunea sa non-critică trebuie să facă acest lucru încât să nu interfere cu celelalte procese. 3. Trebuie să nu fie posibil ca pentru un proces care cere accesul la o resursă critică

întârzierea să fie infinită. 4. Când în secțiunea critică nu este nici un proces, oricare proces care cere intrarea în secțiunea critică trebuie să o facă fără întârzieri. 5. Nu se face nici o presupunere privind viteza relativă a proceselor sau numărul de procesoare. 6. Un proces rămâne în secțiunea sa critică pentru un timp finit.

8. Ce operații trebuie realizate pe un semafor?

1. Un semafor trebuie inițializat cu o valoare nenegativă. 2. Operația wait decrementează valoarea semaforului. Dacă valoarea devine negativă, atunci procesul care a executat wait se blochează. 3. Operația signal incrementează valoarea semaforului. Dacă valoarea nu este pozitivă, atunci un proces blocat de operația wait devine neblocat.

9. Care este diferența între semafoarele binare și cele generale?

Un semafor binar poate lua doar valorile 0 și 1. Un semafor general poate lua orice valoare întreagă.

10. Care este diferența între semafoarele puternice și cele slabe?

Un semafor puternic cere ca procesele care sunt blocate pe acel semafor să fie deblocate pe baza politicii first-in-first-aut. Un semafor slab nu dictează ordinea în care procesele blocate sunt deblocate.

11. Ce este un monitor?

Este o construcție a unui limbaj de programare care oferă tipuri de date abstracte și accesul cu escludere mutuală la un set de proceduri.

12. Care este diferența între mesajele cu blocare și fără blocare?

Există două aspecte, trimiterea și recepționarea de primitive. Atunci când se execută o primitivă send într-un proces, există două posibilități: fie procesul care emite se blochează până când se recepționează mesajul, sau nu se blochează. Similar, când un proces apelează o primitivă receive, există două posibilități: dacă mesajul a fost trimis anterior, mesajul este recepționat și execuția continuă. Dacă nu există nici un mesaj în așteptare, atunci fie (a) procesul este blocat până sosește un mesaj, sau (b) procesul continuă să se execute, abandonând așteptarea receptiei mesajului.

13. Ce condiții sunt în general asociate cu problema cititorilor scriitorilor?

1. Orice număr de cititori pot citi simultan fișierul. 2. Numai un scriitor la un moment dat poate scrie în fișier. 3. Dacă un scriitor scrie în fișier, nici un cititor nu poate citi fișierul.

14. La începutul secțiunii 5.1 , s-a afirmat că multiprogramarea și multiprocesarea prezintă unele probleme referitoare la concurență. Totuși, citați două diferențe cu privire la concurență între multiprogramare și multiprocesare.

a. Pe sistemele uniprocesor se pot evita întreruperile și ca urmare concurența prin dezactivarea întreruperilor. Pe sistemele multiprocesor apare altă problemă: ordinea accesului la memorie (procesoarele multiple accesează unitatea de memorie).

15. Se dă următorul program :

```
P1: {  
    shared int x;  
    x = 10;  
    while (1) {  
        x = x - 1;  
        x = x + 1;  
        if (x != 10)  
            printf("x is %d",x)  
    }  
}  
  
P2: {  
    shared int x;  
    x = 10;  
    while (1) {  
        x = x - 1;  
        x = x + 1;  
        if (x!=10)  
            printf("x is %d",x)  
    }  
}
```

Amintim că planificatorul (scheduler) în sistemele uniprocesor trebuie să implementeze aşa numita execuție pseudo-paralelă a acestor două procese concurente prin întreținerea instrucțiunilor lor, fără restricții privind ordinea de întreținere. Prezentați o secvență astfel încât este tipărită linia "x is 10".

Raspuns:

	M(x)
P1: x = x - 1;	9
P1: x = x + 1;	10
P2: x = x - 1;	9
P1: if(x != 10)	9
P2: x = x + 1;	10
P1: printf("x is %d", x);	10

16. Se dă următorul program :

```
P1: {  
    shared int x;  
    x = 10;  
    while (1) {  
        x = x - 1;  
        x = x + 1;  
        if (x != 10)  
            printf("x is %d",x)  
    }  
}  
  
P2: {  
    shared int x;  
    x = 10;  
    while (1) {  
        x = x - 1;  
        x = x + 1;  
        if (x!=10)  
            printf("x is %d",x)  
    }  
}
```

Amintim că planificatorul (scheduler) în sistemele uniprocesor trebuie să implementeze aşa numita execuție pseudo-paralelă a acestor două procese concurente prin întreținerea instrucțiunilor lor, fără restricții privind ordinea de întreținere. Prezentați o secvență astfel încât este tipărită linia "x is 8". Trebuie să vă amintiți că incrementarea/decrementarea la nivelul limbajului sursă nu se face atomic, codul în asembler poate să arate astfel:

```

LD R0,X    /* load R0 from memory location x */
INCR R0    /* increment R0 */
STO R0,X   /* store the incremented value back in X */

```

implementând o singură linie C cu instrucțiunea de incrementare ($x = x + 1$).

a.

Instruction	M(x)	P1-R0	P2-R0
P1: LD R0, x	10	10	--
P1: DECR R0	10	9	--
P1: STO R0, x	9	9	--
'			
P2: LD R0, x	9	9	9
P2: DECR R0	9	9	8
P2: STO R0, x	8	9	8
'			
P1: LD R0, x	8	8	8
P1: INCR R0	8	9	--
'			
P2: LD R0, x	8	9	8
P2: INCR R0	8	9	9
P2: STO R0, x	9	9	9
P2: if($x \neq 10$) printf("x is %d", x);			
P2: "x is 9" is printed.			
'			
P1: STO R0, x	9	9	9
P1: if($x \neq 10$) printf("x is %d", x);			
P1: "x is 9" is printed.			
'			
P1: LD R0, x	9	9	9
P1: DECR R0	9	8	--
P1: STO R0, x	8	8	--
'			
P2: LD R0, x	8	8	8
P2: DECR R0	8	8	7
P2: STO R0, x	7	8	7
'			
P1: LD R0, x	7	7	7
P1: INCR R0	8	8	7
P1: STO R0, x	8	8	7
P1: if($x \neq 10$) printf("x is %d", x);			
P1: "x is 8" is printed.			

17. Fie următorul program:

```
const int n = 50;
int tally;
void total()
{
    int count;
    for (count = 1; count<= n; count++) {
        tally++;
    }
}
void main()
{
    tally = 0;
    parbegin (total (), total ());
    write (tally);
}
```

Determinați marginea inferioară și marginea superioară pentru valoarea finală a variabilei partajate tally ca ieșire a acestui program concurrent. Presupuneți că procesele se pot executa cu orice viteză relativă și că o valoare poate fi implementată doar după ce a fost încărcată într-un registru de o instrucție în asembler separată.

$2 \leq \text{tally} \leq 100$.

18. Fie următorul program:

```
const int n = 50;
int tally;
void total()
{
    int count;
    for (count = 1; count<= n; count++) {
        tally++;
    }
}
void main()
{
    tally = 0;
    parbegin (total (), total ());
    write (tally);
}
```

Determinați marginea inferioară și marginea superioară pentru valoarea finală a variabilei partajate tally ca ieșire a acestui program concurrent. Presupuneți că procesele se pot executa cu orice viteză relativă și că o valoare poate fi implementată doar după ce a fost încărcată într-un registru de o instrucțiune în asembler separată. Presupuneți că se permite execuția în paralel a unui număr arbitrar de procese care respectă presupunerea anterioară. Ce efect va avea această modificare asupra gamei valorilor finale ale variabilei tally?

$2 \leq \text{tally} \leq (N \times 50)$.

19. Este busy waiting (așteptarea ocupată) întotdeauna mai puțin eficientă (referitor la utilizarea timpului procesor) decât blocking wait (așteptarea cu blocare)?
Explicați.

Da, pe medie, deoarece busy-waiting consumă inutil cicli instrucțiune. Totuși, într-un caz particular, dacă un proces apare într-un punct în program unde trebuie să aștepte ca să fie satisfăcută o condiție, și dacă acea condiție este deja satisfăcută, atunci busy-wait va ieși imediat din așteptare, în timp ce blocking wait consumă resurse ale OS prin comutarea în afară și înapoi în proces.

20. Fie următorul program:

```
boolean blocked [2];
int turn;
void P (int id)
{
    while (true) {
        blocked[id] = true;
        while (turn != id) {
            while (blocked[1-id])
                /* do nothing */;
            turn = id;
        }
        /* critical section */
        blocked[id] = false;
        /* remainder */
    }
}
void main()
{
    blocked[0] = false;
    blocked[1] = false;
    turn = 0;
    parbegin (P(0), P(1));
}
```

Această soluție software la problema excluderii mutuale este propusă în [HYMA66]. Găsiți un contraexemplu care demonstrează că această soluție este incorectă. Este interesant că în suși comitetul de la Communications of the ACM a fost păcălit de această soluție.

Considerați cazul în care turn este egal cu 0 și P(1) setează blocked [1] pe true și apoi găsim blocked[0] setat pe false. P(0) va seta apoi blocked[0] pe true, găsind turn = 0, și

intră în secțiunea sa critică. P(1) va asigna apoi 1 pentru turn și va intra de asemenea în secțiunea sa critică.

21.O abordare software la excluderea mutuală este Lamport's bakery algorithm [LAMP74], astfel denumit deoarece se bazează pe practica din brutări sau alte magazine în care fiecare client recepționează la sosire un bilet numerotat, în schimbul căruia va fi servit. Algoritmul este după cum urmează:

```
boolean choosing[n];
int number[n];
while (true) {
    choosing[i] = true;
    number[i] = 1 + getmax(number[], n);
    choosing[i] = false;
    for (int j = 0; j < n; j++) {
        while (choosing[j]) { };
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
    }
    /* critical section */;
    number [i] = 0;
    /* remainder */;
}
```

Aria aleasă și numărul sunt inițializate cu false și respectiv 0. Al i-lea element al fiecărei arii poate fi citit sau scris de procesul i dar poate fi doar citit de alt proces. Notația (a, b) < (c,d) este definită ca: (a < c) or (a = c and b < d). Descrieți algoritmul în cuvinte.

Atunci când un proces dorește să intre în secțiunea sa critică, î se asignează nu număr. Numărul asignat este calculat prin adunarea cu 1 a numărului cel mai mare deținut de un proces care dorește de asemenea să intre în secțiunea sa critică pentru aceeași resursă, și există deja un proces în secțiunea critică Procesul cu numărul cel mai mic are precedența cea mai mare pentru a intra în secțiunea sa critică. În cazul în care există mai multe procese cu același număr, procesul cu cel mai mic nume numeric va intra în secțiunea sa critică. Când un proces părăsește secțiunea sa critică, acesta resetează valoarea numărului la zero.

22.O abordare software la excluderea mutuală este Lamport's bakery algorithm [LAMP74], astfel denumit deoarece se bazează pe practica din brutări sau alte magazine în care fiecare client recepționează la sosire un bilet numerotat, în schimbul căruia va fi servit. Algoritmul este după cum urmează:

```

boolean choosing[n];
int number[n];
while (true) {
    choosing[i] = true;
    number[i] = 1 + getmax(number[], n);
    choosing[i] = false;
    for (int j = 0; j < n; j++){
        while (choosing[j]) { };
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
    }
    /* critical section */;
    number [i] = 0;
    /* remainder */;
}

```

Aria aleasă și numărul sunt inițializate cu false și respectiv 0. Al i-lea element al fiecărei arii poate fi citit sau scris de procesul i dar poate fi doar citit de alt proces. Notația $(a, b) < (c, d)$ este definită ca: $(a < c)$ or $(a = c \text{ and } b < d)$. Arătați că acest algoritm evită blocajul (deadlock).

Dacă fiecare proces este asignat cu un număr unic de proces, atunci există o ordine unică și strictă al proceselor în orice moment. Ca urmare, blocajul nu poate să apară.

23.O abordare software la excluderea mutuală este Lamport's bakery algorithm [LAMP74], astfel denumit deoarece se bazează pe practica din brutării sau alte magazine în care fiecare client receptionează la sosire un bilet numerotat, în schimbul căruia va fi servit. Algoritmul este după cum urmează:

```

boolean choosing[n];
int number[n];
while (true) {
    choosing[i] = true;
    number[i] = 1 + getmax(number[], n);
    choosing[i] = false;
    for (int j = 0; j < n; j++){
        while (choosing[j]) { };
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
    }
    /* critical section */;
    number [i] = 0;
    /* remainder */;
}

```

Aria aleasă și numărul sunt inițializate cu false și respectiv 0. Al i-lea element al fiecărei arii poate fi citit sau scris de procesul i dar poate fi doar citit de alt proces. Notația $(a, b) < (c, d)$ este definită ca: $(a < c)$ or $(a = c \text{ and } b < d)$. Arătați că asigură excluderea mutuală.

Pentru a demonstra excluderea mutuală, prima dată trebuie să demonstrăm următoarea lemură: dacă P_i este în secțiunea sa critică, și P_k are calculat numărul său $\text{number}[k]$ și dorește să intre în secțiunea sa critică, atunci trebuie îndeplinită următoarea relație: $(\text{number}[i], i) < (\text{number}[k], k)$. Odată demonstrată lema este ușor de a arăta că este îndeplinită excluderea mutuală. Presupunem că P_i este în secțiunea sa critică și P_k așteaptă să intre în secțiunea sa critică. P_k nu va fi capabil să intre în secțiunea sa critică, deoarece va găsi $\text{number}[i] \neq 0$ și $(\text{number}[i], i) < (\text{number}[k], k)$.

24. Să considerăm acum algoritmul brutăriei (bakery algorithm) fără alegerea variabilei. Vom avea:

```
1 int number[n];
2 while (true) {
3     number[i] = 1 + getmax(number[], n);
4     for (int j = 0; j < n; j++) {
5         while ((number[j] != 0) && (number[j], j) < (number[i], i)) { };
6     }
7     /* critical section */;
8     number [i] = 0;
9     /* remainder */;
10 }
```

Violează această versiune excluderea mutuală? Explicați de ce sau de ce nu!

Presupunem că avem două procese care tocmai au început; le vom denumi p_0 și p_1 . Ambele ating linia 3 în același timp. Acum, presupunem că ambele citesc $\text{number}[0]$ și $\text{number}[1]$, dar p_0 se blochează înainte de atribuire. Apoi p_1 execută bucla while la linia 5 și intră în secțiunea critică. Întrând în secțiunea critică se blochează; p_0 se deblochează și atribuie 1 lui $\text{number}[0]$ la linia 3. Execută bucla while la linia 5. Când intră în buclă, pentru $j = 1$, prima condiție este adevărată. Mai departe, a doua condiție de la linia 5 este falsă, astfel p_0 intră în secțiunea critică. Acum p_0 și p_1 sunt ambele în secțiunea critică violând excluderea mutuală.

25. Fie următorul program care furnizează o abordare software la excluderea mutuală:

integer array control [1 :N]; integer k
where **1 ≤ k ≤ N**, and each element of “control” is either 0, 1,
or 2. All elements of “control” are initially zero; the initial value
of k is immaterial.

Aceste este denumit ca Eisenberg-McGuire algorithm. Explicați modul de operare și caracteristicile cheie.

```
begin integer j;
L0: control [i] := 1;
L1: for j:=k step 1 until N, 1 step 1 until k do
    begin
        if i = i then goto L2;
        if control [j] ≠ 0 then goto L1
    end;
L2: control [i] := 2;
    for j := 1 step 1 until N do
        if j ≠ i and control [j] = 2 then goto L0;
L3: if control [k] ≠ 0 and k ≠ i then goto L0;
L4: k := i;
    critical section;
L5: for j := k step 1 until N, 1 step 1 until k do
    if j ≠ k and control [j] ≠ 0 then
        begin
            k := j;
            goto L6
        end;
L6: control [i] := 0;
L7: remainder of cycle;
    goto L0;
end
```

Acesta este un program care oferă excluderea mutuală pentru accesul la o resursă critică între N procese, care pot numai să utilizeze resursa o dată la un anumit moment de timp. Trăsătura unică a acestui algoritm este aceea că un proces nu trebuie să aștepte pentru acces mai mult de N-1 turns (ronduri). Valoarea *control[i]* pentru process *i* este interpretată după cum urmează: 0 = în afara secțiunii critice și fără a căuta o intrare; 1 = dorește să acceseze secțiunea critică; 2 = a obținut precedența pentru a intra în secțiunea critică. Valoarea *k* reflectă care este turns-ul pentru a intra în secțiunea critică. Algoritmul de intrare: Procesul *i* își exprimă intenția de a intra în secțiunea sa critică setând *control[i]* = 1. Dacă nici un proces între *k* și *i* (în ordine circulară) nu a exprimat o intenție similară, atunci procesul *i* își obține precedent pentru a intra în secțiunea sa critică setând *control[i]* = 2. Dacă *i* este singurul proces care și-a exprimat intenția, el intră în secțiunea critică setând *k*=1; dacă este o coliziune, îl repornește algoritmul de intrare. Algoritmul de ieșire: procesul *i* examinează aria *control* într-o manieră circulară începând cu intrarea *i*+1. Dacă procesul *i* găsește un proces cu o intrare *control* diferită de zero, atunci *k* este setat să identifice acest proces. Mai trebuie făcute următoarele observații: prima dată trebuie observat că nu pot fi două procese simultan între liniile L3 și L6. În al doilea rând, trebuie observat că sistemul nu poate fi blocat; În final, trebuie observat că nici un singur proces nu poate fi blocat.

26. Considerați o primă instanță a linie bolt = 0 din figura 5.2b . Atingeți același rezultat utilizând instrucțiunea de interschimbare.

```
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(a) Compare and swap instruction

```
/* program mutual_exclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (&keyi, &bolt);
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Raspuns:

exchange (keyi, blot).

27. Considerați o primă instanță a linie bolt = 0 din figura 5.2b . Atingeți același rezultat utilizând instrucțiunea de interschimbare. Care metodă este preferabilă?

```
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(a) Compare and swap instruction

```
/* program mutual_exclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (&keyi, &bolt);
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Este preferabilă linia bolt = 0. O linie atomică precum exchange va utiliza mai multe resurse.

28. Atunci când se utilizează o instrucțiune mașină (procesor) specială pentru a asigura excluderea mutuală în stilul din figura 5.2, nu există nici un control asupra timpului cât trebuie să aștepte un proces înainte de a-i se permite accesul în secțiunea sa critică. Propuneți un algoritm care utilizează instrucțiunea compare&swap, dar care garantează că orice proces care așteaptă să intre în secțiunea sa critică o va face în $n-1$ turns, unde n este numărul de procese care ar putea cere accesul la secțiunea critică iar un “turn” este un eveniment care constă în părăsirea de către un proces a secțiunii critice și primirea accesului de către alt proces.

```
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (ccmpare_and_swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(a) Compare and swap instruction

```
/* program mutual_exclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (&keyi, &bolt);
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Raspuns:

```
var j: 0..n-1;
    key: boolean;
while (true) {
    waiting[i] = true;
    key := true;
    while (waiting[i] && key)
        key = (compare_and_swap(lock, 0, 1) == 0);
    waiting[i] = false;
    < critical section >
    j = i + 1 mod n;
    while (j != i && !waiting[j]) j = j + 1 mod n;
    if (j == i) lock := false
    else waiting = false;
    < remainder section >
}
```

Algoritmul utilizează structurile comune de date

var waiting: **array** [0..n - 1] **of** boolean

lock: boolean

Acstea structuri de date sunt inițializate cu false. Atunci când un proces părăsește secțiunea sa critică, acesta scanează aria de așteptare într-o ordine ciclică ($i+1, i+2, \dots, n-1, 0, \dots, i-1$). Aceasta desemnează primul proces în ordine care este în secțiunea de intrare ($\text{waiting}[j] = \text{true}$) ca următorul care va intra în secțiunea critică. Oricare proces va aștepta ca să intre în secțiunea sa critică $n-i$ turns (ronduri).

29. Fie următoarea definiție a semafoarelor:

```
void semWait (s)
{
    if (s.count > 0) {
        s.count--;
    }
    else {
        place this process in s.queue;
        block;
    }
}
void semSignal (s)
{
    if (there is at least one process blocked on
        semaphore s) {
        remove a process P from s.queue;
        place process P on ready list;
    }
    else
        s.count++;
}
```

Comparați acest set de definiții cu acela din figura 5.3. Există o diferență: Cu definiția precedentă, un semafor nu poate lua niciodată o valoare negativă. Există vreo diferență ca efect al celor două seturi de definiții când acestea se utilizează într-un program? Dacă este așa, puteți substitui un set cu celălalt fără a altera înțelesul programului?

```

struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

```

Figure 5.3 A Definition of Semaphore Primitives

Cele două sunt echivalente. În definiția din figura 5.3, atunci când valoarea semaforului este negativă, valoarea sa arată câte procese sunt în aşteptare. Cu definiția din această problemă, informația nu este disponibilă pentru a fi citită imediat. Totuși, cele două versiuni ale funcției sunt identice.

30. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Ca urmare putem crea următoarea soluție:

```

1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /* counters, and */
3  boolean must_wait = false;               /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                 /* the mutual exclusion first */
9      semWait(block);                  /* Wait for all current users to depart */
10     SemWait(mutex);                  /* Reenter the mutual exclusion */
11     --waiting;                        /* and update the waiting count */
12 }
13 ++active;                            /* Update active count, and remember */
14 must_wait = active == 3;              /* if the count reached 3 */
15 semSignal(mutex);                   /* Leave the mutual exclusion */
16
17 /* critical section */
18
19 semWait(mutex);                     /* Enter mutual exclusion */
20 --active;                           /* and update the active count */
21 if(active == 0) {                  /* Last one to leave? */
22     int n;
23     if (waiting < 3) n = waiting;
24     else n = 3;                      /* If so, unblock up to 3 */
25     while( n > 0 ) {                /* waiting processes */
26         semSignal(block);
27         --n;
28     }
29     must_wait = false;              /* All active processes have left */
30 }
31 semSignal(mutex);                  /* Leave the mutual exclusion */

```

Soluția pare că realizează totul în regulă : Toate accesele la variabilele partajate sunt protejate prin excludere mutuală, procesele nu se autoblochează în timpul excluderii mutuale, noile procese nu pot folosi resursa dacă există (sau au existat) trei utilizatori activi, și ultimul proces care eliberează resursa o deblochează permitând activarea a trei procese care erau în așteptare. Programul totuși este incorrect. Explicați de ce.

Există două probleme. Prima, deoarece procesele neblocațe pot să reintre în excluderea mutuală (linia 10), există o șansă ca noile procese care ajung (la linia 5) să se bată pentru secțiunea critică. A doua, există un timp de întârziere între momentul în care procesele care așteaptă sunt deblocate și momentul în care își reiau execuția și actualizează contorul. Procesele care așteaptă trebuie contabilizate cât de repede după momentul în care au fost deblocate (deoarece ele își pot relua execuția în orice moment), dar poate fi ceva timp înainte ca procesele să își reia execuția și să actualizeze contorul pentru a reflecta acest lucru. Pentru clarificare, considerați cazul în care trei procese sunt blocați la linia 9. Ultimul proces activ le va debloca (liniile 25-28) la ieșire. Nu există însă nici o cale de a prezice când aceste procese își vor relua

execuția și vor actualiza contorul pentru a reflecta faptul că au devenit active. Dacă un proces nou atinge linia 6 înainte ca un proces deblocat să își reia execuția, noul proces trebuie blocat. Dar variabilele de stare nu au fost încă actualizate astfel încât noul proces va câștiga accesul la resursă. În momentul în care un proces deblocat își reia execuția, acesta va câștiga și el accesul la resursă. Soluția cade deoarece a permis ca patru procese să acceseze împreună resursa.

31. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Ca urmare putem crea următoarea soluție:

```
1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /* counters, and */
3  boolean must_wait = false;                /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                  /* the mutual exclusion first */
9      semWait(block);                   /* Wait for all current users to depart */
10     SemWait(mutex);                   /* Reenter the mutual exclusion */
11     --waiting;                        /* and update the waiting count */
12 }
13 ++active;                            /* Update active count, and remember */
14 must_wait = active == 3;              /* if the count reached 3 */
15 semSignal(mutex);                   /* Leave the mutual exclusion */
16
17 /* critical section */
18
19 semWait(mutex);                     /* Enter mutual exclusion */
20 --active;                           /* and update the active count */
21 if(active == 0) {                  /* Last one to leave? */
22     int n;
23     if (waiting < 3) n = waiting;
24     else n = 3;                      /* If so, unblock up to 3 */
25     while( n > 0 ) {               /* waiting processes */
26         semSignal(block);
27         --n;
28     }
29     must_wait = false;              /* All active processes have left */
30 }
31 semSignal(mutex);                  /* Leave the mutual exclusion */
```

Soluția pare că realizează totul în regulă : Toate accesele la variabilele partajate sunt protejate prin excludere mutuală, procesele nu se autoblochează în timpul excluderii mutuale, noile procese nu pot folosi resursa dacă există (sau au existat) trei utilizatori activi, și ultimul proces care eliberează resursa o deblochează permitând activarea a trei procese care erau în așteptare. Presupuneți că schimbați linia 6 într-un while. Rezolvă această schimbare problemele din program? Ce alte dificultăți rămân?

Aceasta forțează procesele deblocate să testeze din nou dacă pot începe utilizarea resursei. Această soluție însă este mult mai expusă la infometare deoarece încurajează noi și să se plaseze în afara ordinii ("cut in line") în fața celor care deja așteaptă.

32. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```
1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /* counters, and */
3  boolean must_wait = false;               /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                  /* the mutual exclusion first */
9      semWait(block);                   /* Wait for all current users to depart */
10 } else {                                /* Update active count, and */
11     ++active;                           /* remember if the count reached 3 */
12     must_wait = active == 3;            /* Leave mutual exclusion */
13     semSignal(mutex);
14 }
15
16 /* critical section */
17
18 semWait(mutex);                         /* Enter mutual exclusion */
19 --active;                            /* and update the active count */
20 if(active == 0) {                      /* Last one to leave? */
21     int n;
22     if (waiting < 3) n = waiting;
23     else n = 3;                        /* If so, see how many processes to unblock */
24     waiting -= n;                     /* Deduct this number from waiting count */
25     active = n;                      /* and set active to this number */
26     while( n > 0 ) {                 /* Now unblock the processes */
27         semSignal(block);            /* one by one */
28         --n;
29     }
30     must_wait = active == 3;          /* Remember if the count is 3 */
31 }
32 semSignal(mutex);                     /* Leave the mutual exclusion */
```

Explicați cum lucrează acest program și de ce este corect.

Această abordare elimină întârzierea în timp. Dacă procesul care ieșe actualizează numărătoarele pentru procesele de așteptare și active pentru a debloca procesele

active, atunci numărătoarele reflectă cu acuratețe noua stare a sistemului înainte ca un proces nou să ajungă în excluderea mutuală. Deoarece actualizarea este deja făcută, procesele deblocate nu trebuie să reentre toate în secțiunea critică. Implementarea acestui şablon este simplă. Identificați totă munca care a fost făcută de un proces deblocat și lăsați în loc să o facă procesul care deblochează.

33. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```

1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /* counters, and */
3  boolean must_wait = false;               /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                 /* the mutual exclusion first */
9      semWait(block);                  /* Wait for all current users to depart */
10 } else {                                /* Update active count, and */
11     ++active;                           /* remember if the count reached 3 */
12     must_wait = active == 3;            /* Leave mutual exclusion */
13     semSignal(mutex);
14 }
15
16 /* critical section */
17
18 semWait(mutex);                         /* Enter mutual exclusion */
19 --active;                            /* and update the active count */
20 if(active == 0) {                      /* Last one to leave? */
21     int n;
22     if (waiting < 3) n = waiting;
23     else n = 3;                        /* If so, see how many processes to unblock */
24     waiting -= n;                     /* Deduct this number from waiting count */
25     active = n;                      /* and set active to this number */
26     while( n > 0 ) {                /* Now unblock the processes */
27         semSignal(block);            /* one by one */
28         --n;
29     }
30     must_wait = active == 3;          /* Remember if the count is 3 */
31 }
32 semSignal(mutex);                     /* Leave the mutual exclusion */

```

Această soluție nu previne complet ca noile procese care sosesc să nu intre în afara rândului (cutting in line) dare face lucrul acest mai puțin probabil. Dați un exemplu de comportament în afara rândului (cutting in line).

Presupunetiți trei procese care sosesc când resursa este ocupată, dar unul dintre ele și-a pierdut cuanta chiar înainte de a se autobloca la linia 9 (care este improbabil, dar cu siguranță posibil). Când ieșe ultimul proces activ, acesta va face trei operații semSignal

și setează must_wait pe true. Dacă sosește un proces nou înainte ca unul mai vechi să-și reia execuția, procesul nou va decide să se autoblocheze. Totuși, este posibil ca noul proces să execute semWait din linia 9 fără a se bloca, și atunci procesul care a pierdut cuanta anterior se va executa, acesta se blochează în locul noului proces. Aceasta nu este o eroare – problema nu dictează care proces va avea acces la resurse, ci numai câtor proceze li se permite accesul la resursă. Într-adevăr, deoarece ordinea de deblocare a semafoarelor este implementată independent, singurul mod portabil de a asigura că procezele procedează într-o anumită ordine este de a bloca fiecare pe propriul semafor.

34. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei proceze care utilizează resursa , noile proceze pot porni utilizarea sa imediat (2) Odată ce există trei proceze care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de proceze care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```

1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /*      /* counters, and */
3  boolean must_wait = false;                /*      /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                  /* the mutual exclusion first */
9      semWait(block);                   /* Wait for all current users to depart */
10 } else {                                /* Update active count, and */
11     ++active;                           /* remember if the count reached 3 */
12     must_wait = active == 3;            /* Leave mutual exclusion */
13     semSignal(mutex);                  */
14 }
15
16 /* critical section */
17
18 semWait(mutex);                         /* Enter mutual exclusion */
19 --active;                            /* and update the active count */
20 if(active == 0) {                      /* Last one to leave? */
21     int n;
22     if (waiting < 3) n = waiting;
23     else n = 3;                        /* If so, see how many processes to unblock */
24     waiting -= n;                     /* Deduct this number from waiting count */
25     active = n;                      /* and set active to this number */
26     while( n > 0 ) {                /* Now unblock the processes */
27         semSignal(block);           /* one by one */
28         --n;
29     }
30     must_wait = active == 3;          /* Remember if the count is 3 */
31 }
32 semSignal(mutex);                      /* Leave the mutual exclusion */

```

Acest program este un exemplu de proiectare generală a unui şablon care reprezintă un mod uniform de a implementa o soluție la multe probleme concurente care utilizează semafore. Este denumit ca I'll Do It For You pattern. Descrieți şablonul.

Procesul care ieșe actualizează starea sistemului în folosul procesului pe care îl deblochează.

35. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```
1 semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2 int active = 0, waiting = 0;             /* counters, and */
3 boolean must_wait = false;               /* state information */
4
5 semWait(mutex);                         /* Enter the mutual exclusion */
6 if(must_wait) {                         /* If there are (or were) 3, then */
7     ++waiting;                          /* we must wait, but we must leave */
8     semSignal(mutex);                  /* the mutual exclusion first */
9     semWait(block);                   /* Wait for all current users to depart */
10    --waiting;                         /* We've got the mutual exclusion; update count */
11 }
12 ++active;                            /* Update active count, and remember */
13 must_wait = active == 3;              /* if the count reached 3 */
14 if(waiting > 0 && !must_wait)      /* If there are others waiting */
15     semSignal(block);                /* and we don't yet have 3 active, */
16                                         /* unblock a waiting process */
17 else semSignal(mutex);               /* otherwise open the mutual exclusion */
18
19 /* critical section */
20
21 semWait(mutex);                     /* Enter mutual exclusion */
22 --active;                           /* and update the active count */
23 if(active == 0)                     /* If last one to leave? */
24     must_wait = false;              /* set up to let new processes enter */
25 if(waiting == 0 && !must_wait)    /* If there are others waiting */
26     semSignal(block);              /* and we don't have 3 active, */
27                                         /* unblock a waiting process */
28 else semSignal(mutex);            /* otherwise open the mutual exclusion */
```

Explicați cum lucrează acest program și de ce este corect.

După ce se deblochează o resursă la care se așteaptă, se părăsește secțiunea critică (sau se autoblochează) fără a deschide excluderea mutuală. Procesul deblocat nu reintră în secțiunea sa critică. Procesul poate ca urmare să actualizeze în siguranță starea sistemului cu starea sa. Când termină, redeschide excluderea mutuală. Procesele noi care sosesc nu se pot plasa în afara ordinii deoarece ele nu pot intra în excluderea mutuală până ce procesul deblocat nu se termină. Deoarece procesul deblocat are grija de propria actualizare, coeziunea acestei soluții este mai bună. Totuși, odată ce s-a deblocat un proces, trebuie ca imediat să se opreasă accesul la variabilele protejate de excluderea mutuală. Abordarea cea mai sigură este părăsirea sau autoblocarea (după linia 26, procesul părăsește secvența fără a deschide mutex-ul).

36. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```

1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /*      /* counters, and */
3  boolean must_wait = false;               /*      /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                 /* the mutual exclusion first */
9      semWait(block);                  /* Wait for all current users to depart */
10     --waiting;                        /* We've got the mutual exclusion; update count */
11 }
12 ++active;                            /* Update active count, and remember */
13 must_wait = active == 3;              /* if the count reached 3 */
14 if(waiting > 0 && !must_wait)       /* If there are others waiting */
15     semSignal(block);                /* and we don't yet have 3 active, */
16                                         /* unblock a waiting process */
17 else semSignal(mutex);               /* otherwise open the mutual exclusion */
18
19 /* critical section */
20
21 semWait(mutex);                     /* Enter mutual exclusion */
22 --active;                           /* and update the active count */
23 if(active == 0) {                  /* If last one to leave? */
24     must_wait = false;              /* set up to let new processes enter */
25 if(waiting == 0 && !must_wait)   /* If there are others waiting */
26     semSignal(block);            /* and we don't have 3 active, */
27                                         /* unblock a waiting process */
28 else semSignal(mutex);           /* otherwise open the mutual exclusion */

```

Diferă această soluție de precedenta referitor la numărul de procese care pot fi deblocate la un moment dat de timp? Explicați.

Numai un singur proces care așteaptă poate fi deblocat chiar dacă așteaptă mai multe – deblocarea mai multor procese va viola excluderea mutuală a variabilelor de stare. Această problemă este rezolvată prin testarea noilor procese neblocate când mai multe procese trebuie deblocate (linia 14). Dacă este așa, se pasează ștafeta unuia dintre ele (linia 15); dacă nu, se deschide excluderea mutuală pentru noii soșiți (linia 17).

37. Fie o resursă partajabilă cu următoarele caracteristici : (1) Cât timp există mai puțin de trei procese care utilizează resursa , noile procese pot porni utilizarea sa imediat (2) Odată ce există trei procese care utilizează resursa, toate trei trebuie să elibereze resursa înainte ca un nou proces să o poată utiliza. Realizăm că trebuie folosite contoare care să urmărească numărul de procese care așteaptă și care sunt active, și că aceste contoare sunt la rândul lor resurse partajate care trebuie protejate cu excluderea mutuală. Fie următoarea soluție:

```
1  semaphore mutex = 1, block = 0;           /* share variables: semaphores, */
2  int active = 0, waiting = 0;             /*      /* counters, and */
3  boolean must_wait = false;               /*      /* state information */
4
5  semWait(mutex);                         /* Enter the mutual exclusion */
6  if(must_wait) {                         /* If there are (or were) 3, then */
7      ++waiting;                          /* we must wait, but we must leave */
8      semSignal(mutex);                 /* the mutual exclusion first */
9      semWait(block);                  /* Wait for all current users to depart */
10     --waiting;                        /* We've got the mutual exclusion; update count */
11 }
12 ++active;                            /* Update active count, and remember */
13 must_wait = active == 3;              /*      /* if the count reached 3 */
14 if(waiting > 0 && !must_wait)       /* If there are others waiting */
15     semSignal(block);                /* and we don't yet have 3 active, */
16
17 else semSignal(mutex);              /* otherwise open the mutual exclusion */
18
19 /* critical section */
20
21 semWait(mutex);                     /* Enter mutual exclusion */
22 --active;                           /* and update the active count */
23 if(active == 0) {                  /* If last one to leave? */
24     must_wait = false;             /* set up to let new processes enter */
25 if(waiting == 0 && !must_wait)   /* If there are others waiting */
26     semSignal(block);            /* and we don't have 3 active, */
27
28 else semSignal(mutex);           /* otherwise open the mutual exclusion */
```

Acest program este un exemplu de proiectare generală a unui şablon care reprezintă un mod uniform de a implementa o soluţie la multe probleme concurente care utilizează semafoare. Este denumit ca Pass The Baton pattern. Descrieţi şablonul.

Acest şablon sincronizează procesele ca și alergătorii din cursele de ștafetă. Imediat cum un alergător își termină turul său, el pasează ștafeta următorului alergător. Având ștafeta, este ca și cum ai permisiunea să fi pe pistă. În lumea sincronizării, intrarea în excludere mutuală este analogă cu a avea ștafeta – numai o persoană poate să o aibă.

38. Poate fi posibil să implementăm semafoare generale utilizând semafoare binare. Putem utiliza operațiile semWaitB și semSignalB și două semafoare binare, delay și mutex . Fie următorul program:

```
void semWait (semaphore s)
{
    semWaitB (mutex) ;
    s-- ;
    if (s < 0) {
        semSignalB (mutex) ;
        semWaitB (delay) ;
    }
    else SemsignalB (mutex) ;
}
void semSignal (semaphore s) ;
{
    semWaitB (mutex) ;
    s++ ;
    if (s <= 0)
        semSignalB (delay) ;
    semSignalB (mutex) ;
}
```

Initial s este setat la valoare dorită pentru semafor. Fiecare operație semWait decrementează s, și fiecare operație semSignal incrementează s . Semaforul binar mutex, care este inițializat cu 1, asigură că există excludere mutuală pentru actualizarea lui s . Semaforul binar delay, care este inițializat cu 0, este utilizat pentru a bloca procesele. Există o imperfecțiune în programul precedent. Demonstrați imperfecțiunea și propuneți o modificare care să o eliminate. Sugestie: Presupuneți două procese fiecare apelând semWait(s) atunci când s este initial 0, și după ce primul tocmai a realizat semSignalB(mutex) dar nu a realizat semWaitB(delay) , al doilea apelează semWait(s) în același punct. Tot ceea ce trebuie făcut este de a muta o singură linie în program.

Presupuneți că două procese apelează ambele semWait(s) când s este initial 0, și după ce primul tocmai face semSignal(mutex) dar nu face semWaitB(delay), al doilea apel la

`semWait(s)` conduce în același punct. Deoarece `s = -2` și `mutex` este deblocat, dacă alte două procese execută apoi succesiv apelul la `semSignal(s)` în acel moment, acestea vor face fiecare `semSignalB(delay)`, dar efectul celui de-al doilea `semSignalB` nu este definit. Soluția este de a muta linia `else`, care apare chiar înainte de linia `end` în `semWait` înainte de linia `end` din `semSignal`. Astfel ultimul `semSignalB(mutex)` în `semWait` devine necondițional și `semSignalB (mutex)` în `semSignal` devine condițional.

39. În 1978, Dijkstra definește o conjunctură pentru care nu există soluție privind problema excluderii mutuală pentru a evita înfometarea (starvation), aplicabilă unui număr n necunoscut dar finit de procese, utilizând un număr finit de semafoare slabe (weak semaphores). În 1979, J. M. Morris rezolvă această problemă prin publicarea unui algoritm care utilizează trei semafoare slabe (weak semaphores). Comportarea algoritmului poate fi descrisă după cum urmează: Dacă unul sau mai multe procese așteaptă în operația `semWait(S)` și alt proces execută `semSignal(S)`, valoarea semaforului `S` nu este modificată și unul din procesele care așteaptă este deblocat independent de `semWait(S)`. Pe lângă aceste trei semafoare algoritmul utilizează ca și contoare două variabile întregi nenegative pentru numărul de procese aflate în anumite secțiuni ale algoritmului. Astfel, semafoarele `A` și `B` sunt inițializate cu 1, în timp ce semaforul `M` și numărătoarele `NA` și `NM` sunt inițializate cu 0. Semaforul `B` pentru excludere mutuală protejează accesul la variabila partajată `NA`. Un proces care încearcă să intre în secțiunea sa critică trebuie să treacă două bariere reprezentate de semafoarele `A` și `M`. Contoarele `NA` și respectiv `NM`, conțin numărul de procese gata să treacă bariera `A` și cele care deja au trecut bariera `A` dar nu au trecut încă de bariera `M`. În partea a doua a protocolului procesul `NM` blocat pe `M` va intra în secțiunea sa critică unul câte unul, utilizând o tehnică de cascadare similară cu aceea utilizată în prima parte. Definiți un algoritm care se conformează cu această descriere.

a.

```

var a, b, m: semaphore;
    na, nm: 0 ... +∞;
a := 1; b := 1; m := 0; na := 0; nm := 0;
semWait(b); na ← na + 1; semSignal(b);
semWait(a); nm ← nm + 1;
semWait(b); na ← na - 1;
if na = 0 then semSignal(b); semSignal(m)
    else semSignal(b); semSignal(a)
endif;
semWait(m); nm ← nm - 1;
<critical section>;
if nm = 0 then semSignal(a)
    else semSignal(m)
endif;
```

40. Următoarea problemă a fost dată la un examen: Jurassic Park constă dintr-un muzeu al dinozaurilor și un parc pentru un safari de echitație. Există m pasageri și o singură mașină pentru pasageri. Pasagerii își petrec un timp în jurul muzeului după care se aliniază pentru a face o plimbare cu mașina pentru safari. Atunci când mașina este disponibilă, aceasta încarcă un pasager și face o plimbare în jurul parcului pentru o perioadă aleatoare de timp. Dacă toate cele n mașini plimbă pasageri, atunci un pasager care vrea să se plimbe trebuie să aștepte; dacă o mașină este gata de a prelua un pasager dar nu este nici unul care să aștepte, atunci așteaptă mașina. Utilizați semafoare pentru a sincroniza cele m procese pasageri cu cele n procese mașini. Următoarea schiță de cod a fost găsită pe o bucată de hârtie în camera unde s-a dat examenul. Analizați-o pentru corectitudine. Taxa și lipsa declarării variabilelor. Amintiți-vă că P și V corespund la `semWait` și `semSignal`.

```
resource Jurassic_Park()
    sem car_avail := 0, car_taken := 0, car_filled := 0, passenger_released := 0
    process passenger(i := 1 to num_passengers)
        do true -> nap(int(random(1000*wander_time)))
            P(car_avail); V(car_taken); P(car_filled)
            P(passenger_released)
        od
    end passenger

    process car(j := 1 to num_cars)
        do true -> V(car_avail); P(car_taken); V(car_filled)
            nap(int(random(1000*ride_time)))
            V(passenger_released)
        od
    end car
end Jurassic_Park
```

Raspuns:

Codul are o problemă majoră. `V(passenger_released)` din codul pentru mașină poate debloca un pasager blocat pe `P (passenger_released)` care nu este unul care călătorește în mașina care ia dat `V()`.

41. În comentariile din figura Figure 5.9 și tabelul 5.4 , s-a statuat că “nu trebuie realizată o simplă mutare a liniei condiționale în interiorul secțiunii critice (controlate de s) a consumatorului pentru că aceasta poate conduce la blocaj (deadlock).” Demonstrați aceasta cu un tabel similar tabelului 5.4.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

Note: White areas represent the critical section controlled by semaphore s.

Raspuns:

	Producer	Consumer	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		if (n==0) (semWaitB(delay))			
10	semWaitB(s)				

Atât producătorul cât și consumatorul sunt blocați.

42. Luati în considerare soluția problemei infinite-buffer producer/consumer definită în figura 5.10. Presupuneți că avem cazul (comun) în care producătorul și consumatorul rulează la aproximativ aceeași viteză. Scenariul poate fi be:
 Producer: append; semSignal ; produce; ... ; append; semSignal ; produce; ...
 Consumer: consume; ... ; take; semWait ; consume; ... ; take; semWait ; ...
 Producătorul gestionează întotdeauna în sensul de a ataşa un nou element la buffer și de a semnaliza pe durata consumării elementului precedent de către consumator. Producătorul atașează întotdeauna la un buffer gol și consumatorul preia întotdeauna singurul element din buffer. Deși consumatorul nu se blochează niciodată pe semafor, sunt realizate o multitudine de apeluri pe mecanismul semaforului creeând un supracontrol considerabil. Construiți un program nou care să fie mult mai eficient sub aceste circumstanțe. Construct a new program that will be more efficient under these circumstances. Sugestii: permiteți ca n să aibă valoarea -1, care înseamnă nu numai că bufferul este gol dar și că, consumatorul a detectat acest fapt și s-a blocat până când producătorul alimentează cu date proaspete. Soluția nu necesită utilizarea variabilei m din figura 5.10.

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
  
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```

program producerconsumer;
  var
    n: integer;
    s: (*binary*) semaphore (:= 1);
    delay: (*binary*) semaphore (:= 0);
  procedure producer;
  begin
    repeat
      produce;
      semWaitB(s);
      append;
      n := n + 1;
      if n=0 then semSignalB(delay);
      semSignalB(s)
    forever
  end;
  procedure consumer;
  begin
    repeat
      semWaitB(s);
      take;
      n := n - 1;
      if n = -1 then
        begin
          semSignalB(s);
          semWaitB(delay);
          semWaitB(s)
        end;
      consume;
      semSignalB(s)
    forever
  end;
begin (*main program*)
  n := 0;
  parbegin
    producer; consumer
  parend
end.

```

43. Considerați Figura 5.13 . Se schimbă înțelesul programului dacă dacă se realizează următoarele interschimbări?
- semWait(e) ; semWait(s)
 - semSignal(s) ; semSignal(n)
 - semWait(n) ; semWait(s)
 - semSignal(s) ; semSignal(e).

```

/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Toate schimbările vor determina comportări incorecte ale programului. Semaforul s controlează accesul la regiunea critică și doriți ca regiunea critică să includă doar funcția append sau take.

44. Următorul pseudocod reprezintă o implementare corectă problemei producător/consumator cu buffer mărginit:

<pre> item[3] buffer; // initially empty semaphore empty; // initialized to +3 semaphore full; // initialized to 0 binary_semaphore mutex; // initialized to 1 void producer() { ... while (true) { item = produce(); p1: wait(empty); / wait(mutex); p2: append(item); \ signal(mutex); p3: signal(full); } } </pre>	<pre> void consumer() { ... while (true) { c1: wait(full); / wait(mutex); c2: item = take(); \ signal(mutex); c3: signal(empty); consume(item); } } </pre>
--	--

Etichetele p1, p2, p3 și c1, c2, c3 se referă la liniile de cod prezentate anterior (p2 și c2 acoperă fiecare trei linii de cod). Semafoarele empty și full sunt semafoare liniare care pot lua valori negative sau pozitive nemărginire. Există procese producător multiple, denumite ca Pa, Pb, Pc, etc., și procese

consumator multiple denumite ca Ca, Cb, Cc, etc. Fiecare semafor menține o coadă FIFO (first-in-first-out) pentru procesele blocate.

În diagrama de planificare prezentată dedesubt, fiecare linie reprezintă starea bufferului și a semafoarelor după execuția planificării. Pentru simplificare, presupunem că planificarea este astfel încât procesele nu sunt niciodată întrerupte în timpul execuției în timp ce execută o porțiune dată din codul p1, sau p2, ..., sau c3. Sarcina voastră este să completați următoarea diagramă:

Scheduled Step of Execution	full's State and Queue	Buffer	empty's State and Queue
Initialization	full = 0	OOO	empty = +3
Ca executes c1	full = -1 (Ca)	OOO	empty = +3
Cb executes c1	full = -2 (Ca, Cb)	OOO	empty = +3

Scheduled Step of Execution	full's State and Queue	Buffer	empty's State and Queue
Pa executes p1	full = -2 (Ca, Cb)	OOO	empty = +2
Pa executes p2	full = -2 (Ca, Cb)	X OO	empty = +2
Pa executes p3	full = -1 (Cb) Ca	X OO	empty = +2
Ca executes c2	full = -1 (Cb)	OOO	empty = +2
Ca executes c3	full = -1 (Cb)	OOO	empty = +3
Pb executes p1	full =		empty =
Pa executes p1	full =		empty =
Pa executes __	full =		empty =
Pb executes __	full =		empty =
Pb executes __	full =		empty =
Pc executes p1	full =		empty =
Cb executes __	full =		empty =
Pc executes __	full =		empty =
Cb executes __	full =		empty =
Pa executes __	full =		empty =
Pb executes p1-p3	full =		empty =
Pc executes __	full =		empty =
Pa executes p1	full =		empty =
Pd executes p1	full =		empty =
Ca executes c1-c3	full =		empty =
Pa executes __	full =		empty =
Cc executes c1-c2	full =		empty =
Pa executes __	full =		empty =
Cc executes c3	full =		empty =
Pd executes p2-p3	full =		empty =

Scheduled step of execution	full's state & queue	Buffer	empty's state & queue
Initialization	full = 0	000	empty = +3
Ca executes c1	full = -1 (Ca)	000	empty = +3
Cb executes c1	full = -2 (Ca, Cb)	000	empty = +3
Pa executes p1	full = -2 (Ca, Cb)	000	empty = +2
Pa executes p2	full = -2 (Ca, Cb)	XOO	empty = +2
Pa executes p3	full = -1 (Cb) Ca	XOO	empty = +2
Ca executes c2	full = -1 (Cb)	000	empty = +2
Ca executes c3	full = -1 (Cb)	000	empty = +3
Pb executes p1	full = -1 (Cb)	000	empty = +2
Pa executes p1	full = -1 (Cb)	000	empty = +1
Pa executes p2	full = -1 (Cb)	XOO	empty = +1
Pb executes p2	full = -1 (Cb)	XXO	empty = +1
Pb executes p3	full = 0 (Cb)	XXO	empty = +1
Pc executes p1	full = 0 (Cb)	XXO	empty = 0
Cb executes c2	full = 0	XOO	empty = 0
Pc executes p2	full = 0	XXO	empty = 0
Cb executes c3	full = 0	XXO	empty = +1
Pa executes p3	full = +1	XXO	empty = +1
Pb executes p1-p3	full = +2	XXX	empty = 0
Pc executes p3	full = +3	XXX	empty = 0
Pa executes p1	full = +3	XXX	empty = -1(Pa)
Pd executes p1	full = +3	XXX	Empty = -2(Pa, Pd)
Ca executes c1-c3	full = +2	XXO	empty = -1(Pd) Pa

Pa executes p2	full = +2	XXX	empty = -1(Pd)
Cc executes c1-c2	full = +1	XXO	empty = -1(Pd)
Pa executes p3	full = +2	XXO	empty = -1(Pd)
Cc executes c3	full = +2	XXO	empty = 0(Pd)
Pd executes p2-p3	full = +3	XXX	empty = 0

Diferențele de la un pas la altul sunt subliniate cu culoarea roșie.

45. Această problemă demonstrează utilizarea semafoarelor pentru a coordona trei tipuri de procese.

Santa Claus doarme în magazinul său din North Pole și poate fi trezit fie doar de (1) toți cei nouă reni care se întorc din vacanța lor în South Pacific, sau (2) de unul dintre spiridușii care au dificultăți în a construi jucării; pentru că permite lui Santa să doarmă, spiridușii pot să-l trezească doar atunci când trei dintre ei au probleme. Atunci când trei spiriduși au problemele lor rezolvate, oricare alti spiriduși care doresc să-l viziteze pe Santa trebuie să aștepte ca precedenții să se reîntoarcă. Dacă Santa se trezește cu trei spiriduși așteptând la ușa magazinului său, împreună cu ultimii reni întorși de la tropice, Santa a decis că spiridușii pot să aștepte până după Crăciun, pentru că este mult mai important ca să aibă sania gata. (se presupune că renii nu vor să părăsească tropicele și vor sta acolo până în ultimul moment). Ultimul ren care ajunge trebuie să-l ia pe Santa în timp ce ceilalți așteaptă într-o colibă încălzită înainte de a fi înămati la sanie. Rezolvați această problemă utilizând semafoare.

a.

```
#define REINDEER 9 /* max # of reindeer */
#define ELVES 3 /* size of elf group */
/* Semaphores */
only_elves = 3, /* 3 go to Santa */
emutex = 1, /* update elf_cnt */
rmutex = 1, /* update rein_ct */
rein_semWait = 0, /* block early arrivals
    back from islands */
sleigh = 0, /* all reindeer semWait
    around the sleigh */
done = 0, /* toys all delivered */
santa_semSignal = 0, /* 1st 2 elves semWait on
    this outside Santa's shop */
santa = 0, /* Santa sleeps on this
    blocked semaphore */
```

```
/* Elf Process */
for (;;) {
    semWait (only_elves) /* only 3 elves "in" */
    semWait (emutex)
    elf_ct++
    if (elf_ct == ELVES) {
        semSignal (emutex)
        semSignal (santa) /* 3rd elf wakes Santa
    */
    }
    else {
        semSignal (emutex)
        semWait (santa_semSignal) /* semWait
outside
    Santa's shop door */
```

```

problem = 0; /* semWait to pose the
question to Santa */
elf_done = 0; /* receive reply */
/* Shared Integers */
rein_ct = 0; /* # of reindeer back */
elf_ct = 0; /* # of elves with problem */
/* Reindeer Process */
for (;;) {
    tan on the beaches in the Pacific until
        Christmas is close
    semWait (rmutex)
    rein_ct++
    if (rein_ct == REINDEER) {
        semSignal (rmutex)
        semSignal (santa)
    }
    else {
        semSignal (rmutex)
        semWait (rein_semWait)
    }
/* all reindeer semWaiting to be attached to sleigh */
    semWait (sleigh)
    fly off to deliver toys
    semWait (done)
    head back to the Pacific islands
} /* end "forever" loop */

}

semWait (problem)
ask question /* Santa woke elf up */
semWait (elf_done)
semSignal (only_elves)
} /* end "forever" loop */
/* Santa Process */
for (;;) {
    semWait (santa) /* Santa "rests" */
    /* mutual exclusion is not needed on rein_ct
       because if it is not equal to REINDEER,
       then elves woke up Santa */
    if (rein_ct == REINDEER) {
        semWait (rmutex)
        rein_ct = 0 /* reset while blocked */
        semSignal (rmutex)
        for (i = 0; i < REINDEER - 1; i++)
            semSignal (rein_semWait)
        for (i = 0; i < REINDEER; i++)
            semSignal (sleigh)
        deliver all the toys and return
        for (i = 0; i < REINDEER; i++)
            semSignal (done)
    }
    else { /* 3 elves have arrive */
        for (i = 0; i < ELVES - 1; i++)
            semSignal (santa_semSignal)
        semWait (emutex)
        elf_ct = 0
        semSignal (emutex)
        for (i = 0; i < ELVES; i++) {

            semSignal (problem)
            answer that question
            semSignal (elf_done)
        }
    }
} /* end "forever" loop */

```

46. Arătați că trimiterea de mesaje și semafoarele au funcționalitate echivalentă prin: Implementați trimiterea de mesaje prin semafoare. Sugestie: Utilizați buffere partajate pentru a păstra mailbox-urile , fiecare constând dintr-o arie cu sloturi pentru mesaje.

Există o arie de sloturi pentru mesaje care constituie bufferul. Fiecare proces menține o listă cu legături pentru sloturi în bufferul care constituie căsuța poștală (mailbox) pentru acel proces. Operatiile cu mesaje pot fi implementate astfel:

send (message, dest)
semWait (mbuf) semWait for message buffer available
semWait (mutex) mutual exclusion on message queue
acquire free buffer slog
copy message to slot
link slot to other messages
semSignal (dest.sem) wake destination process
semSignal (mutex) release mutual exclusion

receive (message)
semWait (own.sem) semWait for message to arrive
semWait (mutex) mutual exclusion on message queue
unlink slot from own.queue
copy buffer slot to message
add buffer slot to freelist
semSignal (mbuf) indicate message slot freed
semSignal (mutex) release mutual exclusion

unde mbuf este inițializat cu numărul total de sloturi disponibile pentru mesaje: own și dest se referă la cozile de mesaje pentru fiecare proces și sunt inițializate cu 0.

47. Arătați că trimiterea de mesaje și semafoarele au funcționalitate echivalentă prin:
Implementați un semafor utilizând trimiterea de mesaje. Sugestie: Introduceți un proces separat de sincronizare.

Procesul de sincronizare gestionează un numărator și o listă de legături a proceselor în așteptare pentru fiecare semafor. Pentru a executa un WAIT sau un SIGNAL, procesul apelează procedurile corespunzătoare din librărie, wait și signal, care trimit un mesaj la procesul de sincronizare, specificând atât operația dorită cât și semaforul ce urmează a fi utilizat. Procedura din librărie executată apoi un RECEIVE pentru a prelua răspunsul de la procesul de sincronizare. Atunci când sosesc un mesaj, procesul de sincronizare testează număratorul ca să vadă dacă operația cerută poate fi completată. SIGNAL poate fi întotdeauna completată dar WAIT se va bloca dacă valoarea semaforului este 0. Dacă operația este permisă, procesul de sincronizare trimită înapoi un mesaj gol, deblocând astfel apelantul. Dacă, totuși, operația este un WAIT și semaforul este 0, procesul de sincronizare introduce apelantul într-o coadă și nu trimită un răspuns. Rezultatul este acela că procesul care face WAIT este blocat, așa cum ar și trebui să fie. Mai târziu, când se execută SIGNAL, procesul de sincronizare ia unul dintre procesele blocate pe semafor, fie într-o ordine FIFO, sau bazată pe priorități, sau altă ordine, și trimită un răspuns. Condițiile de cursă sunt evitate aici deoarece procesul de sincronizare gestionează doar o cerere la un moment dat.

48. Explicați care este neregula (problema) cu această implementare a problemei one-writer many-readers?

```
int readcount;                                // shared and initialized to 0
Semaphore mutex, wrt;                         // shared and initialized to 1;

// Writer :
semWait(mutex);
readcount := readcount + 1;
if readcount == 1 then semWait(wrt);
semSignal(mutex);
/* Writing performed */
semSignal(wrt);

// Readers :
semWait(wrt);
readcount := readcount - 1;
if readcount == 0 then Up(wrt);
semSignal(mutex);
```

Codul pentru un scriitor și mulți cititori este bun dacă presupunem că cititorii au întotdeauna prioritate. Problema este aceea că cititorii pot informa scriitorii deoarece aceștia nu pot să părăsească toată regiunea critică, cum ar fi de exemplu, există întotdeauna un cititor în regiunea critică, și atunci semaforul "wrt" nu poate fi niciodată semnalizat către scriitori și procesele scriitor nu vor avea acces la semaforul wrt ca să scrie în regiunea critică.

Cap. 6 Concurență: Deadlock and starvation

1. Dați exemple de resurse reutilizabile și consumabile.

Resurse reutilizabile sunt procesoarele, canalele I/O, memoria principală și secundară, dispozitivele și structurile de date cum ar fi fișierile, bazele de date și semafoarele. Exemple de resurse consumabile sunt întreruperile, semnalele, mesajele și informațiile din bufferele I/O.

2. Care sunt trei condiții care trebuie să fie prezente pentru ca blocajul (deadlock) să fie posibil?

Excluderea mutuală. Numai un proces poate utiliza o resursă la un moment dat. Hold and Wait. Un proces poate păstra resurse alocate în timp ce în timp ce așteaptă atribuirea altor resurse. No preemption. Nici o resursă nu poate fi luată cu forță de la un proces care o deține.

3. Care sunt cele patru condiții care creează blocaj?

Excluderea mutuală. Numai un proces poate utiliza o resursă la un moment dat. Hold and Wait. Un proces poate păstra resurse alocate în timp ce în timp ce așteaptă atribuirea altor resurse. No preemption. Nici o resursă nu poate fi luată cu forță de la un proces care o deține. Așteptarea circulară. Există un lanț închis de procese, astfel încât fiecare proces așteaptă cel puțin o resursă de care are nevoie următorul proces din lanț.

4. Cum poate fi prevenită condiția hold-and-wait?

Condiția hold-and-wait poate fi prevenită prin cerința ca un proces să-și ceară toate resurse o dată, și să blocheze procesul până ce toate cererile pot fi satisfăcute simultan.

5. Enumerați două căi prin care poate fi prevenită condiția fără suspendare (no-preemption).

Prima dată, dacă un proces deține anumite resurse și o cerere ulterioară este întârziată, acel proces poate elibera resursele originale și, dacă este necesar, să le ceară din nou împreună cu resursele adiționale. Alternativ, dacă un proces cere o resursă care este curent deținută de alt proces, sistemul de operare poate întrerupe al doilea proces și să-si ceară să elibereze resursele sale.

6. Cum poate fi prevenită condiția de așteptare circulară (circular wait condition)?

Condiția de așteptare circulară poate fi prevenită prin definirea unei ordini liniare a tipurilor de resurse. Dacă unui proces i s-au alocat resurse de tip R, atunci el poate cere în secvență numai acel tip de resursă care urmează în ordine tipului R.

7. Care este diferența între evitarea, detecția și prevenirea blocajului (deadlock)?

Prevenirea blocajului constrângă ca cererile de resurse să prevină cel puțin una dintre cele patru condiții care conduc la blocaj; aceasta este făcută fie indirect, prin prevenirea

unei dintre cele trei condiții (excluderea mutuală, hold and wait și no preemption) sau direct prin prevenirea așteptării circulare. Evitarea blocajului permite apariția celor trei condiții necesare, dar face o alegere judicioasă pentru a asigura că punctul de blocaj nu este niciodată atins. Cu detecția blocajului, resursele cerute sunt accesibile proceselor atunci când este posibil; periodic sistemul de operare rulează un algoritm care permite detectarea condiției de apariție a așteptării circulare.

Problems

8. Prezentați patru condiții de blocaj care se aplică figurii 6.1a

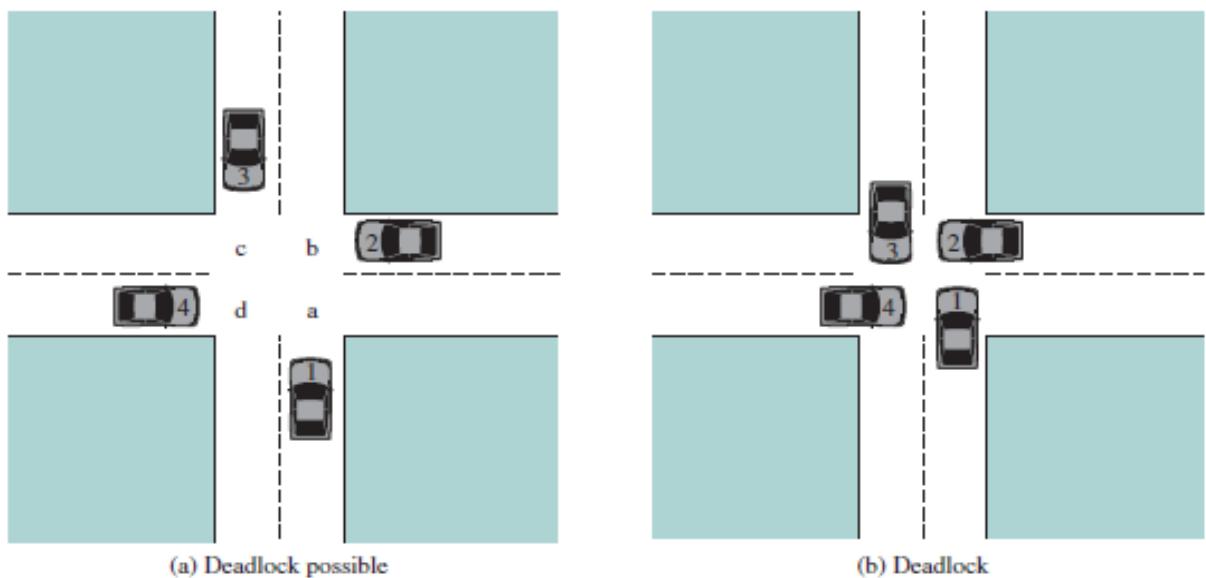


Figure 6.1 Illustration of Deadlock

Excluderea mutuală: numai o mașină poate ocupa un sector al intersecției la un moment dat. Hold and wait: nici o mașină nu se întoarce niciodată; fiecare mașină din intersecție așteaptă până când este disponibil sectorul din față sa. No preemption: Nici o mașină nu poate forța o altă mașină să nu-și urmeze calea. Așteptarea circulară: fiecare mașină așteaptă pentru un sector ocupat de altă mașină.

9. Prezentați cum fiecare dintre tehniciile de prevenire, evitare și detectie pot fi aplicate figurii 6.1

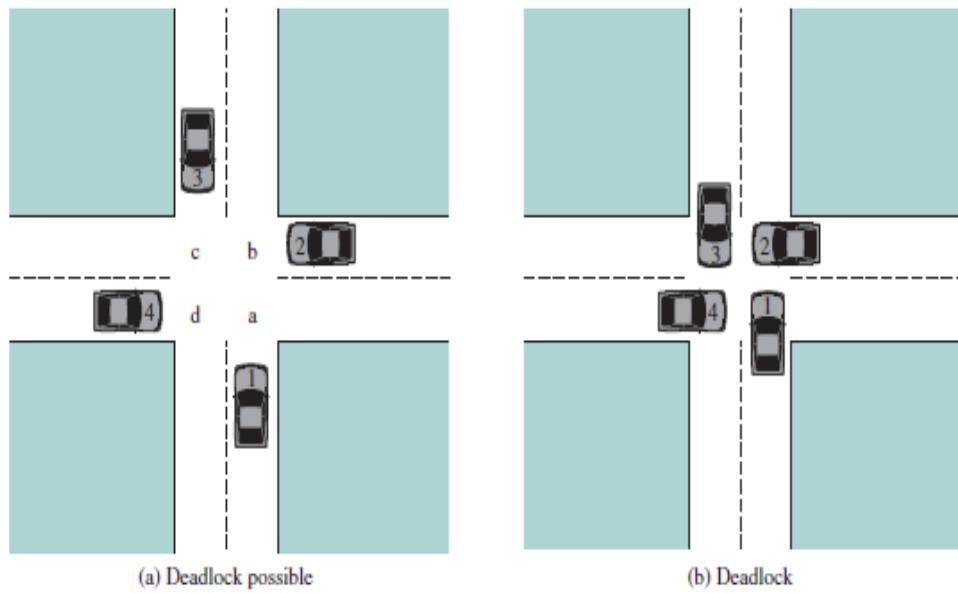


Figure 6.1 Illustration of Deadlock

Prevenirea: Abordarea hold-and-wait: necesită ca o mașină să solicite ambele sectoare de care este nevoie și să blocheze mașina până ce ambele sectoare pot fi obținute. Abordarea no preemption: eliberarea unui sector asignat deja este problematică, deoarece aceasta ar însemna ca mașina să dea cu spatele, fapt care poate să nu fie posibil deoarece altă mașină poate fi în spatele acesteia. Abordarea cu așteptare circulară: implică asignarea în ordine liniară a sectoarelor. Evitarea: Se pot aplica algoritmii discuți în capitol. În esență, blocajul este evitat prin refuzarea alocărilor care duc la blocaj. Detectia: Problema aici este din nou datul cu spatele.

10. Pentru figura 6.3, furnizați o descriere narativă a fiecărei dintre cele șase căi descrise, similar cu descrierea căilor din figura 6.2 furnizată în secțiunea 6.1.

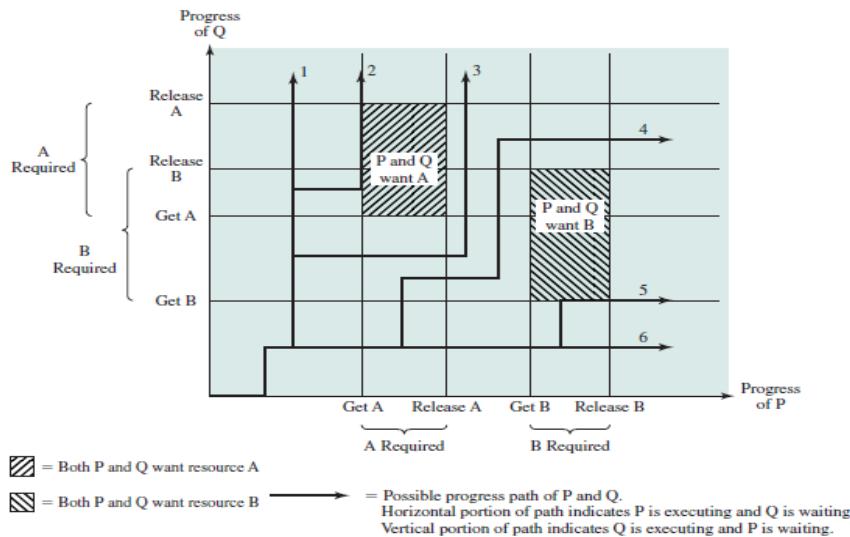


Figure 6.3 Example of No Deadlock [BACO03]

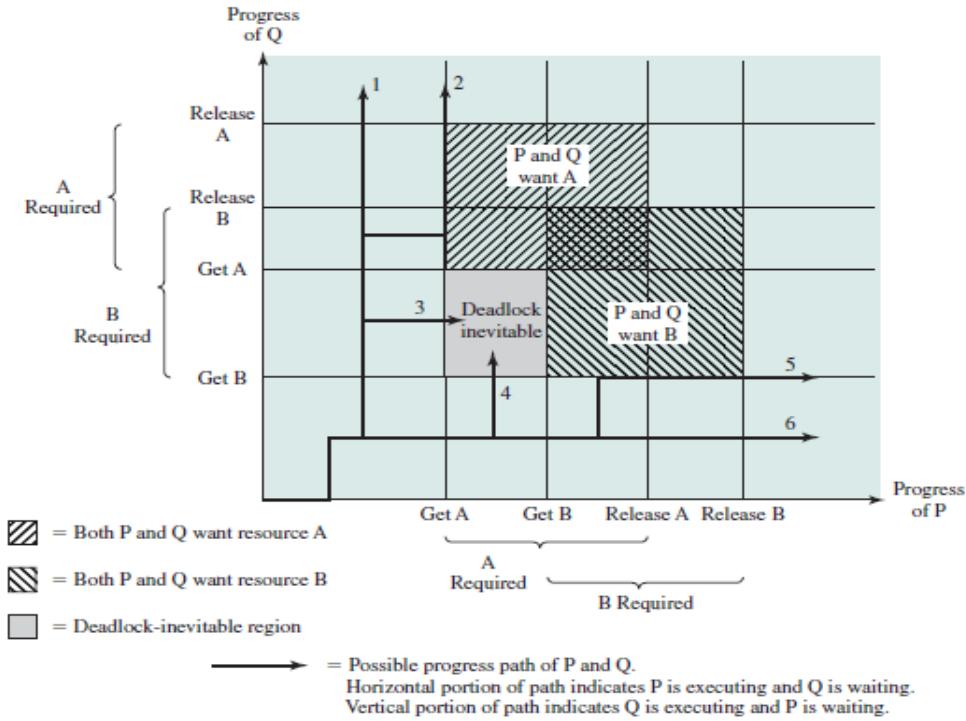


Figure 6.2 Example of Deadlock

1. Q achiziționează A și B, după care eliberează B și A. Când P își reia execuția acesta este capabil să achiziționeze ambele resurse.
2. Q achiziționează A și B. P se execută și se blochează pe cererea pentru A. Q eliberează B și A. Când P își reia execuția acesta este capabil să achiziționeze ambele resurse.
3. Q achiziționează B și apoi P achiziționează și eliberează A. Q achiziționează A și apoi eliberează B și A. Când P își reia execuția acesta este capabil să achiziționeze B.
4. P achiziționează A și apoi Q achiziționează B. P eliberează A. Q achiziționează A și apoi eliberează B. P achiziționează B și apoi eliberează B.
5. P achiziționează și apoi eliberează B. P achiziționează B. Q se execută și se blochează pe cererea pentru B. P eliberează B. Când Q își reia execuția acesta este capabil să achiziționeze ambele resurse.
6. P achiziționează A și eliberează A, apoi achiziționează și eliberează B. Când Q își reia execuția acesta este capabil să achiziționeze ambele resurse.

11. S-a afirmat că blocajul nu poate să apară pentru situația reflectată în figura 6.3. Justificați această declarație.

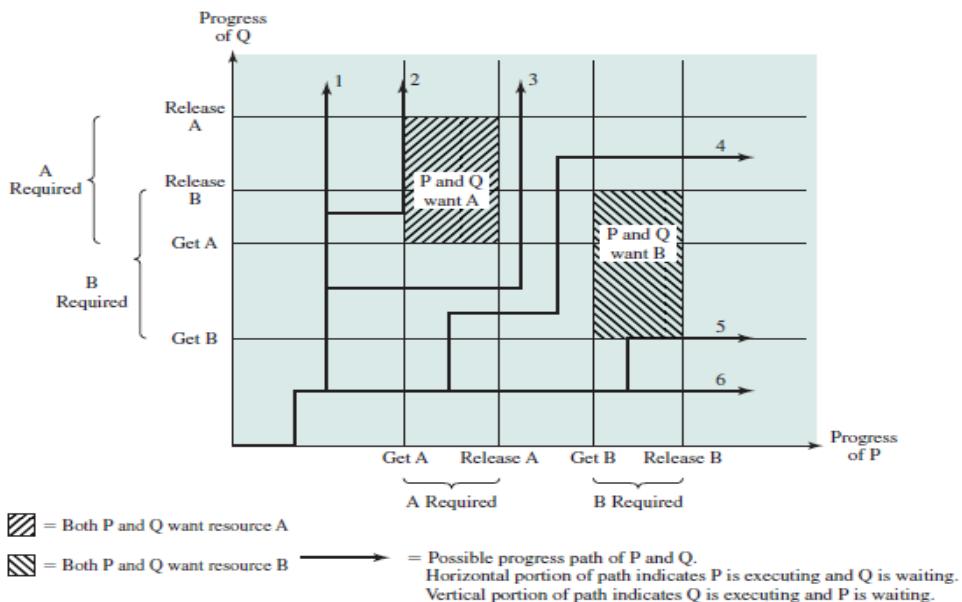


Figure 6.3 Example of No Deadlock [BACO03]

Dacă Q achiziționează B și A înainte ca P să ceară A, atunci Q poate utiliza aceste două resurse și apoi să le elibereze, permîțând lui A să avanseze. Dacă P achiziționează A înainte ca Q să ceară pe A, atunci Q poate să avanseze până la punctul unde îl cere pe A și apoi se blochează. Totuși, odată ce P eliberează A, Q poate să avanseze. Odată ce Q eliberează pe B, P poate să avanseze.

12. Fie următoarea stare dată pentru algoritmul bancherului (Banker's Algorithm).

6 procese P0 la P5

4 tipuri de resurse : A (15 instanțe); B (6 instanțe); C (9 instanțe); D (10 instanțe)

Instantaneu la momentul T0:

Available								
	A	B	C	D		A	B	C
	6	3	5	4				
Current allocation								
Process	A	B	C	D	A	B	C	D
P0	2	0	2	1	9	5	5	5
P1	0	1	1	1	2	2	3	3
P2	4	1	0	2	7	5	4	4
P3	1	0	0	1	3	3	3	2
P4	1	1	0	0	5	2	2	1
P5	1	0	1	1	4	4	4	4

Verificați că aria Available a fost calculată corect.

$$15 - (2+0+4+1+1+1) = 6$$

$$6 - (0+1+1+0+1+0) = 3$$

$$9 - (2+1+0+0+0+1) = 5$$

$$10 - (1+1+2+1+0+1) = 4$$

13. Fie următoarea stare dată pentru algoritmul bancherului (Banker's Algorithm).

6 procese P0 la P5

4 tipuri de resurse : A (15 instanțe); B (6 instanțe); C (9 instanțe); D (10 instanțe)

Instantaneu la momentul T0:

		Available							
		A	B	C	D				
		6	3	5	4				
Process		Current allocation		Maximum demand					
Process		A	B	C	D	A	B	C	D
P0		2	0	2	1	9	5	5	5
P1		0	1	1	1	2	2	3	3
P2		4	1	0	2	7	5	4	4
P3		1	0	0	1	3	3	3	2
P4		1	1	0	0	5	2	2	1
P5		1	0	1	1	4	4	4	4

Calculați matricea Need.

Raspuns:

process		need			
		A	B	C	D
P0		7	5	3	4
P1		2	1	2	2
P2		3	4	4	2
P3		2	3	3	1
P4		4	1	2	1
P5		3	4	3	3

14. Fie următoarea stare dată pentru algoritmul bancherului (Banker's Algorithm).

6 procese P0 la P5

4 tipuri de resurse : A (15 instanțe); B (6 instanțe); C (9 instanțe); D (10 instanțe)

Instantaneu la momentul T0:

		Available							
		A	B	C	D				
		6	3	5	4				
Process		Current allocation		Maximum demand					
Process		A	B	C	D	A	B	C	D
P0		2	0	2	1	9	5	5	5
P1		0	1	1	1	2	2	3	3
P2		4	1	0	2	7	5	4	4
P3		1	0	0	1	3	3	3	2
P4		1	1	0	0	5	2	2	1
P5		1	0	1	1	4	4	4	4

Arătați că starea curentă este sigură, adică, prezentați o secvență sigură de procese. În plus față de secvență arătați cum Available (aria de lucru - working array) se modifică pe măsură ce fiecare proces se termină.

process	available			
	A	B	C	D
P5	7	3	6	5
P4	8	4	6	5
P3	9	4	6	6
P2	13	5	6	8
P1	13	6	7	9
P1	15	6	9	10

15. Fie următoarea stare dată pentru algoritmul bancherului (Banker's Algorithm).

6 procese P0 la P5

4 tipuri de resurse : A (15 instanțe); B (6 instanțe); C (9 instanțe); D (10 instanțe)

Instantaneu la momentul T0:

Process	Available				Current allocation				Maximum demand			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	2	1	9	5	5	5	2	0	2	1
P1	0	1	1	1	2	2	3	3	0	1	1	0
P2	4	1	0	2	7	5	4	4	4	1	0	0
P3	1	0	0	1	3	3	3	2	1	0	0	1
P4	1	1	0	0	5	2	2	1	1	1	0	0
P5	1	0	1	1	4	4	4	4	1	0	1	1

Fie cererea (3,2,3,3) formulată de procesul P5. Poate fi această cerere satisfăcută? De ce sau de ce nu?

- a. Răspunsul este NU din următorul motiv: DACĂ această cerere ar fi fost satisfăcută, atunci noua matrice de alocare ar fi:

process	allocation			
	A	B	C	D
P0	2	0	2	1
P1	0	1	1	1
P2	4	1	0	2
P3	1	0	0	1
P4	1	1	0	0
P5	4	2	4	4

Noua matrice need ar fi

process	allocation			
	A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	0	2	0	0

Ca urmare Available este:

Available			
A	B	C	D
3	1	2	1

Ceea ce înseamnă că Nu putem satisface orice nevoie a proceselor.

16. În codul prezentat în continuare, trei procese sunt în competiție pentru șase resurse etichetate de la A la F.

Utilizând graful de alocare a resurselor (figurile 6.5 și 6.6) prezentați o posibilitate de apariție a blocajului pentru această implementare.

<pre>void P0() { while (true) { get(A); get(B); get(C); // critical region: // use A, B, C release(A); release(B); release(C); } }</pre>	<pre>void P1() { while (true) { get(D); get(E); get(B); // critical region: // use D, E, B release(D); release(E); release(B); } }</pre>	<pre>void P2() { while (true) { get(C); get(F); get(D); // critical region: // use C, F, D release(C); release(F); release(D); } }</pre>
--	--	--

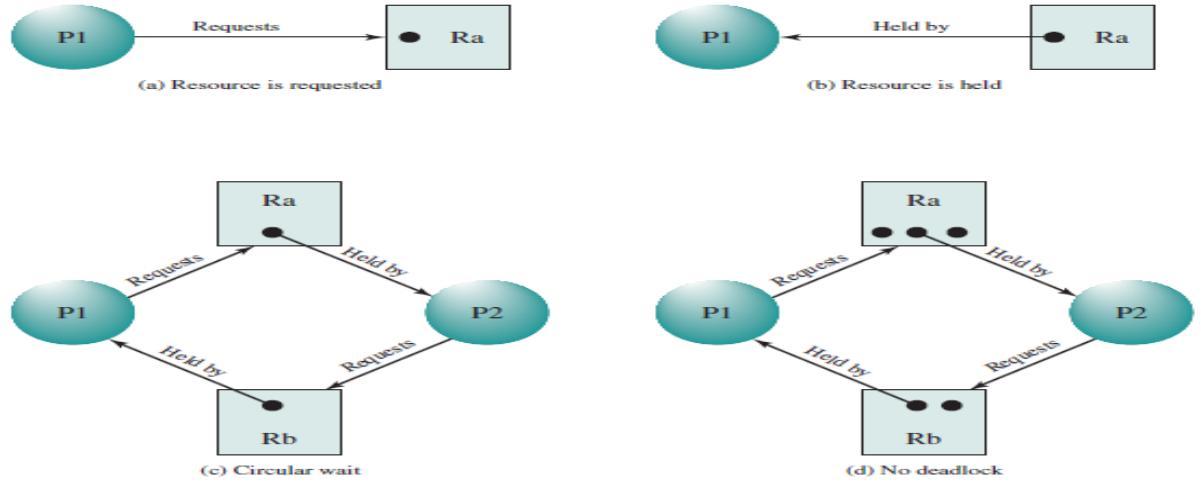


Figure 6.5 Examples of Resource Allocation Graphs

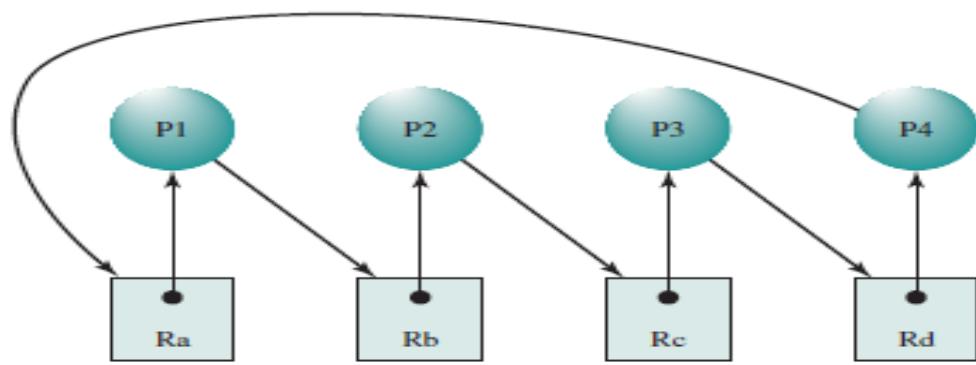
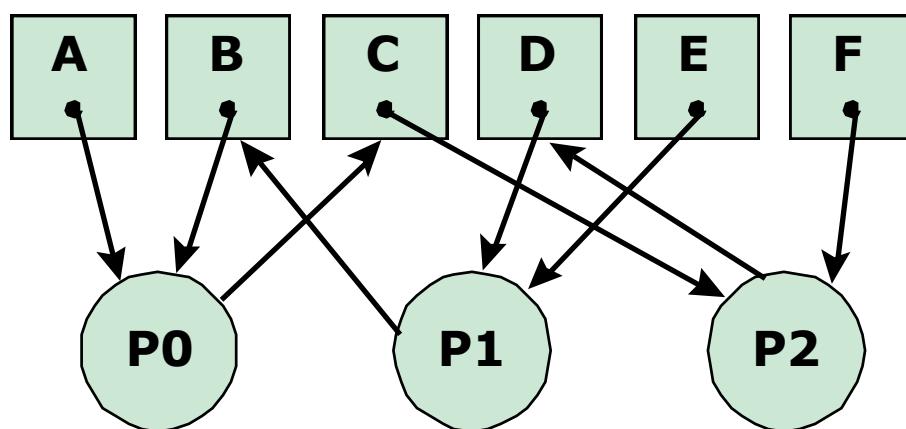
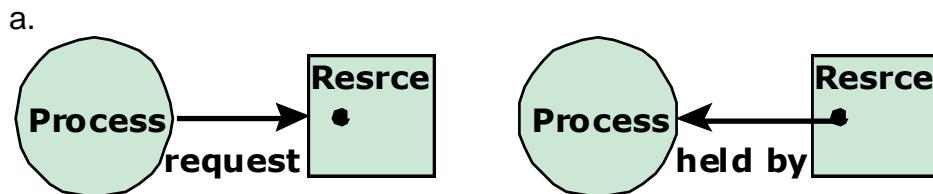
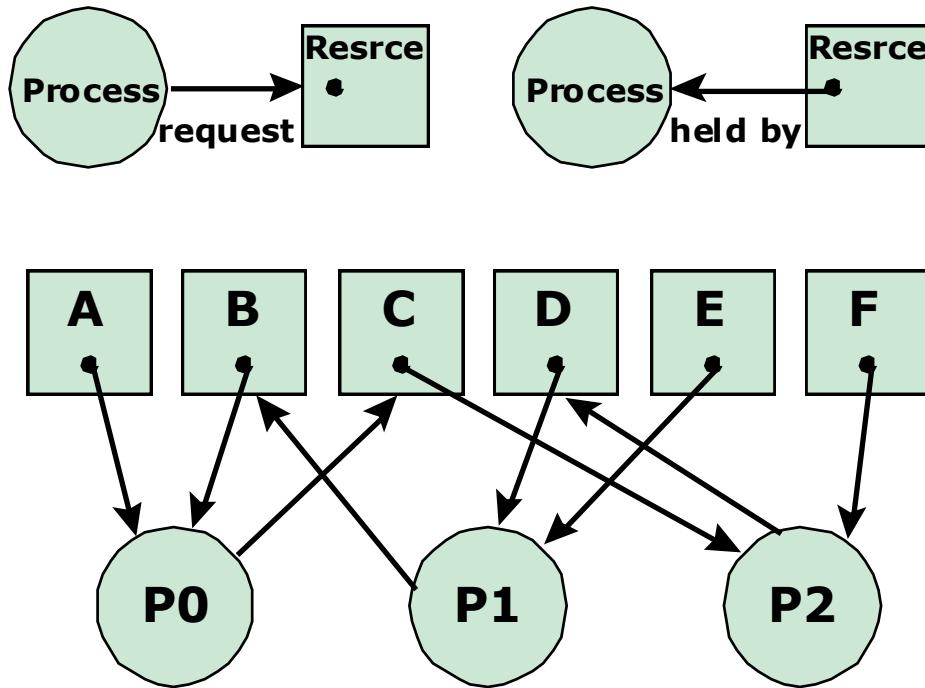


Figure 6.6 Resource Allocation Graph for Figure 6.1b



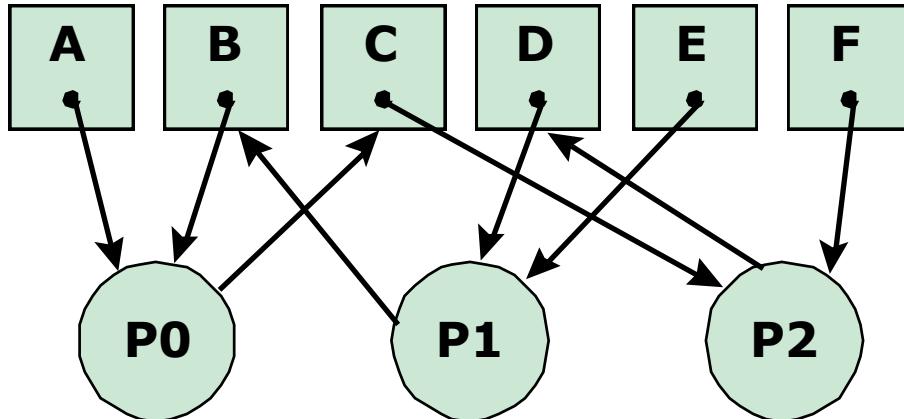
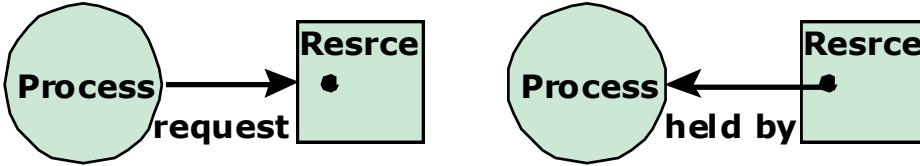
Există un blocaj dacă planificare este, de exemplu P0-P1-P2-P0-P1-P2 (linie cu linie): Fiecare din cele 6 resurse vor fi apoi păstrate de către unul dintre procese, astfel încât toate cele 3 procese sunt acum blocate la cea de a treia linie înăuntrul buclei, așteptând resursa pe care o detine alt proces. Aceasta este ilustrat prin așteptarea circulară în din figura de mai sus: P0→C→P2→D→P1→B→P0.

b.



Există un blocaj dacă planificare este, de exemplu P0-P1-P2-P0-P1-P3 (linie cu linie): Fiecare din cele 6 resurse vor fi apoi păstrate de către unul dintre procese, astfel încât toate cele 3 procese sunt acum blocate la cea de a treia linie înăuntrul buclei, așteptând resursa pe care o detine alt proces. Aceasta este ilustrat prin așteptarea circulară în din figura de mai sus: P0→C→P2→D→P1→B→P0.

a.



Există un blocaj dacă planificare este, de exemplu P0-P1-P2-P0-P1-P2 (linie cu linie): Fiecare din cele 6 resurse vor fi apoi păstrate de către unul dintre procese, astfel încât toate cele 3 procese sunt acum blocate la cea de a treia linie înăuntrul buclei, așteptând resursa pe care o detine alt proces. Aceasta este ilustrat prin așteptarea circulară din RAG-ul (Resource Allocation Graph – graful de alocare a resurselor) din figura de mai sus: P0→C→P1→D→P2→B→P1.

17. În codul prezentat în continuare, trei procese sunt în competiție pentru șase resurse etichetate de la A la F. Modificați ordinea unor cereri de a prelua resurse pentru a preveni orice blocaj. Nu puteți muta cererile între proceduri, doar schimbați ordinea în cadrul procedurii. Utilizați un graf de alocare a resurselor pentru a justifica răspunsul.

<pre>void P0() { while (true) { get(A); get(B); get(C); // critical region: // use A, B, C release(A); release(B); release(C); } }</pre>	<pre>void P1() { while (true) { get(D); get(E); get(B); // critical region: // use D, E, B release(D); release(E); release(B); } }</pre>	<pre>void P2() { while (true) { get(C); get(F); get(D); // critical region: // use C, F, D release(C); release(F); release(D); } }</pre>
--	--	--

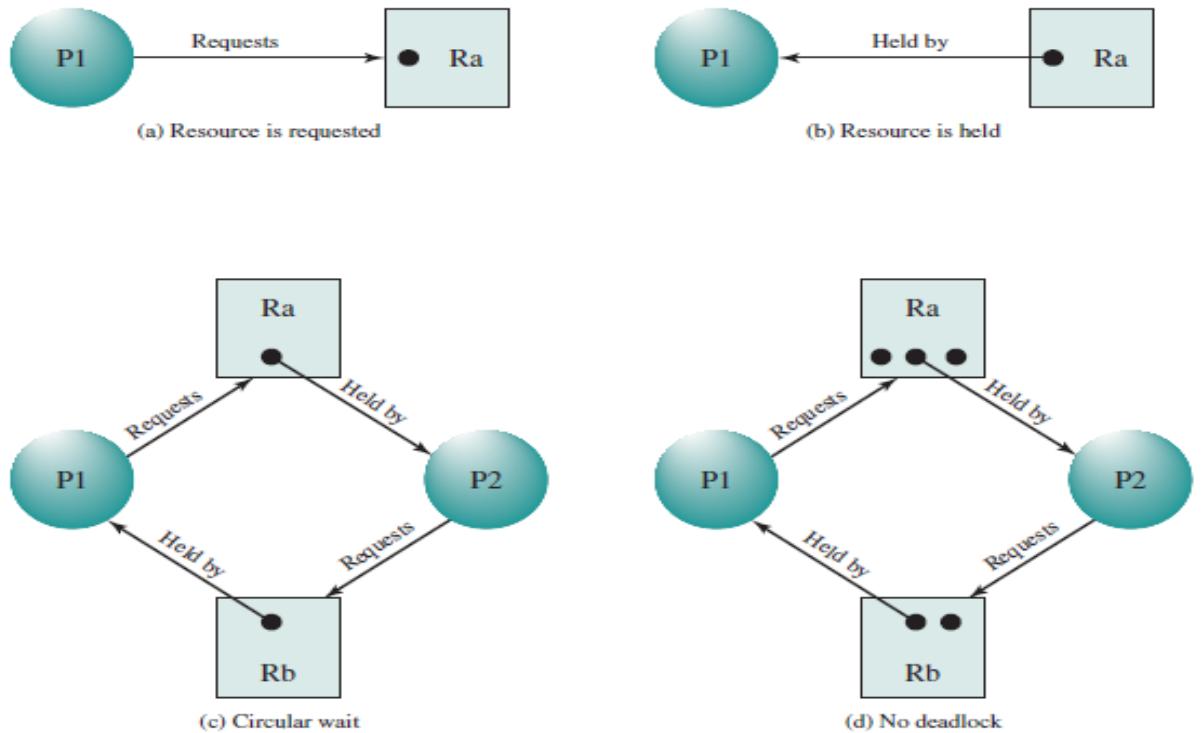


Figure 6.5 Examples of Resource Allocation Graphs

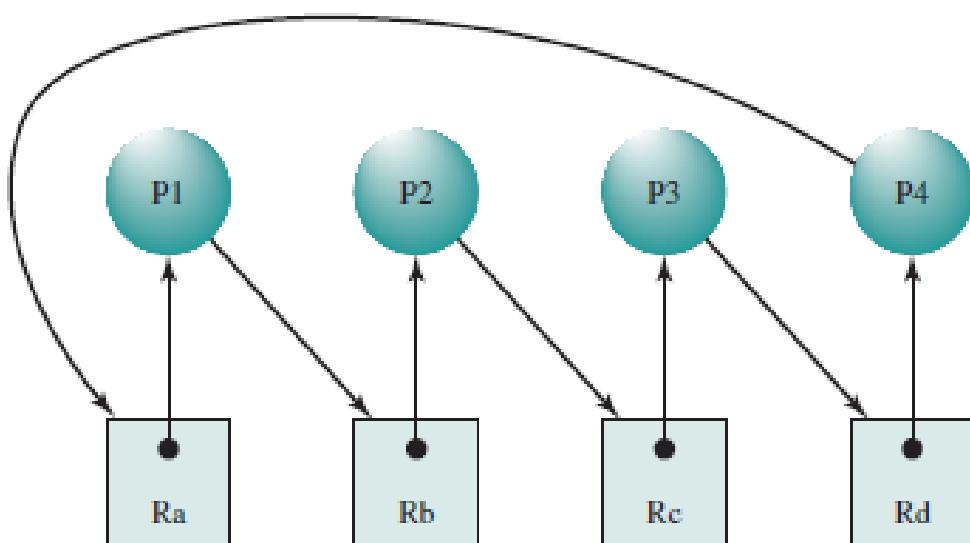


Figure 6.6 Resource Allocation Graph for Figure 6.1b

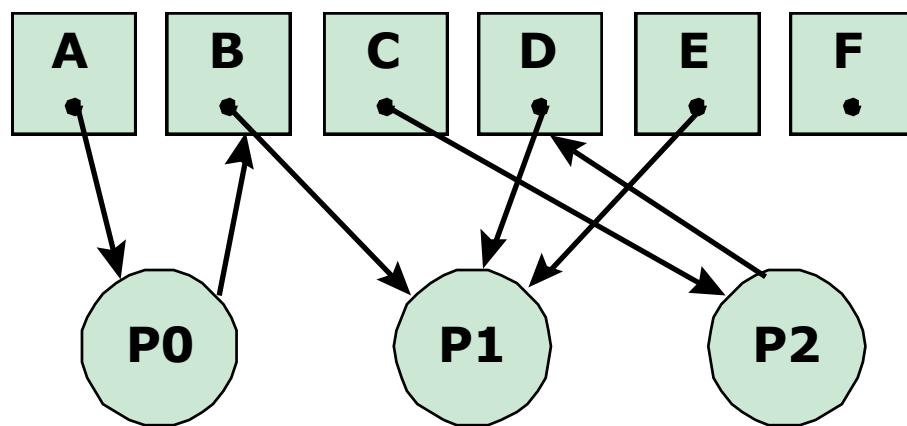
a.

Orice schimbare a ordinii apelului `get()` care alfabetizează resursele înăuntru codului fiecărui procesă evita blocajul. Mai general, poate fi o ordine alfabetică directă sau inversă, sau orice ordine arbitrară predefinită cu o listă de ordonare a resurselor care

trebuie respectată înăuntrul fiecărui proces. Explicația: dacă resursele sunt unic ordonate, nu sunt posibile ciclări deoarece un proces nu poate deține o resursă care vine după ce altă resursă este deținută din lista de ordonare. De exemplu:

A	B	C
B	D	D
C	E	F

Cu acest cod, și pornind cu același scenariu de planificare pentru cazul cel mai defavorabil P0-P1-P2, putem continua fie cu P1-C1-CR1... sau cu P2-P2-CR2.... De exemplu, în cazul P1-P1 avem următorul RAG (Resource Allocation Graph – graful de alocare a resurselor) fără așteptare circulară:



După intrarea în CR1 (critical region), P1 eliberează apoi toate resursele sale și P0 și P2 sunt libere să avanseze. În general același lucru se întâmplă cu orice ordine fixă a resurselor: unul dintre cele trei procese va fi capabil întotdeauna să intre în regiunea sa critică, și după ieșire să lase celelalte două procese să progreseze.

18. Un sistem de tip spooling (Figura 6.16) conține un proces de intrare I, un proces utilizator P, și un proces de ieșire O conectate la două buffere. Procesele schimbă date sub formă de blocuri de mărime egală. Aceste blocuri sunt bufferate pe disc utilizând o delimitare flotantă între bufferele de intrare și cele de ieșire, în funcție de viteza proceselor. Primitivele de comunicație asigură satisfacerea următoarelor constrângeri:

$$i + o \leq \max$$

Unde

\max = numărul maxim de blocuri pe disc

i = numărul de blocuri de intrare pe disc

o = numărul blocurilor de ieșire pe disc.

Se cunosc următoarele despre procese :

Atât timp cât mediul alimentează date, procesul I eventual le va prelua pe disc (spațiul oferit pe disc devine disponibil).

Cât timp sunt disponibile intrări pe disc, procesul P eventual le va consuma și va trimite le ieșire o cantitate finită de date pe disc pentru fiecare bloc de intrare

(spațiul oferit pe disc devine disponibil). 3. Cât timp sunt disponibile pe disc ieșiri, procesul O eventual le va consuma.

Arătați că acest sistem poate ajunge la blocaj.

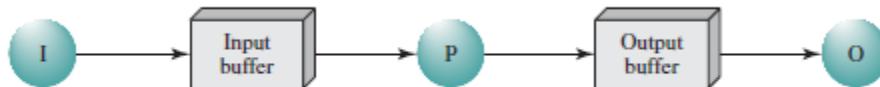


Figure 6.16 A Spooling System

Apare un blocaj atunci când procesul I umple discul cu intrarea ($i=\max$) și procesul I așteaptă să transfere mai multă ieșire pe disc și procesul O așteaptă să transfere mai multă ieșire de pe disc.

19. Un sistem de tip spooling (Figura 6.16) conține un proces de intrare I, un proces utilizator P, și un proces de ieșire O conectate la două buffere. Procesele schimbă date sub formă de blocuri de mărime egală. Aceste blocuri sunt bufferate pe disc utilizând o delimitare flotantă între bufferele de intrare și cele de ieșire, în funcție de viteza proceselor. Primitivile de comunicație asigură satisfacerea următoarelor constrângeri:

$$i + o \leq \max$$

Unde

\max = numărul maxim de blocuri pe disc

i = numărul de blocuri de intrare pe disc

o = numărul blocurilor de ieșire pe disc.

Se cunosc următoarele despre procese :

Atât timp cât mediul alimentează date, procesul I eventual le va prelua pe disc (spațiul oferit pe disc devine disponibil).

Cât timp sunt disponibile intrări pe disc, procesul P eventual le va consuma și va trimite le ieșire o cantitate finită de date pe disc pentru fiecare bloc de intrare (spațiul oferit pe disc devine disponibil). 3. Cât timp sunt disponibile pe disc ieșiri, procesul O eventual le va consuma.

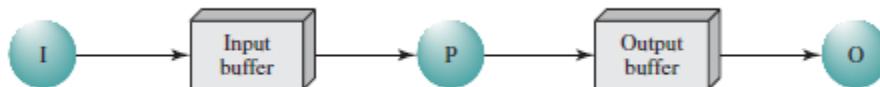


Figure 6.16 A Spooling System

Sugerați o constrângere suplimentară referitoare la resurse care să prevină blocajul din problema 6.7, dar care să permită ca delimitarea între bufferele de intrare și ieșire să varieze în concordanță cu nevoile prezente ale proceselor.

Rezervați pentru bufferul de ieșire) permanent un număr minim de blocuri (denumit reso), și permiteți ca numărul de blocuri de ieșire să depășească această limită atunci când este spațiu disponibil pe disc. Constrângerile pentru resurse devin acum:

$$i + o \leq \max$$

$$i \leq \max - reso$$

unde

$$0 < reso < max$$

Dacă procesul P așteaptă să furnizeze ieșire pentru disc, procesul O va consuma eventual toată ieșirea anterioară și va face cel puțin reso pagini disponibile pentru viitoarea ieșire, permitându-i astfel lui P să continue. Procesul I poate fi întârziat dacă discul este plin de I/O; dar curând sau mai târziu, toate intrările anterioare vor fi consumate de P și ieșirea corespunzătoare va fi consumată de O, validând astfel ca I să continue.

20. În sistemul cu multiprogramare THE [DIJK68], un suport extern (drum = predecesorul discului pentru memoria secundară) este divizat în buffere de intrare, arii de procesare și buffere de ieșire cu delimitări flotante, care depind de viteza proceselor implicate. Starea curentă a memorie secundare poate fi caracterizată de următorii parametrii:

max = numărul maxim de pagini pe memoria secundară

i = numărul de pagini de intrare de pe memoria secundară

p = numărul de pagini de procesare de pe memoria secundară

o = numărul de pagini de ieșire de pe memoria secundară

$reso$ = numărul minim de pagini rezervate pentru ieșire

$resp$ = numărul minim de pagini rezervate pentru procesare.

Formulați constrângările necesare de resurse care garantează că, capacitatea memoriei secundare nu va fi depășită, și că este rezervat numărul minim de pagini pentru ieșire și procesare.

$$i + o + p \leq max$$

$$i + o \leq max - resp$$

$$i + p \leq max - reso$$

$$\leq max - (reso + resp)$$

21. În sistemul cu multiprogramare THE o pagină poate realiza următoarele tranziții de stare :

1. empty : input buffer (producerea unei intrări)
2. input buffer : processing area (consumarea unei intrări)
3. processing area : output buffer (producerea unei ieșiri)
4. output buffer : empty (consumarea unei ieșiri)
5. empty : processing area (apel de procedură)
6. processing area : empty (revenire din procedură)

Definiți efectul acestor tranziții în funcție de cantitățile i , o , și p .

1. $i \leftarrow i + 1$
2. $i \leftarrow i - 1; p \leftarrow p + 1$
3. $p \leftarrow p - 1; o \leftarrow o + 1$
4. $o \leftarrow o - 1$
5. $p \leftarrow p + 1$
6. $p \leftarrow p - 1$

22. În sistemul cu multiprogramare THE o pagină poate realiza următoarele tranziții de stare :

1. empty : input buffer (producerea unei intrări)
2. input buffer : processing area (consumarea unei intrări)
3. processing area : output buffer (producerea unei ieșiri)
4. output buffer : empty (consumarea unei ieșiri)
5. empty : processing area (apel de procedură)
6. processing area : empty (revenire din procedură)

a. Definiți efectul acestor tranziții în funcție de cantitățile i , o , și p .

Poate oricare dintre ele să conducă la blocaj dacă se realizează presupunerea din problema 6.7 (Un sistem de tip spooling (Figura 6.16) conține un proces de intrare I, un proces utilizator P, și un proces de ieșire O conectate la două buffere. Procesele schimbă date sub formă de blocuri de mărime egală. Aceste blocuri sunt bufferate pe disc utilizând o delimitare flotantă între bufferele de intrare și cele de ieșire, în funcție de viteza proceselor. Primitivele de comunicație asigură satisfacerea următoarelor constrângeri:

$$i + o \leq max$$

Unde

$max =$ numărul maxim de blocuri pe disc

$i =$ numărul de blocuri de intrare pe disc

$o =$ numărul blocurilor de ieșire pe disc.

Se cunosc următoarele despre procese :

Atât timp cât mediul alimentează date, procesul I eventual le va prelua pe disc (spațiul oferit pe disc devine disponibil).

Cât timp sunt disponibile intrări pe disc, procesul P eventual le va consuma și va trimite le ieșire o cantitate finită de date pe disc pentru fiecare bloc de intrare (spațiul oferit pe disc devine disponibil). 3. Cât timp sunt disponibile pe disc ieșiri, procesul O eventual le va consuma.

Arătați că acest sistem poate ajunge la blocaj.

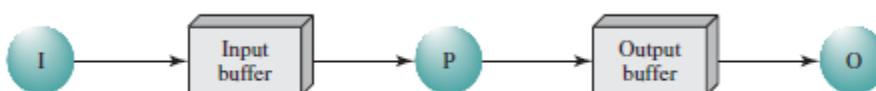


Figure 6.16 A Spooling System

)

privind resursele deținute de procesele de intrare, utilizator și de ieșire?

Raspuns

Prin examinarea constrângerilor din problema 6.7 se pot trage următoarele concluzii:

1. Revenirea din procedură poate avea loc imediat deoarece se eliberează doar resurse.
2. Apelul de procedură poate golii discul ($p=max-reso$) și poate conduce la blocaj.
3. Consumarea ieșirii poate avea loc imediat după ce ieșirea devine disponibilă.
4. Producerea ieșirii poate fi întârziată temporar până ce a fost consumată ieșirea anterioară și sunt eliberate cel puțin reso pagini pentru viitoarea ieșire.

5. Consumarea intrării poate avea loc imediat după ce intrarea devine disponibilă.
6. Producerea intrării poate fi întârziată până când au fost consumate toate intrările anterioare și ieșirile corespunzătoare. În acest punct, când $i=0=0$, intrarea poate fi produsă și furnizată procesului utilizator și aceasta nu va goli discul deoarece ($p < \max - reso$).

Concluzia: cantitatea necontrolată de date de memorat asignată procesului utilizator este singura sursă posibilă de blocaj al memorării pe disc.

23. Considerați un sistem cu un total de 150 de unități de memorie, alocate la trei procese după cum urmează:

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Aplicați algoritmul bancherului pentru a determina în ce condiții ar fi sigură alocarea cererilor următoare. Dacă da, indicați o secvență de terminare care poate fi posibil garantată. Dacă nu arătați reducerea tabelului rezultat de alocare. Sosesc patru procese, cu un necesar maxim de memorie de 60 și un necesar inițial de 25 de unități.

- a. Crearea procesului va conduce la starea:

Process	Max	Hold	Claim	Free
1	70	45	25	25
2	60	40	20	
3	60	15	45	
4	60	25	35	

Există suficientă memorie liberă pentru a garanta terminarea fie a lui P1 sau a lui P2. După aceasta, cele trei job-uri care au rămas pot fi completate în orice ordine.

24. Considerați un sistem cu un total de 150 de unități de memorie, alocate la trei procese după cum urmează:

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Aplicați algoritmul bancherului pentru a determina în ce condiții ar fi sigură alocarea cererilor următoare. Dacă da, indicați o secvență de terminare care poate fi posibil garantată. Dacă nu arătați reducerea tabelului rezultat de alocare. Sosesc patru procese, cu un necesar maxim de memorie de 60 și un necesar inițial de 35 de unități.

Crearea procesului va conduce la o stare trivială nesigură.

Process	Max	Hold	Claim	Free
1	70	45	25	15
2	60	40	20	
3	60	15	45	
4	60	35	25	

25. Evaluati dacă algoritmul bancherului este util într-un SO.

Este nerealist: nu putem să în avans cererea maximă, numărul de procese se poate schimba în timp, numărul de resurse se poate schimba în timp (unele pot să cadă). Majoritatea OS ignorează blocajul. Doar Solaris permite super-utilizatorului să utilizeze slotul ultimului proces.

26. Un algoritm de tip pipeline este implementat astfel încât un sir de elemente de date de tip T produse de un proces P0 trece printr-o secvență de procese P1,P2,...,Pn-1, care operează în această ordine pe elementele sirului.

Definiți un buffer generalizat de mesaje care conține toate elementele de date parțial consumate și scrieți un algoritm pentru procesele Pi (0 ≤ i ≤ n-1) de forma

repeat

receive from predecessor;

consume element;

send to successor:

forever

Presupuneti că P0 receptionează elementele de intrare de p n-1. Algoritmul trebuie să valideze procesele pentru a opera direct cu mesajele memorate în buffer astfel încât copierea să nu fie necesară.

Bufferul este declarat ca fiind o arie de elemente partajate de tip T. O altă arie definește numărul de elemente de intrare disponibile pentru fiecare proces. Fiecare proces ține urma indexului j al elementelor din buffer la care se referă la un moment dat.

```
var buffer: array 0..max-1 of shared T;
available: shared array 0..n-1 of 0..max;
```

"Initialization"

var K: 1..n-1;

region available do

begin

available(0) := max;

for every k do available (k) := 0;

end

"Process i"

var j: 0..max-1; succ: 0..n-1;

begin

j := 0; succ := (i+1) mod n;

```

repeat
    region available do
        await available (i) > 0;
        region buffer(j) do consume element;
        region available do
            begin
                available (i) := available(i) – 1;
                available (succ) := available (succ) + 1;
            end
            j := (j+1) mod max;
        forever
    end

```

În programul anterior, regiunea de construcție definește regiunea critică utilizând mecanisme corespunzătoare pentru excluderea mutuală. Notația **region v do S** înseamnă că la un moment dat poate intra în secțiunea critică cel mult un proces asociat cu variabila **v** pentru a executa linia **S**.

27. Un algoritm de tip pipeline este implementat astfel încât un sir de elemente de date de tip T produse de un proces P0 trece printr-o secvență de procese P1,P2,...,Pn-1, care operează în această ordine pe elementele sirului.

Definiți un buffer generalizat de mesaje care conține toate elementele de date parțial consumate și scrieți un algoritm pentru procesele Pi (0 ≤ i ≤ n-1) de forma

```

repeat
    receive from predecessor;
    consume element;
    send to successor:
forever

```

Presupuneți că P0 recepționează elementele de intrare de pn-1. Algoritmul trebuie să valideze procesele pentru a opera direct cu mesajele memorate în buffer astfel încât copierea să nu fie necesară.

Arătați că procesele nu pot crea blocaj referitor la bufferul comun.

Un blocaj este situația în care:

P₀ waits for P_{n-1} AND

P₁ waits for P₀ AND

.....

P_{n-1} waits for P_{n-2}

deoarece

(available (0) = 0) AND

(available (1) = 0) AND

.....
(available (n-1) = 0)

Dar dacă $\max > 0$ această condiție nu poate fi menținută deoarece regiunea critică satisface următoarea invariантă:

$$\sum_{i=1}^N \text{claim}(i) < N \sum_{i=0}^{n-1} \text{available}(i) = \max$$

28. Presupunem că procesele foo și bar sunt executate în mod concurrent și partajează variabilele S și R (fiecare inițializată cu 1) și variabila întreg x (inițializată cu 0).

<pre>void foo() { do { semWait (S) ; semWait (R) ; x++ ; semSignal (S) ; SemSignal (R) ; } while (1) ; }</pre>	<pre>void bar() { do { semWait (R) ; semWait (S) ; x-- ; semSignal (S) ; SemSignal (R) ; } while (1) ; }</pre>
---	---

Poate execuția concurrentă a acestor două procese să determine blocarea unuia sau a ambelor procese să pentru totdeauna? Dacă da, dați o secvență de execuție în care unul sau ambele sunt blocați pentru totdeauna.

Da, dacă foo() execută semWait (S) și apoi bar execută semWait(R), ambele procese se vor bloca când vor executa următoarea instrucțiune. Deoarece fiecare va aștepta apoi apelul semSignal () de la celălalt, niciodată nu-și vor relua execuția.

29. Presupunem că procesele foo și bar sunt executate în mod concurrent și partajează variabilele S și R (fiecare inițializată cu 1) și variabila întreg x (inițializată cu 0).

<pre>void foo() { do { semWait (S) ; semWait (R) ; x++ ; semSignal (S) ; SemSignal (R) ; } while (1) ; }</pre>	<pre>void bar() { do { semWait (R) ; semWait (S) ; x-- ; semSignal (S) ; SemSignal (R) ; } while (1) ; }</pre>
---	---

Poate execuția concurrentă a acestor două procese să determine blocarea unuia sau a ambelor procese să pentru totdeauna? Dacă da, dați o secvență de execuție în care unul sau ambele sunt blocați pentru totdeauna.

Nu, Dacă oricare proces se blochează pe apelul semWait() atunci oricare alt alt proces se va bloca sau celălalt proces va execuția secțiunea sa critică. În ultimul caz, când procesul în execuție părăsește secțiunea sa critică, va execuția apelul la semSignal(), care va trezi procesul blocat.

30. Considerați un sistem constând din patru procese și o singură resursă. Stările curente ale matricelor de cerere și alocare sunt:

$$\mathbf{C} = \begin{pmatrix} 3 \\ 2 \\ 9 \\ 7 \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$

Care este numărul minim de unități de resurse necesare a fi disponibile pentru ca această stare să fie sigură?

Numărul de unități disponibile necesare pentru ca starea să fie sigură este 3, realizându-se un total în sistem de 10 unități. În starea prezentată în problemă, dacă este disponibilă o unitate adițională, P2 se poate execuția pentru completare, eliberând resursele sale. Acum sunt două unități disponibile. Aceasta va permite ca P1 să ruleze pentru completarea execuției și vor rezulta 3 unități disponibile. În acest punct însă p3 are nevoie de 6 unități și p4 de 5 unități. Dacă începem cu trei unități disponibile în loc de una, acum vor fi 5 unități disponibile și ceea ce permite completarea execuției lui P4 care eliberează 2 unități. Ca urmare numărul unităților disponibile va fi 7, ceea ce va permite și completarea execuției lui P3.

31. Comentați următoarea soluție a problemei cinei filozofilor. Un filozof flămând ia prima dată bețișorul din stânga; dacă bețișorul din dreapta este de asemenea disponibil, el ia acest bețișor din dreapta și începe să mănânce; altfel el pune din înapoi bețișorul din stânga și repetă ciclul.

Filozofii pot să se informeze în timp ce repetă preluarea și depunerea bețișorului din stânga la un unison perfect.

32. 6.18 Presupuneți că există două tipuri de filozofi. Un tip întotdeauna ia prima dată bețișorul din stânga (stângaciul) și celălalt tip întotdeauna ia bețișorul din dreapta (dreptaciul). Comportarea stângaciului este prezentată în figura 6.12. Comportarea dreptaciului este după cum urmează:

```
begin
repeat
    think;
    wait ( fork[ (i+1) mod 5] );
    wait ( fork[i] );
```

```

eat;
signal ( fork[i] );
signal ( fork[ (i+1) mod 5 ] );
forever
end;

```

Demonstrați că orice aranjament de așezare a stângacilor și dreptacilor cu cel puțin unul dintre ei elimină blocajul.

```

/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}

```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Presupuneți că masa este în blocaj, cum ar fi cazul în care există un set D diferit de nul de filozofi astfel încât fiecare P_i din D deține un bețișor și așteaptă pentru un bețișor deținut de vecin. Fără a pierde din generalitate, presupuneți că $P_j \in G$ este stângaci. Deoarece P_j prinde bețișorul său din stânga și nu poate avea bețișorul din dreapta, vecinul său din dreapta P_k care este tot un stângaci nu-și va completa niciodată cina. Ca urmare, $P_k \in D$. Continuând spre dreapta de-a lungul mesei cu aceleasi argumente rezultă că toți filozofii sunt stângaci. Aceasta contrazice existența a cel puțin unui dreptaci. Ca urmare, blocajul nu este posibil.

33. Presupuneți că există două tipuri de filozofi. Un tip întotdeauna ia prima dată bețișorul din stânga (stângaciul) și celălalt tip întotdeauna ia bețișorul din dreapta (dreptaciul). Comportarea stângaciului este prezentată în figura 6.12. Comportarea dreptaciului este după cum urmează:

```

begin
repeat
    think;
    wait ( fork[ (i+1) mod 5 ] );
    wait ( fork[i] );
    eat;

```

```

    signal ( fork[i] );
    signal ( fork[ (i+1) mod 5] );
    forever
end;

```

Demonstrați că orice aranjament de așezare a stângacilor și dreptacilor cu cel puțin unul dintre ei elimină înfometarea.

```

/* program  diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2),      philosopher (3),
              philosopher (4));
}

```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Presupuneți că stângaciul P_j se înfometează, adică există un şablon stabil al cinei în care P_j nu mănâncă niciodată. Presupuneți că P_j nu deține nici un bețișor. Atunci vecinul din dreapta P_i trebuie să dețină în mod continuu bețișorul său din dreapta, dar nu va termina niciodată de mâncat., Astfel P_i este un dreptaci care ține bețișorul său din dreapta, dar care nu are niciodată bețișorul din stânga pentru ași completa masa deci și P_i este înfometat. Acum vecinul din stânga a lui P_i trebuie să fie un dreptaci care deține continuu bețișorul său din dreapta. Continuând spre stânga de-a lungul mesei cu acest argument se arată că toți filozofii sunt înfometăți și dreptaci. Dar P_j este stângaci: o contradicție. Astfel P_j trebuie să dețină un bețișor. Deoarece P_j deține în mod continuu un bețișor li aşteaptă bețișorul din dreapta, vecinul din dreapta P_k nu va lăsa niciodată bețișorul său din stânga și nu-și va completa niciodată masa, și ca urmare P_k este un stângaci înfometat. Dacă P_k nu va continua să-și dețină bețișorul din stânga în mod continuu, P_j va putea mâncă; ca urmare P_k păstrează bețișorul său din stânga. Continuând cu argumentul spre dreapta de-a lungul mesei se arată că toți filozofii sunt înfometăți și stângaci: o contradicție. Înfometarea este astfel exclusă.

34. Figura 6.17 prezintă o altă soluție la problema cinei filozofilor utilizând monitoare. Comparați cu figura 6.14 și raportați concluziile voastre.

```
monitor dining_controller;
cond ForkReady[5]; /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork(left))
        cwait(ForkReady[left]); /* queue on condition variable */
    fork(left) = false;
    /*grant the right fork*/
    if (!fork(right))
        cwait(ForkReady[right]); /* queue on condition variable */
    fork(right) = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left]))/*no one is waiting for this fork */
        fork(left) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right]))/*no one is waiting for this fork */
        fork(right) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}

void philosopher[k=0 to 4] /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k); /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k); /* client releases forks via the monitor */
    }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

```

monitor dining_controller;
enum states {thinking, hungry, eating} state[5];
cond needFork[5]                                /* condition variable */

void get_forks(int pid)             /* pid is the philosopher id number */
{
    state[pid] = hungry;           /* announce that I'm hungry */
    if (state[(pid+1) % 5] == eating || (state[(pid-1) % 5] == eating)
        cwait(needFork[pid]);      /* wait if either neighbor is eating */
    state[pid] = eating;          /* proceed if neither neighbor is eating */
}

void release_forks(int pid)
{
    state[pid] = thinking;
    /* give right (higher) neighbor a chance to eat */
    if (state[(pid+1) % 5] == hungry) && (state[(pid+2) % 5]) != eating)
        csignal(needFork[pid+1]);
    /* give left (lower) neighbor a chance to eat */
    else if (state[(pid-1) % 5] == hungry) && (state[(pid-2) % 5]) != eating)
        csignal(needFork[pid-1]);
}

void philosopher[k=0 to 4]           /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k);           /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k);       /* client releases forks via the monitor */
    }
}

```

Figure 6.17 Another Solution to the Dining Philosophers Problem Using a Monitor

Soluția (6.14) așteaptă pe o ramificație disponibilă; cealaltă soluție (6.17) așteaptă pentru ca filozoful vecin să fie liber. Logica este în esență aceeași. Soluția din figura 6.17 este totuși mai compactă.

35. În tabelul 6.3, unele operații atomice din Linux nu implică două accese la variabile, cum ar fi `atomic_read(atomic_t *v)`. O operație simplă de citire este în mod clar atomică în orice arhitectură. Atunci, de ce această operație a fost adăugată la repertoriul operațiilor atomice?

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
ATOMIC_INIT (int i)	At declaration: initialize an atomic_t to i
int atomic_read(atomic_t *v)	Read integer value of v
void atomic_set(atomic_t *v, int i)	Set the value of v to integer i
void atomic_add(int i, atomic_t *v)	Add i to v
void atomic_sub(int i, atomic_t *v)	Subtract i from v
void atomic_inc(atomic_t *v)	Add 1 to v
void atomic_dec(atomic_t *v)	Subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Subtract i from v; return 1 if the result is zero; return 0 otherwise
int atomic_add_negative(int i, atomic_t *v)	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
int atomic_dec_and_test(atomic_t *v)	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
int atomic_inc_and_test(atomic_t *v)	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
void set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr
void clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr
void change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr
int test_and_set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr; return the old bit value
int test_bit(int nr, void *addr)	Return the value of bit nr in the bitmap pointed to by addr

Operațiile atomice operează pe tipurile de date atomice, care au propriul format intern. Ca urmare, nu se poate utiliza o simplă operație de citire, și ca urmare este nevoie de o operație specială de citire pentru tipurile de date atomice.

36. Considerați următorul fragment din codul sistem din Linux.

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

Unde mr_rwlock este un lacăt (lock) reader-writer. Care este efectul acestui cod?

Acest cod cauzează un blocaj, deoarece lacătul scriitorului se va partaja, așteptând că toți cititorii să elibereze lacătul, inclusiv acest fir de execuție.

37. Două variabile a și b au valorile inițiale 1 și respectiv 2. Următorul cod este pentru un sistem Linux :

Thread 1	Thread 2
a = 3;	—
mb();	—
b = 4;	c = b;
—	rmr();
—	d = a;

Ce erori posibile sunt evitate prin utilizarea barierelor de memorie?

Fără utilizarea barierelor de memorie, pe unele procesoare este posibil ca c să recepționeze o valoare new pentru b, în timp ce d recepționează vechea valoare a lui a. De exemplu, c poate fi 4 (ceea ce se așteaptă), totuși d poate fi egal cu 1 (ceea ce nu este de așteptat). Utilizând mb() se asigură că a și b sunt scrise în ordinea dorită, în timp ce rmb(), asigură că c și d sunt citite în ordinea dorită.

Cap. 7. GESTIUNEA MEMORIEI

1. Ce cerințe intenționează să satisfacă managementul memoriei?

Relocarea, protecția, partajarea, organizarea logică, organizarea fizică.

2. De ce este necesară capabilitatea de a reloca procesele dorite?

Tipic, nu este posibil ca un programator să știe în avans ce alte programe vor fi rezidente în memoria principală a calculatorului în momentul execuției programului său. În plus, ne-ar plăcea să fim capabili să comutăm procesele active în și în afara memoriei principale pentru a maximiza utilizarea procesorului prin furnizarea unei arii largi de procese gata de execuție. În ambele aceste cazuri, locația specifică a procesului în memoria principală este nepredictibilă.

3. De ce nu este posibil de a forța protecția memoriei pe durata compilării?

Deoarece locația unui program în memoria principală este nepredictibilă, este imposibil de a testa adresa absolută la momentul compilării pentru a asigura protecția. Mai mult, majoritatea limbajelor de programare permit calculul dinamic al adresei în momentul execuției, de exemplu calculând indicele unei arii sau un pointer la o structură de date. Ca urmare toate referințele la memorie generate de un proces trebuie verificate în timpul execuției pentru a se asigura faptul că ele sunt realizate în spațiul de memorie alocat aceluia proces.

4. Care sunt unele dintre motivele de a permite ca două sau mai multe procese să să aibă acces la o regiunea anume de memorie?

Dacă mai multe procese execută același program, este avantajos să se acceseze aceeași copie a programului mai degrabă decât să existe câte o copie separată pentru fiecare proces. De asemenea, procesele care cooperează pentru aceeași sarcină trebuie să partajeze accesul la aceeași structură de date.

5. Care sunt avantajele de a utiliza partiții de mărime inegală atunci când se folosește o schemă cu partiții fixe?

Prin utilizarea partiților fixe de lungime inegală avem: 1. Este posibil să se furnizeze una sau două partiții mari complete și totuși să avem un număr mare de partiții. Partițile mari pot permite încărcarea completă a unui program mare. 2. Fragmentarea internă este redusă deoarece un program mic poate fi plasat într-o partiție mică.

6. Care este diferența între fragmentarea internă și fragmentarea externă?

Fragmentarea internă se referă la spațiul intern al unei partiții care se pierde datorită faptului că blocul de date încărcat este mai mic decât partiția. Fragmentarea externă este un fenomen asociat cu partiționarea dinamică, și se referă la faptul că un număr mare de arii mici din memoria principală care se acumulează în zona externă a partiților.

7. Care sunt distincțiile între adresarea logică, relativă și fizică?

- a. O adresă logică este o referință la o locație de memorie independentă de asignarea curentă a datei la memorie; trebuie realizată o translație la adresa fizică înainte de a realiza accesul la memorie. O adresă relativă este un exemplu particular de adresă logică, cu care se exprimă o adresă ca o locație relativă la un punct cunoscut, uzual începutul unui program. O adresă fizică, sau absolută, este o locație actuală din memoria principală

8. Care este diferența între pagină și cadru?

Într-un sistem cu paginare, programele și datele memorate pe disc sunt divizate în blocuri egale de lungime fixă numite pagini, și memoria principală este divizată în blocuri de aceeași mărime numite cadre.

9. Care este diferența între pagină și segment?

Un mod alternativ în care un program utilizator poate fi subdivizat este segmentarea. În acest caz, programul și datele sale asociate sunt divizate într-un anumit număr de segmente. Nu este necesar ca toate segmentele să aibă aceeași lungime, dar există o lungime maximă pentru segment.

Problems

10. În secțiunea 2.3, s-au enumerate cinci obiective ale managementului memoriei, și în secțiunea 7.1, s-au enumerate cinci cerințe. Argumentați că fiecare listă înglobează toate problemele adresate în cealaltă.

Raspuns:

Relocarea	≈	suport pentru programarea modulară
Protectia	≈	izolarea proceselor; controlul accesului și al protecției
Partajarea	≈	controlul accesului și al protecției
Organizarea		
logică	≈	suport pentru programarea modulară
Organizarea		
Fizică	≈	memorare pe termen lung; alocare și management automat

11. Considerați o schemă cu partitōnare fixă cu partiții egale având 216 octeți și o memorie totală de 224 octeți. Se gestionează un tabel cu procese care include un pointer la o partiție pentru fiecare proces rezident. Căți biți sunt necesari pentru pointer?

8 biți.

12. Considerați o schemă de partitōnare dinamică. Arătați că, în medie, memoria conține jumătate goluri, jumătate segmente.

Se notează cu s și h numărul mediu de segmente și goluri. Probabilitatea ca un anumit segment să fie urmat de un gol în memorie (și nu de un alt segment) este 0,5, deoarece

ștergerea și crearea sunt probabil egale la echilibru, astfel cu s segmente în memorie, numărul mediu de goluri este $s/2$. Intuitiv, este rezonabil să afirmăm că numărul de goluri este mai mic decât numărul de segmente, deoarece segmentele vecine pot fi combinate la ștergere într-un singur gol.

13. Pentru a implementa diferiți algoritmi de plasament discutați pentru partităionarea dinamică (Section 7.2), trebuie păstrată o listă de blocuri libere de memorie. Care este lungimea medie a căutării pentru fiecare dintre cele trei metode discutate (best-fit, first-fit, next-fit)?

$N, \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \dots + \frac{N}{2^N} + \frac{N}{2^N}$ (valoarea este cuprinsă între 1 și 2), la fel ca la First-fit cu deosebirea că începe căutarea unde se termină căutarea la first-fit.

14. Un alt algoritm pentru partităionarea dinamică este denumit potrivirea cea mai defavorabilă (worst-fit). În acest caz, cel mai mare bloc liber este utilizat pentru a aduce un proces în memorie. Discutați avantajele și dezavantajele acestei metode în comparație cu prima- (first-), următoarea- (next-) și cea mai bună (best-) potrivire (-fit).

O critică a algoritmului best-fit este aceea că spațiul rămas după alocarea unui bloc de mărimea cerută este prea mic și în general nu mai este de folos real. Algoritmul worst-fit maximizează șansa ca spațiul liber rămas să fie suficient de mare pentru a satisface altă cerere, minimizând astfel frecvența compactărilor. Dezavantajul acestei metode este acela că sunt alocate prima dată blocurile mari; ca urmare o cerere târzie pentru un bloc mare este foarte probabil să eșueze.

15. Un alt algoritm pentru partităionarea dinamică este denumit potrivirea cea mai defavorabilă (worst-fit). În acest caz, cel mai mare bloc liber este utilizat pentru a aduce un proces în memorie. Care este lungimea medie pentru potrivirea cea mai defavorabilă (worst-fit)?

La fel ca la best-fit.

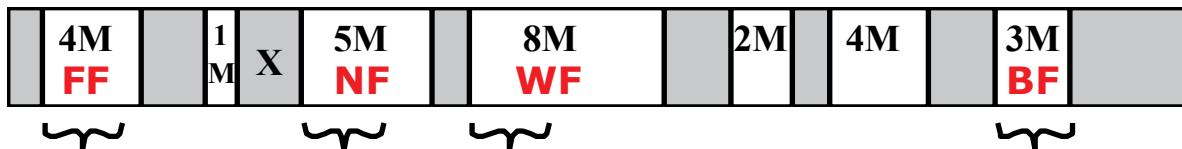
16. Diagrama din figură prezintă un exemplu de configurare a memoriei utilizând partităionarea dinamică, după ce au fost deja realizate operații de plasare și de comutare (swapping-out). Adresele evoluează de la stânga la dreapta; ariile gri indică blocuri ocupate de procese; ariile albe indică blocuri libere de memorie. Ultimul proces plasat este de 2 Mocteți și este marcat cu X. Numai un singur proces a fost scos afară (swapping-out).



Care a fost mărimea maximă a unui proces scos-comutat din memorie (swapping-out)?

Raspuns: 1M

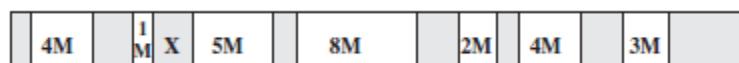
17. Diagrama din figură prezintă un exemplu de configurare a memoriei utilizând partitōnarea dinamică, după ce au fost deja realizate operații de plasare și de comutare (swapping-out). Adresele evoluează de la stāngă la dreapta; ariile gri indică blocuri ocupate de procese; ariile albe indică blocuri libere de memorie. Ultimul proces plasat este de 2 Moctēti și este marcat cu X. Numai un singur proces a fost scos afară (swapping-out).



Care a fost mārimea unui bloc liber de memorie imediat înainte de a fi partitōnat de X?

Raspuns : 7M

18. Diagrama din figură prezintă un exemplu de configurare a memoriei utilizând partitōnarea dinamică, după ce au fost deja realizate operații de plasare și de comutare (swapping-out). Adresele evoluează de la stāngă la dreapta; ariile gri indică blocuri ocupate de procese; ariile albe indică blocuri libere de memorie. Ultimul proces plasat este de 2 Moctēti și este marcat cu X. Numai un singur proces a fost scos afară (swapping-out).



Trebuie satisfăcută o cerere nouă de 3 Moctēti. Indicați intervalele de memorie unde se va crea partitōia pentru noul proces pentru patru algoritmi de plasare: best-fit, first-fit, next-fit, worst-fit. Pentru fiecare algoritm desenați un segment orizontal sub banda de memorie (vezi figura) și etichetați-l în mod clar.

a.



19. Un bloc de memorie de 1-Moctet este alocat utilizând buddy system. Prezentați rezultatul următoarei secvențe cu o figură similară celei din figura 7.6; Request 70; Request 35; Request 80; Return A; Request 60; Return B; Return D; Return C.

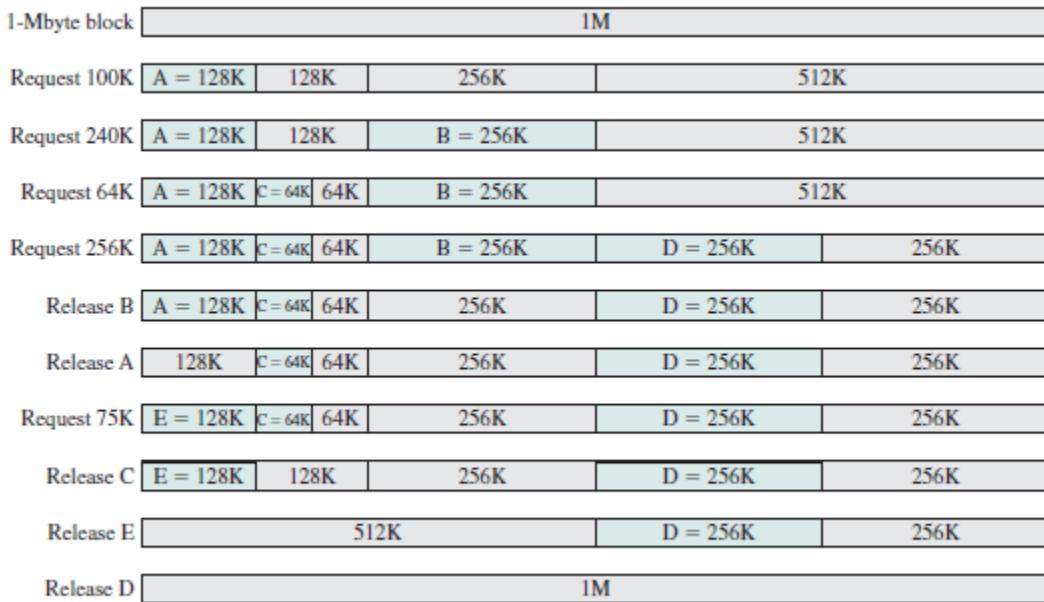


Figure 7.6 Example of Buddy System

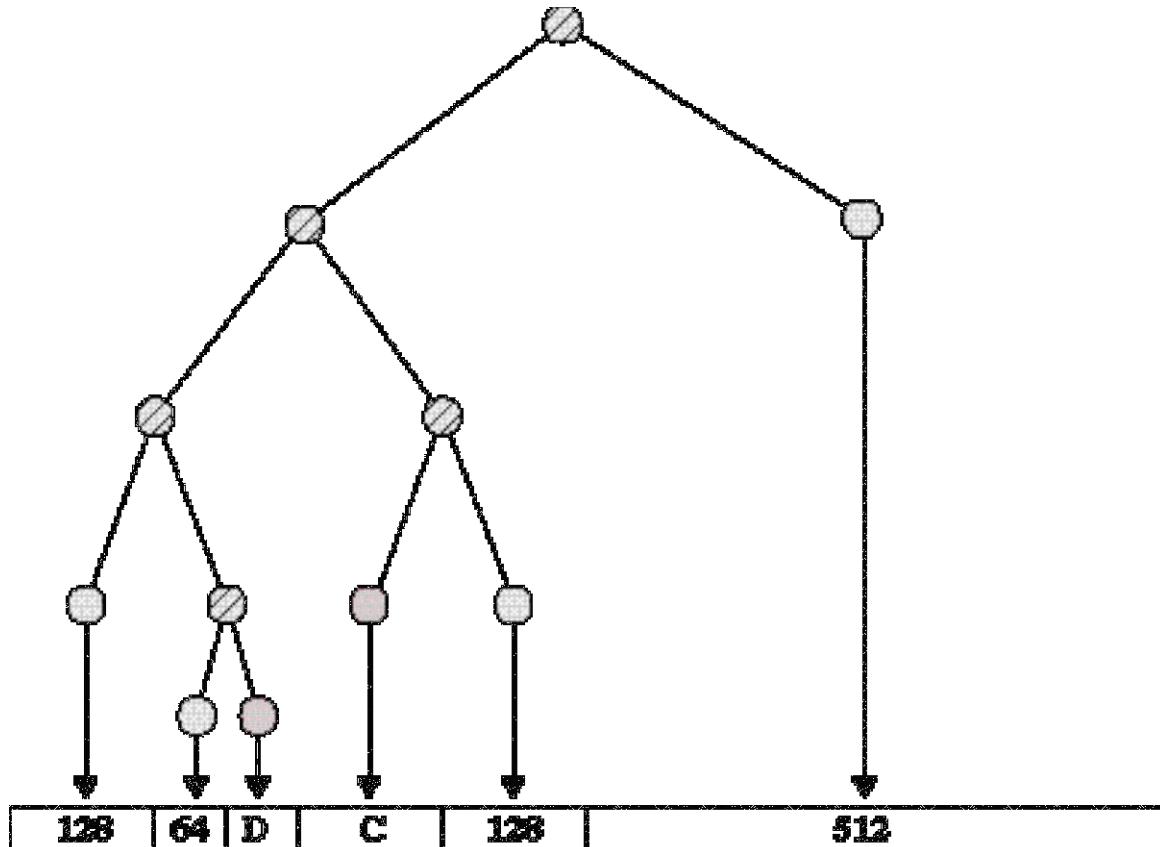
a.

Request 70	A	128	256		512
Request 35	A	B	64	256	512
Request 80	A	B	64	C	128
Return A	128	B	64	C	128
Request 60	128	B	D	C	128
Return B	128	64	D	C	128
Return D	256		C	128	512
Return C	1024				

20. Un bloc de memorie de 1-Moctet este alocat utilizând buddy system. Prezentați rezultatul următoarei secvențe cu o figură similară celei din figura 7.6; Request 70;

Request 35; Request 80; Return A; Request 60; Return B; Return D; Return C.
Prezentați reprezentarea arborelui binar ce urmează după Return B.

Raspuns:



21. Considerați un buddy system în care un anumit bloc sub alocarea curentă are o adresă având valoarea 011011110000. Dacă blocul are mărimea 4, care este adresa binară pentru buddy-ul său?

011011110100

22. Considerați un buddy system în care un anumit bloc sub alocarea curentă are o adresă având valoarea 011011110000. Dacă blocul are mărimea 16, care este adresa binară pentru buddy-ul său?

011011100000

23. 7.9 Fie adresa buddy k (x) pentru buddy-ul unui bloc de mărime 2^k a cărui adresă este x . Scrieți o expresie generală pentru buddy k (x).

Raspuns:

$$\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{if } x \bmod 2^{k+1} = 2^k \end{cases}$$

24. Secvența Fibonacci este definită după cum urmează : $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$, $n \geq 0$. Poate fi folosită această secvență pentru a stabili un buddy system?
Da, mărimea blocului poare satisface $F_n = F_{n-1} + F_{n-2}$.

25. Secvența Fibonacci este definită după cum urmează : $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$, $n \geq 0$. Care va avantajul acestui sistem față de binary buddy system descris în acest capitol?

Această schemă oferă mai multe mărimi de blocuri în raport cu un sistem buddy binar, și ca urmare are un potențial mai mare pentru a reduce fragmentarea internă, dar poate cauza fragmentare externă suplimentară, deoarece sunt create multe blocuri mici fără utilitate.

26. În timpul execuției unui program, procesorul va incrementa conținutul registrului program counter cu un cuvânt după extragerea fiecărei instrucțiuni, dar va altera conținutul acestui registru dacă se întâlnește o instrucțiune de ramificație sau de apel de procedură care cauzează execuția altundeva în program. Considerați acum figura 7.8.

Există două alternative cu privire la adresele instrucțiunilor:

- Menține o adresă relativă în registrul pentru instrucțiuni și realizarea unei translații dinamice de adresă utilizând registrul de adresă ca intrare. Atunci când se întâlnește o ramificație sau un apel de procedură corecte, adresa relativă generată de aceste instrucțiuni este încărcată în registrul instrucțiunilor (program counter)
- Menține o adresă absolută în registrul instrucțiunilor. Când apare o ramificație sau un apel de procedură care se execută cu succes, se activează translatarea dinamică a adresei, cu rezultatul memorat în registrul pentru instrucțiuni.
Care este abordarea preferabilă?

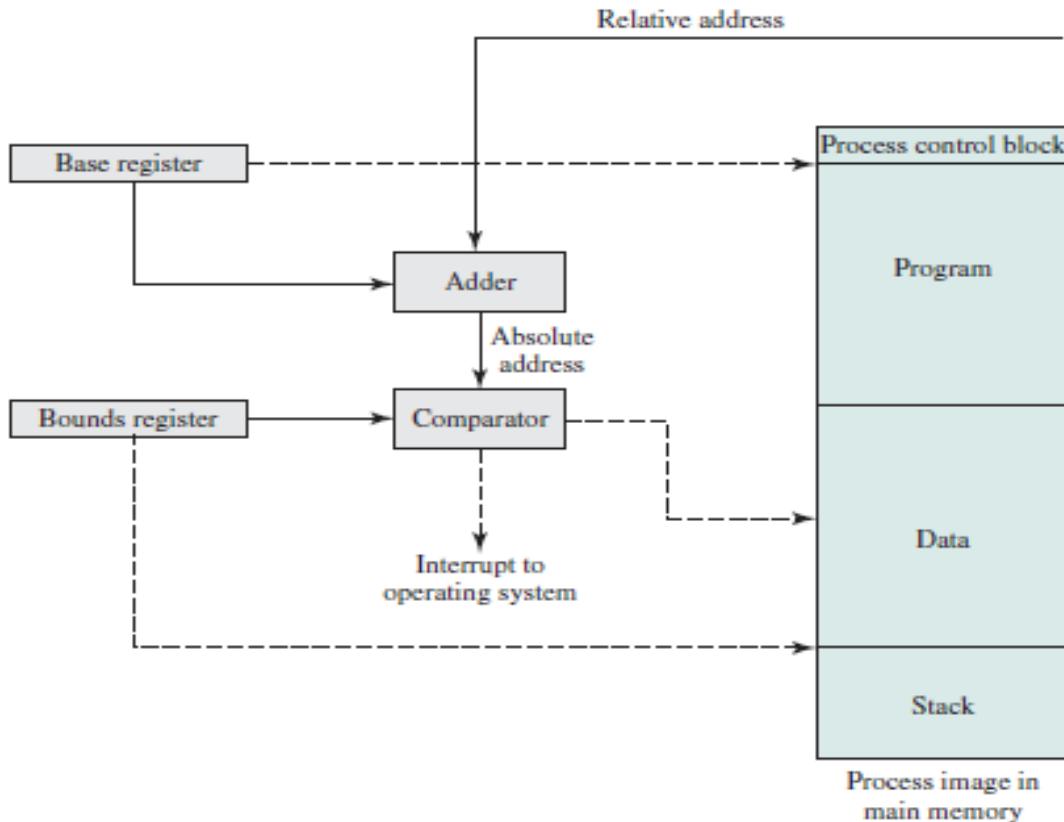


Figure 7.8 Hardware Support for Relocation

Utilizarea adresării absolute reduce numărul de ori când trebuie realizată translatarea dinamică a adresei. Totuși, dorim ca programul să fie relocabil. Ca urmare, este de preferat să utilizăm adresarea relativă la registrul pentru instrucțiuni. Alternativ, adresa din registrul de instrucțiuni poate fi convertită la una relativă atunci când procesul este scos din memorie (swapped-out).

27. Considerați un sistem simplu de paginare cu următorii parametrii: 2^{32} bytes de memorie fizică; mărimea unei pagini de 2^{10} bytes; 2^{16} pagini din spațiul logic de adrese. Câți biți are adresa logică?

26 biți,

28. Considerați un sistem simplu de paginare cu următorii parametrii: 2^{32} bytes de memorie fizică; mărimea unei pagini de 2^{10} bytes; 2^{16} pagini din spațiul logic de adrese. Câți octeți sunt într-un cadru?

2^{10} biți,

29. Considerați un sistem simplu de paginare cu următorii parametrii: 2^{32} bytes de memorie fizică; mărimea unei pagini de 2^{10} bytes; 2^{16} pagini din spațiul logic de adrese. Câți biți din adresa fizică specifică un cadru?

22 biți

30. Considerați un sistem simplu de paginare cu următorii parametrii: 2^{32} bytes de memorie fizică; mărimea unei pagini de 2^{10} bytes; 2^{16} pagini din spațiul logic de adrese. Câte intrări sunt în tabela cu pagini?

2^{16} intrări,

31. Considerați un sistem simplu de paginare cu următorii parametrii: 2^{32} bytes de memorie fizică; mărimea unei pagini de 2^{10} bytes; 2^{16} pagini din spațiul logic de adrese. Câți biți are fiecare intrare în tabelele cu pagini? Presupuneți că fiecare intrare în tabelă conține un bit de tipul valid/ invalid.

23 biți,

32. Scrieți translatarea adresei binare logice 0001010010111010 ținând cont de următoarele scheme ipotetice privind gestiunea memoriei, și explicați răspunsul vostru: Un sistem de paginare cu 256 – adrese de pagină, ce utilizează o tabelă pentru pagini în care cadrele sunt de patru ori mai mici ca numărul de pagină.

a. Numărul paginii este în cei 8 biți mai semnificativi: 00010100. Îl preluăm din adresă și îl înlocuim cu numărul de cadru, care este de 4 ori mai mic, adică îl deplasăm cu doi biți la dreapta: 00000101. Ca urmare, rezultatul este acest număr de cadru concatenat cu offsetul 10111010.

Numărul de segment este de dat de cei 6 biți mai semnificativi: 000101. Îl preluăm din adresă și adunăm offsetul rămas 0010111010 la segmentul de bază. Baza este $2^2 = 10110$ adunat de un număr de 4096 ori, ceea ce înseamnă deplasarea la stânga cu 12 biți: $10110 + 0101000000000000 = 010100000010110$. Acum adunând cele două numere subliniate obținem: adresa fizică binară = 0101000011010000

33. Scrieți translatarea adresei binare logice 0001010010111010 ținând cont de următoarele scheme ipotetice privind gestiunea memoriei, și explicați răspunsul vostru: Un sistem de segmentare cu 1K-adrese de segment, utilizând un tabel pentru segmente în care baza este plasată la adresa reală de segment $2^2 + 4,096 \times$ segment #.

Numărul de segment este de dat de cei 6 biți mai semnificativi: 000101. Îl preluăm din adresă și adunăm offsetul rămas 0010111010 la segmentul de bază. Baza este $2^2 = 10110$ adunat de un număr de 4096 ori, ceea ce înseamnă deplasarea la stânga cu 12 biți: $10110 + 0101000000000000 = 010100000010110$. Acum adunând cele două numere subliniate obținem: adresa fizică binară = 0101000011010000

34. Considerați un sistem cu segmentare simplă care are următorul tabel:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

Pentru fiecare din următoarele adrese logice, determinați adresa fizică sau indicați dacă apare o eroare de segment: 0, 198.

858

35. Considerați un sistem cu segmentare simplă care are următorul tabel:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

Pentru fiecare din următoarele adrese logice, determinați adresa fizică sau indicați dacă apare o eroare de segment: 2, 156.

378

36. Considerați un sistem cu segmentare simplă care are următorul tabel:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

Pentru fiecare din următoarele adrese logice, determinați adresa fizică sau indicați dacă apare o eroare de segment: 1, 530.

Segmentul 1 are o lungime de 422, ca urmare adresa determină apariția unei erori de segment.

37. Considerați un sistem cu segmentare simplă care are următorul tabel:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

Pentru fiecare din următoarele adrese logice, determinați adresa fizică sau indicați dacă apare o eroare de segment: 3, 444. Raspuns: 1440

38. Considerați un sistem cu segmentare simplă care are următorul tabel:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

Pentru fiecare din următoarele adrese logice, determinați adresa fizică sau indicați dacă apare o eroare de segment: 0, 222.

882

39. Considerați o memorie în care segmentele continue S_1, S_2, \dots, S_n sunt plasate în ordinea creării lor de la un capăt al memoriei la celălalt capăt aşa cum se sugerează în figura următoare:



Când se creează segmentul S_{n+1} acesta este plasat imediat după segmentul S_n chiar dacă unele dintre segmentele S_1, S_2, \dots, S_n pot fi deja șterse. Atunci când granița dintre segmente (în utilizare sau șterse) și goliurile ating capătul celălalt al memoriei, segmentele în utilizare sunt compactate. Arătați că fracția F de timp petrecută pentru compactare respectă următoarea inegalitate:

$$F \geq \frac{1-f}{1+kf} \quad \text{where} \quad k = \frac{t}{2s} - 1$$

where

s = average length of a segment, in words

t = average lifetime of a segment, in memory references

f = fraction of the memory that is unused under equilibrium conditions

Sugestie: Găsiți viteza medie la care marginea trece de memorie și presupuneți că pentru a copia un singur cuvânt sunt necesare cel puțin două referințe la memorie

$$F = 1 - t_0/(t_0 + t_c),$$

40. Considerați o memorie în care segmentele continue S_1, S_2, \dots, S_n sunt plasate în ordinea creării lor de la un capăt al memoriei la celălalt capăt aşa cum se sugerează în figura următoare:



Când se creează segmentul S_{n+1} acesta este plasat imediat după segmentul S_n chiar dacă unele dintre segmentele S_1, S_2, \dots, S_n pot fi deja șterse. Atunci când granița dintre segmente (în utilizare sau șterse) și golurile ating capătul celălalt al memoriei, segmentele în utilizare sunt compactate. Fracția F de timp petrecută pentru compactare respectă următoarea inegalitate:

$$F \geq \frac{1-f}{1+kf} \quad \text{where} \quad k = \frac{t}{2s} - 1$$

where

s = average length of a segment, in words

t = average lifetime of a segment, in memory references

f = fraction of the memory that is unused under equilibrium conditions

Sugestie: Găsiți viteza medie la care marginea trece de memorie și presupuneți că pentru a copia un singur cuvânt sunt necesare cel puțin două referințe la memorie

Găsiți F pentru $f = 0.2$, $t = 1,000$, and $s = 50$.

$$k = (t/2s) - 1 = 9; \quad F \geq (1 - 0.2)/(1 + 1.8) = 0.29$$

Cap. 7. Memoria virtuală

1. Care este diferența între paginarea simplă și paginarea memoriei virtuale?

Paginarea simplă: toate paginile unui proces trebuie să fie în memoria principală pentru ca procesul să se execute, exceptie fiind cazul utilizării suprapunerilor (overlays). Paginarea memoriei virtuale: pentru ca procesul să se execute, nu trebuie menținute toate paginile unui proces în cadrele memoriei principale; paginile trebuie citite la nevoie.

2. Explicați ce este thrashing.

Thrashing este un fenomen care apare la memoria virtuală, conform căruia procesorul petrece majoritatea timpului comutând pagini în loc să execute instrucțiuni utile ale programelor.

3. De ce este principiul localității crucial pentru utilizarea memoriei principale?

Algoritmii pot fi proiectați pentru a exploata principiul localității pentru a evita thrashing-ul. În general, principiul localității permite unui algoritm să prezică care pagini rezidente vor fi referite în viitorul apropiat și ca urmare devin candidate potențiale pentru a fi înlocuite.

4. Ce elemente se găsesc în mod tipic într-o intrare dintr-o tabelă cu pagini?

Descrieți pe scurt fiecare element.

Numărul de cadru: un număr secvențial care identifică o pagină în memoria principală; bitul de prezență: indică dacă pagina curentă este în memoria principală; bitul de modificare: indică dacă pagina a fost modificată din momentul aducerii ei în memorie.

5. Care este scopul bufferului de translatare anticipată (translation lookaside buffer)?

TLB-ul este un cache care conține acele intrări în tabelele cu pagini care au fost cel mai recent utilizate. Scopul său este de a evita, majoritatea timpului, accesul la disc pentru a aduce o intrare din tabela cu pagini.

6. Descrieți succint politicile alternative de extragere a unei pagini.

Cu paginarea la cerere o pagină este adusă în memoria principală numai atunci când se realizează o referință la o locație din acea pagină. Cu prepaginarea sunt aduse pagini, altele decât cele cerute de o eroare de pagină.

7. Care este diferența între politicile de gestiune cu set rezident (resident set management) și de înlocuire a paginilor (page replacement policy)?

Gestiunea setului rezident se ocupă cu două probleme: (1) câte cadre sunt alocate fiecărui proces activ; și (2) dacă setul de pagini care trebuie luat în considerare pentru înlocuire trebuie să fie limitat la acele din proces care cauzează apariția de erori de pagină sau trebuie aduse toate paginile în memoria principală. Politica de înlocuire a paginilor se ocupă de următoarea problemă: dintr-un set de pagini care se iau în considerare, care pagină anume va fi selectată pentru înlocuire.

8. Care este relația între algoritmii FIFO și cel de înlocuire în sensul acelor de ceasornic (clock page replacement algorithms)?

Politica clock este similară cu FIFO, cu excepția că în politica clock, orice cadru cu bitul de utilizare pe 1 este amânat de către algoritm de la înlocuire.

9. Ce se înțelege prin bufferarea paginilor?

(1) Dacă o pagină este înlăturată din setul rezident dar este nevoie în curând de ea, aceasta este încă în memoria principală, salvându-se citirile de pe disc. (2) Paginile modificate pot fi actualizate sub formă de clustere în loc de a fi salvate una la un moment dat, reducându-se semnificativ numărul operațiilor I/O și ca urmare cantitatea de timp pentru accesul la disc.

10. De ce nu este posibilă combinarea unei politici globale de înlocuire cu o politică ce utilizează alocarea fixă?

Deoarece o politică de alocare fixă necesită ca numărul de cadre alocate unui proces să fie fix, atunci când vine momentul să se aducă o nouă pagină pentru proces, una din paginile rezidente pentru acel proces va trebui comutată înapoi pe disc (swapped out), pentru a ține numărul de cadre alocate la aceeași valoare, ceea ce reprezintă o politică locală de înlocuire.

11. Care este diferența între un set rezident și un set de lucru (working set)?

Setul rezident al unui proces îl reprezintă numărul curent de pagini ale procesului aflate în memoria principală. Setul de lucru al unui proces este numărul de pagini ale unui proces care au fost accesate recent.

12. Care este diferența între ștergerea la cerere și pre-ștergerea (precleaning)?

Cu ștergerea la cerere, o pagină este scrisă în memoria secundară numai atunci când a fost selectată pentru înlocuire. Politica de tip precleaning scrie paginile modificate înainte ca și cadrele lor să fie necesare astfel încât paginile pot fi scrise pe loturi.

Problems

13. Presupuneți că o tabelă cu pagini pentru procesul curent în execuție arată ca în figura următoare. Toate numerele sunt zecimale, totul fiind numerotat începând de la zero, și toate adresele sunt adrese de memorie de la nivelul octet. Mărimea paginii este de 1,024 octeți.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

Descrieți exact, cum, în general, o adresă virtuală generată de UCP este translatată într-o adresă fizică de memorie principală.

Împarte adresa binară în numărul virtual de pagină și offset: utilizează VPN ca index în tabela cu pagini; extrage numărul cadrului de pagină; concatenează cu offsetul și preia adresa fizică de memorie.

14. Presupuneți că o tabelă cu pagini pentru procesul curent în execuție arată ca în figura următoare. Toate numerele sunt zecimale, totul fiind numerotată începând de la zero, și toate adresele sunt adrese de memorie de la nivelul octet. Mărimea paginii este de 1,024 octeți.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

Ce adresă fizică, dacă există vreo una, va avea fiecare următoarele adrese virtuale corespunzătoare? (Nu încercați să gestionați erorile de pagină dacă există vreo una.)

- i. 1,052
- ii. 2,221
- iii. 5,499

7196 (ii) 2221 – eroare de pagină (iii) 5499

15. Considerați următorul program.

```
#define Size 64
int A[Size; Size], B[Size; Size], C[Size; Size];
int register i, j;
for (j = 0; j < Size; j++)
    for (i = 0; i < Size; i++)
        C[i; j] = A[i; j] + B[i; j];
```

Presupuneți că programul este executat pe un sistem utilizând paginarea la cerere și mărimea paginii este de 1 Kocet. Fiecare întreg are lungimea de 4 octeți. Este clar că fiecare arie necesită un spațiu de 16 pagini. Ca un exemplu, A[0, 0]-A[0, 63], A[1, 0]-A[1, 63], A[2, 0]-A[2, 63], și A[3, 0]-A[3, 63] vor fi memorate în prima pagină de date. Un şablon similar poate fi derivat pentru restul ariei A și pentru ariile B și C. Presupuneți că sistemul alocă pentru acest proces un set de lucru de 4 pagini. Una dintre pagini va fi utilizată de program și

trei pagini pot fi utilizate pentru date. De asemenea, doi registri de index se vor asigna pentru i și j (astfel, nu va fi nevoie de acces la memorie pentru referirea acestor variabile). Discutați cât de frecvent vor apărea erorile de pagină (în funcție de numărul de ori cât se execută $C[i, j] = A[i, j] + B[i, j]$).

3 erori de pagină la fiecare 4 execuții ale $C[i, j] = A[i, j] + B[i, j]$.

16. Considerați următorul program.

```
#define Size 64
int A[Size; Size], B[Size; Size], C[Size; Size];
int register i, j;
for (j = 0; j < Size; j++)
for (i = 0; i < Size; i++)
C[i; j] = A[i; j] + B[i; j];
```

Presupuneți că programul este executat pe un sistem utilizând paginarea la cerere și mărimea paginii este de 1 Kocet. Fiecare întreg are lungimea de 4 octeți. Este clar că fiecare arie necesită un spațiu de 16 pagini. Ca un exemplu, A[0, 0]-A[0, 63], A[1, 0]-A[1, 63], A[2, 0]-A[2, 63], și A[3, 0]-A[3, 63] vor fi memorate în prima pagină de date. Un şablon similar poate fi derivat pentru restul ariei A și pentru ariile B și C. Presupuneți că sistemul alocă pentru acest proces un set de lucru de 4 pagini. Una dintre pagini va fi utilizată de program și trei pagini pot fi utilizate pentru date. De asemenea, doi registri de index se vor asigna pentru i și j (astfel, nu va fi nevoie de acces la memorie pentru referirea acestor variabile). Puteți modifica programul pentru a minimiza frecvența erorilor de pagină?

Da, frecvența erorilor de pagină poate fi minimizată prin comutarea buclelor internă și externă.

17. Considerați următorul program.

```
#define Size 64
int A[Size; Size], B[Size; Size], C[Size; Size];
int register i, j;
for (j = 0; j < Size; j++)
for (i = 0; i < Size; i++)
C[i; j] = A[i; j] + B[i; j];
```

Presupuneți că programul este executat pe un sistem utilizând paginarea la cerere și mărimea paginii este de 1 Kocet. Fiecare întreg are lungimea de 4 octeți. Este clar că fiecare arie necesită un spațiu de 16 pagini. Ca un exemplu, A[0, 0]-A[0, 63], A[1, 0]-A[1, 63], A[2, 0]-A[2, 63], și A[3, 0]-A[3, 63] vor fi memorate în prima pagină de date. Un şablon similar poate fi derivat pentru restul ariei A și pentru ariile B și C. Presupuneți că sistemul alocă pentru acest proces un set de lucru de 4 pagini. Una dintre pagini va fi utilizată de program și trei pagini pot fi utilizate pentru date. De asemenea, doi registri de index se vor asigna pentru i și j (astfel, nu va fi nevoie de acces la memorie pentru referirea

acestor variabile). Care va fi frecvența erorilor de pagină după modificarea pe care ați făcut-o?

După modificare, vor fi 3 erori de pagină la fiecare 256 de execuții.

18. Cât spațiu este necesar pentru tabela cu pagini utilizator din figura 8.4?

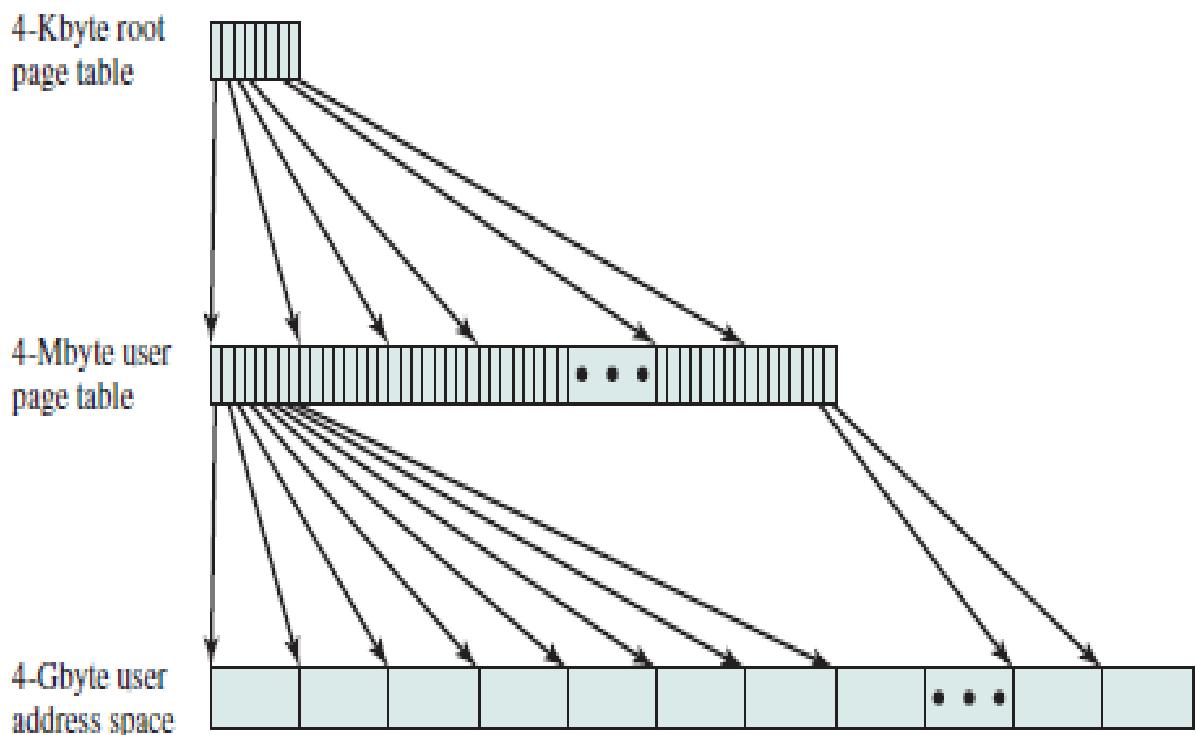


Figure 8.4 A Two-Level Hierarchical Page Table

4Mocteți

19. Presupuneți că doriți să implementați un table cu pagini inversate hashed pentru aceeași schemă de adresare prezentată în figura 8.4., utilizând o funcție hash care mapează numărul de pagină pe 20 de biți într-o valoare hash pe 6 biți. Intrările în tabelă conțin numărul de pagină, numărul de cadru, și un pointer de înlățuire. Dacă tabele cu pagini alocă spațiu pentru până la trei intrări cu depășire pe o intrare hash, cât spațiu de memorie va lua tabela de pagini inversată hashed de memorie?

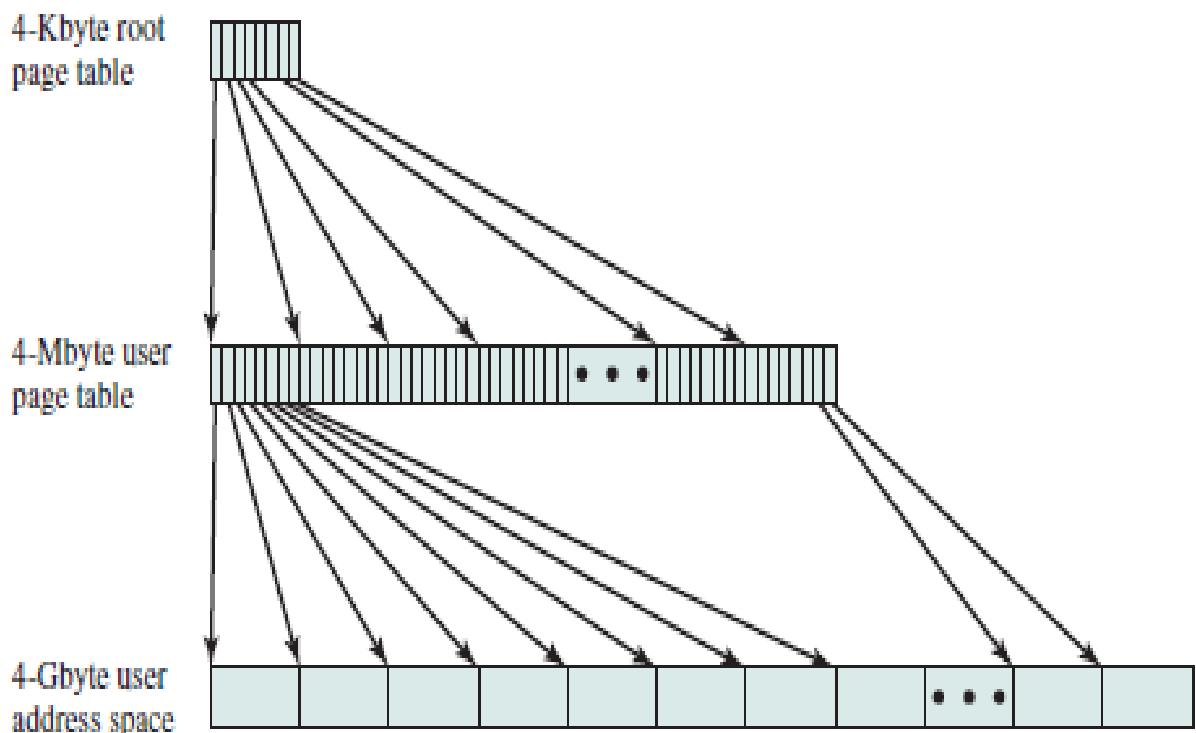


Figure 8.4 A Two-Level Hierarchical Page Table

768 octeți

20. Considerați următorul sir de referințe la pagini 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Completăți o figură asemănătoare cu figura 8.5, care să arate alocarea cadrelor pentru: FIFO (first-in-first-out).

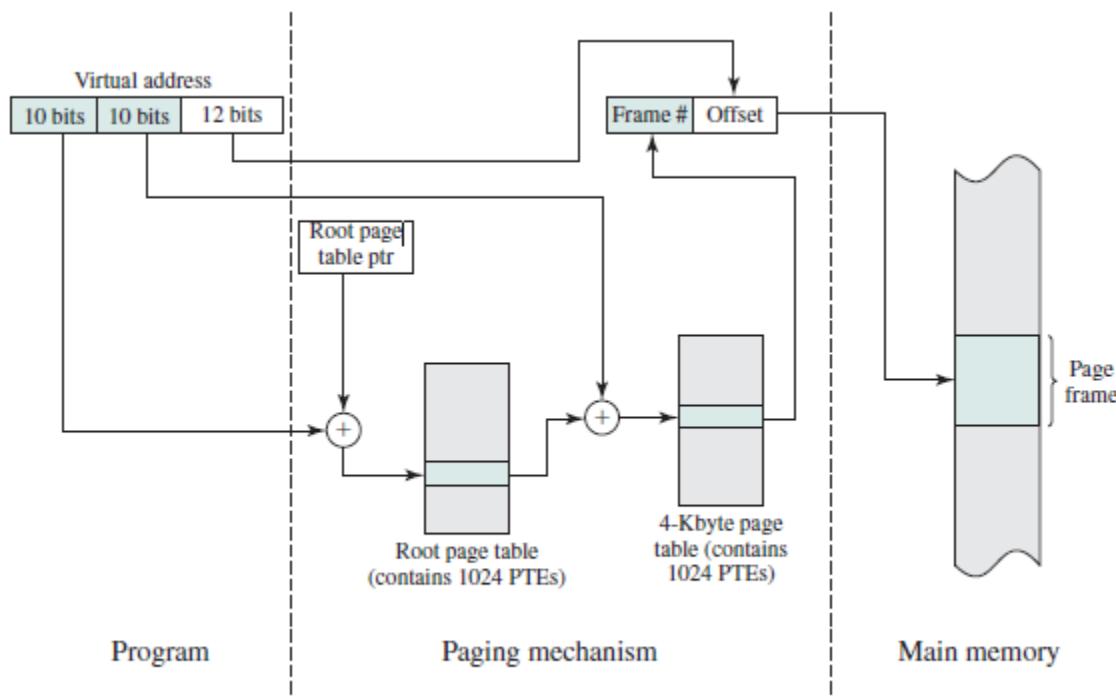


Figure 8.5 Address Translation in a Two-Level Paging System

FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2
		1	1	1	1	0	0	0	3	3	3	3
F			F			F			F			

21. Considerați următorul sir de referințe la pagini 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Completați o figură asemănătoare cu figura 8.5, care să arate alocarea cadrelor pentru: LRU (least recently used).

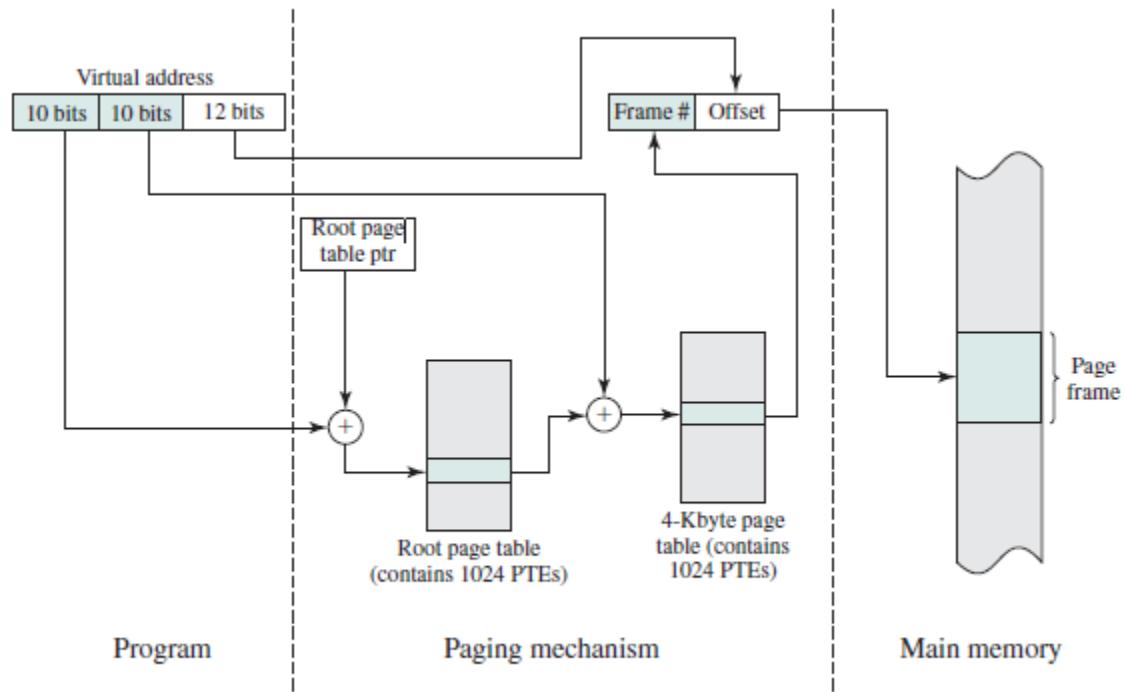


Figure 8.5 Address Translation in a Two-Level Paging System

LRU

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
0	0	0	0	3	3	3	2	2	2	2	2	2

F F F F F F F F F F

22. Considerați următorul sir de referințe la pagini 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Completați o figură asemănătoare cu figura 8.5, care să arate alocarea cadrelor pentru: Clock

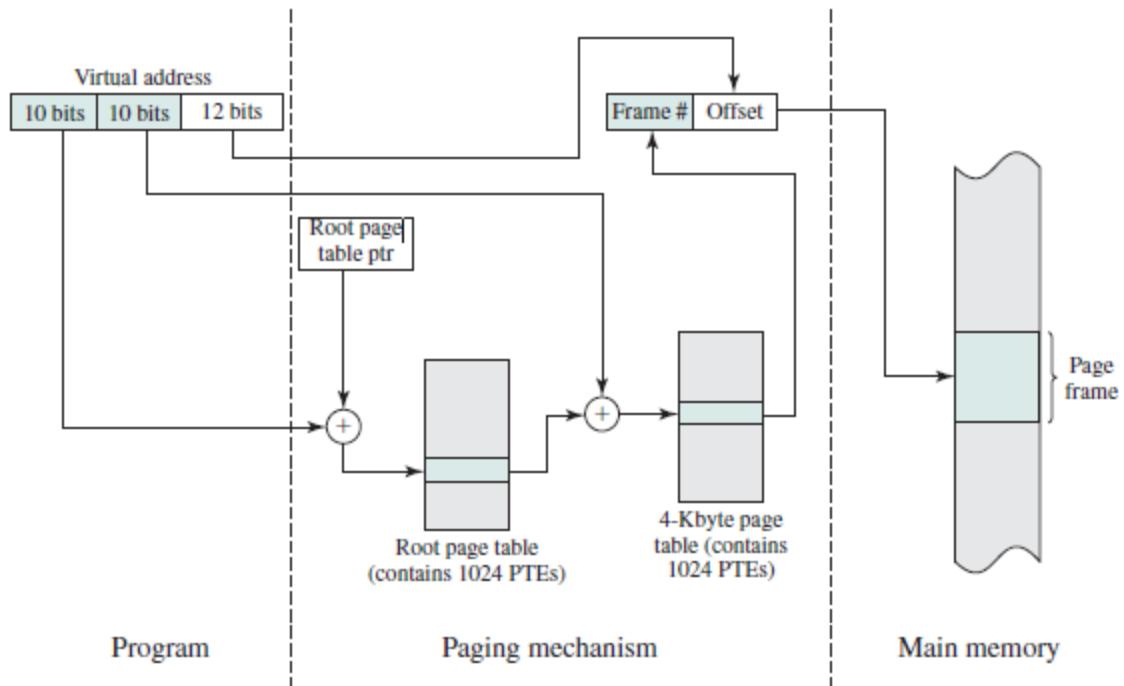
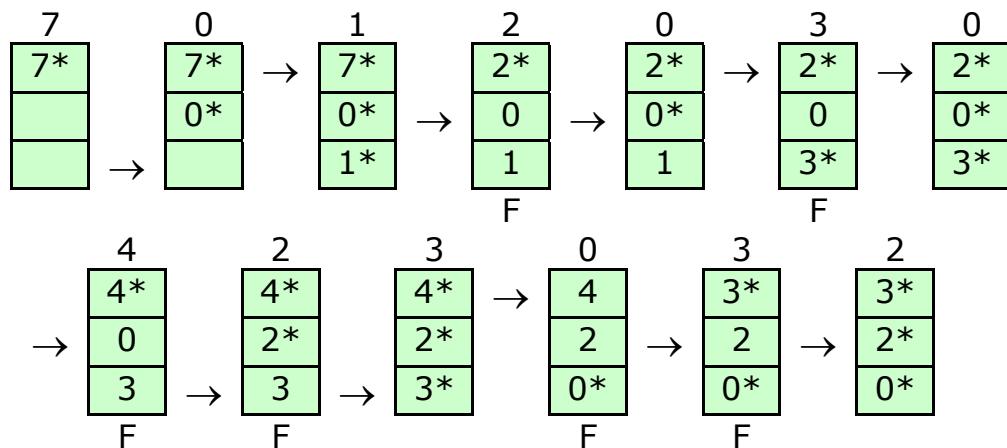


Figure 8.5 Address Translation in a Two-Level Paging System

a. Clock



23. Considerați următorul sir de referințe la pagini 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Completați o figură asemănătoare cu figura 8.5, care să arate alocarea cadrelor pentru: Optimal (presupuneți că sirul de referințe continuă cu 1, 2, 0, 1, 7, 0, 1)

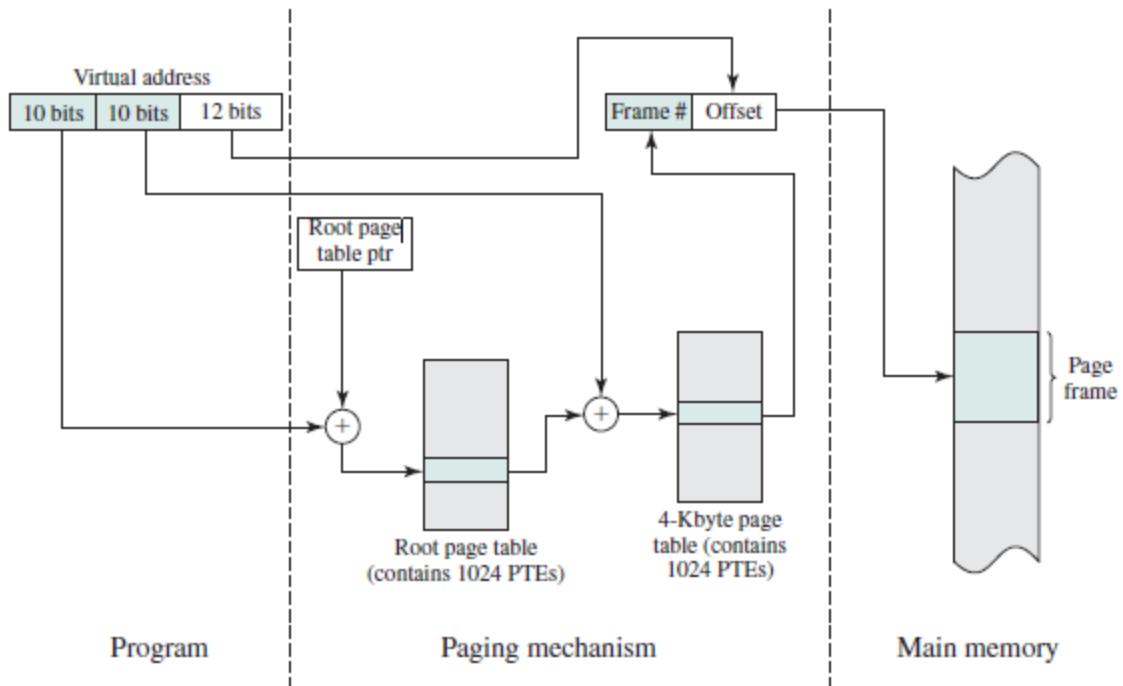


Figure 8.5 Address Translation in a Two-Level Paging System

a. OPT

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	2	2	2	2	2	2
0	0	0	0	0	0	0	4	4	4	0	0	0
		1	1	3	3	3	3	3	3	3	3	3

F F F

24. Considerați următorul sir de referințe la pagini 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Completăți o figură asemănătoare cu figura 8.5, care să arate alocarea cadrelor pentru: Listați numărul total de erori de pagină și rata pentru lipsă pagină (miss) pentru fiecare politică. Numărarea erorilor de pagină începe doar după ce toate cadrele au fost inițializate.

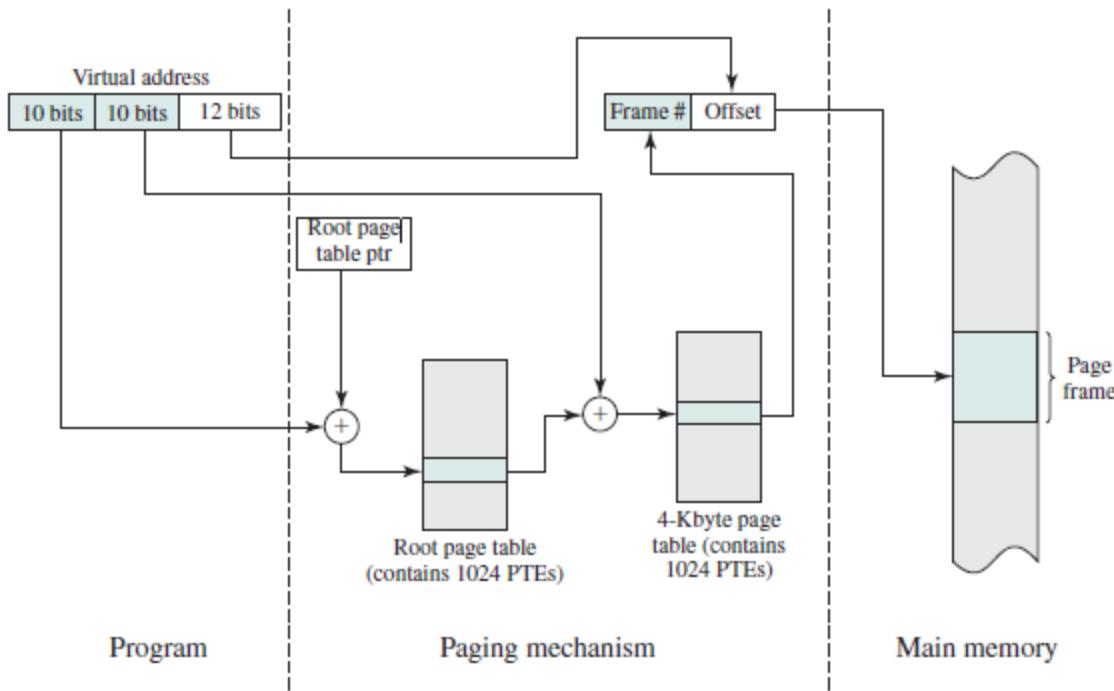


Figure 8.5 Address Translation in a Two-Level Paging System

a.

FIFO: page faults = 7 miss rate = 70%
 LRU: page faults = 6 miss rate = 60%
 Clock: page faults = 6 miss rate = 60%
 OPT: page faults = 3 miss rate = 30%

25. Un proces face referințe la cinci pagini, A, B, C, D, și E, în următoarea ordine :

A; B; C; D; A; B; E; A; B; C; D; E

Presupuneți că algoritmul de înlocuire este first-in-first-out și găsiți numărul de transferuri de pagină pe durata acestei secvențe de referințe începând cu o memorie principală goală având trei cadre de tip pagină. Repetați pentru patru cadre de tip pagină.

9 și respectiv 10 pagini transferate, Este denumită anomalia lui Belady.

26. Un proces conține 8 pagini virtuale pe disc și are asignată o alocare fixă de patru cadre în memorie principală. Apare următoarea trasare a paginilor : 1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3. Arătați paginile succesive rezidente în cele patru cadre utilizând politica LRU de înlocuire. Calculați rata de potrivire (hit ratio) în memorie principală. Presupuneți că, cadrele sunt inițial goale.

Raspuns:

LRU rata de potrivire 16/33

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2
-	0	0	0	0	6	6	6	6	2	2	2	2	2	5	5	5	5	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4	
-	-	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	7	7	7	7	7	7	7	7	7	7	
-	-	-	-	7	7	7	7	7	7	7	3	3	3	3	1	1	1	1	1	6	6	6	6	6	6	3	3	3	3	3	3		
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

27. Un proces conține 8 pagini virtuale pe disc și are asignată o alocare fixă de patru cadre în memorie principală. Apare următoarea trasare a paginilor : 1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3. Arătați paginile succesive rezidente în cele patru cadre utilizând politica FIFO de înlocuire. Calculați rata de potrivire (hit ratio) în memorie principală. Presupuneți că, cadrele sunt inițial goale.

FIFO: rata de potrivire = 16/33

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3	
1	1	1	1	1	1	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6	6	6	2	2	
-	0	0	0	0	0	0	0	1	1	1	1	1	1	1	5	5	5	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	
-	-	2	2	2	2	2	2	2	2	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4	4	4	4	4
-	-	-	-	7	7	7	7	7	7	7	7	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

28. Un proces conține 8 pagini virtuale pe disc și are asignată o alocare fixă de patru cadre în memorie principală. Apare următoarea trasare a paginilor : 1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3. Arătați paginile succesive rezidente în cele patru cadre utilizând politica LRU de înlocuire. Calculați rata de potrivire (hit ratio) în memorie principală. Presupuneți că, cadrele sunt inițial goale. Repetați a pentru politica de înlocuire FIFO. Comparați cele două rate de potrivire și comentați eficacitatea utilizării FIFO pentru a aproxima LRU cu respectarea acestei trasări particulare.

Cele două politici sunt efectiv egale pentru această trasare de pagini.

29. În VAX, tabela cu pagini utilizator este localizată la o adresă virtuală în spațiul sistemului. Care este avantajul de a avea tabela cu pagini în spațiul virtual de adrese mai degrabă decât în memoria principală? Care este dezavantajul?

Principalul avantaj este salvarea spațiului de memorie fizică. Aceasta apare din două motive: (1) tabela cu pagini a utilizatorului poate fi paginată în memorie la nevoie. (2)

Sistemul de operare poate aloca dinamic tabelele cu pagini, creând doar una singură la crearea procesului.

30. Fie următoarele linii de program

```
for (i = 1; i <= n; i++)  
    a[i] = b[i] + c[i];
```

Care sunt executate într-o memorie cu mărimea paginii de 1,000 de cuvinte. Fie $n = 1,000$. Utilizând un calculator care are o gamă întreagă de instrucțiuni registrula-registru care implică și regiștrii de index, scrieți un program ipotetic pentru a implementa liniile anterioare. Arătați apoi secvența de referințe la pagini pe durata execuției.

a.

Versiunea în limbajul mașină a acestui program, încărcat în memoria principală începând de la adresa 4000, poate să apară ca:

```
4000      (R1) ← ONE Establish index register for i  
4001      (R1) ← n   Establish n in R2  
4002      compare R1, R2   Test i > n  
4003      branch greater 4009  
4004      (R3) ← B(R1)   Access B[i] using index register R1  
4005      (R3) ← (R3) + C(R1)   Add C[i] using index register R1  
4006      A(R1) ← (R3)   Store sum in A[i] using index register R1  
4007      (R1) ← (R1) + ONE   Increment i  
4008      branch 4002  
6000-6999 storage for A  
7000-7999 storage for B  
8000-8999 storage for C  
9000     storage for ONE  
9001     storage for n
```

Sirul de referințe generat de această buclă este $494944(47484649444)^{1000}$ constând din peste 11000 de referințe, dar care implică numai cinci pagini distincte.

31. Sistemul IBM -IBM System/370- utilizează o structură de memorie organizată pe două niveluri care sunt referite ca segmente și pagini, deși abordarea bazată pe segmentare blochează multe dintre trăsăturile mai devreme în acest capitol. Pentru arhitectura 370 de bază, mărimea paginii poate fi fie 2 Kocteți sau 4 Kocteți, iar mărimea segmentului este fixată fie la 64 Kocteți sau 1 Mocet. Pentru arhitecturile 370/XA și 370/ESA, mărimea paginii este de 4 Kocteți și mărimea segmentului este de 1 Mocet. Care avantaj al segmentării îl blochează această schemă? Care este avantajul segmentării la 370?

a. Segmentele S/370 sunt fixe în mărime și nu sunt vizibile programatorului. Astfel, nici unul dintre beneficiile enumerate pentru segmentare nu se realizează pe S/370, cu excepția protecției. Bitul P din fiecare intrare din tabela cu segmente oferă protecție pentru întregul segment.

32. Presupuneți o mărime a paginii de 4 Kocteți și că o intrare în tabela cu pagini ia 4 octeți, cât de multe niveluri de paginare vor fi necesare pentru a mapa un spațiu de adrese pe 64 de biți, dacă tabela cu pagini de la nivelul cel mai de sus se încadrează într-o singură pagină.

Deoarece fiecare intrare de pagină din tabel este de 4 octeți și fiecare pagină conține 4 Kocteți, atunci o tabelă de pagini va pointa $1024 = 2^{10}$ pagini, adresând un total de $2^{10} \times 2^{12} = 2^{22}$ octeți. Spațiul de adrese este totuși de 2^{64} octeți. Adăugând un al doilea nivel la tabelele cu pagini, tabelul de pagini din top va pointa 2^{10} tabele cu pagini, adresând un total de 232 octeți. Continuând acest proces,

Depth	Address Space
1	2^{22} bytes
2	2^{32} bytes
3	2^{42} bytes
4	2^{52} bytes
5	2^{62} bytes
6	2^{72} bytes ($> 2^{64}$ bytes)

Se poate vedea că cinci niveluri nu sunt de ajuns pentru a adresa întreg spațiul de adrese de 64 de biți, astfel încât este necesar un al șaselea nivel. Astfel încât să cerem ca spațiul de adrese să fie pe 72 de biți, putem masca și ignora toți biții mai puțin cei doi mai puțin semnificativ ai nivelului al șaselea. Vom obține astfel o adresă pe 64 de biți. Pagina din top va avea doar 4 intrări. Totuși o altă opțiune este de a revizui criteriul ca tabela cu pagini din top să se potrivească într-o singură pagina, și în loc o facem să ocupe patru pagini. Aceasta va salva o pagină fizică ceea ce nu este prea mult.

33. Considerați un sistem cu maparea memoriei făcută pe bază de pagini și utilizând un singur nivel de paginare. Presupuneți că necesarul de tabele cu pagini este întotdeauna în memorie. Dacă o referință la memorie ia 200 ns, cât de mult ia o referință la o memorie paginată?

400 ns

34. Considerați un sistem cu maparea memoriei făcută pe bază de pagini și utilizând un singur nivel de paginare. Presupuneți că necesarul de tabele cu pagini este întotdeauna în memorie. O referință la memorie ia 200 ns. Acum adăugăm o MMU care impune un supracontrol de 20ns la o potrivire sau la o lipsă. Dacă presupunem că 85% din toate referințele la memorie se potrivesc în MMU TLB, care este timpul efectiv de acces la memorie (Effective Memory Access Time (EMAT))?

250

35. Considerați un sistem cu maparea memoriei făcută pe bază de pagini și utilizând un singur nivel de paginare. Presupuneți că necesarul de tabele cu pagini este întotdeauna în memorie. Explicați cum este afectat EMAT-ul de rata de potrivire la TLB.

Cu cât este mai mare potrivirea la TLB cu atât este mai mic EMAT-ul, deoarece penalizarea adițională de 200 ns pentru a prelua o intrare în TLB nu va contribui la EMAT.

36. Considerați un sir de referințe la pagini pentru un proces având un set de lucru notat cu M, inițial gol. Sirul de referințe la memorie este de lungime P cu numerotat cu N pagini distințe. Pentru oricare algoritm de înlocuire, care este marginea inferioară a erorilor de pagină?

N

37. Considerați un sir de referințe la pagini pentru un proces având un set de lucru notat cu M, inițial gol. Sirul de referințe la memorie este de lungime P cu numerotat cu N pagini distințe. Pentru oricare algoritm de înlocuire, care este marginea superioară a erorilor de pagină?

P

38. În discuția despre algoritmii de înlocuire a paginilor, un autor a făcut o analogie cu mișcarea unei mașini de curățat zăpada (snowplow) pe o pistă circulară. Zăpada cade uniform pe pistă și o mașină solitară de curățat zăpada circulă pe pistă continuu cu viteză constantă. Dacă zăpada este curățată pista dispare din sistem. Pentru care dintre algoritmii de înlocuire discutați în 8.2 este utilă analogia?

- a. Aceasta este o analogie bună cu algoritmul CLOCK. Căderea zăpezii pe pistă este analogă cu potrivirea la pagină pe bufferul circular în sensul acelor de ceasornic. Mișcarea pointerului CLOCK este analogă cu mișcarea plugului.

39. În discuția despre algoritmii de înlocuire a paginilor, un autor a făcut o analogie cu mișcarea unei mașini de curățat zăpada (snowplow) pe o pistă circulară. Zăpada cade uniform pe pistă și o mașină solitară de curățat zăpada circulă pe pistă continuu cu viteză constantă. Dacă zăpada este curățată pista dispare din sistem. Ce sugerează această analogie despre comportarea algoritmului de înlocuire a paginilor în cauză?

De notat că densitatea paginilor înlocuibile este cea mai mare imediat înaintea pointerului CLOCK, analog cum densitatea zăpezii este cea mai mare în fața plugului. Astfel, ne putem aștepta că algoritmul CLOCK să fie foarte eficient în găsirea paginilor ca să le înlocuiască. De fapt, se poate arăta că adâncimea zăpezii din fața plugului este de două ori în medie mai adâncă decât zăpada de pe toată pistă. Prin această analogie, numărul de pagini înlocuite de politica CLOCK pe un singur circuit trebuie să fie de două ori numărul care este înlocuit I pe o durată aleatoare. Analogia este imperfectă deoarece pointerul CLOCK nu se mișcă cu o viteză constantă, dar ideea intuitivă rămâne.

40. În arhitectura S/370 o cheie de memorare este un câmp de control asociat cu fiecare cadru de memorie paginată din memoria reală. Doi biți din această cheie care sunt relevanți pentru înlocuirea paginii sunt bitul de referință și bitul de modificare. Bitul de referință este setat pe 1 atunci când orice adresă din cadru este accesată pentru citire sau scriere, și este setat pe 0 când se încarcă o pagină nouă într-un cadru. Bitul de modificare este setat pe 1 atunci când se realizează o operație de scriere pe orice locație din cadru. Sugerați o abordare pentru a determina care pagini de tip cadru sunt cel mai puțin recent utilizate (least-recently-used) utilizând numai bitul de referință.

Hardware-ul procesorului setează bitul de referință pe 0 când o nouă pagină se încarcă în cadru, și pe 1 când o locație din cadru este accesată. Sistemul de operare poate menține un număr de cozi de tabele pentru cadre. O intrare în tabelă pentru cadru de pagină se mută de la o coadă la alta în concordanță de timpul în care bitul de referință pentru acea pagină cadru stă pe 0. Când pagina trebuie înlocuită, paginile de înlocuit sunt alese din coada cu cadrele pentru care timpul cât nu au fost accesate este cel mai lung.

41. Considerați următoarea secvență de referințe la pagini (fiecare element din secvență reprezintă un număr de pagină): 1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5. Definiți setul de lucru mediu după a k-a referință ca

$$s_k(\Delta) = \frac{1}{k} \sum_{t=1}^k |W(t, \Delta)|$$

și definiți probabilitatea de lipsă a unei pagini după a k-a referință ca

$$m_k(\Delta) = \frac{1}{k} \sum_{t=1}^k F(t, \Delta)$$

unde $F(t, \Delta) = 1$ dacă apare o eroare de pagină la momentul de timp virtual t și 0 altfel. Trasați o diagramă similară cu aceea din figura 8.19 pentru secvența de referințe tocmai definită pentru valorile 1, 2, 3, 4, 5, 6.

Sequence of Page References	Window Size, Δ			
W	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figure 8.19 Working Set of Process as Defined by Window Size

a.

Seq of page refs	Window Size, Δ					
	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	1 2	1 2	1 2	1 2	1 2
3	3	2 3	1 2 3	1 2 3	1 2 3	1 2 3
4	4	3 4	2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
5	5	4 5	3 4 5	2 3 4 5	1 2 3 4 5	1 2 3 4 5
2	2	5 2	4 5 2	3 4 5 2	3 4 5 2	1 3 4 5 2
1	1	2 1	5 2 1	4 5 2 1	3 4 5 2 1	3 4 5 2 1
3	3	1 3	2 1 3	5 2 1 3	4 5 2 1 3	4 5 2 1 3
3	3	3	1 3	2 1 3	5 2 1 3	4 5 2 1 3
2	2	3 2	3 2	1 3 2	1 3 2	5 1 3 2
3	3	2 3	2 3	2 3	1 2 3	1 2 3
4	4	3 4	2 3 4	2 3 4	2 3 4	1 2 3 4
5	5	4 5	3 4 5	2 3 4 5	2 3 4 5	2 3 4 5
4	4	5 4	5 4	3 5 4	2 3 5 4	2 3 5 4
5	5	4 5	4 5	4 5	3 4 5	2 3 4 5
1	1	5 1	4 5 1	4 5 1	4 5 1	3 4 5 1
1	1	1	5 1	4 5 1	4 5 1	4 5 1

3	3	1 3	1 3	5 1 3	4 5 1 3	4 5 1 3
2	2	3 2	1 3 2	1 3 2	5 1 3 2	4 5 1 3 2
5	5	2 5	3 2 5	1 3 2 5	1 3 2 5	1 3 2 5

42. Considerați următoarea secvență de referințe la pagini (fiecare element din secvență reprezintă un număr de pagină): 1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5. Definiți setul de lucru mediu după a k-a referință ca

$$s_k(\Delta) = \frac{1}{k} \sum_{t=1}^k |W(t, \Delta)|$$

și definiți probabilitatea de lipsă a unei pagini după a k-a referință ca

$$m_k(\Delta) = \frac{1}{k} \sum_{t=1}^k F(t, \Delta)$$

unde $F(t, \Delta) = 1$ dacă apare o eroare de pagină la momentul de timp virtual t și 0 altfel. Desenați $s_{20}(\Delta)$ ca o funcție de Δ .

Sequence of Page References	Window Size, Δ			
	2	3	4	5
W				
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figure 8.19 Working Set of Process as Defined by Window Size

Raspuns:

Δ	1	2	3	4	5	6
s₂₀(Δ)	1	1.85	2.5	3.1	3.55	3.9

43. Considerați următoarea secvență de referințe la pagini (fiecare element din secvență reprezintă un număr de pagină): 1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5. Definiți setul de lucru mediu după a k-a referință ca

$$s_k(\Delta) = \frac{1}{k} \sum_{t=1}^k |W(t, \Delta)|$$

și definiți probabilitatea de lipsă a unei pagini după a k-a referință ca

$$m_k(\Delta) = \frac{1}{k} \sum_{t=1}^k F(t, \Delta)$$

unde $F(t, \Delta) = 1$ dacă apare o eroare de pagină la momentul de timp virtual t și 0 altfel. Desenați $m_{20}(\Delta)$ ca o funcție de Δ .

Sequence of Page References	Window Size, Δ			
	2	3	4	5
W				
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figure 8.19 Working Set of Process as Defined by Window Size

Raspuns:

Δ	1	2	3	4	5	6
$m_{20}(\Delta)$	0.9	0.75	0.75	0.65	0.55	0.5

44. Cheia performanțelor politicii de gestiune cu set rezident VSWS este alegerea valorii lui Q. Experiența a arătat că, cu o valoare fixă pentru Q pentru un proces, există diferențe considerabile privind frecvența erorilor de pagină la diferite stagiile ale execuției. Mai mult, dacă este utilizată o singură valoare a lui Q pentru diferite procese, apar diferențe dramatice ale frecvenței de apariție a erorilor de pagină. Aceste diferențe indică cu tărie faptul că un mecanism care să schimbe dinamic valoarea lui Q pe durata execuției procesului va îmbunătăți comportarea algoritmului. Sugerați un mecanism simplu pentru acest scop.

[PIZZ89] a sugerat următoarea strategie. Utilizați un mecanism care ajustează valoarea lui Q la fiecare fereastră de timp ca o funcție de rata erorilor de pagină experimentate pe durata ferestrei. Rata erorilor de pagină este calculată și comparată cu o valoare a erorilor de pagină pentru un job, dorită la nivelul sistemului. Valoarea lui Q este ajustată prin ridicare (coborâre) cât timp rata actuală a erorilor de pagină este mai mare (mai mică) decât valoarea dorită. Experimentele utilizând acest mecanism de ajustare au arătat că execuția unor job-uri de test cu ajustarea dinamică a lui Q a produs în mod consistent un număr mai mic de erori de pagină pe execuție și o descreștere a mediei setului rezident față de cazul execuției cu o valoare constantă pentru Q (într-o gamă foarte largă).

45. Presupuneți că un task este divizat în patru segmente de mărime egală și că sistemul construiește un tabel cu descriptori pentru pagini cu 8 intrări pentru fiecare segment. Astfel, sistemul are o combinație de segmentare și paginare. Presupuneți că pagina are 2 Kocteți. Care este mărimea maximă a fiecarui segment?

16k

46. Presupuneți că un task este divizat în patru segmente de mărime egală și că sistemul construiește un tabel cu descriptori pentru pagini cu 8 intrări pentru fiecare segment. Astfel, sistemul are o combinație de segmentare și paginare. Presupuneți că pagina are 2 Kocteți. Care spațiul maxim de adrese logice pentru task?

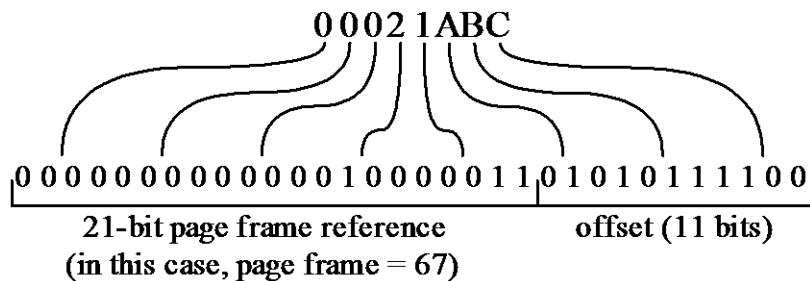
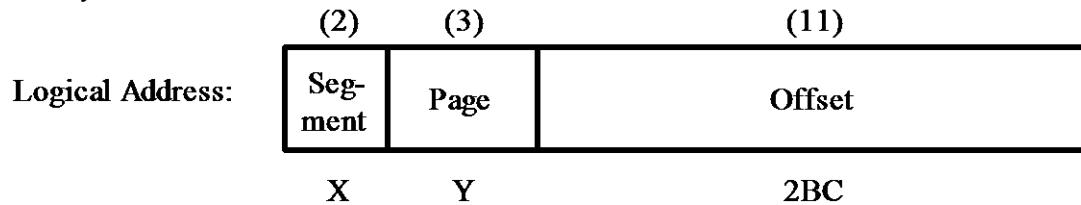
64k

47. Presupuneți că un task este divizat în patru segmente de mărime egală și că sistemul construiește un tabel cu descriptori pentru pagini cu 8 intrări pentru fiecare segment. Astfel, sistemul are o combinație de segmentare și paginare. Presupuneți că pagina are 2 Kocteți. Presupuneți că un element de la locația fizică 00021ABC este accesat de acest task. Care este formatul adresei logice pe

care o generează taskul pentru aceasta? Care este maximul spațiului de adrese fizice pentru sistem?

Raspuns:

4 Gbyte



48. Considerați un spațiu de adrese logice paginat (compus din 32 de pagini fiecare pagină având 2 Kocteți) mapate într-un spațiu de memorie de 1 Mocet. Care este formatul adresei logice a procesorului?

Raspuns:

page number (5)	offset (11)
-----------------	-------------

49. Considerați un spațiu de adrese logice paginat (compus din 32 de pagini fiecare pagină având 2 Kocteți) mapate într-un spațiu de memorie de 1 Mocet. Care este lungimea și lățimea tabelului cu pagini (neglijând “access rights” bits)?

32 de intrări a către 9 biți fiecare.

50. Considerați un spațiu de adrese logice paginat (compus din 32 de pagini fiecare pagină având 2 Kocteți) mapate într-un spațiu de memorie de 1 Mocet. Care este efectul asupra tabelului cu pagini dacă mărimea spațiului de memorie fizică este redus la jumătate?

Dacă numărul total de intrări se stabilizează pe 32 și mărimea paginii nu se schimbă, atunci fiecare intrare devine de 8 biți.

Cap. 9. Plannificare uniprosesor

1. Descrieți pe scurt trei tipuri de planificare a procesorului.

Planificarea pe termen lung: decizia de a adăuga un proces grupului de procese care urmează a fi executate. Planificarea pe termen mediu: decizia de a adăuga la numărul de procese care sunt parțial sau în întregime în memoria. Principală. Planificarea pe termen scurt: decizia privind care proces va fi executat de procesor.

2. Care sunt în mod ușual cerințele critice privind performanța într-un sistem de operare interactiv?

Timpul de răspuns.

3. Care sunt diferențele între timpul turnaround (timpul între momentul intrării în sistem și cel al ieșirii sau execuției) și timpul de răspuns?

Turnaround time (timpul intra-ieșire) este timpul total petrecut în sistem (timpul de așteptare plus timpul de servire). Timpul de răspuns este timpul consumat între depunerea unei cereri și momentul în care răspunsul apare la ieșire.

4. Pentru planificarea proceselor, o valoare mică reprezintă o prioritate mică sau o prioritate mare?

În UNIX și alte multe sisteme, valori mari pentru prioritate reprezintă procese cu prioritate mică. Unele sisteme precum Windows, utilizează semnificația opusă: un număr mare reprezintă o prioritate mare.

5. Care este diferența între planificarea cu suspendare (preemptive) și planificarea fără suspendare (nonpreemptive) ?

Planificarea fără suspendare: dacă un proces este în starea în execuție. El continuă să se execute până când se termină, se autoblochează așteptând completarea unei operațiilor I/O sau cere un serviciu al sistemului de operare. Planificarea cu suspendare: procesul care este în execuție poate fi întrerupt și mutat în starea ready de către sistemul de operare. Decizia de suspendare poate fi luată atunci când sosesc un nou proces, când apare o întrerupere care deblochează un proces pe care îl placează în lista ready și acest din urmă proces este mai priorită decât procesul curent, sau periodic pe baza întreruperii de ceas.

6. Definiți succint planificarea FCFS.

Imediat ce un proces devine gata de execuție (ready), acesta se alătură listei ready. Imediat ce un proces își încetează execuția, procesul care a stat cel mai mult în coada ready este selectat pentru execuție.

7. Definiți succint planificarea round-robin.

Se generează o întrerupere de ceas în mod periodic. Când apare întreruperea, procesul curent în execuție este plasat în coada ready, și următorul proces gata de execuție este selectat pe baza unui algoritm FCFS.

8. Definiți succint planificarea shortest-process-next scheduling.

Aceasta este o politică fără suspendare unde este selectată pentru execuție procesul cu timpul estimat ca fiind cel mai scurt timp de execuție.

9. Definiți succint planificarea shortest-remaining-time scheduling.

Aceasta este o versiune cu suspendare a SPN. În acest caz, planificatorul alege întotdeauna procesul care are cel mai scurt timp estimat de completare a execuție. Atunci când la coada ready se alătură un nou proces, poate fi și acela care are cel mai scurt timp rămas pentru execuție. Ca urmare, planificatorul poate suspenda dacă un nou proces devine gata de execuție.

10. Definiți succint planificarea highest-response-ratio-next scheduling.

Atunci când procesul curent își completează execuția sau devine blocat, se alege un proces gata de execuție care are valoarea cea mai mare pentru $R = (w+s)/s$ unde w = timpul petrecut așteptând procesorul și s = timpul estimat de servire.

11. Definiți succint planificarea feedback scheduling.

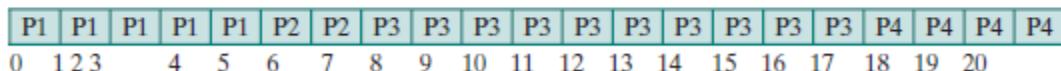
Planificare este făcută cu suspendare (pe o cantă de timp), și se utilizează un mecanism dinamic de priorități. Atunci când un proces intră prima dată în sistem el este plasat în RQ0 (vezi figura 9.4). După prima sa execuție, când se reîntoarce în starea ready, va fi plasat în RQ1. De fiecare dată când este suspendat este degradat și plasat în următoarea coadă cu prioritatea mai mică. Procesele scurte se completează repede fără a migra foarte jos în ierarhie. Procesele lungi, în mod gradat vor aluneca treptat în jos. Astfel, procesele noi și scurte sunt favorite față de cele vechi și lungi. În cadrul fiecărei cozi, cu excepția celei cu prioritatea cea mai mică, se utilizează un mecanism simplu de tip FCFS. O dată ajuns în coada cea mai puțin priorităță, procesul nu poate merge mai jos, dar revine în această coadă până își completează execuția.

12. Considerați următoarea încărcare:

Process	Burst Time	Priority	Arrival Time
P1	50 ms	4	0 ms
P2	20 ms	1	20 ms
P3	100 ms	3	40 ms
P4	40 ms	2	60 ms

Prezentați planificarea utilizând shortest remaining time, nonpreemptive priority (un număr mic al priorității implică o prioritate mai mare) și o cantă round robin de 30 ms. Utilizați o diagramă cu diviziuni de timp așa cum se prezintă mai jos pentru un exemplu ce utilizează FCFS, pentru a arăta planificarea pentru fiecare politică cerută de planificare.

Example for FCFS (1 unit = 10 ms):



Shortest Remaining Time:

P1	P1	P2	P2	P1	P1	P1	P4	P4	P4	P4	P3								
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Non-preemptive Priority:

P1	P1	P1	P1	P1	P1	P2	P2	P4	P4	P4	P4	P3							
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Round Robin with quantum of 30 ms:

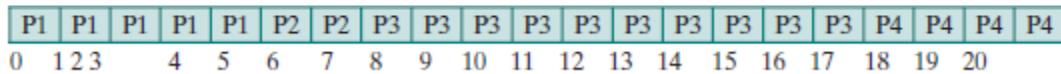
P1	P1	P1	P2	P2	P1	P1	P3	P3	P3	P4	P4	P4	P3	P3	P3	P4	P3	P3	P3
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

13. Considerați următoarea încărcare:

Process	Burst Time	Priority	Arrival Time
P1	50 ms	4	0 ms
P2	20 ms	1	20 ms
P3	100 ms	3	40 ms
P4	40 ms	2	60 ms

Prezentați planificarea utilizând shortest remaining time, nonpreemptive priority (un număr mic al priorității implică o prioritate mai mare) și o cantă round robin de 30 ms. Utilizați o diagramă cu diviziuni de timp așa cum se prezintă mai jos pentru un exemplu ce utilizează FCFS, pentru a arăta planificarea pentru fiecare politică cerută de planificare.

Example for FCFS (1 unit = 10 ms):



Care este media timpului de așteptare pentru politicile de planificare menționate anterior?

Shortest Remaining Time: = 25 ms.

Non-preemptive Priority: = 27.5ms

Round-Robin: = 42.5ms

Round-Robin: = 42.5ms

14. Considerați următorul set de procese:

Process	Arrival Time	Processing Time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Realizați aceeași analiză aşa cum este aceasta prezentată în Tabelul 9.5 și Figura 9.5 din acest capitol.

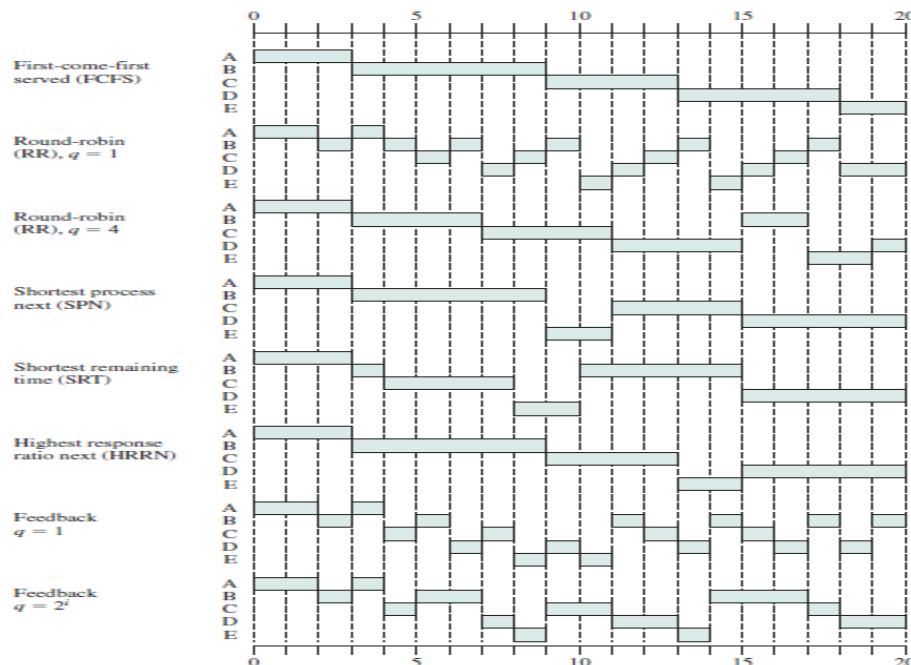


Figure 9.5 A Comparison of Scheduling Policies

Table 9.5 A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$						
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

a.

FCFS

A	A	A	B	B	B	B	C	C	C	D	D	D	D	D	D	E	E	E	E	E
A	B	A	B	C	A	B	C	B	D	B	D	E	D	E	D	E	D	E	D	E
A	A	A	B	B	B	B	C	C	B	D	D	D	D	D	D	E	E	E	E	D
A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	D	D	E	E	E	E
A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	D	D	E	E	E	E
A	A	A	B	B	B	B	B	C	C	D	D	D	D	D	D	D	E	E	E	E
A	B	A	C	B	C	A	B	B	D	B	D	E	D	E	D	E	D	E	D	E
A	B	A	A	C	B	B	C	B	B	D	D	E	D	D	D	E	E	D	E	E

RR, $q = 1$

RR, $q = 4$

SPN

SRT

HRRN

Feedback, $q = 1$

Feedback, $q = 2^i$

		A	B	C	D	E
	T_a	0	1	3	9	12
	T_s	3	5	2	5	5
FCFS	T_f	3	8	10	15	20
	T_r	3.00	7.00	7.00	6.00	8.00
	T_r/T_s	1.00	1.40	3.50	1.20	1.60
RR $q = 1$	T_f	6.00	11.00	8.00	18.00	20.00
	T_r	6.00	10.00	5.00	9.00	8.00
	T_r/T_s	2.00	2.00	2.50	1.80	1.60
RR $q = 4$	T_f	3.00	10.00	9.00	19.00	20.00

	T_r	3.00	9.00	6.00	10.00	8.00	7.20
	T_r/T_s	1.00	1.80	3.00	2.00	1.60	1.88
SPN	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
SRT	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
HRRN	T_f	3.00	8.00	10.00	15.00	20.00	
	T_r	3.00	7.00	7.00	6.00	8.00	6.20
	T_r/T_s	1.00	1.40	3.50	1.20	1.60	1.74
FB $q = 1$	T_f	7.00	11.00	6.00	18.00	20.00	
	T_r	7.00	10.00	3.00	9.00	8.00	7.40
	T_r/T_s	2.33	2.00	1.50	1.80	1.60	1.85
FB $q = 2^i$	T_f	4.00	10.00	8.00	18.00	20.00	
	T_r	4.00	9.00	5.00	9.00	8.00	7.00
	T_r/T_s	1.33	1.80	2.50	1.80	1.60	1.81

15. Presupuneți următorul şablon pentru timpul de execuție în salvă (continuă): 6, 4, 6, 4, 13, 13, 13, și presupuneți că valoare inițială este 10. Trasați un grafic similar celui din figura 9.9.

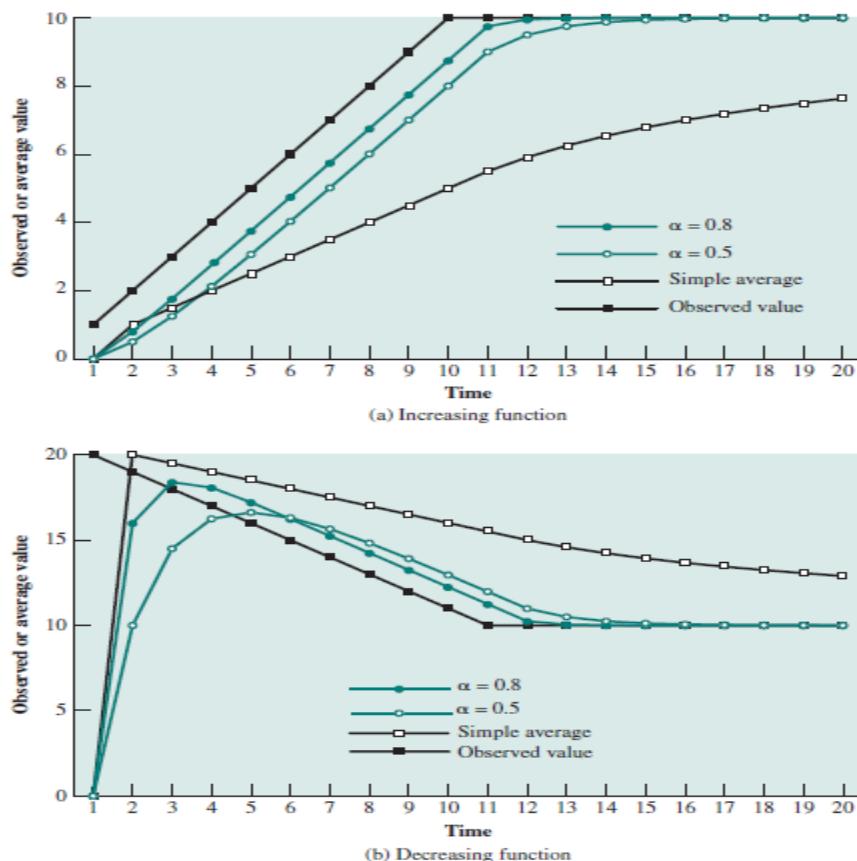


Figure 9.9 Use of Exponential Averaging

a.

Age of Observation	Observed Value	Simple Average	alpha = 0.8	alpha = 0.5
1	6	0.00	0.00	0.00
2	4	3.00	4.80	3.00
3	6	3.33	4.16	3.50
4	4	4.00	5.63	4.75
5	13	4.00	4.33	4.38
6	13	5.50	11.27	8.69
7	13	6.57	12.65	10.84

16.9.5 Considerați următoarea pereche de ecuații ca o alternativă la ecuația (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n$$

$$X_{n+1} = \min[Ubound, \max[Lbound, (\beta S_{n+1})]]$$

unde Ubound și Lbound sunt marginile superioară și inferioară pre-alese pentru valoarea estimată a lui T. Valoarea X_{n+1} este utilizată în algoritmul shortest-

process-next, în loc de valoarea S_{n+1} . Ce funcții realizează α și β , și care este efectul unor valori mici sau mari ale acestora?

Prima ecuație este identică cu ecuația 9.3, astfel că parametrul α oferă un efect de netezire exponențială. Parametrul β un factor care exprimă varianța întârzierii (de ex. între 1,3 și 2,0). O valoare mică pentru β va determina o adaptare mai rapidă la schimbările date de timpul de observare, dar va determina și o fluctuație mai mare a estimărilor.

17. În exemplul din figura Figure 9.5, procesul A se execută două unități de timp înainte ca să fie pasat controlul lui B. Un alt scenariu plauzibil poate fi acela că A să ruleze trei unități de timp înainte ca să fie pasat controlul lui B.. Ce diferențe între politicile algoritmului feedbackscheduling vor trebui luate în calcul pentru cele două scenarii?

Depinde de momentul când se pune job-ul A în coadă, după prima unitate de timp sau nu. Dacă da, atunci are dreptul la două unități adiționale de timp înainte de a putea fi suspendat (întrerupt).

18. Într-un sistem uniprocesor nonpreemptive, coada ready conține trei job-uri la timpul t imediat după completarea unui job curent. Aceste job-uri au sosit la momentele de timp t_1 , t_2 , și t_3 cu timpii de execuție estimati având valoarea r_1 , r_2 , și respectiv r_3 . Figura 9.18 prezintă creșterea liniară a raportului lor de răspuns (response ratios) în funcție de timp. Utilizați acest exemplu pentru a găsi o variantă a planificării în funcție de raportul lor de răspuns, cunoscut ca minimax response ratio scheduling, care minimizează maximul raportul lor de răspuns pentru un lot dat de job-uri ignorând viitoarele sosiri (Sugestie: Decideți prima dată care job să-l planificați ultimul.)

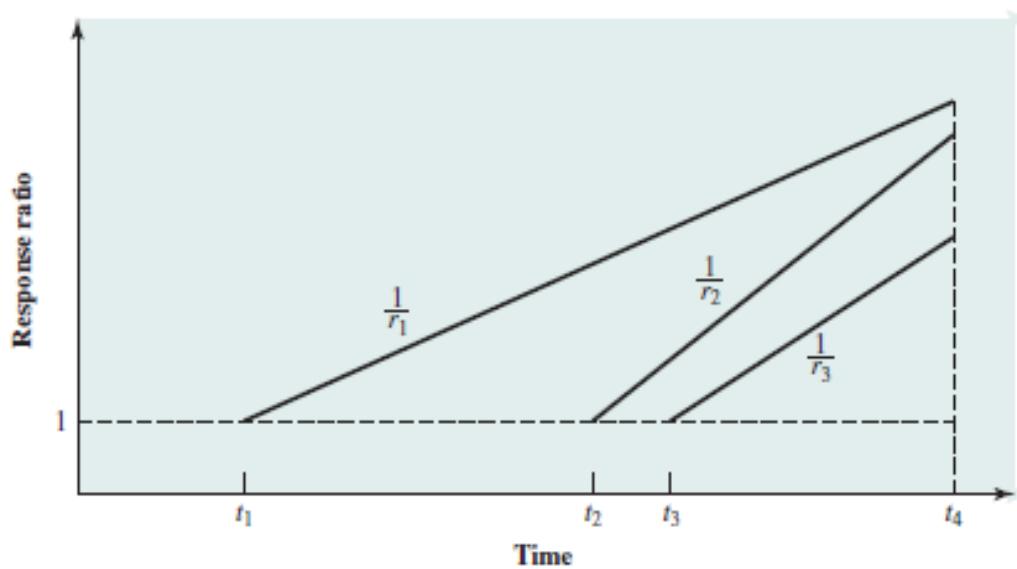
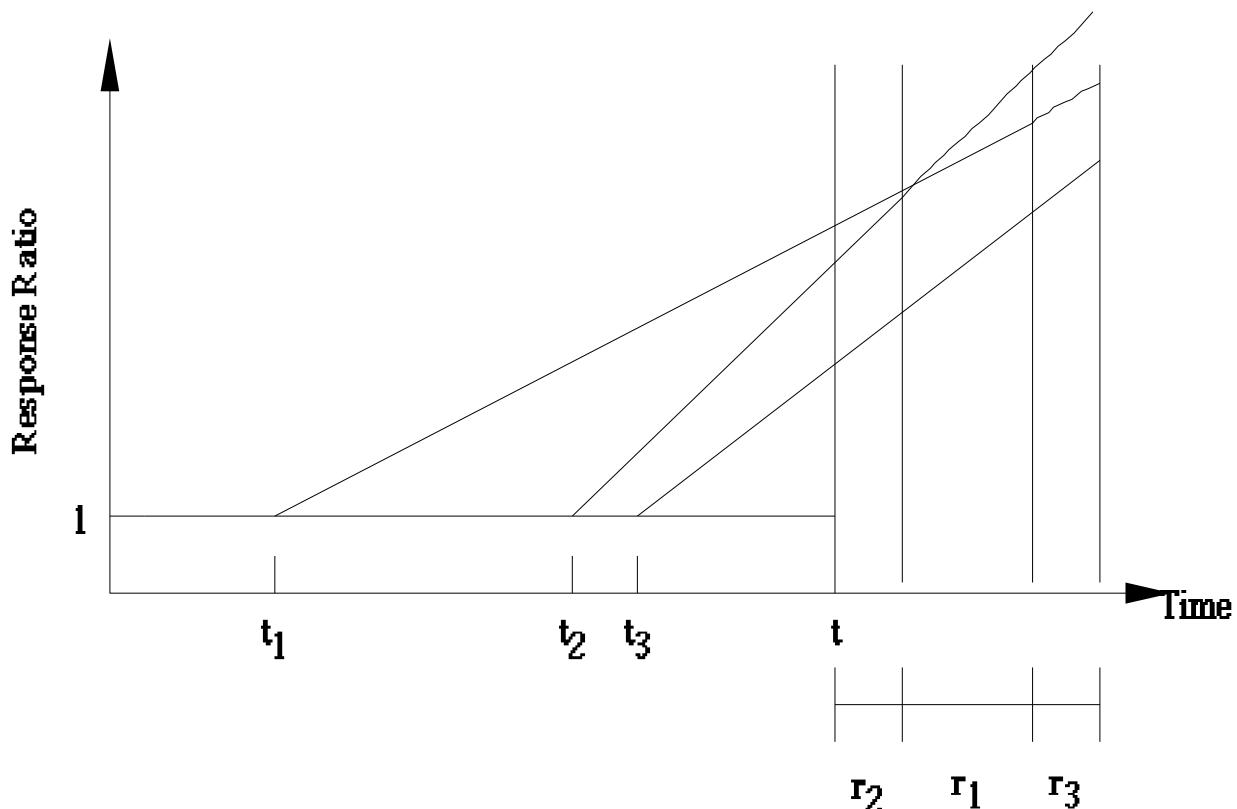


Figure 9.18 Response Ratio as a Function of Time

a.

Prima dată, planificatorul calculează rapoartele de răspuns la timpul $t + r_1 + r_2 + r_3$, când sunt terminate toate cele trei job-uri (vezi figura). La acel moment job-ul trei are cel mai mic raport de răspuns din cele trei: ca urmare planificatorul decide să execute acest ultim job și trece la examinarea job-urilor 1 și doi la momentul $t + r_1 + r_2$, când ambele se vor termina. Aici raportul de răspuns al job-ului 1 este mai mic, și în consecință job-ul 2 va fi selectat pentru execuție la momentul t . Acest este repetat de fiecare dată când un job este completat pentru a lua în calcul noile job-uri sosite.



19. Considerați o variantă a algoritmului de planificare RR unde intrările în coada ready sunt pointeri la PCB-uri. Care va fi efectul plasării a doi pointeri la același proces în coada ready?

Deoarece coada ready are pointeri multipli la același proces, sistemul oferă aceluiași proces un tratament preferențial. Aceasta înseamnă că, acel proces va prelua un timp procesor dublu față de un proces cu un singur pointer.

20. Considerați o variantă a algoritmului de planificare RR unde intrările în coada ready sunt pointeri la PCB-uri.

Care ar fi avantajul major al acestei scheme?

Avantajul este acela că unui job mai important i se poate da mai mult timp procesor prin simpla adăugare a unui pointer adițional (un supracontrol redus pentru implementare).

21. Considerați o variantă a algoritmului de planificare RR unde intrările în coada ready sunt pointeri la PCB-uri. Cum puteți modifica algoritmul RR de bază ca să obțineți același efect fără a duplica pointerii?

Oferă procesului o cantă de timp mai mare prin utilizarea a două priorități. Se adaugă un bit în PCB care spune că un proces poate rula două cuante de timp. Adaugă un întreg în PCB care indică numărul de cuante pe care le poate executa un proces. Se prevăd două cozi ready, una din ele permite cuante mai lungi pentru job-uri cu prioritate mai mare.

22. Un sistem interactiv ce utilizează planificarea round-robin și comutarea (swapping) încearcă să dea un răspuns garantat unei cereri triviale cum este următoarea: după completarea unui ciclul round-robin pentru toate procesele ready, sistemul determină cuanta de timp pe care să o aloce pentru următorul ciclul, divizând timpul maxim de răspuns la numărul de procese care cer a fi servite. Este această politică rezonabilă?

Numai dacă există doar puțini utilizatori în sistem. Atunci când cuanta descrește pentru a satisface rapid doi utilizatori se întâmplă două lucruri: (1) descrește utilizarea procesorului, și (2) la un anumit punct, cuanta devine prea mică pentru a satisface cele mai triviale cereri. Utilizatorii vor experimenta o creștere supărătoare a timpului de răspuns, deoarece cererile lor trebuie să treacă de mai multe ori prin coada round-robin.

23. Ce tip de proces este în general favorizat de un planificator de tip multilevel feedback queueing – un proces orientat pe procesor (processor bound - calcule) sau un proces orientat pe operații I/O (I/O-bound process)? Explicați pe scurt de ce.

Dacă un proces utilizează un timp procesor prea mare, acesta va fi mutat într-o coadă de prioritate mai mică. Aceasta lasă procesele orientate pe operații (I/O bound) în cozile cu prioritate mai mare.

24. În planificarea proceselor pe baza priorității (priority-based process scheduling), planificatorul dă controlul unui anumit proces dacă nu există procese cu prioritate mai mare în coada ready curentă. Presupuneți că nu există altă informație pentru a realiza decizia de planificare. De asemenea presupuneți că prioritățile proceselor sunt stabilite la momentul creării procesului și nu se schimbă. Într-un sistem de operare cu astfel de presupuneri, de ce soluția Dekker (see Section A.1) la problema excluderii mutuale poate deveni "periculoasă"? Explicați aceasta prin a spune ce eveniment nedorit poate să apară și cum poate să apară.

Algoritmul lui Dekker se bazează pe corectitudinea hardware-ului și a sistemului de operare. Utilizarea priorităților riscă infometarea după cum urmează. Se poate întâmpla ca P0 să fie un proces care se repetă foarte repede, și care găsește constant flag[1] = false, neputând intra în secțiunea critică, în timp ce P1, care părăsește bucla internă în care a intrat așteptând turn-ul său, nu poate seta flag-ul pe true, deoarece P0 citește prea des această variabilă.

25. Cinci job-uri dintr-un lot, de la A la E, sosesc la un centru de calcul în principiu în același timp. Acestea au un timp estimat de execuție de 15, 9, 3, 6, și respectiv 12 minute. Prioritățile lor (definite extern) sunt 6, 3, 7, 9, și respectiv 4, cu valoarea mai mică corespunzând unei priorități mai mari. Determinați pentru fiecare dintre următorii algoritmi și pentru fiecare proces timpul de intrare-iesire (turnaround time) și pentru toate job-urile timpul mediu intrare-iesire (average turnaround). Ignorați supracontrolul dat de comutarea proceselor. Explicați cum ați ajuns la răspunsul vostru. Pentru ultimi-le trei cazuri, presupuneți că doar un singur job la un moment dat se execută până se termină și că toate job-urile sunt orientate pe procesor (processor-bound).

- a. round robin with a time quantum of 1 minute
- b. priority scheduling
- c. FCFS (run in order 15, 9, 3, 6, and 12)
- d. shortest job first

Răspundeți la a.

1	2	3	4	5	Elapsed time
A	B	C	D	E	5
A	B	C	D	E	10
A	B	C	D	E	15
A	B		D	E	19
A	B		D	E	23
A	B		D	E	27
A	B			E	30
A	B			E	33
A	B			E	36
A				E	38
A				E	40
A				E	42
A					43
A					44
A					45

Turnaround time = 42 min

The average turnaround time = 32,2

26. Cinci job-uri dintr-un lot, de la A la E, sosesc la un centru de calcul în principiu în același timp. Acestea au un timp estimat de execuție de 15, 9, 3, 6, și respectiv 12 minute. Prioritățile lor (definite extern) sunt 6, 3, 7, 9, și respectiv 4, cu valoarea mai mică corespunzând unei priorități mai mari. Determinați pentru fiecare dintre următorii algoritmi și pentru fiecare proces timpul de intrare-iesire (turnaround time) și pentru toate job-urile timpul mediu intrare-iesire (average turnaround). Ignorați supracontrolul dat de comutarea proceselor. Explicați cum ați ajuns la răspunsul vostru. Pentru ultimi-le trei cazuri, presupuneți că doar un singur job la un moment dat se execută până se termină și că toate job-urile sunt orientate pe procesor (processor-bound).
- round robin with a time quantum of 1 minute
 - priority scheduling
 - FCFS (run in order 15, 9, 3, 6, and 12)
 - shortest job first
- Răspundeți la b

Priority	Job	Turnaround Time
3	B	9
4	E	$9 + 12 = 21$
6	A	$21 + 15 = 36$
7	C	$36 + 3 = 39$
9	D	$39 + 6 = 45$

The average turnaround time = 30

27. Cinci job-uri dintr-un lot, de la A la E, sosesc la un centru de calcul în principiu în același timp. Acestea au un timp estimat de execuție de 15, 9, 3, 6, și respectiv 12 minute. Prioritățile lor (definite extern) sunt 6, 3, 7, 9, și respectiv 4, cu valoarea mai mică corespunzând unei priorități mai mari. Determinați pentru fiecare dintre următorii algoritmi și pentru fiecare proces timpul de intrare-iesire (turnaround time) și pentru toate job-urile timpul mediu intrare-iesire (average turnaround). Ignorați supracontrolul dat de comutarea proceselor. Explicați cum ați ajuns la răspunsul vostru. Pentru ultimi-le trei cazuri, presupuneți că doar un singur job la un moment dat se execută până se termină și că toate job-urile sunt orientate pe procesor (processor-bound).
- round robin with a time quantum of 1 minute
 - priority scheduling
 - FCFS (run in order 15, 9, 3, 6, and 12)
 - shortest job first
- Răspundeți la c

Job	Turnaround Time
A	15
B	$15 + 9 = 24$
C	$24 + 3 = 27$

D	$27 + 6 = 33$
E	$33 + 12 = 45$

The average turnaround time = 33

28. Cinci job-uri dintr-un lot, de la A la E, sosesc la un centru de calcul în principiu în același timp. Acestea au un timp estimat de execuție de 15, 9, 3, 6, și respectiv 12 minute. Prioritățile lor (definite extern) sunt 6, 3, 7, 9, și respectiv 4, cu valoarea mai mică corespunzând unei priorități mai mari. Determinați pentru fiecare dintre următorii algoritmi și pentru fiecare proces timpul de intrare-iesire (turnaround time) și pentru toate job-urile timpul mediu intrare-iesire (average turnaround). Ignorați supracontrolul dat de comutarea proceselor. Explicați cum ați ajuns la răspunsul vostru. Pentru ultimi-le trei cazuri, presupuneți că doar un singur job la un moment dat se execută până se termină și că toate job-urile sunt orientate pe procesor (processor-bound).

- a. round robin with a time quantum of 1 minute
- b. priority scheduling
- c. FCFS (run in order 15, 9, 3, 6, and 12)
- d. shortest job first

Răspundetă la d

Running Time	Job	Turnaround Time
3	C	3
6	D	$3 + 6 = 9$
9	B	$9 + 9 = 18$
12	E	$18 + 12 = 30$
15	A	$30 + 15 = 45$

The average turnaround time = 21

Cap. 10. Planificarea multiprocesor și de timp real

1. Enumerați și descrieți succint cinci categorii diferite de granularitate ale sincronizării.

Fine: paralelism intrinsec într-un singur sir de instrucțiuni. Medium: procesare paralelă sau multitasking în cadrul unei singure aplicații. Coarse: multiprocesarea proceselor concurente într-un mediu cu multiprogramare. Very Coarse: procesarea distribuită între nodurile unei rețele pentru a forma un singur mediu de calcul. Independent: procese multiple fără interdependențe.

2. Enumerați și descrieți succint patru tehnici pentru planificarea taskurilor.

Load Sharing: Procesele nu sunt asignate unui anumit procesor. Este menținută o coadă globală cu firele gata de execuție, și fiecare procesor când ajunge în idle, delectează un fir din coadă. Termenul load sharing este utilizat pentru a distinge această strategie de schemele de tip load-balancing unde sarcinile sunt alocate pe o bază mai permanentă. Round robin scheduling: un set de fire asociate este planificat pentru a rula pe un set de procesoare în același timp, utilizând o relație de tip unul-la-unul. Dedicated processor assignment: Fiecare program este alocat unui număr de procesoare egal cu numărul firelor sale de execuție. Când se termină programul, procesoarele revin la grupul general pentru o posibilă alocare a altui program. Dynamic scheduling: Numărul de fire dintr-un program poate fi alterat pe durata execuției.

3. Enumerați și descrieți succint trei versiuni de partajare a încărcării.

First Come First Serve (FCFS): Atunci când apare un job, acesta este plasat întotdeauna la sfârșitul cozii. Când procesorul devine idle, acesta preia următorul fir gata de execuție din capul cozii, și-l execută până se termină sau se blochează. Smallest Number of Threads First: Coada ready partajată este organizată ca o coadă cu priorități, cu prioritatea cea mai mare dată firelor din job-ul cu numărul cel mai mic de fire neplanificate. Job-urile cu prioritate egală sunt ordonate în funcție de care a sosit primul. Ca și la FCFS, un fir planificat se execută până se termină sau se blochează. Preemptive Smallest Number of Threads First: Prioritatea cea mai mare este dată job-ului cu cel mai mic număr de fire neplanificate. Dacă sosște un job cu un număr de fire mai mic decât job-ul care se execută acesta va suspenda firele aparținând job-ului curent în execuție.

4. Care este diferența între hard și soft real-time tasks?

Un hard real time task (task cu constrângeri dure) trebuie să-și respecte deadline-ul (timpul maxim de execuție); altfel va cauza stricării nedoreite sau erori fatale în sistem.

Un soft real time task (task cu constrângeri ușoare) are un deadline asociat care este de dorit a fi îndeplinit dar nu este indezirabil; are încă sens de a planifica și a completa execuția taskului chiar dacă s-a depășit deadline-ul său.

5. Care este diferența între taskurile de timp real periodice și cele aperiodice?

Un task aperiodic are un deadline pentru care trebuie să pornească sau să se termine, sau poate avea o constrângere atât pentru timpul de start cât și pentru timpul de stop. În cazul unui task periodic, cerința poate fi definită ca “unul pe perioadă” sau “exact la T unități”.

6. Enumerați și descrieți succint cinci arii generale de cerințe pentru sistemele de operare de timp real.

Determinism: un sistem de operare este determinist dacă poate realiza operații la momente fixe și determinate de timp, sau într-un interval de timp predeterminat. Responsiveness: se referă la cât de mult, după recunoaștere, îi va lua sistemului de operare să servească o intrerupere. User control: utilizatorul trebuie să fie capabil să distingă între taskurile hard și soft și să specifice priorități relative pentru fiecare clasă. Un RTS poate de asemenea să permită utilizatorului să specifice unele caracteristici cum ar fi ce pagini sau procese pot fi comutate, ce procese trebuie să fie întotdeauna rezidente în memoria principală, ce algoritm să fie utilizat pentru transferul cu discul, ce drepturi are un proces în diferite zone de priorități, și alteșe. Reliability: fiabilitatea trebuie oferită într-un mod pentru a oferi continuu îndeplinirea deadline-ului. Fail-soft operation: este o caracteristică care se referă la abilitatea unui sistem de a-și întrerupe execuția (fail) într-un astfel de mod încât să prezeve cât mai multe date și capabilități posibile.

7. Enumerați și descrieți succint patru clase de algoritmi de planificare pentru timpul real.

Static table-driven approaches: aceasta realizează o analiză statică a fiabilității și planificării pentru dispecerizare. Rezultatul analizei este o planificare care determină, în timpul execuției, când trebuie un task să-și înceapă execuția. Static priority-driven preemptive approaches: din nou, are loc o analiză statică, dar planificarea nu este decisă. Mai degrabă, analiza este utilizată pentru a asigna priorități taskurilor, astfel încât se poate utiliza un planificator cu suspendare tradițional bazat priorități. Dynamic planning-based approaches: fezabilitatea este determinată în execuție (dynamic) și nu offline înainte de începerea execuției (static). Un task care sosetează este acceptat pentru execuție numai dacă este fezabilă îndeplinirea constrângerilor sale de timp. Ca rezultat al analizei fezabilității este o planificare sau un plan care se utilizează pentru a decide când să se dispecereze acest task. Dynamic best efforts approaches: nu se realizează nici o analiză de fezabilitate. Sistemul încearcă să îndeplinească toate deadline-urile și abandonează orice proces lansat în execuție pentru care s-a pierdut deadline-ul.

8. Care informație referitoare la task poate fi utilă în planificarea de timp real?
 Ready time, Starting deadline, Completion deadline, Processing time, Resource requirements, Priority, Subtask structure.
9. Considerați un set de trei taskuri periodice cu profilul de execuție dat în tabelul 10.6. Dezvoltați pentru aceste taskuri o diagramă de execuție similară aceleia din figura 10.5.

Table 10.6 Execution Profile for Problem 10.1

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	10	50
B(2)	50	10	100
•	•	•	•
•	•	•	•
•	•	•	•
C(1)	0	15	50
C(2)	50	15	100
•	•	•	•
•	•	•	•
•	•	•	•

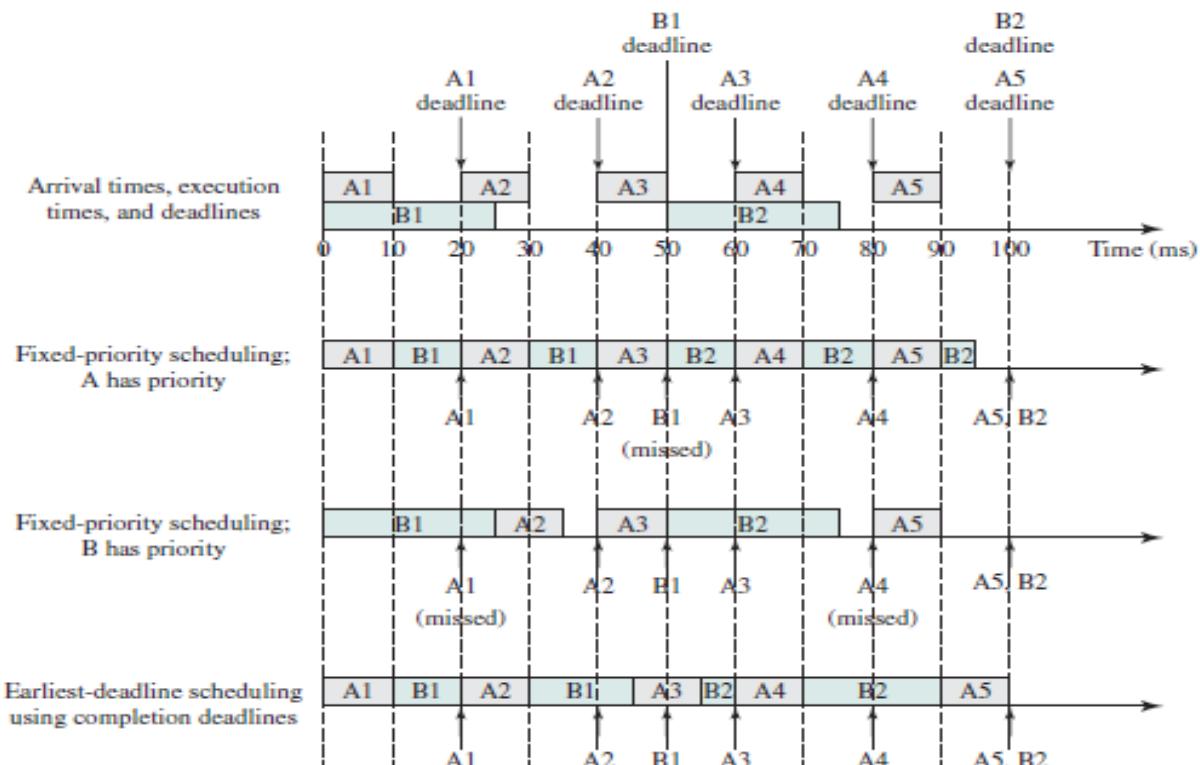


Figure 10.5 Scheduling of Periodic Real-Time Tasks with Completion Deadlines
 (Based on Table 10.2)

A	A	B	B	A	A	C	C	A	A	B	B	A	A	C	C	A	A		
A	A	B	B	A	C	C	A	C	A	A	B	B	A	A	C	C	C	A	A

Pentru planificarea cu priorități fixe, procesul C întotdeauna își pierde deadline-ul.

10. Considerați un set de cinci taskuri aperiodice cu profilul de execuție dat în tabelul 10.7. Dezvoltați pentru aceste taskuri o diagramă de execuție similară aceleia din figura 10.6.

Table 10.7 Execution Profile for Problem 10.2

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

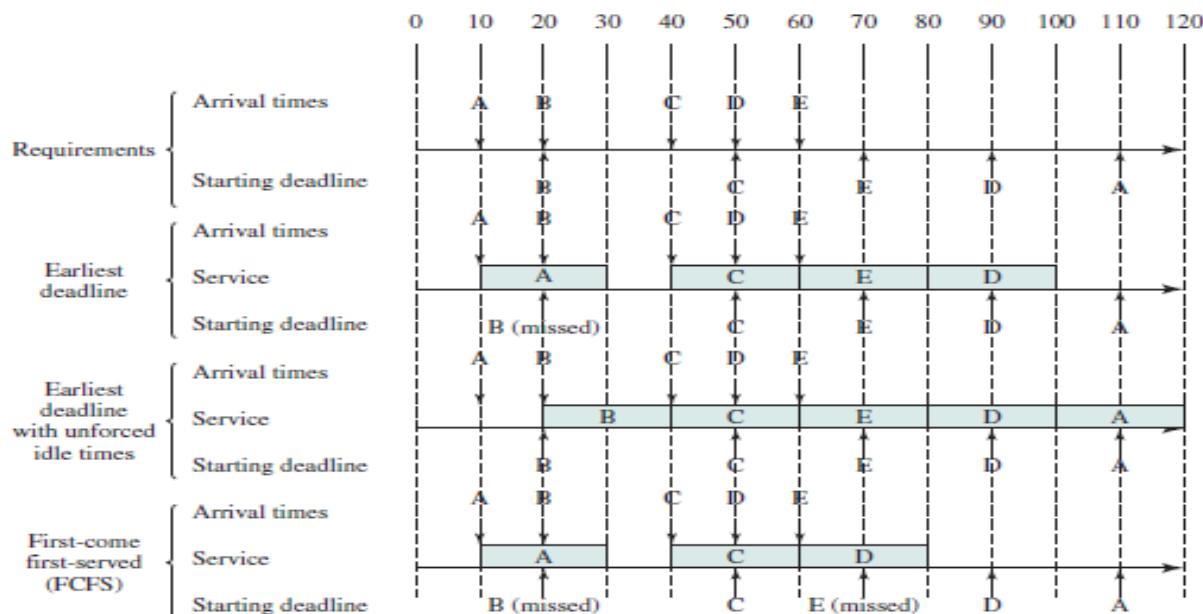


Figure 10.6 Scheduling of Aperiodic Real-Time Tasks with Starting Deadlines

Fiecare dreptunghi reprezintă 10 unități.

Earliest deadline

Earliest deadline with unforced idle times

FCFS

	A	A		C	C	E	E	D	D		
		B	B	C	C	E	E	D	D	A	A
	A	A		C	C	D	D				

11. Least laxity first (LLF) este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$

LLF selectează pentru execuția viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Presupuneți că un task are timpul de relaxare curent t. Pentru cât timp poate planificatorul întârziu pornirea acestui task astfel încât să respecte încă deadline-ul său?

Taskul poate fi întârziat până la un interval t și totuși își va menține deadline-ul.

12. Least laxity first (LLF) este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$

LLF selectează pentru execuția viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Presupuneți că un task are timpul de relaxare curent t. Presupuneți că taskul curent are timpul de relaxare 0. Ce înseamnă aceasta?

Laxitate 0 înseamnă că taskul trebuie executat acum ori va eșua în a-și îndeplini deadline-ul.

13. Least laxity first (LLF) este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$

LLF selectează pentru execuția viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Ce înseamnă un timp de relaxare negativ?

Un task cu laxitate negativă nu își poate îndeplini deadline-ul.

14. Least laxity first (LLF) este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$

LLF selectează pentru execuția viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Considerați un set de trei taskuri periodice cu profilul de execuție dat în tabelul 10.8a. Dezvoltați o diagramă de planificare similară cu aceea din figura 10.4 pentru acest set de taskuri care să compare rate monotonic, earliest deadline first, și LLF. Presupuneți că suspendarea poate să apară la intervale de 5 ms. Comentați rezultatul.

Table 10.8 Execution Profiles for Problems 10.3 through 10.6

(a) Light load		
Task	Period	Execution Time
A	6	2
B	8	2
C	12	3

(b) Heavy load		
Task	Period	Execution Time
A	6	2
B	8	5
C	12	3

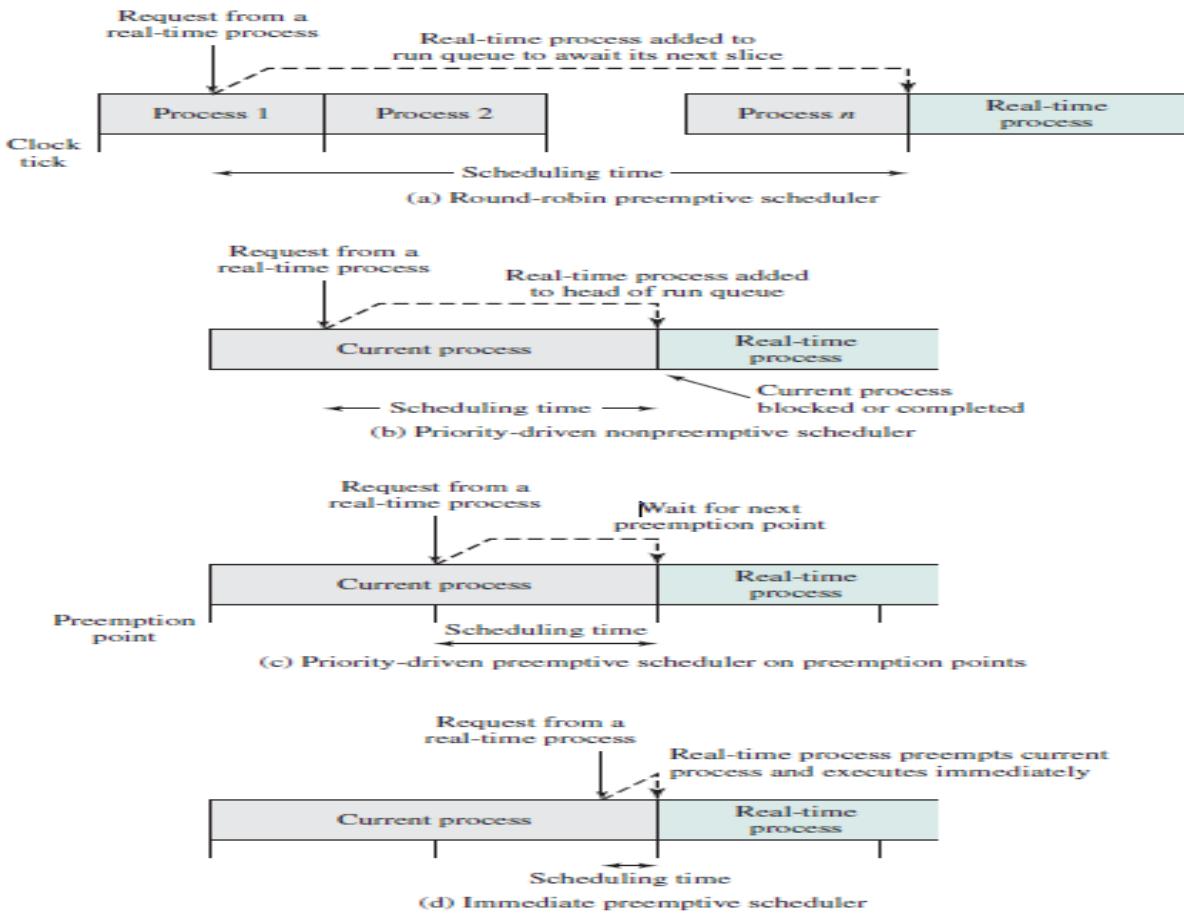
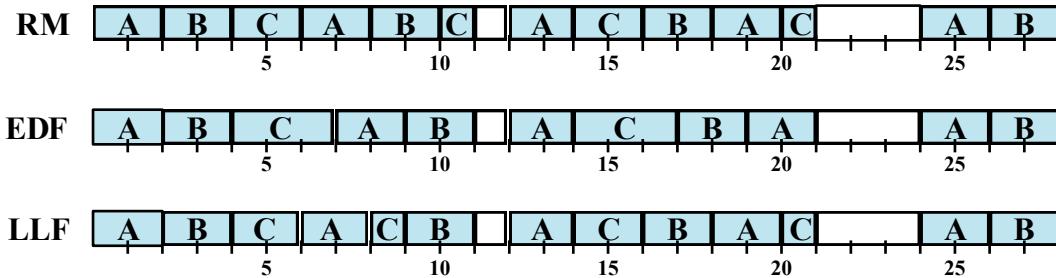


Figure 10.4 Scheduling of Real-Time Process



15. Repetați problema 10.3d (Least laxity first (LLF) este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$
 LLF selectează pentru execuție viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Considerați un set de trei taskuri periodice cu profilul de execuție dat în tabelul 10.8a. Dezvoltați o diagramă de planificare similară cu aceea din figura 10.4 pentru acest set de taskuri care să compare

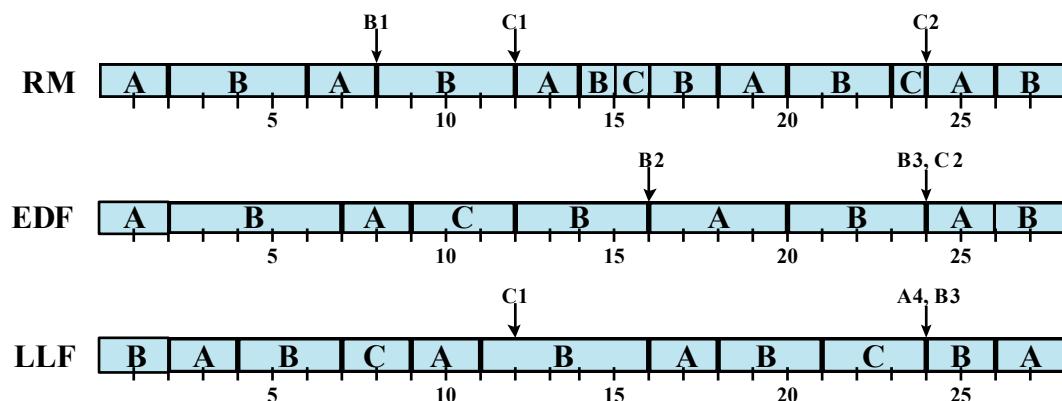
rate monotonic, earliest deadline first, și LLF. Presupuneți că suspendarea poate să apară la intervale de 5 ms. Comentați rezultatul.) pentru profilul de execuție din tabelul 10.8b . Comentați rezultatul.

Table 10.8 Execution Profiles for Problems 10.3 through 10.6

(a) Light load		
Task	Period	Execution Time
A	6	2
B	8	2
C	12	3

(b) Heavy load		
Task	Period	Execution Time
A	6	2
B	8	5
C	12	3

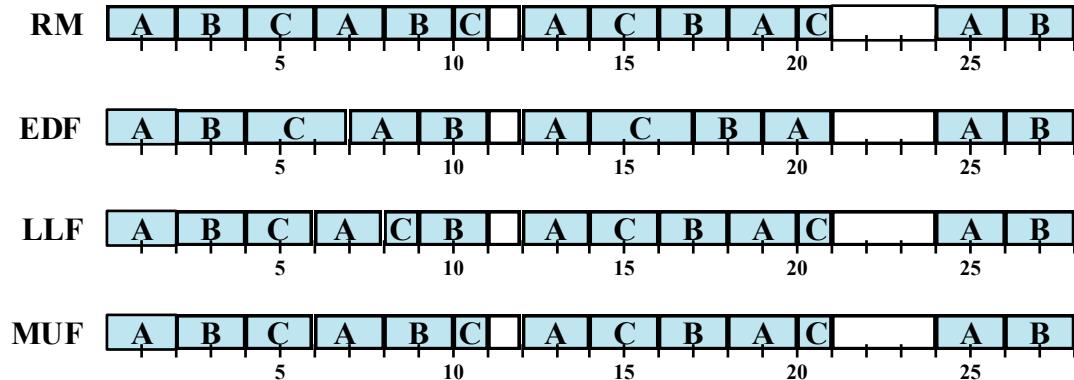
Încărcarea totală depășește acum 100 %. Niciuna dintre metode nu poate gestiona încărcarea. Taskul care va eșua în îndeplinirea deadline-ului variază în funcție de metodă.



16. Maximum urgency first (MUF) este un algoritm de planificare de timp real pentru taskurile periodice. Fiecare task are asignată o urgență care se definește ca o combinație a două priorități fixe și a uneia dinamice. Una din prioritățile fixe, starea critică (the criticality), are precedență peste prioritatea dinamică. În același timp, prioritatea dinamică are precedență peste cealaltă prioritate fixă. Prioritatea dinamică este invers proporțională cu relaxarea unui task. MUF poate fi explicat după cum urmează: Prima dat, taskurile sunt ordonate de la perioada cea mai scurtă la perioada cea mai lungă. Se definește setul critic de taskuri ca fiind primele N taskuri, astfel încât utilizarea pentru cazul cel mai defavorabil a procesorului să nu depășească 100 %. Dintre setul de taskuri critice care sunt gata de execuție planificatorul selectează taskul cu relaxarea cea mai mică. Dacă nici un set de taskuri critice nu este gata de execuție, planificatorul alege dintre taskurile non-critice pe acela cu relaxarea cea mai mică. Mai departe taskurile sunt aranjate

pentru execuție pe baza unei priorități optionale date de utilizator și mai apoi pe baza algoritmului FCFS. Repetați problema 10.3d, adăugând MUF la diagramă. Presupuneți că prioritățile definite de utilizator sunt A cea mai mare, B următoare și C cea mai mică. Comentați rezultatul.

Pe acest profil de taskuri, comportarea MUF este identică cu RM. De notat că EDF are mai puține comutări de context (11) față de celelalte care au 13.



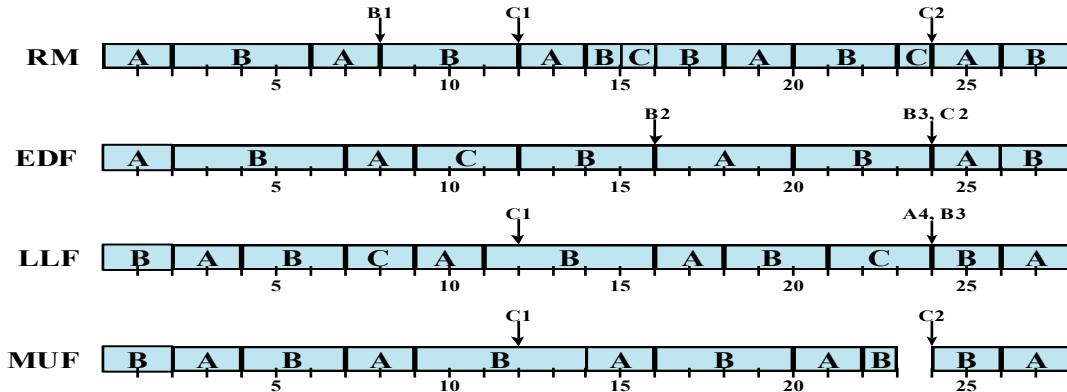
17. Repetați problema 10.4 (Repetați problema 10.3d (*Least laxity first (LLF)* este un algoritm de planificare în timp real pentru taskurile periodice. Timpul de inactivitate (Slack time) sau relaxare (laxity), este cantitatea de timp dintre momentul când un task se termină dacă ar fi pornit acum și următorul deadline. Aceasta este mărimea ferestrei disponibile pentru planificare. Relaxarea (Laxity) poate fi exprimată ca:

$$\text{Laxity} = (\text{deadline time}) - (\text{current time}) - (\text{processor time needed})$$

LLF selectează pentru execuția viitoare taskul cu timpul minim de relaxare. Dacă două sau mai multe taskuri au același timp de relaxare, atunci acestea sunt servite pe baza unui algoritm FCFS. Considerați un set de trei taskuri periodice cu profilul de execuție dat în tabelul 10.8a. Dezvoltați o diagramă de planificare similară cu aceea din figura 10.4 pentru acest set de taskuri care să compare rate monotonic, earliest deadline first, și LLF. Presupuneți că suspendarea poate să apară la intervale de 5 ms. Comentați rezultatul.) pentru profilul de execuție din tabelul 10.8b . Comentați rezultatul.

), adăugând MUF la diagramă. Comentați rezultatul.

Setul critic de taskuri conține A și B. De observat că MUF este singurul algoritm care execută setul critic la timp.



18. Această problemă demonstrează că deși ecuația (10.2) pentru planificarea rate monotonic este o condiție suficientă, nu este și o condiție necesară (se poate realiza o planificare cu succes chiar dacă nu este satisfăcută ecuația (10.2)). Considerați un set de taskuri cu următoarele taskuri periodice independente :

- Task P 1 : $C_1 = 20 ; T_1 = 100$
- Task P 2 : $C_2 = 30 ; T_2 = 145$

Pot fi planificate cu succes aceste taskuri utilizând planificarea rate monotonic?

Utilizarea totală este de 0, 41 mai mică decât marginea de planificabilitate de 0,828, deci taskurile sunt planificabile.

19. Această problemă demonstrează că deși ecuația (10.2) pentru planificarea rate monotonic este o condiție suficientă, nu este și o condiție necesară (se poate realiza o planificare cu succes chiar dacă nu este satisfăcută ecuația (10.2)). Considerați un set de taskuri cu următoarele taskuri periodice independente :

- Task P 1 : $C_1 = 20 ; T_1 = 100$
- Task P 2 : $C_2 = 30 ; T_2 = 145$

Adăugați acum următoarele taskuri la set :

- Task P 3 : $C_3 = 68 ; T_3 = 150$

Este satisfăcută ecuația (10.2)?

Utilizarea totală este de 0,86 care excede marginea de planificabilitate de 0,779, deci nu este satisfăcută.

20. Această problemă demonstrează că deși ecuația (10.2) pentru planificarea rate monotonic este o condiție suficientă, nu este și o condiție necesară (se poate realiza o planificare cu succes chiar dacă nu este satisfăcută ecuația (10.2)). Considerați un set de taskuri cu următoarele taskuri periodice independente :

- Task P 1 : $C_1 = 20 ; T_1 = 100$

- Task P 2 : C₂ = 30 ; T₂ = 145

Adăugați acum următoarele taskuri la set :

- Task P 3 : C₃ = 68 ; T₃ = 150

Presupuneți că prima instanță a celor trei taskuri sosește la momentul t = 0.

Presupuneți că deadline-urile pentru fiecare task sunt:

$$D_1 = 100; D_2 = 145; D_3 = 150$$

Utilizând planificarea rate monotonic, își vor respecta cele trei taskuri deadline-ul lor? Ce puteți spune despre deadline în cazul unor repetări ulterioare ale fiecărui task?

DA,

21. Trasați o diagramă similară cu aceea din figura 10.9b care prezintă secvența de evenimente care pentru același exemplu dar utilizând plafonul maxim (priority ceiling).

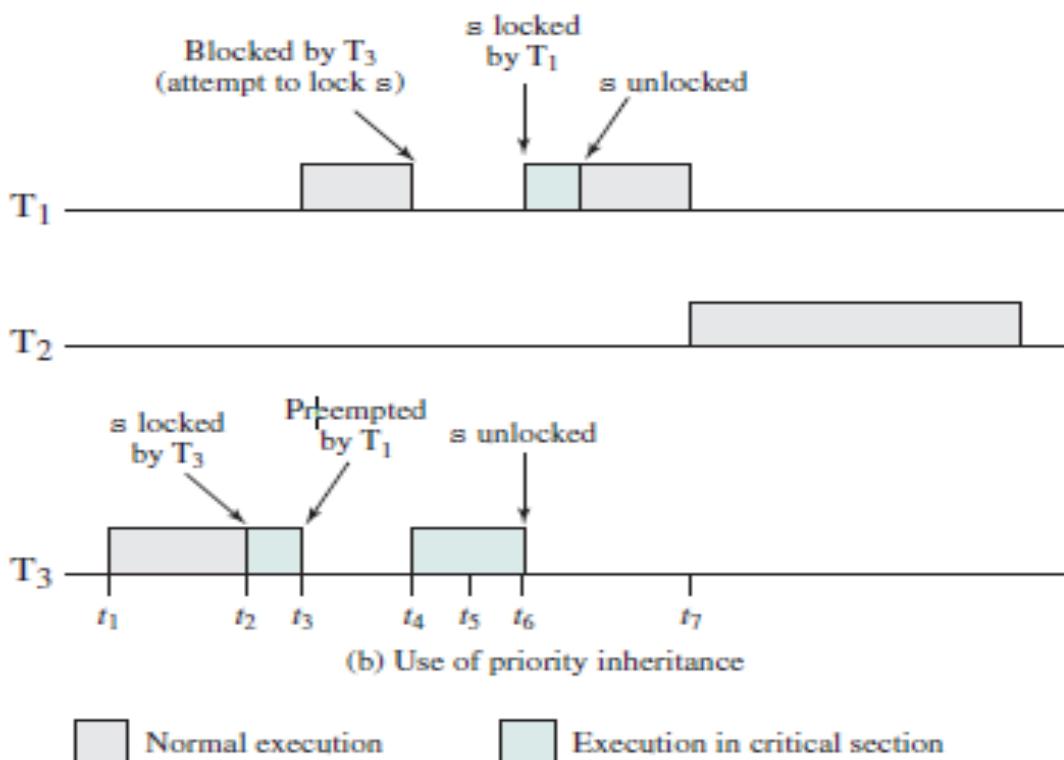
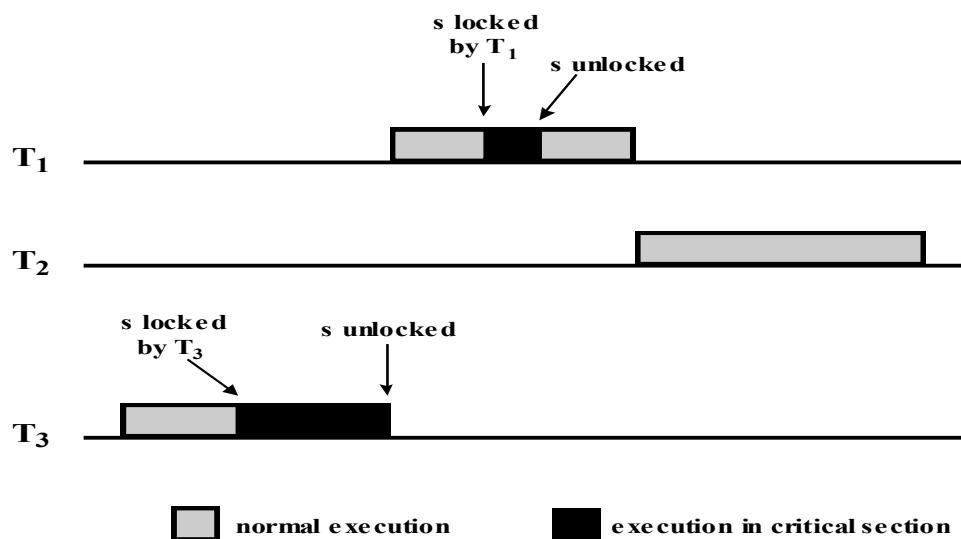


Figure 10.9 Priority Inversion

Odată ce T3 intră în secțiunea critică, îi este asignată o prioritate mai mare ca T1. Când T3 părăsește secțiunea sa critică, acest este suspendat de T1.



Cap. 11. I/O MANAGEMENT AND DISK SCHEDULING

1. Enumerați și descrieți succint trei tehnici pentru a realiza transfer I/O.

Transfer I/O programat: procesorul lansează o comandă I/O, din cadrul unui proces către un modul I/O; procesul apoi așteaptă ocupat (busy-waits) pentru ca operația să se termine înainte de a continua. Transfer I/O prin întreruperi: procesorul emite o comandă din cadrul unui proces, și continuă să execute instrucțiunile care urmează, și este întrerupt de modulul I/O când acesta, mai târziu, completează execuția comenzii. Altfel procesul este suspendat în așteptarea întreruperii și sunt realizate alte acțiuni. Accesul direct la memorie (DMA): un modul DMA controlează schimbul de date dintre memoria principală și modulul I/O. Procesorul trimite o cerere de transfer a unui bloc de date către modulul DMA și este întrerupt numai după ce a fost transferat întregul bloc.

2. Care este diferența între transfer I/O logic (logical I/O) și transfer I/O la nivel de dispozitiv (device I/O)?

Logical I/O: Modulele I/O logice lucrează cu dispozitivele sub formă de resurse logice și nu sunt preocupate cu detaliile de control al dispozitivului actual. Modulele I/O logice sunt preocupate de gestiunea funcțiilor I/O generale în folosul proceselor utilizator, permitându-le acestora să lucreze cu dispozitivele utilizând identificatori de dispozitiv și comenzi simple cum ar fi open, close, read, write. Device I/O: operațiile cerute și datele (caractere buferate, înregistrări, etc.) sunt convertite în secvențe de instrucțiuni I/O. comenzi pentru canale, și ordine pentru controlere. Tehnicile de bufferare pot fi utilizate pentru a îmbunătății utilizarea.

3. Care este diferența între dispozitivele orientate pe bloc și dispozitivele orientate pe caracter (stream)? Dați câteva exemple din fiecare.

Dispozitivele orientate pe bloc memorează informația în blocuri care de obicei sunt de dimensiune fixă, și transferurile sunt făcute de genul un bloc la un moment dat. În general este posibil să se facă referințe la date prin numărul de bloc. Discurile, benzile sunt exemple de dispozitive orientate pe bloc. Dispozitivele orientate pe caracter transferă date în și dintr-un sir de octeți, fără o structură de bloc. Terminalele, imprimantele, porturile de comunicație, mouse și alte dispozitive de localizare, și multe alte dispozitive care nu sunt pentru memorarea secundară sunt orientate pe caracter.

4. De ce vă așteptați la îmbunătățirea performanțelor atunci când utilizați dubla bufferare în raport cu simpla buferare pentru transferul I/O?

Dubla bufferare permite execuția în paralel a două operații. În mod specific un proces poate transfera date într-un (sau dintr-un) buffer în timp ce sistemul de operare golește (sau umple) un altul.

5. Ce elemente de întârziere sunt implicate în citirea sau scrierea pe disc?

Timpul de căutare, întârzierea de rotație, timpul de acces,

6. Definiți succint politicile de planificare a discului ilustrate în figura 11.7.

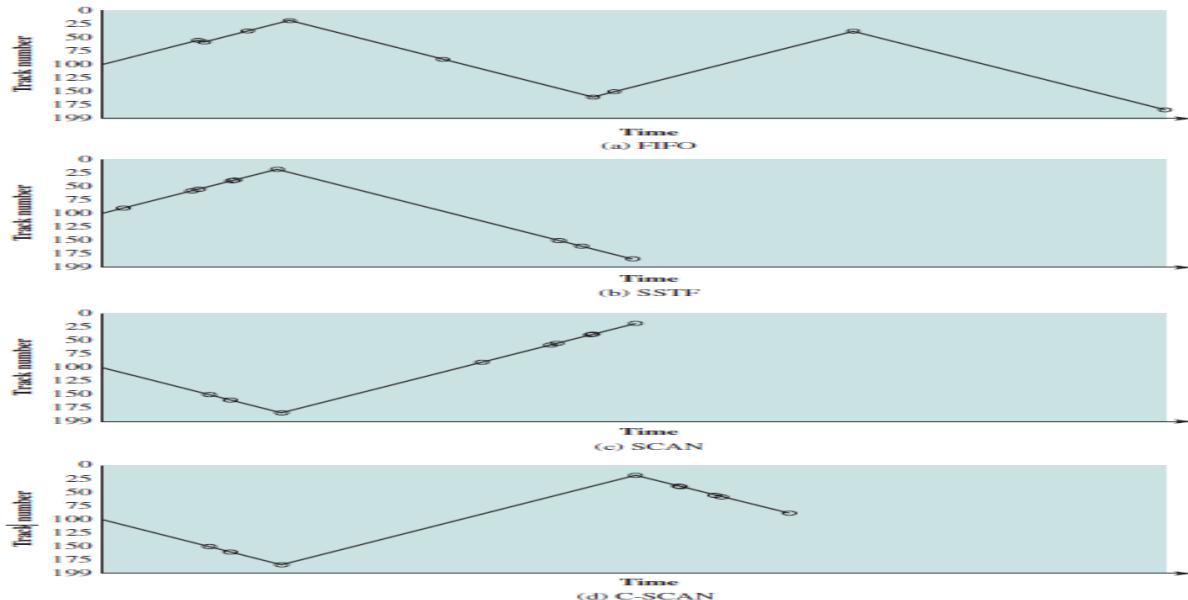


Figure 11.7 Comparison of Disk Scheduling Algorithms (see Table 11.2)

FIFO: pistele sunt procesate din coadă într-o ordine secvențială de tipul primul venit primul servit. SSTF: Selectează cererea I/O pentru disc care necesită mișcarea minimală a brațului discului de la poziția curentă. SCAN: brațul discului se mișcă doar într-o direcție, satisfăcând în drumul său toate cererile în așteptare, până atinge ultima pistă în acea direcție sau până când nu mai sunt cereri în acea direcție. Direcția servită este apoi schimbată și scanarea se realizează în direcția opusă, din nou luând cererile în ordine. C-SCAN: similar cu SCAN, dar se restricționează scanarea la o singură direcție. Astfel, când s-a vizitat ultima pistă într-o direcție, brațul se reîntoarce la capătul opus al discului și scanarea începe din nou.

7. Definiți pe scurt cele șapte niveluri RAID.

0: neredundant. 1: în oglindă; fiecare disk are un disc în oglindă conținând aceeași date. 2: redundanță via codul Hamming; este calculat un cod de corectare a erorii pentru biții corespunzători fiecarui disc de date, și biții codului sunt memorati în poziția corespunzătoare a biților pe discuri multiple de paritate. 3: bit de paritate întrețesut; similar cu nivelul 2, dar în loc de un cod de eroare, se calculează un simplu bit de paritate pentru un set de biți individuali în aceeași poziție pe toate discurile de date. 4: paritate întrețesută la nivel de bloc; o bandă cu paritatea bit-la-bit este calculată pentru benzile corespunzătoare pentru fiecare disc, și biții de paritate sunt memorati în banda corespunzătoare de pe discul de paritate. 5: paritate întrețesută la nivel de bloc și distribuită; similar cu nivelul 4 dar benzile cu bitul de paritate sunt distribuite pe toate discurile. 6: paritate întrețesută la nivel de bloc cu distribuție duală. Se calculează două parități diferite care sunt memorate în blocuri separate pe diferite discuri.

8. Care este mărimea tipică a unui sector?

512 octeți

9. Considerați un program care accesează un singur dispozitiv I/O și comparați transferul I/O fără bufferare cu cel cu bufferare. Arătați că utilizarea bufferului reduce timpul de execuție de cel puțin două ori.

Dacă timpul de calcul este egal cu timpul I/O (situația cea mai favorabilă) atât procesorul cât și dispozitivul periferic vor rula jumătate din timpul pe care l-ar consuma dacă ar rula separat. Fie C timpul total pentru întregul program și T timpul total cerut de operațiile I/O. Atunci timpul de execuție cel mai bun posibil cu bufferare este max (C,T), în timp ce timpul de execuție fără bufferare este C + T; rezultă $((C + T)/2) \leq \max(C, T) \leq (C + T)$.

10. Generalizați rezultatul problemei 11.1 (*Considerați un program care accesează un singur dispozitiv I/O și comparați transferul I/O fără bufferare cu cel cu bufferare. Arătați că utilizarea bufferului reduce timpul de execuție de cel puțin două ori.*) la cazul în care un program utilizează n dispozitive.

Raportul cel mai bun este $(n+1):n$.

11. Realizați același tip de analiză ca în tabelul 11.2 pentru următoarea cerere de piste de disc : 27, 129, 110, 186, 147, 41, 10, 64, 120. Presupuneți că, capetele discului sunt poziționate inițial pe pista 100 și se mișcă inițial în direcția unei piste cu un număr de pistă descrescător.

Table 11.2 Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

a.

FIFO		SSTF		SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
27	73	110	10	64	36	64	36
129	102	120	10	41	23	41	23
110	19	129	9	27	14	27	14
186	76	147	18	10	17	10	17
147	39	186	39	110	100	186	176
41	106	64	122	120	10	147	39
10	31	41	23	129	9	129	18
64	54	27	14	147	18	120	9
120	56	10	17	186	39	110	10
Average	61.8	Average	29.1	Average	29.6	Average	38

12. Realizați același tip de analiză ca în tabelul 11.2 pentru următoarea cerere de piste de disc : 27, 129, 110, 186, 147, 41, 10, 64, 120. Presupuneți că, capetele discului sunt poziționate inițial pe pista 100 și se mișcă inițial în direcția unei piste cu un număr de pistă crescător.

Table 11.2 Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

FIFO		SSTF		SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
27	73	110	10	110	10	110	10
129	102	120	10	120	10	120	10
110	19	129	9	129	9	129	9
186	76	147	18	147	18	147	18
147	39	186	39	186	139	186	39
41	106	64	122	64	122	10	176
10	31	41	23	41	23	27	17
64	54	27	14	27	14	41	14
120	56	10	17	10	17	64	23
Average	61.8	Average	29.1	Average	29.1	Average	35.1

13. Considerați un disc cu N piste numerotate de la 0 la N-1 și presupuneți că sectoarele cerute sunt distribuite aleatoriu și uniform pe disc. Dorim să calculăm numărul mediu de piste traversate de căutare. Calculați probabilitatea unei căutări de lungime j, capetele fiind poziționate pe pista t. (Sugestie: Scopul este de a calcula numărul total de combinații, recunoscând că toate pozițiile pistelor ca destinație pentru căutare sunt egale ca șanse.)

$$Ps[j / t] = \frac{1}{N} \quad \text{if} \quad t \leq j - 1 \quad \text{OR} \quad t \geq N - j$$

$$Ps[j / t] = \frac{2}{N} \quad \text{if} \quad j - 1 < t < N - j$$

14. Considerați un disc cu N piste numerotate de la 0 la N-1 și presupuneți că sectoarele cerute sunt distribuite aleatoriu și uniform pe disc. Dorim să calculăm numărul mediu de piste traversate de căutare.

Calculați probabilitatea unei căutări de lungime K, pentru o poziție curentă a capetelor arbitrară. (Sugestie: Aceasta implică sumarea tuturor combinațiilor posibile ale mutărilor pentru K piste.)

$$Ps[K] = 2/N^2 \times (N - K).$$

15. Considerați un disc cu N piste numerotate de la 0 la N-1 și presupuneți că sectoarele cerute sunt distribuite aleatoriu și uniform pe disc. Dorim să calculăm numărul mediu de piste traversate de căutare. Calculați numărul mediu de piste traversate de o căutare utilizând formula pentru valoarea așteptată (căutări de lungime K).

$$E[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

Hint: Use the equalities $\sum_{i=1}^n i = \frac{n(n + 1)}{2}$; $\sum_{i=1}^n i^2 = \frac{n(n + 1)(2n + 1)}{6}$

$$E[K] = (N^2 - 1)/3N$$

16. Considerați un disc cu N piste numerotate de la 0 la N-1 și presupuneți că sectoarele cerute sunt distribuite aleatoriu și uniform pe disc. Dorim să calculăm numărul mediu de piste traversate de căutare. Arătați că pentru valori mari ale lui N, numărul mediu de piste traversate de o căutare tinde la N/3 (căutări de lungime K)

$$\text{Rezultă din relația } E[K] = (N^2 - 1)/3N$$

17. Următoarea ecuație a fost sugerată atât pentru memoria cache a procesorului cât și pentru memoria cache a discului:

$$TS = TC + M * TD$$

Generalizați această ecuație pentru o ierarhie de memorie cu N niveluri în loc de doar două niveluri.

$$T_s = T_1 + \sum_{L=2}^N \sum_{j=1}^{L-1} B_j H_i$$

18. Pentru algoritmul frequency-based replacement algorithm (Figura 11.9), definiți F_{new} , F_{middle} , și F_{old} ca fracție a cache-ului care cuprinde secțiunile new, middle, și respectiv old. În mod clar $F_{new} + F_{middle} + F_{old} = 1$. Caracterizați politica atunci când $F_{old} = 1 - F_{new}$

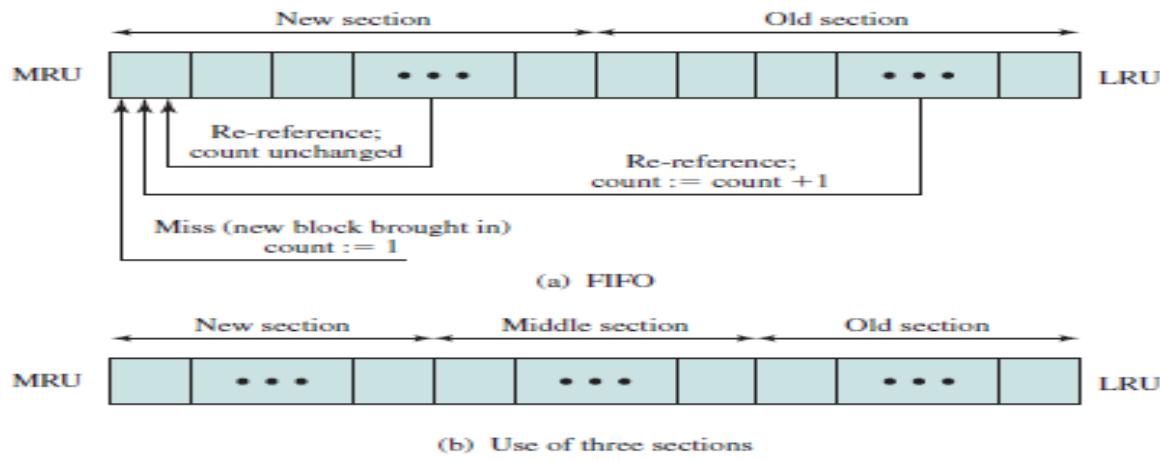


Figure 11.9 Frequency-Based Replacement

- Secțiunea din mijloc din figura 11.9b este goală. Astfel, aceasta se reduce la strategia din figura 11.9a.
- Vechea secțiune constă dintr-un bloc, și avem politica LRU de înlocuire.
- FIFO.

19. Pentru algoritmul frequency-based replacement algorithm (Figura 11.9), definiți Fnew , Fmiddle, și Fold ca fracție a cache-ului care cuprinde sectiunile new, middle, și respectiv old. În mod clar $F_{new} + F_{middle} + F_{old} = 1$. Caracterizați politica atunci când $F_{old} = 1/(cache\ size)$.

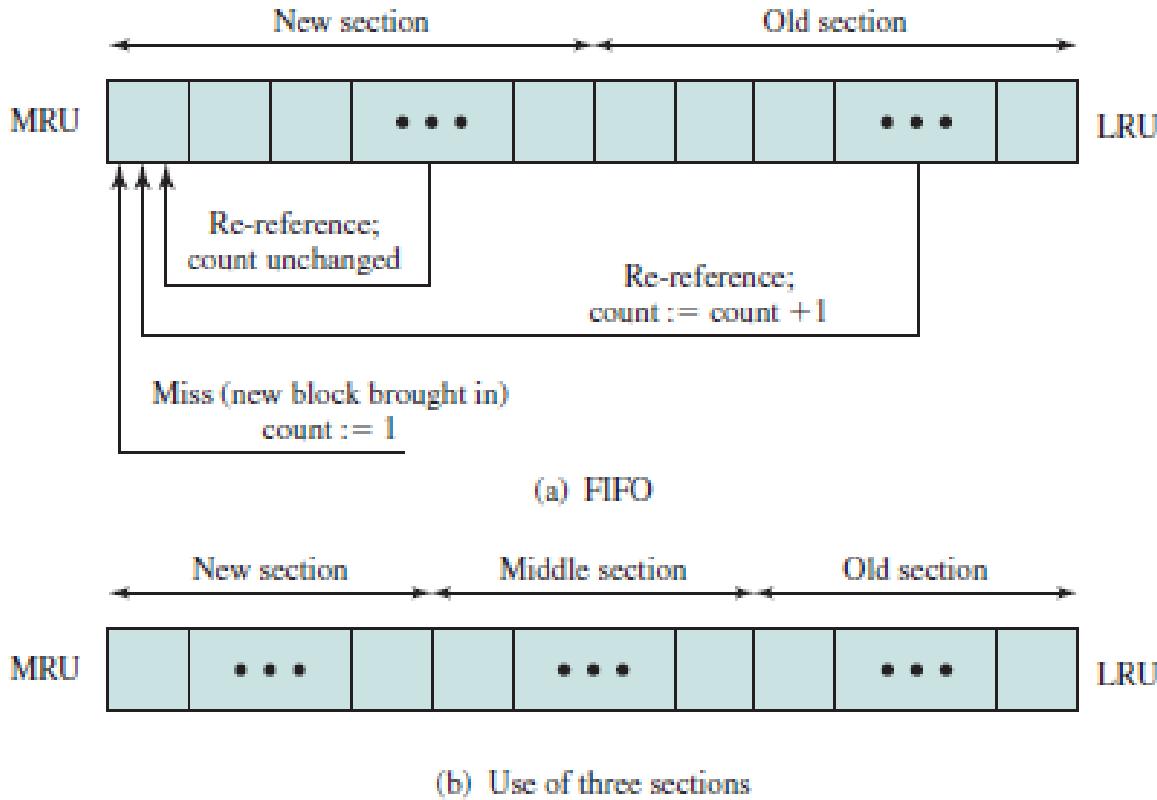


Figure 11.9 Frequency-Based Replacement

Vechea sectiune constă dintr-un bloc, și avem politica LRU de înlocuire.

20. Calculați cât spațiu de disc (în sectoare, piste și suprafață) va fi necesară pentru a memora 300000 de înregistrări logice a căte 120 de octeți fiecare, dacă diacul are sectoare fixe de mărime 512 octeți/ sector și 96 de sectoare pe pistă, 110 piste pe suprafață, și 8 suprafete utilizabile. Ignorați orice înregistrare legată de antetul fișierului și indecșii pistelor, și presupuneți că înregistrările nu pot să se regăsească în două sectoare.

75000 sectoare, 782 piste, 8 suprafete.

21. Considerați discul descris în problema 11.7, și presupuneți că discul se rotește cu 360 rpm. Un procesor citește un sector de pe disc utilizând transferul I/O bazat pe întreruperi, cu o întrerupere pe octet. Dacă îl ia 2,5 μ s pentru a procesa fiecare întrerupere, ce procent de timp va petrece procesorul gestionând I/O (nu se ia în considerare timpul de căutare)?

74%

22. Repetați problema precedentă (Considerați discul descris în problema 11.7, și presupuneți că discul se rotește cu 360 rpm. Un procesor citește un sector de pe disc utilizând transferul I/O bazat pe întreruperi, cu o întrerupere pe octet. Dacă îl ia 2,5 μ s pentru a procesa fiecare întrerupere, ce procent de timp va petrece procesorul gestionând I/O (nu se ia în considerare timpul de căutare)?) utilizând DMA, și presupuneți că se generează o singură întrerupere pe sector.

0,14 %

23. Un calculator pe 32 de biți are două canale de sector și un canal de multiplexor. Fiecare canal selector suportă două unități de discuri magnetice și două de bandă. Canalul multiplexor are conectate la el două imprimante la nivel de linie, două cititoare de cartele, și 10 terminale VDT. Presupuneți următoarele viteze de transfer:

- Disk drive 800 Kbytes/s
- Magnetic tape drive 200 Kbytes/s
- Line printer 6.6 Kbytes/s
- Card reader 1.2 Kbytes/s
- VDT 1 Kbyte/s

Estimați viteză agregată maximă de transfer I/O în acest sistem.

1625.6 KBytes/sec

24. Poate fi clar că partitōnarea discului în benzi (disk striping) poate îmbunătăti viteza de transfer atunci când mărimea benzii (strip size) este mică în comparație cu mărimea cererii I/O. Este de asemenea clar că RAID 0 oferă o îmbunătățire a performanțelor relativ la un singur disk mare, deoarece pot fi gestionate în paralel cereri I/O multiple. Totuși, în acest din urmă caz este necesară partitōnarea discului în benzi? Dacă da, îmbunătățește partitōnarea discului în benzi (disk striping) performanțele privind viteza cererilor I/O în comparație cu un disc fără benzi (without striping)?

Depinde de natura şablonului pentru cererea I/O. La o extremă dacă un singur proces realizează operații I/O și singur realizează o operație I/O mare la un moment dat, atunci partajarea în benzi a discului va îmbunătății performanțele. Dacă sunt multe procese ce realizează operații I/O mici, atunci un disc fără benzi va da performanțe comparabile cu RAID 0.

25. Considerați o arie RAID cu 4 drivere, fiecare având 200 GB-per-drive. Care este capacitatea de memorare disponibilă pentru nivelurile RAID 0, 1, 3, 4, 5, și 6?

- | | |
|----------------|----------------|
| RAID 0: 800 GB | RAID 4: 600 GB |
| RAID 1: 400 GB | RAID 5: 600 GB |
| RAID 3: 600 GB | RAID 6: 400 GB |

Cap 12 – File management

1. Care este diferența între un câmp și o înregistrare?

Un câmp este un element de bază privind datele ce conține o singură valoare. O înregistrare este o colecție de câmpuri interactive care pot fi tratate ca o unitate de unele programe aplicație.

2. Care este diferența între un fișier și o bază de date?

Un fișier este o colecție de înregistrări similare, și este tratat ca o entitate singulară de către utilizatori și aplicații și poate fi referit (accesat) prin nume. O bază de date este o colecție de date interactive (asociate). Aspectul esențial al unei baze de date sunt relațiile care există între elemente care sunt explicate și că baza de date este proiectată pentru a fi utilizată de mai multe aplicații diferite.

3. Ce este un sistem de gestiune a fișierelor?

Un sistem de gestiune a fișierelor este acel set de software sistem care furnizează servicii utilizatorilor și aplicațiilor pentru utilizarea fișierelor.

4. Ce criterii sunt importante în alegerea organizării fișierelor?

Accesul rapid, actualizarea ușoară, economia memorării, întreținerea simplă, fiabilitatea.

5. List and briefly define five file organizations.

Pile: datele sunt colectate în ordinea în care sosesc. Fiecare înregistrare constă dintr-o salvă de date. Fișiere secvențiale: se utilizează un format fix pentru înregistrări. Toate înregistrările sunt de aceeași lungime, constând din același număr de câmpuri de lungime fixă având o anumită ordine. Deoarece lungimea și poziția fiecărei înregistrări este cunoscută, trebuie memorată numai valorile câmpurilor; numele câmpurilor și lungimea pentru fiecare sunt atribuite ale structurii fișierului. Fișier indexat secvențial: un astfel de fișier menține chei caracteristice fișierului secvențial: înregistrările sunt organizate în secvențe bazate pe un câmp cheie. Sunt adăugate două facilități; un index la fișier pentru a suporta accesul aleatoriu, și un fișier de depășire (overflow file). Indexul furnizează posibilitatea de căutare pentru a atinge rapid vecinătatea înregistrării dorite. Fișierul de depășire este similar cu fișierul log utilizat de fișierele secvențiale, dar este integrat astfel încât înregistrările din fișierul de depășire sunt localizate urmând un pointer din înregistrarea precedentă. Fișier indexat: înregistrările sunt accesate doar prin indexul lor. Rezultatul este că nu sunt restricții privind plasarea înregistrărilor cât timp pointerul din cel puțin un index se referă la acea înregistrare. Mai mult, se pot

utiliza înregistrări de lungime variabilă. Fișiere directe sau hashed: Aceste fișiere utilizează hashing pe valoarea cheii.

6. De ce timpul mediu de căutare pentru a găsi o înregistrare într-un fișier este mai mic pentru un fișier indexat secvențial decât pentru un fișier secvențial?

Într-un fișier secvențial, o căutare poate implica o testare secvențială a fiecărei înregistrări până se găsește una la care se potrivește cheia. Fișierele secvențiale indexate furnizează o structură care permite o căutare mai puțin extinsă.

7. Care sunt operațiile tipice care se pot realiza pe un director?

Căutare, creare, ștergere, listare, actualizare.

8. Care este relația între o cale cu nume și un director de lucru

Calea de nume este o enumerare explicită a căii prin structura arborescentă a directorului către un anumit punct din director. Directorul de lucru este directorul cu acea structură arborescentă încărcată care aparține directorului curent în care lucrează utilizatorul.

9. Care sunt drepturile de acces tipice care pot permite sau anula ca un utilizator anume să aibă acces la un anume fișier?

Nici unul, are cunoștință de (knowledge of), read, write, execute, schimbă protecția, șterge.

10. Enumerați și descrieți pe scurt trei metode de blocare.

Blocare fixă: sunt utilizate înregistrări de lungime fixă, și într-un bloc sunt stocate un număr integral de înregistrări. Pot exista spații neutilizate la sfârșitul blocului. Blocare cu acoperire cu lungime variabilă: sunt utilizate înregistrări cu lungime variabilă care sunt împachetate în blocuri fără spațiu neutilizat. Astfel unele înregistrări pot acoperi două blocuri, cu continuarea indicată de un pointer la blocul successor. Blocare cu lungime variabilă fără acoperire: sunt utilizate înregistrări de lungime variabilă dar acoperirea nu este implementată. Există mult spațiu irosit în majoritatea blocurilor datorită incapacității de a folosi zona nefolositoare a unui bloc dacă următoarea înregistrare este mai mare decât spațiul neutilizat rămas.

11. Enumerați și descrieți pe scurt trei metode de alocare a fișierelor.

Alocare continua; alocare înlănțuită, alocare indexată.

12. Se definește :

B block size

R record size

P size of block pointer

F blocking factor; numărul de înregistrări așteptate dintr-un bloc.

Dăți o formulă pentru F pentru cele trei metode de blocare din figura 12.8 .

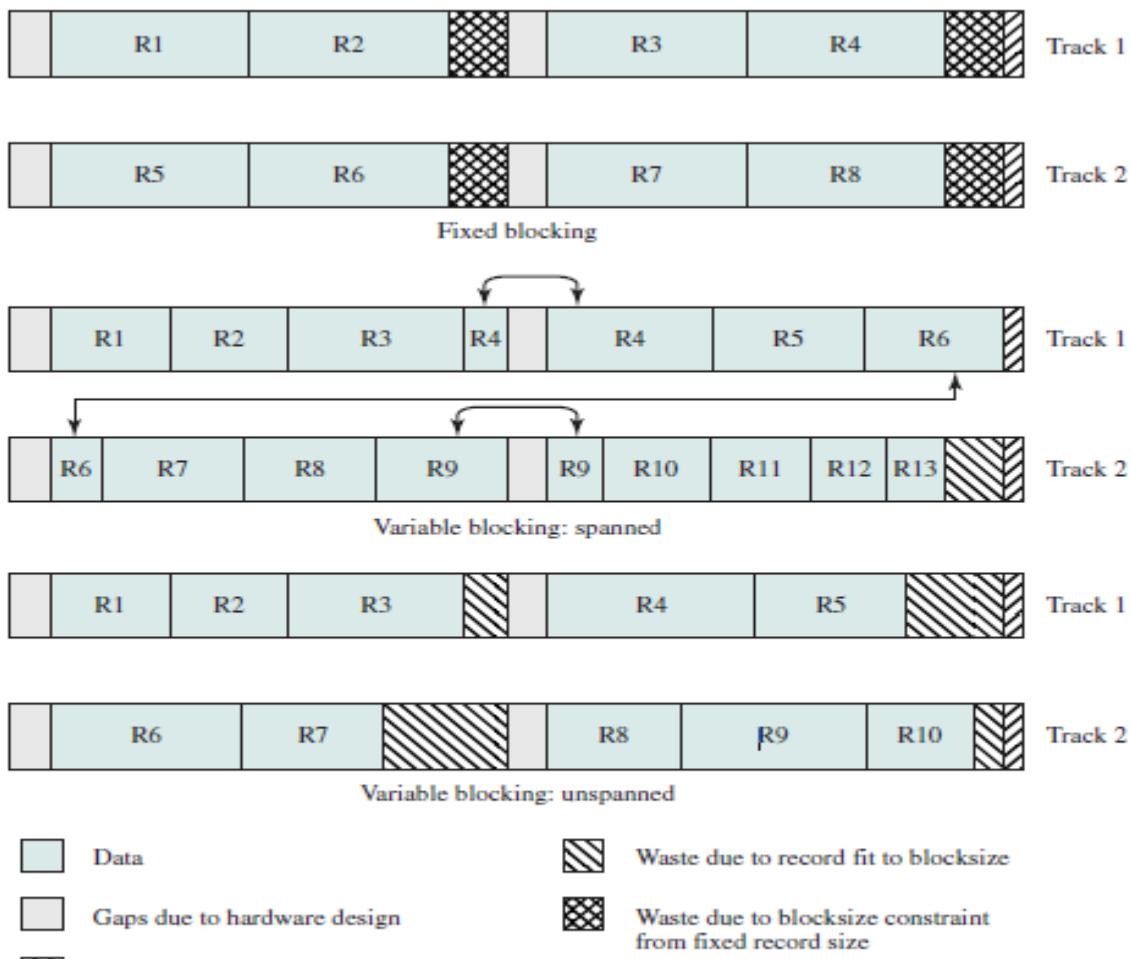


Figure 12.8 Record Blocking Methods [WIED87]

Blocare fixă: $F = \text{cel mai mare întreg } \leq B/R$; blocare cu acoperire cu lungime variabilă: $F = (B-P)/(R+P)$; blocare fără acoperire de lungime variabilă: $F = (B - R/2)/R$.

13. Una dintre schemele care evită problema preallocării versus lipsa și degradarea continuității este aceea de a aloca porțiuni de mărime crescătoare pe măsură ce mărimea fișierului crește. De exemplu, începând cu o porțiune din mărimea unui bloc, și dublarea mărimiții porțiunii pentru fiecare alocare. Considerați un fișier cu n înregistrări cu un factor F de blocare, și presupuneți că se utilizează un tabel de alocare a fișierului cu un singur nivel de indexare. Dați limita superioară a numărului de intrări în tabelul de alocare a fișierului ca o funcție de F și n.

Raspuns: $\log_2 N/F$

14. Una dintre schemele care evită problema preallocării versus lipsa și degradarea continuității este aceea de a aloca porțiuni de mărime crescătoare pe măsură ce mărimea fișierului crește. De exemplu, începând cu o porțiune din mărimea unui bloc, și dublarea mărimiilor porțiunii pentru fiecare alocare. Considerați un fișier cu n înregistrări cu un factor F de blocare, și presupuneți că se utilizează un tabel de alocare a fișierului cu un singur nivel de indexare. Care este cantitatea maximă din spațiul alocat fișierului care este neutilizată la orice moment de timp?

Mai puțin de jumătate din spațiul din spațiul alocat fișierului este neutilizat în orice moment.

15. Ce organizare de fișier ați alege pentru a maximiza eficiența din punctul de vedere al viteze de acces, utilizarea spațiului de memorare, și actualizarea ușoară (adăugare, ștergere, modificare) când datele sunt actualizate infrecvent și accesate frecvent într-o ordine aleatoare?

Indexat.

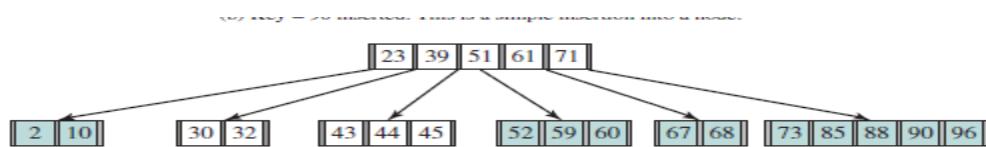
16. Ce organizare de fișier ați alege pentru a maximiza eficiența din punctul de vedere al viteze de acces, utilizarea spațiului de memorare, și actualizarea ușoară (adăugare, ștergere, modificare) când datele sunt actualizate frecvent și accesate în întregime aproape frecvent?

Indexat secvențial.

17. Ce organizare de fișier ați alege pentru a maximiza eficiența din punctul de vedere al viteze de acces, utilizarea spațiului de memorare, și actualizarea ușoară (adăugare, ștergere, modificare) când datele sunt cu actualizare frecventă și accesate frecvent într-o ordine aleatoare?

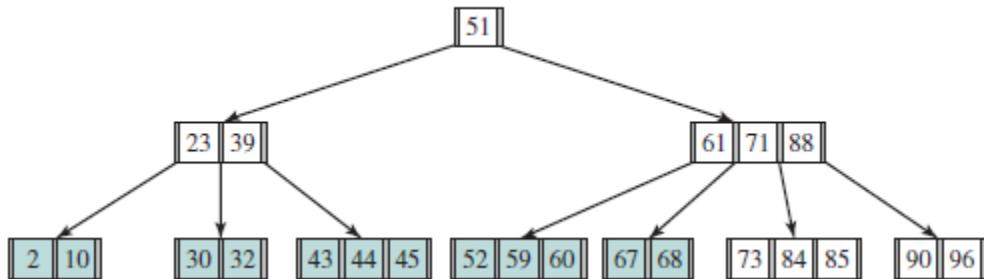
Hashed sau indexat,

18. Pentru B-tree din figura 12.4c , arătați rezultatul inserării cheii 97.



(c) Key = 45 inserted. This requires splitting a node into two parts and promoting one key to the root node.

Rezultatul este identic ca cel din figura 12.5 d, exceptând următoarele: la nivelul cel mai de jos. Blocurile cele mai din dreapta conțin 90,96, 97. Blocul de la stânga blocului cel mai din dreapta conține 73, 85.



19. Un algoritm alternativ pentru inserarea într-un arbore binar (B-tree) este următorul: îndată ce algoritmul de inserție traversează în jos arborele, fiecare nod plin care este întâlnit este imediat împărțit, chiar dacă acea împărțire nu este necesară. Care este avantajul acestei tehnici?

Această tehnică ne permite să inserăm chei într-un B-tree într-un singur pas în jos arborului de la rădăcină la frunze. Aceasta salvează timp pentru operația curentă.

20. Un algoritm alternativ pentru inserarea într-un arbore binar (B-tree) este următorul: îndată ce algoritmul de inserție traversează în jos arborele, fiecare nod plin care este întâlnit este imediat împărțit, chiar dacă acea împărțire nu este necesară. Care este dezavantajul?

Principalul dezavantaj este că arborele este subdivizat și, potențial, sunt adăugate noi niveluri la arbore care nu sunt necesare. Aceasta poate încetini viitoarele căutări.

21. Atât timpul de căutare cât și cel de inserție într-un arbore binar (B-tree) sunt funcții de greutatea nodului. Dorim să dezvoltăm o măsură a cazului de căutare cel mai defavorabil pentru timpul de inserție . Considerați un arbore binar de gradul d care conține un total de n chei. Dezvoltați o inegalitate care să dea o margine superioară a greutății h a arborelui ca o funcție de n și d.

$$n \geq 2d^h - 1.$$

22. Considerați un sistem de fișiere pentru care fișierele sunt memorate în blocuri de 16 K octeși, ignorând informația dată de supracontrolul pentru directoare și descriptorii de fișiere. Calculați procentul de spațiu irosit de fișier datorită umplerii incomplete a ultimului bloc pentru următoarele mărimi de fișiere : 41,600 octeți; 640,000 octeți; 4.064,000 octeți.

File size	41,600 bytes	640,000 bytes	4,064,000 bytes
No. of blocks	3 (rounded up to whole number)	40	249
Total capacity	$3 \times 16K = 48K = 49,152$ bytes	$40 \times 16K = 640K = 655,360$ bytes	$249 \times 16K = 3984K = 4,079,616$ bytes
Wasted space	7,552 bytes	15,360 bytes	15,616 bytes
% of wasted space	15.36%	2.34%	0.38%

23. Care sunt avantajele utilizării directoarelor?

Directoarele fac ca fișierele să fie organizate și clar separate pe bază de proprietar, aplicație ori alte criterii. Aceasta îmbunătățește securitatea, integritatea și evită problemele date de confruntarea de nume.

24. Directoarele pot fi implementate fie ca fișiere speciale care pot fi accesate fie în mod limitat fie ca fișiere ordinare de date. Care sunt avantajele și dezavantajele fiecărei abordări?

În mod clar, securitatea este mult mai ușor de întărit dacă directoarele sunt ușor de recunoscut ca fișiere speciale de către sistemul de operare. Tratarea directoarelor ca fișiere ordinare având asignate doar câteva restricții de acces oferă un set de obiecte mult mai uniform pentru a fi gestionat de sistemul de operare și permite crearea și gestiunea mai ușoară a directoarelor utilizator.

25. Unele sisteme de operare au un sistem de fișiere cu organizare ierarhică arborescentă dar care limitează adâncimea arborului la un număr mic de niveluri. Ce efect are această limitare asupra utilizatorilor? Cum simplifică acest aspect sistemul de fișiere (dacă o face) ?

Aceasta este o trăsătură rară. Dacă sistemului de operare structurează sistemul de fișiere astfel încât sunt permise subdirectoare sub un director master, nu există o logică sau are puțină logică definirea unei adâncimi arbitrară a subdirectoarelor. Limitarea adâncimii în arborele cu subdirectoare aduce o limitare care nu este necesară în organizarea spațiului utilizator.

26. Considerați un sistem ierarhic de fișiere în care spațiul liber de pe disc este ținut de o listă a spațiilor libere. Presupuneți că s-a pierdut pointerul la spațiul liber. Poate sistemul să reconstruiască lista cu spațiul liber?

Da

27. Considerați un sistem ierarhic de fișiere în care spațiul liber de pe disc este ținut de o listă a spațiilor libere. Sugerați o schemă care să asigure că pointerul nu este niciodată pierdut ca rezultat al unei singure căderi de memorie.

Păstrați un backup a listeii cu pointerii la spațiile libere pe unul sau mai multe locuri pe disc. De fiecare dată când lista se schimbă, vor trebui actualizați și pointerii din backup. Aceasta va asigura că se va putea găsi în totdeauna o valoare validă a pointerului chiar dacă este o cădere a blocului în memorie sau pe disc.

28. În sistemul V din UNIX, lungimea unui bloc este de 1 Kocet, și fiecare bloc poate memora 256 de intrări pentru adrese de blocuri. Utilizând schema inode, care este mărimea maximă a unui fișier?

Level	Number of Blocks	Number of Bytes
Direct	10	10K
Single indirect	256	256K
Double indirect	$256 \times 256 = 64K$	64M
Triple indirect	$256 \times 64K = 64M$	16G

29. Considerați organizarea unui fișier UNIX așa cum este reprezentată prin inode (figura 12.16). Presupuneți că sunt 12 pointeri direcți, și câte un pointer indirect pentru indexarea simplă, dublă și triplă în fiecare inode. Mai mult presupuneți că mărimea unui bloc sistem este de 8 Kocteți și mărimea unui sector pe disc are 8K. Dacă pointerul pentru blocurile pe disc are 32 de biți, cu 8 biți care identifică discul fizic și 24 de biți care identifică blocul fizic, atunci: care este mărimea maximă a unui fișier suportată de acest sistem?

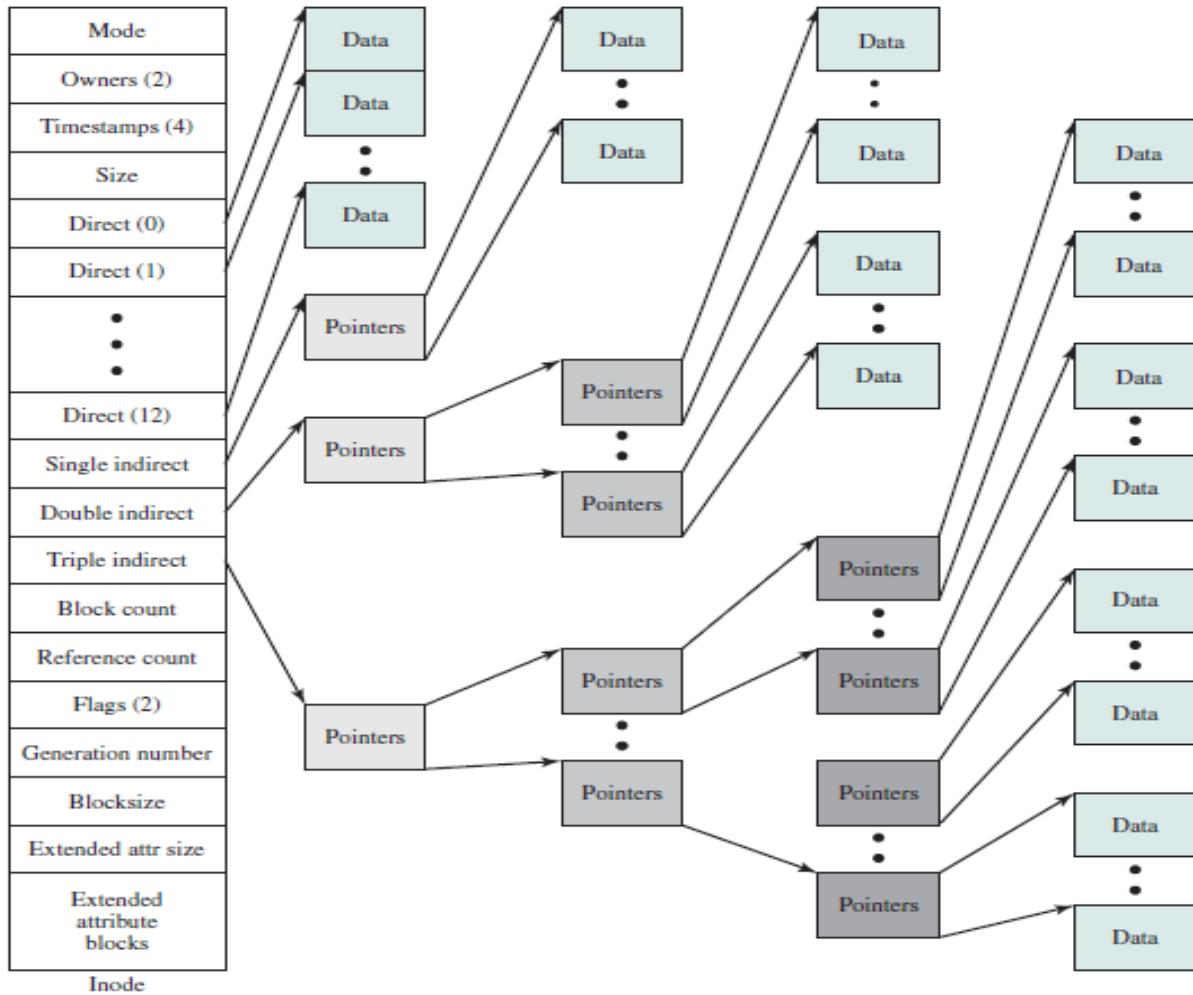


Figure 12.16 Structure of FreeBSD Inode and File

96KB + 16MB + 32MB + 64TB.

30. Considerați organizarea unui fișier UNIX aşa cum este reprezentată prin inode (figura 12.16). Presupuneți că sunt 12 pointeri directi, și câte un pointer indirect pentru indexarea simplă, dublă și triplă în fiecare inode. Mai mult presupuneți că mărimea unui bloc sistem este de 8 Kocteți și mărimea unui sector pe disc are 8K. Dacă pointerul pentru blocurile pe disc are 32 de biți, cu 8 biți care identifică discul fizic și 24 de biți care identifică blocul fizic, atunci: Care este mărimea maximă a partiiiei pentru sistemul de fișiere suportată de acest sistem?

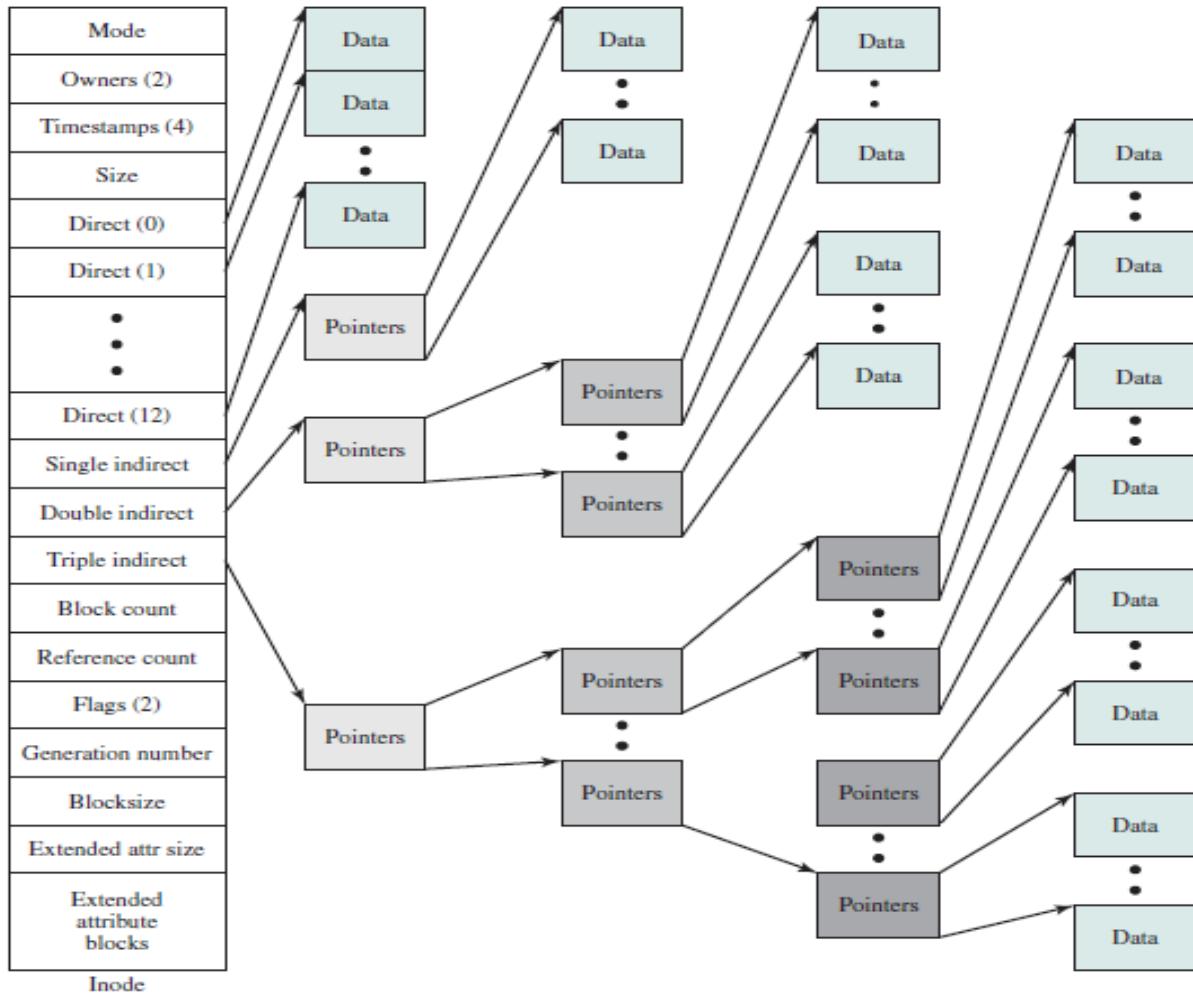


Figure 12.16 Structure of FreeBSD Inode and File

128 GB.

31. Considerați organizarea unui fișier UNIX aşa cum este reprezentată prin inode (figura Figure 12.16). Presupuneți că sunt 12 pointeri direcți, și câte un pointer indirect pentru indexarea simplă, dublă și triplă în fiecare inode. Mai mult presupuneți că mărimea unui bloc sistem este de 8 Kocteți și mărimea unui sector pe disc are 8K. Dacă pointerul pentru blocurile pe disc are 32 de biți, cu 8 biți care identifică discul fizic și 24 de biți care identifică blocul fizic, atunci: Presupunând că nici o altă informație decât aceea referitoare la inode este deja în memoria principală, câte accese la disc sunt necesare pentru a accesa octetul din poziția 13.423.956?

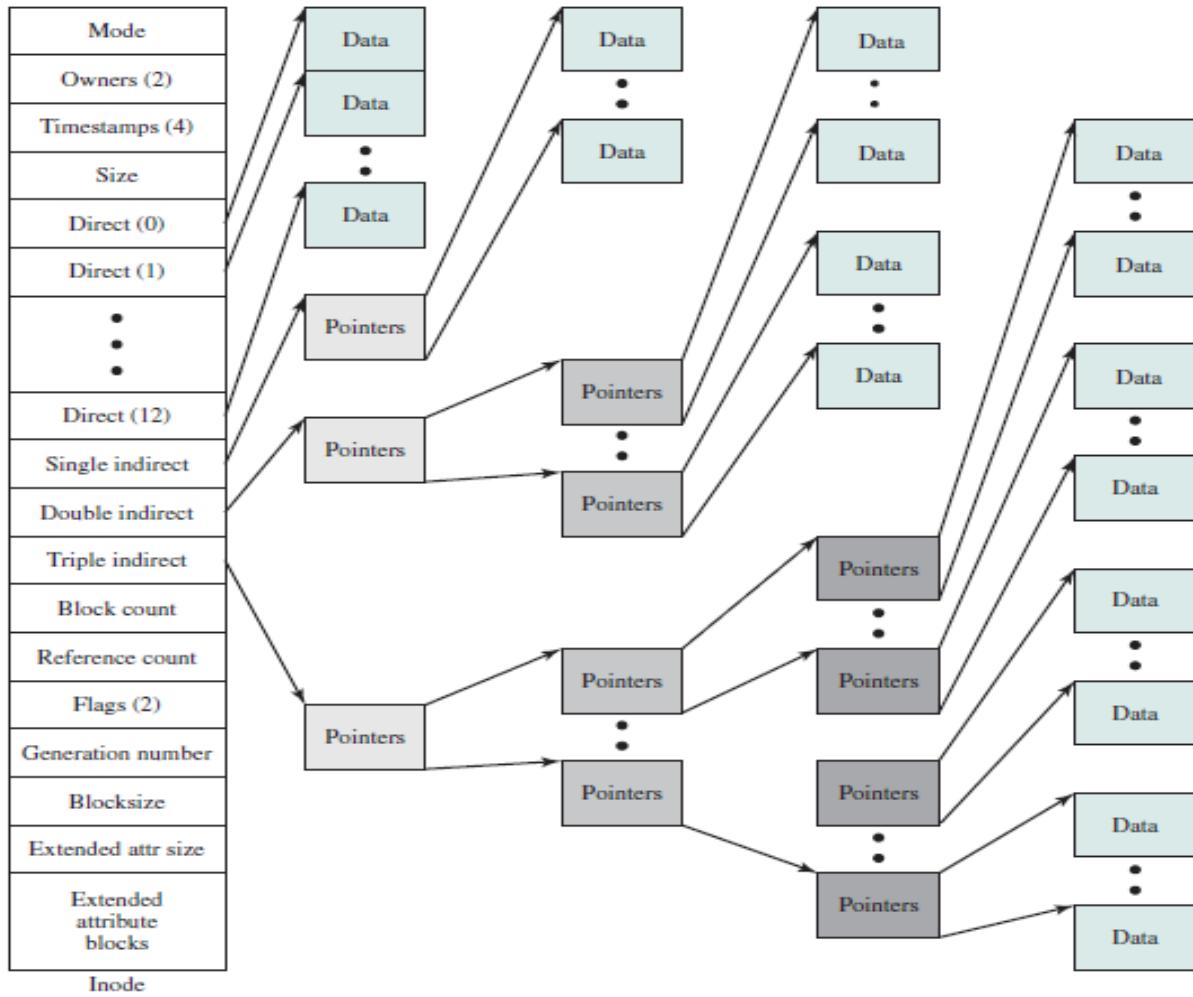


Figure 12.16 Structure of FreeBSD Inode and File

Pozitia ceruta cade in primul bloc indirect.

Cap. 13. EMBEDDED OPERATING SYSTEMS.

1. Ce este un sistem înglobat (an embedded system)?

O combinație de hardware și software, și probabil o parte mecanică adițională sau alte blocuri constructive, proiectate pentru a realiza o funcție dedicată. În multe cazuri, sistemele înglobate sunt părți ale unui sistem mare sau produs, ca în cazul sistemului de anti-blocare a frânelor dintr-un automobil.

2. Care sunt câteva din constrângerile sau cerințele tipice ale unui sistem înglobat?

Sisteme de la mici la mari, ce implică constrângerile diferite privind costul, și ca urmare nevoi diferite privind optimizarea și reutilizarea; cerințe relaxate până la cerințe foarte severe și combinații ale diferențelor cerințe privind calitatea, de exemplu, respectarea siguranței, fiabilitatea, timpul real, flexibilitatea și legislația; timp de viață de la scurt la lung; diferențe condiții de mediu, cum ar fi cele referitoare la radiații, vibrații și umiditate; caracteristici diferențiale ale aplicației cum ar fi încărcare statică versus încărcare dinamică, viteză mică sau viteză mare, sarcini de calcul intens versus sarcini intensive privind interfața, și/sau combinații ale acestora; diferențe modele de calcul de la sisteme cu evenimente discrete la cele care implică o dinamică continuă în timp (denumite sisteme hibride).

3. Ce este un sistem de operare înglobat (embedded OS)?

Un SO înglobat este un SO proiectat pentru mediul sistemelor înglobate.

4. Care sunt câteva dintre caracteristicile cheie ale unui SO înglobat (embedded OS)?

Operare în timp real: în multe sisteme înglobate, corectitudinea calculului depinde, în parte, de momentul de timp la care este furnizat rezultatul. Adesea, constrângerile de timp real sunt dictate de cerințele I/O și de control al stabilității. Operare reactivă: software-ul înglobat poate să se execute ca răspuns la evenimentele externe. Dacă aceste evenimente apar periodic sau la intervale de timp predictibile, software-ul înglobat trebuie să ia în calcul condițiile pentru cazul cel mai defavorabil și să seteze prioritățile pentru execuția rutinelor. Configurabilitate: deoarece există o gamă largă de sisteme înglobate, există o variație mare privind cerințele, atât cantitative cât și calitative, pentru funcționalitățile SO înglobat. Astfel un SO înglobat care tinde să fie utilizat pe o varietate de sisteme înglobate trebuie să se preteze el însuși la o configurare flexibilă astfel încât să furnizeze doar funcționalitățile dorite pentru o aplicație specifică și un set hardware. Flexibilitatea dispozitivelor I/O: nu există în mod virtual nici un dispozitiv care trebuie suportat de toate versiunile de SO, și gama dispozitivelor I/O este mare. Mecanisme de protecție simplificate: sistemele înglobate sunt în mod tipic proiectate pentru funcționalități limitate și bine definite. Programele netestate sunt rar adăugate în cadrul software-ului. După ce software-ul a fost configurat și testat este considerat să fie fiabil. Astfel în afară de măsurile de securitate,

sistemele înglobate au mecanisme de protecție limitate. De exemplu un task poate realiza propriile operații I/O. Similar, mecanismele de protecție a memoriei pot să fie minime. Utilizarea directă a întreruperilor: sistemele de operare de uz general nu permit utilizatorilor să utilizeze direct întreruperile.

5. Explicați avantajele și dezavantajele relative ale unui SO înglobat bazat pe un OS comercial existent în comparație cu un SO înglobat construit pentru un anume scop precizat.

Un avantaj al unui SO înglobat bazat pe un SO comercial existent în comparație cu un SO înglobat specific unei aplicații este acela că SĂ înglobat derivat dintr-un sistem comercial de uz general este bazat pe un set de interfețe familiare, care facilitează portarea. Dezavantajul utilizării unui SO de uz general este acela că nu este optimizat pentru timpul real și aplicațiile înglobate. Astfel, sunt necesare modificări considerabile pentru a atinge performanțe adecvate.

6. Care sunt principalele obiective care ghidează proiectarea nucleului eCos?

Întârziere mică de răspuns la întreruperi; întârziere mică la comutarea taskurilor; o amprentă mică de memorie; comportament deterministic.

7. Care este diferența în eCos între o rutină de tratare și o rutină întârziată de servire a întreruperii?

O ISR este invocată ca răspuns la o întrerupere hardware. Întreruperile hardware sunt debitate cu o intervenție minimală a unei IST. HAL (nivelul de abstractizare a hardware-ului) decodifică sursa hardware a întreruperii și apelează ISR (rutina de servire a întreruperii) a obiectului atașat întreruperii. Această ISR poate manipula hardware-ul dar și se permite să realizeze un set restrâns de apeluri ale API apartinând driverului. La revenire un ISR poate cere ca DSR ei să fie planificat pentru execuție. O DSR este invocată ca răspuns la o cerere ISR. O DSR se va executa atunci când este sigură execuția fără a avea interferențe cu planificatorul. De cele mai multe ori DSR se va rula imediat după ISR, dar dacă firul curent este în planificator, aceasta va fi întârziată până când execuția firului este terminată. Unei DSR (rutină întârziată de servire a întreruperii) și se permite să realizeze un set larg de apeluri API, inclusiv, în particular, a fi capabilă să apeleze `cyg_drv_cond_signal()` pentru a trezi firele care așteaptă.

8. Ce mecanisme concurențiale sunt disponibile în eCos?

Nucleul eCos poate fi configurat să includă unul sau mai mult de șase mecanisme diferite de sincronizare a firelor de execuție. Aceste includ mecanismele clasice de sincronizare: mutexes, semaphores, și condition variables. În plus, eCos suportă două mecanisme de sincronizare/comunicație care sunt comune în sistemele de timp real, denumite event flag și mailboxes. În final nucleul eCos suportă spinlocks, care sunt utile în sistemele multiprocesare simetrică (SMP).

9. Care sunt aplicațiile țintă pentru TinyOS?

Rețele de senzori fără fir.

10. Care sunt principiile de proiectare pentru TinyOS?

Să permită concurență sporită; să opereze cu resurse limitate; să se adapteze la evoluțiile hardware; să suporte o gamă largă de aplicații; să suporte un set diversificat de platforme; să fie robust.

11. Ce este o componentă TinyOS?

Un sistem software înglobat construit utilizând TinyOS constă dintr-un set de module mici, denumite componente, fiecare dintre ele realizând un task simplu sau un set de taskuri, și care interferă unele cu altele precum și cu hardware-ul într-un mod limitat și bine definit.

12. Ce software cuprinde sistemul de operare TinyOS?

Un planificator la care se adaugă un număr de componente.

13. Care este disciplina implicită de planificare pentru TinyOS?

Planificatorul implicit în TinyOS este FIFO.

14. Referitor la referințele la interfața unui driver de dispozitiv (device driver) la nucleul eCos (Tabelul 13.2), este recomandat ca driverul de dispozitiv să utilizeze varianta _intsave() pentru a achiziționa și a elibera un spinlock mai degrabă decât varianta non- intsave(). Explicați de ce.

Aceasta asigură excluderea corectă cu codul care rulează atât pe alte procesoare cât și pe acest procesor.

Table 13.2 Device Driver Interface to the eCos Kernel: Concurrency

<code>cyg_drv_spinlock_init</code> Initialize a spinlock in a locked or unlocked state.
<code>cyg_drv_spinlock_destroy</code> Destroy a spinlock that is no longer of use.
<code>cyg_drv_spinlock_spin</code> Claim a spinlock, waiting in a busy loop until it is available.
<code>cyg_drv_spinlock_clear</code> Clear a spinlock. This clears the spinlock and allows another CPU to claim it. If there is more than one CPU waiting in <code>cyg_drv_spinlock_spin</code> , then just one of them will be allowed to proceed.
<code>cyg_drv_spinlock_test</code> Inspect the state of the spinlock. If the spinlock is not locked, then the result is TRUE. If it is locked, then the result will be FALSE.
<code>cyg_drv_spinlock_spin_intsave</code> This function behaves like <code>cyg_drv_spinlock_spin</code> except that it also disables interrupts before attempting to claim the lock. The current interrupt enable state is saved in * <code>estate</code> . Interrupts remain disabled once the spinlock has been claimed and must be restored by calling <code>cyg_drv_spinlock_clear_intsave</code> . Device drivers should use this function to claim and release spinlocks rather than the non- <code>intsave()</code> variants, to ensure proper exclusion with code running on both other CPUs and this CPU.
<code>cyg_drv_mutex_init</code> Initialize a mutex.
<code>cyg_drv_mutex_destroy</code> Destroy a mutex. The mutex should be unlocked and there should be no threads waiting to lock it when this call is made.
<code>cyg_drv_mutex_lock</code> Attempt to lock the mutex pointed to by the mutex argument. If the mutex is already locked by another thread, then this thread will wait until that thread is finished. If the result from this function is FALSE, then the thread was broken out of its wait by some other thread. In this case, the mutex will not have been locked.
<code>cyg_drv_mutex_trylock</code> Attempt to lock the mutex pointed to by the mutex argument without waiting. If the mutex is already locked by some other thread then this function returns FALSE. If the function can lock the mutex without waiting, then TRUE is returned.
<code>cyg_drv_mutex_unlock</code> Unlock the mutex pointed to by the mutex argument. If there are any threads waiting to claim the lock, one of them is woken up to try and claim it.
<code>cyg_drv_mutex_release</code> Release all threads waiting on the mutex.
<code>cyg_drv_cond_init</code> Initialize a condition variable associated with a mutex with. A thread may only wait on this condition variable when it has already locked the associated mutex. Waiting will cause the mutex to be unlocked, and when the thread is reawakened, it will automatically claim the mutex before continuing.
<code>cyg_drv_cond_destroy</code> Destroy the condition variable.
<code>cyg_drv_cond_wait</code> Wait for a signal on a condition variable.
<code>cyg_drv_cond_signal</code> Signal a condition variable. If there are any threads waiting on this variable, at least one of them will all be awakened.
<code>cyg_drv_cond_broadcast</code> Signal a condition variable. If there are any threads waiting on this variable, they will all be awakened.

15. De asemenea în tabelul 13.2 , este recomandat ca `cyg_drv_spinlock_spin` ar trebui utilizată cu moderație, și doar în situații în care nu pot să apară deadlocks/ livelocks (blocaj/ blocaj cu schimbarea stării). Explicați de ce.

Cât timp aceasta este apelată, această operație efectiv va trece procesorul săn pauză până ce are succes. Ca urmare această operație trebuie utilizată cu moderație, și în situații când nu pot să apară blocaje.

16. În tabelul 13.2, care trebuie să fie limitarea în utilizarea `cyg_drv_spinlock_destroy` ? Explicați.

Nu trebuie să fie nici un procesor care încearcă să preia încuietoarea (lock –ul) pe durata cât se apelează această funcție, altfel comportarea este neînțeleasită.

17. În tabelul 13.2 , ce limitări trebuie plasate în utilizarea cyg_drv_mutex_destroy ? Mutex-ul trebuie descuiat și acolo nu trebuie să fie nici un fir care așteaptă încuietoarea când se face acest apel.

18. De ce planificatorul cu hartă de biți (bitmap scheduler) din eCos nu suportă divizarea timpului (time slicing)?

Divizarea timpului în cadrul unei priorități date este irelevantă deoarece poate fi doar un singur fir pe fiecare nivel de prioritate.

19. Implementarea mutexelor în nucleul eCos nu suportă încuietori recursive (recursive locks). Dacă un fir a încuiat un mutex și încearcă încuierea din nou a mutex-ului, tipic ca urmare a unor apeluri recursive într-un graf complicat de apeluri, atunci fie se va reporta o cădere sau firul se va bloca. Sugerați un motiv pentru această politică.

Această comportare este deliberată. Atunci când un fir tocmai a încuiat un mutex asociat cu o structură de date, se poate presupune acea structură este într-o stare inconsistentă. Înainte de descuierarea din nou a mutex-ului trebuie să se asigure faptul că structura de date este din nou într-o stare consistentă. Mutex-urile recursive permit unui fir să facă schimbări arbitrarie pe o structură de date, ca urmare în apelurile recursive încuierea din nou a mutex-ului are loc în timp ce structura de date este încă inconsistentă. Rezultatul net este că programul nu mai poate face nici o presupunere despre consistența structurii de date, ceea ce contrazice scopul utilizării mutex-urilor.

20. În figura 13.14 este un listing care cuprinde un cod care se intenționează a fi utilizat cu nucleul eCos. Ce să va întâmpla dacă eliberarea (descuierarea) mutex-ului și execuția codului de așteptare în apelul cyg_cond_wait, de pe linia 30, nu au fost atomice?

```

1  unsigned char buffer_empty = true;
2  cyg_mutex_t mut_cond_var;
3  cyg_cond_t cond_var;
4
5  void thread_a( cyg_addrword_t index )
6  {
7      while ( 1 ) // run this thread forever
8      {
9          // acquire data into the buffer...
10         // there is data in the buffer now
11         buffer_empty = false;
12
13         cyg_mutex_lock( &mut_cond_var );
14
15         cyg_cond_signal( &cond_var );
16
17         cyg_mutex_unlock( &mut_cond_var );
18     }
19 }
20
21
22 void thread_b( cyg_addrword_t index )
23 {
24     while ( 1 ) // run this thread forever
25     {
26         cyg_mutex_lock( &mut_cond_var );
27
28         while ( buffer_empty == true )
29         {
30             cyg_cond_wait( &cond_var );
31         }
32
33
34         // get the buffer data...
35
36         // set flag to indicate the data in the buffer has been processed
37         buffer_empty = true;
38
39         cyg_mutex_unlock( &mut_cond_var );
40
41         // process the data in the buffer
42     }
43 }
```

Figure 13.14 Condition Variable Example Code

Dacă acest cod nu a fost atomic, este posibil ca B să piardă semnalizarea de la A chiar dacă bufferul conține date. Aceasta deoarece `cyg_cond_wait` verifică prima dată să vadă dacă variabila de condiție este setată, și în acest caz nu este. Ca urmare, mutexul este eliberat în apelul `cyg_cond_wait`. Acum, A se execută punând date în buffer și apoi semnalizând variabila de condiție (linia 15). Apoi B revine în așteptare. Totuși, variabila de condiție a fost setată.

21. În figura 13.14 este un listing care cuprinde un cod care se intenționează a fi utilizat cu nucleul eCos. De ce este nevoie de bucla while la 26?

```
1  unsigned char buffer_empty = true;
2  cyg_mutex_t mut_cond_var;
3  cyg_cond_t cond_var;
4
5  void thread_a( cyg_addrword_t index )
6  {
7      while ( 1 ) // run this thread forever
8      {
9          // acquire data into the buffer...
10         // there is data in the buffer now
11         buffer_empty = false;
12
13         cyg_mutex_lock( &mut_cond_var );
14
15         cyg_cond_signal( &cond_var );
16
17         cyg_mutex_unlock( &mut_cond_var );
18     }
19 }
20
21
22 void thread_b( cyg_addrword_t index )
23 {
24     while ( 1 ) // run this thread forever
25     {
26         cyg_mutex_lock( &mut_cond_var );
27
28         while ( buffer_empty == true )
29         {
30             cyg_cond_wait( &cond_var );
31         }
32
33
34         // get the buffer data...
35
36         // set flag to indicate the data in the buffer has been processed
37         buffer_empty = true;
38
39         cyg_mutex_unlock( &mut_cond_var );
40
41         // process the data in the buffer
42     }
43 }
```

Figure 13.14 Condition Variable Example Code

Aceasta asigură faptul că, condiția pe care B o așteaptă B este încă true după revenirea din apelul așteptării pe condiție. Aceasta este necesar în cazul în care există alte fire așteptând pe aceeași condiție.

22. În figura 13.14 este un listing care cuprinde un cod care se intenționează a fi utilizat cu nucleul eCos. Explicați ce face codul. Presupuneți că firul B începe primul execuția și firul A începe execuția după ce apar anumite evenimente.

```

1  unsigned char buffer_empty = true;
2  cyg_mutex_t mut_cond_var;
3  cyg_cond_t cond_var;
4
5  void thread_a( cyg_addrword_t index )
6  {
7      while ( 1 ) // run this thread forever
8      {
9          // acquire data into the buffer...
10         // there is data in the buffer now
11         buffer_empty = false;
12
13         cyg_mutex_lock( &mut_cond_var );
14
15         cyg_cond_signal( &cond_var );
16
17         cyg_mutex_unlock( &mut_cond_var );
18     }
19 }
20
21
22 void thread_b( cyg_addrword_t index )
23 {
24     while ( 1 ) // run this thread forever
25     {
26         cyg_mutex_lock( &mut_cond_var );
27
28         while ( buffer_empty == true )
29         {
30             cyg_cond_wait( &cond_var );
31         }
32
33
34         // get the buffer data...
35
36         // set flag to indicate the data in the buffer has been processed
37         buffer_empty = true;
38
39         cyg_mutex_unlock( &mut_cond_var );
40
41         // process the data in the buffer
42     }
43 }
```

Figure 13.14 Condition Variable Example Code

Listingul este un exemplu de utilizare a variabilelor de condiție. Firul A obține date care sunt procesate de firul B. Prima dată, se execută B. La linia 24, B achiziționează un mutex asociat cu variabila de condiție. Nu este nici îndată în buffer și buffer_empty este inițializat cu true, astfel că B apelează cyg_cond_wait la linia 26. Acest apel suspendă B care așteptă ca variabila de condiție să fie setată și acesta va deschide mutex-ul mut_cond_var. Odată ce A achiziționează date, acesta setează buffer_empty pe false (linia 11). Apoi A începe mutex-ul (linia 13), semnalizează variabila de condiție (linia 15), și apoi deschide mutex-ul (linia 17). B poate acum să ruleze. Înainte de a se reîntoarce din cyg_cond_wait, mutex-ul mut_cond_var este încuiat și este în proprietatea lui B. B poate acum să preia bufferul de date (linia 28) și să seteze buffer_empty cu true (linia 31). În final mutex-ul este eliberat de B (linia 33) și data din buffer este procesată (linia 35).

23. Discuția despre spinlocks-urile eCos include un exemplu care arată de ce spinlocks nu trebuie utilizate în sistemele uniprocesor, dacă două fire de execuție cu priorități diferite pot concura pentru același spinlock. Explicați de ce problema persistă chiar dacă numai firele având aceeași prioritate pot pretinde același spinlock.

Chiar dacă două fire s-au executat pe aceeași prioritate, unul care încearcă să preia un spinlock se va roti până când i se consumă cuanta de timp irosindu-se astfel timp procesor prețios. Dacă un handler de intrerupere încearcă să achiziționeze un spinlock care este în proprietatea unui fir, acesta va bucla pentru totdeauna.

24. Planificatorul TinyOS's servește taskurile într-o ordine FIFO. Au fost propuse multe alte planificatoare pentru TinyOS, dar niciunul nu a devenit popular. Ce caracteristici din domeniul rețelelor de senzori au putut cauza blocarea nevoii pentru planificatoare mult mai complexe.

Motivul de bază este acela că sistemul este complet în idle. Este nevoie de o planificare intelligentă atunci când o resursă este în contencios (cereri multiple).

25. Interfața pentru resurse din TinyOS nu permite unei componente care deja a făcut o cerere plasată în coada pentru resurse să facă o a doua cerere. Sugerați un motiv.

Clienții nu sunt capabili să monopolizeze coada de resurse prin cereri multiple.

26. Totuși, interfața pentru resurse din TinyOS permite ca o componentă să dețină o încuietoare de resurse (resource lock) pentru a cere din nou încuietoarea (the lock). Această cerere este plasată în coadă pentru o acceptare ulterioară. Sugerați un motiv pentru această politică. Sugestie (Hint): Ce se poate întâmpla dacă există o întârziere între o componentă care eliberează o încuietoare (lock) și permisiunea pentru următorul fir care a făcut o cerere?

Motivul de bază este acela că o componentă poate încă deține o încuietoare după ce a apelat o eliberare. O poate deține deoarece transferul controlului către următorul deținător necesită reconfigurarea hardware-ului într-o operație de tipul split-phase (descompunere de fază). Încuietoarea poate fi oferită următorului deținător numai după ce apare această reconfigurare. Aceasta înseamnă că tehnic o secvență de cod de tipul release(), request() poate fi văzută ca: încuietoarea este cerută din nou. Există alte soluții ale problemei, dar în acest caz nu mai ținem seama de cicli CPU sau energie.

Cap. 14. COMPUTER SECURITY THREATS

1. Definiți securitatea calculatoarelor (*computer security*).

Protecția oferită unui sistem automat de informații cu scopul de a îndeplini obiectivele aplicabile privind prezervarea (conservarea) integrității, disponibilității și confidențialității resurselor sistemului de informații (inclusiv hardware-ul, software-ul, firmware-ul, informația/data și telecomunicațiile).

2. Care sunt cerințele fundamentale adresate de securitatea calculatoarelor?

Confidențialitatea, integritatea și accesibilitatea.

3. Care este diferența între pericolele de securitate (security threats) active și pasive?

Atacurile pasive au de-a face cu ascultatul (eavesdropping on), sau monitorizarea transmisiei. Poșta electronică, transferul ftp și schimburile client server sunt exemple de transmisii care pot fi monitorizate. Atacurile active includ modificarea datelor transmise și încercarea de a câștiga acces neautorizat la sistemul de calcul.

4. Enumerați și descrieți succint trei clase de intruși.

Masquerader (persoană care se dă drept alta): un individ care nu este autorizat să utilizeze computerul, și care penetreză controlul accesului la sistem pentru a exploata un cont utilizator legitim. Misfeasor (corupt): un utilizator legitim care accesează date, programe, sau resurse pentru care acces nu are autorizație, sau este autorizat pentru astfel de accese dar abuzează de privilegiile sale. Clandestine user (utilizator clandestin): un individ care capturează controlul de supervizare a sistemului și utilizează acest control pentru a evita auditul sau controlul accesului sau pentru a suprima colectarea informațiilor de către audit.

5. Enumerați și descrieți succint trei șablonane de comportare corespunzătoare intrușilor.

Hackers: aceia care atacă (hăcuiesc) calculatoarele pentru senzațiile lor personale sau pentru statut. Comunitatea lor este o meritocrație puternică în care statul este determinat de nivelul de competență. Astfel, atacatorii adesea caută ținte opertune și apoi împart informația cu alții. Criminals: grup organizat de hackeri, având în mod tipic o țintă specifică sau o clasă de ținte. Insider attacks: un intrus din cadrul unei organizații.

6. Care este rolul compresiei în modul de operare a unui virus?

Un virus poate utiliza compresia astfel încât programul infectat să aibă exact aceeași lungime cu o versiune neinfectată.

7. Care este rolul criptării în modul de operare a unui virus?

O porțiune din virus numită în general mutation engine, creează o cheie de criptare aleatoare pentru a cripta restul virusului. Cheia este memorată cu virusul, și este alterat însuși mutation engine.

8. Care sunt fazele tipice de operare a unui virus sau a unui vierme (virus or worm)?

Faza de inactivitate, faza de propagare, faza de activare și faza de execuție.

9. În termeni generali cum se propagă viermii?

1. Caută infectarea altor sisteme prin examinarea tabelelor cu gazde sau depozite similare de adrese pentru sisteme cu acces de la distanță. 2. Stabilește o conexiune cu sistem de la distanță. 3. Se copie pe el pe sistemul de la distanță și cauzează execuția copiei.

10. Care este diferența între bot și rootkit?

Un bot (robot) denumit și zombie sau dronă, este un program care în mod secret se plasează pe un alt calculator atașat la Internet, și apoi utilizează acel calculator pentru a lansa atacuri care sunt greu de trasat spre creatorul bot-ului. Un rootkit este un set de programe instalate pe un sistem pentru a menține acces de tip administrator (root) pentru acel sistem. Accesul root permite accesul la toate funcțiile și serviciile sistemului de operare. Rootkit alterează funcționalitatea standard a gazdei într-un mod răutăcios și pe fură.

11. Considerați un automat de restituit bani (automated teller machine (ATM)) unde utilizatorii furnizează un număr de identificare personală (personal identification number (PIN)) și un card pentru accesul la cont. Dați exemple de cerințe de confidențialitate, integritate și disponibilitate asociate cu sistemul și, în fiecare caz, indicați gradul de importanță al cerinței.

Sistemul trebuie să păstreze confidențialitatea numărului personal de identificare, atât în sistemul gazdă cât și pe durata transmisiei tranzacției. Acesta trebuie să protejeze integritatea înregistrărilor în cont pentru tranzacțiile individuale. Accesibilitatea sistemului gazdă este importantă pentru binele economic al băncii, dar nu este o responsabilitate fiduciară (de încredere). Accesibilitatea unui bancomat individual trebuie să îngrijoreze mai puțin.

12. Repetați problema precedentă (Considerați un automat de restituit bani (automated teller machine (ATM)) unde utilizatorii furnizează un număr de identificare personală (personal identification number (PIN)) și un card pentru accesul la cont. Dați exemple de cerințe de confidențialitate, integritate și disponibilitate asociate cu sistemul și, în fiecare caz, indicați gradul de importanță al cerinței.) pentru un sistem

de telefonie comutată care rutează apelurile printr-o rețea de comutare pe baza numărului de telefon cerut de apelant.

Sistemul nu trebuie să aibă cerințe mari privind integritatea tranzacțiilor individuale, deoarece pierderile de durată nu vor fi influențate pierderea ocazională a unui apel sau a înregistrării unei facturi. Integritatea programelor de control și a înregistrării configurațiilor este critică. Fără acestea funcția de comutare va fi învins și cel mai important atribut dintre toate, disponibilitatea, va fi compromisă. Un sistem de comutare telefonică trebuie de asemenea să conserve confidențialitatea apelurilor individuale, prevenind ca un apelant să fie auzit de un altul.

13. Considerați un sistem de publicare de tip desktop utilizat pentru a produce documente pentru diferite organizații. Dați un exemplu de tip de publicație pentru care confidențialitatea datelor memorate este cerința cea mai importantă.

Sistemul trebuie să asigure confidențialitatea dacă este va fi utilizat pentru a publica un material privat al unei corporații.

14. Considerați un sistem de publicare de tip desktop utilizat pentru a produce documente pentru diferite organizații. Dați un exemplu de tip de publicație pentru care integritatea datelor este cea mai importantă cerință.

Sistemul trebuie să asigure integritatea dacă va fi utilizat pentru legi sau reguli (reglementări).

15. Considerați un sistem de publicare de tip desktop utilizat pentru a produce documente pentru diferite organizații. Dați un exemplu în care disponibilitatea sistemului este cerința cea mai importantă.

Sistemul trebuie să asigure confidențialitatea dacă este va fi utilizat pentru a publica un material privat al unei corporații.

16. Pentru următoarea situație (assets), asignați un nivel de impact mic, moderat, sau mare pentru pierderea confidențialității, disponibilitate și respectiv integritate - o organizație care gestionează informația publică care se află pe serverul ei WEB.

Neaplicabil, moderat, moderat.

17. Pentru următoarea situație (assets), asignați un nivel de impact mic, moderat, sau mare pentru pierderea confidențialității, disponibilitate și respectiv integritate - o organizație de aplicare a legii care gestionează informații de investigare extrem de sensibile.

Mare, moderat, moderat.

18. Pentru următoarea situație (assets), asignați un nivel de impact mic, moderat, sau mare pentru pierderea confidențialității, disponibilitate și respectiv integritate - o organizație financiară care gestionează informație administrativă de rutină (nu sunt informații private).

Mic, mic, mic.

19. Pentru următoarea situație (assets), asignați un nivel de impact mic, moderat, sau mare pentru pierderea confidențialității, disponibilitate și respectiv integritate - un sistem de informații utilizat pentru achiziții mari într-o organizație de contractare conținând atât informații contractuale sensibile din faza de presolicitară cât și informații administrative de rutină. Evaluați impactul pentru cele două seturi de date în mod separat și sistemul de informații ca întreg.

Moderat, mic, moderat – mic, mic, mic.

20. Pentru următoarea situație (assets), asignați un nivel de impact mic, moderat, sau mare pentru pierderea confidențialității, disponibilitate și respectiv integritate - o întreprindere de producere a energiei electrice conținând un sistem SCADA (supervisory control and data acquisition) ce controlează distribuția energiei electrice pentru o instalație militară mare. Sistemul SCADA conține atât date de timp real de la senzori și informații administrative de rutină. Evaluați impactul pentru cele două seturi de date în mod separat și sistemul de informații ca întreg.

Neaplicabil, mare, mare – mic, mic, mic.

21. Presupuneți că parola este selectată dintr-o combinație de patru caractere din 26 de caractere alfabetice. Presupuneți că un adversar este capabil să încerce parola cu o viteză de o încercare pe secundă. Presupunând că adversarul nu are nici o reacție (feedback) până când fiecare încercare este completă, care este timpul așteptat necesar pentru a descoperii parola corectă?

63,5 ore

22. Presupuneți că parola este selectată dintr-o combinație de patru caractere din 26 de caractere alfabetice. Presupuneți că un adversar este capabil să încerce parola cu o viteză de o încercare pe secundă. Presupuneți că adversarul are o reacție de tip eroare odată ce este introdus fiecare caracter eronat, care este timpul așteptat necesar pentru a descoperii parola corectă?

52 secunde

23. Există un defect în programul virus din figura 14.3 . Care este acesta?

```

program V :-

(goto main;
 1234567;

subroutine infect-executable :-
  {loop:
   file := get-random-executable-file;
   if (first-line-of-file = 1234567)
     then goto loop
     else prepend V to file; }

subroutine do-damage :-
  {whatever damage is to be done}

subroutine trigger-pulled :-
  {return true if some condition holds}

main:  main-program :-
  {infect-executable;
  if trigger-pulled then do-damage;
  goto next; }

next:
}

```

Figure 14.3 A Simple Virus

Programul va bucla nedefinit odată ce toate fișierele executabile din sistem vor fi infectate.

24. Întrebarea apare dacă este posibil să dezvolti un program care să poată analiza o piesă de software și să determine dacă este un virus. Considerați că avem un program D care se presupune că poate face asta. Aceasta înseamnă că, pentru orice program P, dacă rulăm D(P), rezultatul întors este TRUE (P este un virus) sau FALSE (P nu este un virus). Considerați acum următorul program :

Program CV :=

```

{ ...
main-program :=
{...
 {
  if D(CV) then goto next;
  else infect-executable;
 }
next:
}

```

În programul precedent, infect-executable este un modul care scanează memoria pentru programele executabile și se reproduce pe el în acele programe. Determinați dacă D poate decide corect dacă CV este un virus.

Se presupune că D examinează un program P și returnează TRUE dacă P este un virus și FALSE dacă nu este. Dar CV apelează D. Dacă D spune că CV este un virus, atunci CV nu va infecta un executabil. Dar dacă D spune că CV nu este un virus, acesta infectează un executabil. D întoarce întotdeauna un răspuns incorect.

25. Scopul acestei probleme este să demonstreze tipul de puzzles care trebuie rezolvat în proiectarea unui cod răutăcios (malicious code) și ca urmare tipul de atitudine pe care trebuie să o adopte cineva care dorește să contracareze aceste atacuri.

Considerați următorul program :

```
begin
print (*begin print (); end.*);
end
```

Ce credeți că vrea să facă programul? Va merge?

Când se execută programul, el produce următoarea ieșire: begin print () : end. Programul probabil intenționa să producă după execuție, o listă exactă a textului programului original. În mod clar nu a mers.

26. Scopul acestei probleme este să demonstreze tipul de puzzles care trebuie rezolvat în proiectarea unui cod răutăcios (malicious code) și ca urmare tipul de atitudine pe care trebuie să o adopte cineva care dorește să contracareze aceste atacuri.

Ce credeți că vrea să facă programul? Va merge?

```
char [] = {'0', ' ', '}', ';', 'm', 'a', 'i', 'n', '(', ')', '{',
and so on... 't', ')', '0'};
main ()
{
    int l;
    printf(*char t[] = (*);
    for (i=0; t[i]!=0; i=i+1)
        printf("%d, ", t[i]);
    printf("%s", t);
}
```

Ce credeți că vrea să facă programul? Va merge?

Acesta merge. La bază, este un proces în trei pași: (1) declară un sir de caractere care corespunde corpului principal al programului; (2) printează individual fiecare caracter din sirul definit; (3) printează valoarea ariei ca un sir definit de caractere.

27. Scopul acestei probleme este să demonstreze tipul de puzzles care trebuie rezolvat în proiectarea unui cod răutăcios (malicious code) și ca urmare tipul de atitudine pe care trebuie să o adopte cineva care dorește să contracareze aceste atacuri.

Considerați următorul program :

```
begin
print (*begin print (); end.*);
end
```

Ce credeți că vrea să facă programul? Va merge?

Răspundetă la aceeași întrebare pentru următorul program :

```
char [] = {'0', ' ', '}', '!', 'm', 'a', 'i', 'n', '(', ')', '{',
```

```
and so on... 't', ')', '0'};
```

```
main ()
```

```
{
```

```
int l;
```

```
printf(*char t[] = (*);
```

```
for (i=0; t[i]!=0; i=i+1)
```

```
printf("%d, ", t[i]);
```

```
printf("%s", t);
```

```
}
```

Întrebarea la care trebuie răspuns pentru această problemă este: care este relevanța specifică a acestei probleme în acest capitol?

Problema arată un program cu auto-replicare, adică tipul de funcționalitate utilizată de un virus.

28. Considerați următorul fragment:

```
legitimate code
```

```
if data is Friday the 13th;
```

```
    crash_computer();
```

```
legitimate code
```

Ce tip de software răutăcios (malicious) este acesta?

Bombă logică (Logic bomb)

29. Considerați următorul fragment dintr-un program de autentificare :

```
username = read_username();
```

```
password = read_password();
```

```
if username is "133t h4ck0r"
```

```
    return ALLOW_LOGIN;
```

```
if username and password are valid
```

```
    return ALLOW_LOGIN
```

```
else return DENY_LOGIN
```

Ce tip de software răutăcios (malicious) este acesta?

Bacdoor

30. Următorul fragment de cod arată o secvență de instrucțiuni ale unui virus și versiunea polimorfică (metamorfică) a virusului. Descrieți efectul produs de codul metamorfic.

Original Code	Metamorphic Code
<pre>mov eax, 5 add eax, ebx call [eax]</pre>	<pre>mov eax, 5 push ecx pop ecx add eax, ebx swap eax, ebx swap ebx, eax call [eax] nop</pre>

Codul original a fost alterat pentru a distruge semnătura fără a afecta semantica codului. Instrucțiunile ineficace (fără efect) în codul metamorfic sunt a doua, a treia, a cincea, a șasea și a opta.

Cap. 15. COMPUTER SECURITY TECHNIQUES

1. În termeni generali, care sunt patru dintre înțelesurile autentificării identității unui utilizator?

Ceva ce știe individul: Ca exemple avem parola, numărul personal de identificare, sau răspunsurile la un set pre-aranjat de întrebări. Ceva ce posedă individul: Ca exemple avem carduri electronice de tip cheie, carduri smart, și chei fizice. Acest tip de identificator este denumit token. Ceva ce este individul (biometrie statică): Ca exemple avem recunoașterea amprentei, retină sau față. Ceva ce face individul: Ca exemple avem recunoașterea vocii, a scrisului de mâna, și ritmul de tipărire.

2. Explicați scopul sării din figura 15.1.

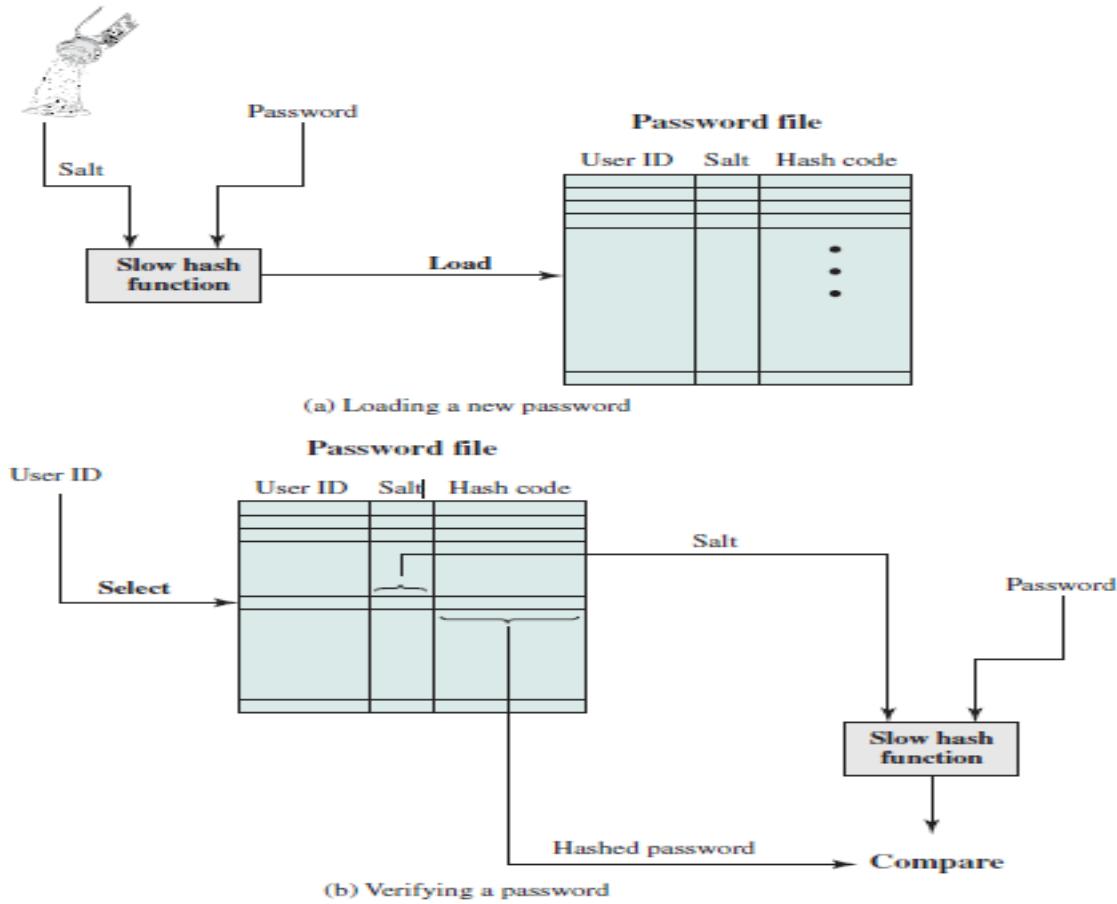


Figure 15.1 UNIX Password Scheme

Sarea este combinată cu parola la intrarea într-o rutină cu o cale de criptare.

3. Explicați diferența dintre un simplu card de memorie și un card deștept (smart card).

Cardurile cu memorie pot memora dar nu procesează datele. Cardurile smart au microprocesor.

4. Enumerați și descrieți pe scurt principalele caracteristici fizice utilizate pentru identificarea biometrică.

Caracteristicile fetii: se definesc caracteristicile fetei după poziția relativă și forma ochilor, sprâncenelor, nasului, buzelor și a bărbiei. O abordare alternativă utilizează o cameră în infraroșu pentru a produce o termogramă. Amprenta: crestele și adânciturile de pe suprafața degetului formează o formă unică pentru fiecare persoană. Geometria mâinii: identifică trăsăturile mâinii, inclusiv formă, lungimea și grosimea degetelor. Amprenta retinei: şablonul format de venele de sub suprafața retinei este unică și, ca urmare, potrivită pentru identificare. Irisul: o altă caracteristică unică este structura detaliată a irisului. Semnătura: fiecare individ are un stil unic de scris și aceasta se reflectă în special în semnătură. Vocea: vocea este mult mai apropiată de caracteristicile fizice și anatomiche ale individului. Totuși variația de la eșantion la eșantion complică sarcina de recunoaștere biometrică.

5. Descrieți pe scurt diferența dintre DAC și RBAC.

Discretionary acces control – DAC (controlul accesului discrețional) controlează accesul pe baza identității celui care cere accesul și pe regulile de acces (autorizații) care statuează ce se permite și ce nu celui care a emis cererea. Această politică este denumită discrețională deoarece o entitate poate avea drepturi de acces care permite entității, prin propria sa voință, să valideze o altă entitate să acceseze anumite resurse. Role-based acces control – RBAC (controlul accesului pe baza rolului) controlează accesul pe baza rolului pe care îl au utilizatorii în cadrul sistemului și a regulilor care statuează că acese sunt permise utilizatorilor pentru un anumit rol dat. RBAC poate avea un mecanism discrețional sau obligatoriu.

6. Explicați diferența dintre detectia intruziunilor anormale și a intruziunilor cu semnătură.

Detectia statistică a unor anomalii implică colectarea de date relativ la comportarea utilizatorului legitim pe o perioadă de timp. Apoi sunt aplicate teste statistice pentru a observa comportarea, și a determina, cu un mare grad de încredere, dacă acest comportament nu este comportamentul utilizatorului legitim. Detectarea semnăturii intrușilor implică o încercare de a defini un set de reguli care pot fi utilizate pentru a decide că o anumită comportare este aceea a unui intrus.

7. Ce este un sistem digital imun?

Un sistem digital imun furnizează un sistem de emulare cu scop general și de detectie a virușilor. Obiectivul este de a furniza un timp de răspuns rapid astfel încât virușii să fie

exterminări cât mai curând posibil din momentul în care au fost introduse. Atunci când un virus nou intră în organizație, sistemul de imunizare îl capturează automat, îl analizează, adaugă detecție și protecție, îl înălță, și transmite informația despre acel virus sistemului care rulează un program antivirus general, astfel încât acesta să poată fi detectat înainte de a se executa oriunde în alt loc.

8. Cum lucrează software-ul de tip blochează-comportamentul (behavior-blocking software)?

Behavior-blocking software se integrează cu sistemul de operare a unui calculator gazdă și monitorizează comportarea în timp real a programelor privind acțiunile răutăcioase. Behavior-blocking software blochează apoi acțiunile potențial răutăcioase înainte ca acestea să aibă o șansă ca să afecteze sistemul.

9. Descrieți câteva măsuri împotriva viermilor.

Signature-based worm scan filtering: în cadrul acestei abordări se generează o semnătură a viermelui. Este destul de greu de aplicat viermilor polimorfici. Filter-based worm containment: Această abordare este similară cu prima, dar se concentrează pe conținutul viermelui mai degrabă decât pe scanarea semnăturii. Payload-classification-based worm containment: aceasta este o tehnică bazată pe rețea conform căreia se examinează pachetele pentru a vedea dacă acestea conțin viermi. Threshold random walk (TRW) scan detection: exploatează hazardul privind culegerea destinațiilor de conectare ca un mod de detectare dacă scannerul este în lucru. TRW este potrivit în dezvoltarea dispozitivelor de mare viteză și cost redus. Rate limiting: această clasă limitează viteza traficului de tip scanare de la o gazdă infectată. Rate halting: Această abordare blochează imediat traficul de ieșire atunci când se depășește un prag fie în traficul de ieșire fie în diversitatea încercărilor de conectare.

10. Ce tipuri de limbaje de programe sunt vulnerabile la depășiri ale bufferelor (buffer overflows)?

Limbajele de programare vulnerabile la depășirea bufferelor sunt aceleia care nu au o notare puternică a tipurilor de variabile, și care sunt operațiile permise pe acestea. Acestea includ limbajul de asamblare, C și alte limbaje similare. Limbajele cu tipuri puternic definite cum ar fi Java, ADA, Python, și multe altele nu sunt vulnerabile la aceste atacuri.

Behavior-blocking software se integrează cu sistemul de operare a unui calculator gazdă și monitorizează comportarea în timp real a programelor privind acțiunile răutăcioase. Behavior-blocking software blochează apoi acțiunile potențial răutăcioase înainte ca acestea să aibă o șansă ca să afecteze sistemul.

11. Care sunt cele două categorii mari de apărare împotriva depășirii bufferelor?

Cele două categorii mari de apărare împotriva depășirii bufferelor sunt: apărarea din timpul compilării care are ca scop întărirea programelor pentru a rezista atacurilor din

noile programe; și apărarea în timpul execuției care are ca scop să detecteze și să esueze atacul în programele existente.

12. Enumerați și descrieți pe scurt posibile apărări împotriva depășirii bufferelor care pot fi utilizate atunci când se compilează un nou program.

Apărarea în timpul compilării include: scrierea de programe utilizând limbi de programare de nivel înalt care nu sunt vulnerabile la atacurile de depășire a bufferelor. Utilizând tehnici de codificare sigure pentru a valida utilizarea bufferelor; utilizând extensii de siguranță a limbajelor și/ sau biblioteci sigur implementate; sau utilizând mecanisme de protecție a stivelor.

13. Enumerați și descrieți pe scurt posibile apărări împotriva depășirii bufferelor care pot fi implementate când se rulează program existente și vulnerabile.

Apărarea în timpul execuției care furnizează protecție pentru programele vulnerabile existente: utilizând “Executable Address Space Protection” care blochează execuția codului pe stivă, heap, sau în datele globale; utilizând “Address Space Randomization” pentru a manipula locația structurilor cheie de date cum ar fi stiva și heap-ul în spațiul de adrese al procesului; sau plasând pagini de gardă între regiunile critice de memorie ale spațiului de adrese ale proceselor.

14. Explicați susținerea sau nu a următoarei parole:
YK 334.

Dacă aceasta este un număr de licență, este ușor de ghicit.

15. Explicați susținerea sau nu a următoarei parole:
mfmitm (for “my favorite movie is tender mercies”)

Potrivită.

16. Explicați susținearea sau nu a următoarei parole:
Natalie1

Ușor de ghicit.

17. 15.1 Explicați susținearea sau nu a următoarei parole:
Washington

Ușor de ghicit.

18. Explicați susținerea sau nu a următoarei parole:
Aristotle
Ușor de ghicit.

19.15.1 Explicați susținerea sau nu a următoarei parole:tv9stove

Potrivită.

20. Explicați susținerea sau nu a următoarei parole:
12345678

Foarte nepotrivită.

21. Explicați susținerea sau nu a următoarei parole:
dribgib.

Aceasta este o bigbird în sens invers. Nu este potrivită.

22. O încercare timpurie de a forța utilizatorii de calculatoare să utilizeze parole mai puțin predictibile a implicat parole de tipul computersupplied (furnizate de calculator). Parolele aveau lungimea de 8 caractere și erau preluate din setul de caractere constând din litere mici și cifre. Ele erau generate cu un generator pseudo-aleator de numere cu 215 valori posibile de start. Utilizând tehnologia timpului, timpul necesar de a căuta toate șirurile de caractere având lungimea de 8 dintr-un alfabet de 36 de caractere a fost de 112 ani. Din nefericire, aceasta nu mai este o reflexie adevărată pentru sistemele actuale de securitate. Explicați problema.

Numărul șiruri de caractere cu lungimea de 8 biți care utilizează un alfabet de 36 de caractere este 36^8 aproximativ 2^{41} . Totuși, numai 2^{15} dintre ele trebuie să fie luate în considerare, deoarece acesta este numărul de ieșiri posibile ale unui generator de numere aleatoare.

23. Presupuneți că elementele sursă de lungime k sunt mapate într-un mod oarecare uniform în elemente întă de lungime p. Dacă fiecare digit poate lua valori de la una la r valori, atunci numărul de elemente sursă este r^k și numărul de elemente întă este numărul mai mic rp. Un element sursă particular x_i este mapat la un element întă y_i . Care este probabilitatea ca elementul sursă corect să poată fi selectat de un adversar la o încercare?

$$p = r^k.$$

24. Presupuneți că elementele sursă de lungime k sunt mapate într-un mod oarecare uniform în elemente întă de lungime p. Dacă fiecare digit poate lua valori de la una la r valori, atunci numărul de elemente sursă este r^k și numărul de elemente întă este numărul mai mic rp. Un element sursă particular x_i este mapat la un element întă y_i . Care este probabilitatea ca un element sursă diferit x_k ($x_i \neq x_k$) care duce în același element întă, y_j , poate fi produs de un adversar?

$$p = (r^k - r^p)/r^{k+p}.$$

25. Presupuneți că elementele sursă de lungime k sunt mapate într-un mod oarecare uniform în elemente ţintă de lungime p. Dacă fiecare digit poate lua valori de la una la r valori, atunci numărul de elemente sursă este rk și numărul de elemente ţintă este numărul mai mic rp. Un element sursă particular xi este mapat la un element ţintă yi. Care este probabilitatea ca elementul ţintă corect să poată să poată fi produs de un adversar cu o încercare?

$$p = (r^k - r^p)/r^{k+p}.$$

26. Presupuneți că parolele sunt limitate la a utiliza 95 de caractere printabile și că toate parolele sunt de lungime 10. Presupuneți un spărgător de parole cu o viteză de codare de 6,4 milioane de codări pe secundă. Cât timp îi va lua să testeze complet toate parolele posibile pe un sistem UNIX?

300000 ani.

27. Din cauza riscurilor cunoscute ale sistemului de parole din UNIX, documentația SunOS-4.0 recomandă ca fișierul cu parole să fie înlăturat și înlocuit un fișier public citibil denumit /etc/publickey. O intrare în fișier pentru utilizatorul A constă dintr-un identificator IDA, cheia publică utilizator, PUa , și o cheie privată corespunzătoare, PRa. Această cheie privată este criptată utilizând DES, cu o cheie derivată din parola de logare a utilizatorului Pa . Când A se loghează, sistemul decriptează E(Pa, PRa) pentru a obține PRa . Sistemul verifică faptul că Pa a fost corect introdusă. Cum?

la un bloc arbitrar X și verifică dacă $X = D(PRa, E[PUa, X])$.

28. Din cauza riscurilor cunoscute ale sistemului de parole din UNIX, documentația SunOS-4.0 recomandă ca fișierul cu parole să fie înlăturat și înlocuit un fișier public citibil denumit /etc/publickey. O intrare în fișier pentru utilizatorul A constă dintr-un identificator IDA, cheia publică utilizator, PUa , și o cheie privată corespunzătoare, PRa. Această cheie privată este criptată utilizând DES, cu o cheie derivată din parola de logare a utilizatorului Pa . Când A se loghează, sistemul decriptează E(Pa, PRa) pentru a obține PRa . Cum poate un oponent ataca acest sistem?

Deoarece fișierul file/etc/publickey este public și citibil. Un atacator poate ghici P (să zicem P') și poate calcula $PR_{a'} = D(P', E[P, PR_a])$. Acum el poate alege un bloc arbitrar Y și poate verifica să vadă dacă $Y = D(PR_{a'}, E[PU_a, Y])$. Dacă este așa este foarte probabil ca $P' = P$. Pot fi alese și alte blocuri pentru a verifica egalitatea.

29. S-a statuat că introducerea sării în schema de parole UNIX crește dificultatea de ghicire cu un factor de 4096. Dar sareea este memorată sub formă de text în aceeași intrare ca și parola ciphertext corespunzătoare. Ca urmare, acele două caractere sunt cunoscute atacatorului și nu trebuie ghicite. De ce se susține că sareea crește securitatea?

Fără sare, atacatorul poate ghici parola și o poate cripta. Dacă ORICARE din utilizatorii din sistem utilizează acea parolă, atunci va fi o potrivire. Cu sare, atacatorul trebuie să ghicească parola și apoi să o cripteze pentru fiecare utilizator, utilizând o sare particulară pentru fiecare utilizator.

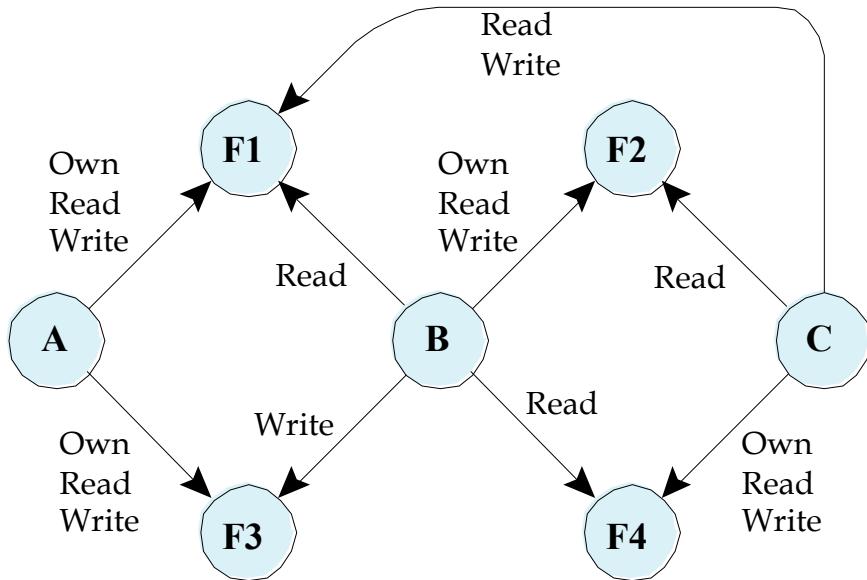
30. Presupunând că ați răspuns cu succes la problema precedentă și ați înțeles semnificația sării, apare aici o altă problemă. Nu poate fi posibil a contracara complet toți crackerii de parole prin creșterea dramatică mărimea sării la, să zicem, 24 sau 48 de biți?

Depinde de mărimea populației de utilizatori și nu de mărimea sării, deoarece atacatorul presupus are acces la sarea fiecărui utilizator. Beneficiul unei sări mari este acela al mărimii sării, pentru că este puțin probabil ca doi utilizatori să aibă aceeași sare. Dacă utilizatorii mulți au aceeași sare, atunci atacatorul poate face o criptare pe fiecare parolă găsită pentru a testa toți acei utilizatori.

31. Pentru modelul DAC discutat în secțiunea 15.2, o reprezentare alternativă a sării de protecție este graful direct. Fiecare subiect și fiecare obiect în starea de protecție este reprezentat de un nod (un singur nod este utilizat pentru o entitate care este atât subiect cât și obiect). O linie directă de la un subiect la un obiect indică un drept de acces, și o etichetă pe legătură definește drepturile de acces. Desenați un graf direct care corespunde matricei de acces din figura 12.15a.

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry credit	
User B	R	Own R W	W	R	Inquiry debit	Inquiry credit
User C	R W	R		Own R W		Inquiry debit

(a) Access matrix

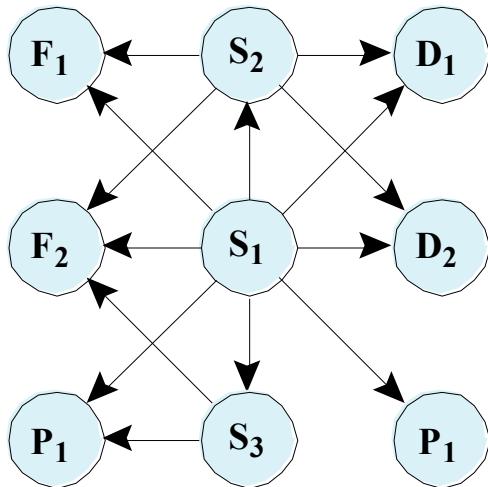


32. Pentru modelul DAC discutat în secțiunea 15.2 , o reprezentare alternativă a stării de protecție este graful direct. Fiecare subiect și fiecare obiect în starea de protecție este reprezentat de un nod (un singur nod este utilizat pentru o entitate care este atât subiect cât și obiect). O linie directă de la un subiect la un obiect indică un drept de acces, și o etichetă pe legătură definește drepturile de acces. Desenați un graf direct care corespunde matricei de acces din figura 15.4 .

		Objects								
		Subjects			Files		Processes		Disk drives	
		S ₁	S ₂	S ₃	F ₁	F ₂	P ₁	P ₂	D ₁	D ₂
Subjects	S ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
	S ₂		control		write *	execute			owner	seek *
	S ₃			control		write	stop			

* = copy flag set

Figure 15.4 Extended Access Control Matrix



33. Pentru modelul DAC discutat în secțiunea 15.2 , o reprezentare alternativă a stării de protecție este graful direct. Fiecare subiect și fiecare obiect în starea de protecție este reprezentat de un nod (un singur nod este utilizat pentru o entitate care este atât subiect cât și obiect). O linie directă de la un subiect la un obiect indică un drept de acces, și o etichetă pe legătură definește drepturile de acces. Există acolo o corespondență unu-la-unu între reprezentarea prin graful direct și reprezentarea prin matricea de acces? Explicați.

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry credit	
User B	R	Own R W	W	R	Inquiry debit	Inquiry credit
User C	R W	R		Own R W		Inquiry debit

(a) Access matrix

		Objects								
		Subjects			Files		Processes		Disk drives	
		S ₁	S ₂	S ₃	F ₁	F ₂	P ₁	P ₂	D ₁	D ₂
Subjects	S ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
	S ₂		control		write *	execute			owner	seek *
	S ₃			control		write	stop			

* = copy flag set

Figure 15.4 Extended Access Control Matrix

matrice de acces dată generează numai un graf direct, și un graf direct dat utilizează numai o matrice de acces, aşadar corespondența este unu-la-unu.

34. UNIX tratează fișierele cu directoare în aceeași manieră ca un fișier; aceasta însemenă că, ambele sunt același tip de structură de date, denumită inode. Ca și cu fișierele, directoarele includ un sir de protecție pe 9 biți. Dacă nu se are de grija, aceasta poate crea probleme privind controlul accesului. De exemplu, considerați un fișier cu modul de protecție 644 (octal) conținut într-un director cu modul de protecție 730. Cum a putut fi compromis fișierul în acest caz?

Presupuneți că directorul d și fișierul f au același proprietar și grup și că f conține textul something. Indiferent de super-utilizator, nimeni sub proprietarul fișierul f nu poate schimba conținutul său, deoarece numai proprietarul are permisiunea de a scrie. Totuși, oricare din grupul proprietarului are permisiunea de scriere pentru d, astfel încât orice astfel de persoană poate înlătura f din d și instala o versiune diferită, care, pentru cele mai multe scopuri, echivalează cu capabilitatea de a modifica f.

35. Presupuneți un sistem cu N poziții de joburi. Pentru job-ul i, numărul de utilizatori individuali în acea poziție este U_i și numărul de permisii cerut pentru poziția job-ului este P_i. Câte relații trebuie definite între utilizatori și permisii, pentru o schemă DAC tradițională?

$$\text{Sum}_{i=1}^N (U_i \times P_i).$$

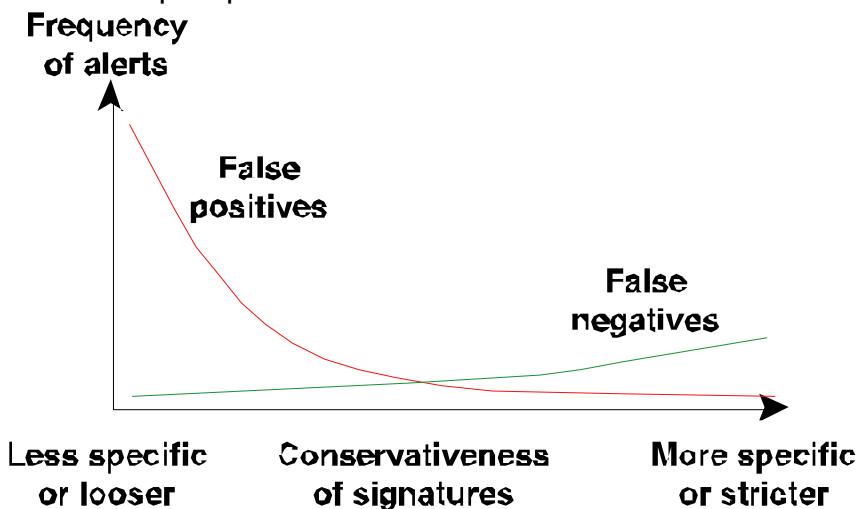
36. Presupuneți un sistem cu N poziții de joburi. Pentru job-ul i, numărul de utilizatori individuali în acea poziție este U_i și numărul de permisii cerut pentru poziția job-ului este P_i. Câte relații trebuie definite între utilizatori și permisii, pentru o schemă RBAC tradițională?

$$\text{Sum}_{i=1}^N (U_i \times P_i).$$

37.15.14 În contextul unui IDS, s-a definit ca un false positive să fie o alarmă generată de un IDS, în care IDS alertează o condiție care tocmai a început. Un false negative apare atunci când un IDS eșuează să genereze o alarmă atunci când o condiție de alertă evidentă este activă. Utilizând următoarea diagramă, desenați două curbe care indică aproximativ false positives și respectiv false negatives.



Acesta este un exemplu tipic



38. Rescrieți funcția din figura 7.13a astfel încât să nu mai existe vulnerabilitate la o depășire a bufferului de stivă.

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    fgets(str2, sizeof(str2), stdin);
    if (strncmp(str1, str2, sizeof(str2)) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

Cap. 16. Distributed Processing, Client/Server, and Clusters

1. Ce este modelul de calcul client/server?

Un mediu de rețea care include calculatoare client care lansează cereri către calculatoare server.

2. Ce distinge modelul de calcul client/ Server de orice altă formă de procesare de date distribuită?

1. Există un suport bogat pentru a aduce aplicațiile prietenoase cu utilizatorul pe propriul său sistem. 2. Deși aplicațiile sunt dispersate, există o concentrare pe bazele de date centralizate ale corporațiilor și pe multe funcții utilitare și de gestiune a rețelei. 3. Există o obligație, atât pentru organizațiile utilizator cât și pentru furnizări, pentru a dezvolta sisteme deschise și modulare. 4. Rețelistica este fundamentală pentru operare. Astfel, gestiunea rețelei și securitatea acesteia au o mare prioritate în organizarea și operarea sistemului informațional.

3. Care este rolul arhitecturii de comunicație, cum ar fi TCP/IP, într-un mediu client/server?

Un software de comunicație care validează clienții și serverele să interacționeze.

4. Discutați rațiunea pentru localizarea aplicațiilor pe client, server, sau împărțirea între client și server.

Procesarea bazată pe server: rațiunea care stă la baza unei astfel de configurații este că stațiile de lucru utilizator sunt cele mai potrivite pentru a furniza interfețe utilizator prietenoase și că aplicațiile și bazele de date pot fi ușor întreținute pe sisteme centrale. Procesarea bazată pe client: această arhitectură validează utilizatorul să dezvolte aplicații cu nevoile locale. Procesarea cooperativă: acest tip de configurație poate oferi un câștig de productivitate utilizator mai mare și o eficiență mai mare a rețelei decât celelalte abordări client/server.

5. Ce sunt clienții fat (grași) și clienții thin (subțiri, slabii), și care sunt diferențele privind filozofia celor două abordări.?

Fat client: procesare bazată pe client, cu majoritatea software-ului pe client. Principalul beneficiu a acestui model este acela că preia avantajul puterii desktop-ului, eliberând încărcarea la nivelul serverului și făcându-le pe acestea mult mai eficiente și mai puțin supuse la gătuiri (bottleneck). Thin client: procesare bazată pe server, cu partea cea mai importantă de software plasată pe server. Această abordare aproape mimează abordarea tradițională cu gazdă centrală, și este adesea calea de migrare pentru evoluția aplicațiilor de la nivelul corporațiilor larg răspândite către mediul distribuit.

6. Sugerați avantaje și dezavantaje ale strategiilor clientilor fat (grași) și a celor thin (slabi).

Fat client: principalul beneficiu este acela că preia puterea de calcul a desktop-ului, eliberând încărcarea la nivelul serverului și făcându-le pe acestea mult mai eficiente și mai puțin supuse la gâtuiri (bottleneck). Ca dezavantaj menționăm dificultatea în a realiza întreținerea, actualizarea sau înlocuirea aplicațiilor distribuite pentru zeci sau sute de desktop-uri. Thin client: aproape mimează abordarea tradițională cu gazdă centrală, și este adesea calea de migrare pentru evoluția aplicațiilor de la nivelul corporațiilor larg răspândite către mediul distribuit. Ca dezavantaj menționăm faptul că nu oferă flexibilitatea abordării de tip fat client.

7. Explicați rationamentul care stă la baza arhitecturii client/ server pe trei niveluri (the three-tier client/server architecture).

Nivelul din mijloc este în esență o poartă de acces (gateway) între clienți thin și o varietate de servere cu baze de date cu acces indirect. Nivelul din mijloc poate converti protocoale și poate asocia un tip de baze de date la alt tip. În plus, nivelul din mijloc poate alătura/ integra rezultate din surse de date diferite. În final, nivelul din mijloc poate servi ca o poartă de acces între aplicațiile desktop și aplicațiile moștenite cu acces de la distanță, care mediază între cele două lumi.

8. Ce este middleware?

Middleware-ul este un set de interfețe standard de programare care este situat între aplicația de deasupra și software-ul de comunicație și sistemul de operare de operare de dedesubt. Acesta oferă un mijloc și un stil uniform de acces la resursele sistemului de-a lungul oricărei platforme.

9. Deoarece avem standarde precum TCP/IP, de ce este necesar middleware-ul?

TCP/IP nu furnizează API și protocoale de nivel intermediar pentru a suporta o varietate de aplicații pentru diferite platforme hardware și SO.

10. Enumerați unele beneficii și dezavantaje ale primitivelor cu blocare și fără blocare pentru trimiterea de mesaje.

TCP/IP nu furnizează API și protocoale de nivel intermediar pentru a suporta o varietate de aplicații pentru diferite platforme hardware și SO.

11. Enumerați unele beneficii și dezavantaje ale asocierii (binding) persistente și non-persistente pentru RPC.

Asocierea non-persistentă: deoarece o conexiune necesită informații pentru menținerea stării la ambele capete, consumă din resurse. Stilul non-persistent este utilizat pentru a

conserva aceste resurse. Pe de altă parte, supracontrolul implicat în stabilirea conexiunii face asocierea non-persistentă inadecvată pentru procedurile cu acces de la distanță care sunt apelate frecvent de același apelant. Asocierea persistentă: pentru aplicațiile care realizează multe apeluri repetitive la procedurile cu acces de la distanță. Asocierea persistentă menține conectarea logică și permite ca o secvență de apeluri și reveniri să utilizeze aceeași conexiune.

12. Enumerați unele beneficii și dezavantaje ale RPC-ului sincron și respectiv asincron.

RPC-ul sincron este ușor de înțeles și programat deoarece comportarea sa este predictibilă. Totuși, acesta nu reușește să exploateze întregul paralelism inherent în aplicațiile distribuite. Aceasta limitează tipul de interacțiuni pe care le pot avea aplicațiile distribuite, rezultând performanțe slabe. Pentru a furniza o flexibilitate mai mare, RPC-ul asincron facilitează un grad mai mare de paralelism în timp ce reține și familiaritatea și simplitatea RPC-ului. RPC-ul asincron nu blochează apelantul; răspunsul poate fi recepționat atunci când este nevoie, permitându-se astfel clientului să execute sarcini locale în paralel cu invocarea serverului.

13. Enumerați și definiți pe scurt patru metode de clustering.

Passive Standby: un server secundar preia controlul în cazul în care serverul primar cade. Separate Server: servere separate au propriul disk. Data este copiată continuu din serverul primar în cel secundar. Servers Connected to Disk: serverele sunt cablate la aceeași discuri, dar fiecare server are propriul disk. Dacă un server cade, discul său este preluat de alt server. Servers Share Disks: servere multiple partajează simultan accesul la discuri.

14. Fie α procentul de cod program care poate fi executat simultan pe n calculatoare într-un cluster, fiecare calculator utilizând un set diferit de parametri sau condiții initiale. Presupuneți că codul rămas trebuie executat secvențial de un singur procesor. Fiecare procesor are o viteza de execuție de x MIPS. Derivați o expresie pentru viteza efectivă în MIPS atunci când se folosește sistemul pentru execuția exclusivă a acestui program, în funcție de n , α , și x .

$$\text{MIPS rate} = (n \alpha - \alpha + 1) x.$$

15. Fie α procentul de cod program care poate fi executat simultan pe n calculatoare într-un cluster, fiecare calculator utilizând un set diferit de parametri sau condiții initiale. Presupuneți că codul rămas trebuie executat secvențial de un singur procesor. Fiecare procesor are o viteza de execuție de x MIPS. Dacă $n=16$ și $x=4$ MIPS, determinați valoarea pentru care sistemul va atinge o performanță de 40 MIPS.

$$\alpha = 0,6.$$

16. Un program aplicație este executat pe un cluster cu nouă calculatoare. Un program de tip benchmark (evaluare a performanțelor) consumă T timp pe acest cluster. Mai mult, 25 % din timpul T este timpul în care aplicația rulează simultan pe cele nouă calculatoare. În timpul rămas, aplicația trebuie să ruleze pe un singur calculator. Calculați viteza efectivă în condițiile menționate anterior în comparație cu execuția pe un singur calculator. De asemenea, calculați procentul de cod care a fost paralelizat (programat sau compilat astfel încât să utilizeze modul cluster).

Cresterea efectivă de viteză = 3. $\alpha = 0,75$.

17. Un program aplicație este executat pe un cluster cu nouă calculatoare. Un program de tip benchmark (evaluare a performanțelor) consumă T timp pe acest cluster. Mai mult, 25 % din timpul T este timpul în care aplicația rulează simultan pe cele nouă calculatoare. În timpul rămas, aplicația trebuie să ruleze pe un singur calculator. Presupuneți că suntem capabili să utilizăm efectiv, pe porțiunea de paralelizare a codului, 18 calculatoare în loc de 9. Calculați viteza efectivă care se atinge.

Noua creștere efectivă de viteză = 3,43. $\alpha = 0,8$.

18. Următorul program FORTRAN va fi executat pe un calculator, și o versiune paralelă va fi executată pe 32 de calculatoare dintr-un cluster.

L1: DO 10 I = 1, 1024

L2: SUM(I) = 0

L3: DO 20 J = 1, I

L4: 20 SUM(I) = SUM(I) + I

L5: 10 CONTINUE

Presupuneți că liniile 2 și 4 iau fiecare doi cicli mașină inclusând toate activitățile procesorului și accesul la memorie. Ignorați supracontrolul dat de liniile de control al buclei (liniile 1,3,5 și toate celelalte supracontroale date de sistem sau de conflictul la resurse).

Care este timpul total de execuție (în timpi dați de cicli mașină) a programului pe un singur calculator.

Timpul de execuție secvențial = 1051628 cicli.

19. Următorul program FORTRAN va fi executat pe un calculator, și o versiune paralelă va fi executată pe 32 de calculatoare dintr-un cluster.

L1: DO 10 I = 1, 1024

L2: SUM(I) = 0

L3: DO 20 J = 1, I

L4: 20 SUM(I) = SUM(I) + I

L5: 10 CONTINUE

Presupuneți că liniile 2 și 4 iau fiecare doi cicli mașină inclusând toate activitățile procesorului și accesul la memorie. Ignorați supracontrolul dat de liniile de control al buclei (liniile 1,3,5 și toate celelalte supracontroale date de sistem sau de conflictul la resurse).

Divizați iterațiile buclei I-loop între cele 32 de calculatoare după cum urmează: calculatorul 1 execută primele 32 de iterații (I de la 1 la 32), procesorul 2 execută următoarele 32 și aşa mai departe. Care este timpul de execuție și factorul de multiplicare a vitezei (speedup factor) în comparație cu timpul pentru calculul pe un singur procesor? (De nota că încărcarea, dictată de bucla J, este dezechilibrată între calculatoare)

Factorul de multiplicare a vitezei = 16,28

20. Următorul program FORTRAN va fi executat pe un calculator, și o versiune paralelă va fi executată pe 32 de calculatoare dintr-un cluster.

L1: DO 10 I = 1, 1024

L2: SUM(I) = 0

L3: DO 20 J = 1, I

L4: 20 SUM(I) = SUM(I) + I

L5: 10 CONTINUE

Presupuneți că liniile 2 și 4 iau fiecare doi cicli mașină inclusând toate activitățile procesorului și accesul la memorie. Ignorați supracontrolul dat de liniile de control al buclei (liniile 1,3,5 și toate celelalte supracontroale date de sistem sau de conflictul la resurse).

Explicați cum să modificăm paraleлизarea pentru a facilita o execuție echilibrată a întregului calcul pentru cele 32 de calculatoare. O încărcare echilibrată înseamnă un număr egal de adunări asignat fiecărui calculator cu respectarea ambelor bucle.

Fiecare calculator are asignate 32 de iterații echilibrate între începutul și sfârșitul buclei

21. Următorul program FORTRAN va fi executat pe un calculator, și o versiune paralelă va fi executată pe 32 de calculatoare dintr-un cluster.

L1: DO 10 I = 1, 1024

L2: SUM(I) = 0

L3: DO 20 J = 1, I

L4: 20 SUM(I) = SUM(I) + I

L5: 10 CONTINUE

Presupuneți că liniile 2 și 4 iau fiecare doi cicli mașină inclusând toate activitățile procesorului și accesul la memorie. Ignorați supracontrolul dat de liniile de control al buclei (liniile 1,3,5 și toate celelalte supracontroale date de sistem sau de conflictul la resurse).

Care este timpul minim de execuție rezultat din execuția paralelă pe 32 de calculatoare? Care este creșterea de viteză față de un singur calculator?

Este atins factorul de multiplicare ideal 32.

