FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI, UPB

Temă de curs SOA

Inversiunea priorităților în sisteme de operare în timp real

Damian Constantin Cristian, IISC

Coord. Prof. Ștefan Stăncescu

2015-02-4

CUPRINS

2 Sumar 3
3 Concepte de bază 3
3.1 Planificarea execuţiei 3
1.1.1 Primul venit, primul servit 4
1.1.2 Planificarea Round-Robin 4
1.1.3 Planificarea cu priorităţi 4
3.2 Sincronizarea proceselor 5
3.2.1 Mutex-uri 6
3.2.2 Semafoare 6
3.3 Sisteme de operare în timp real 7
4 Inversiunea priorităţii 7
1 Soluţii 8
4.1 Dezactivarea întreruperilor 8
4.2 Mostenirea imediată a priorităţii 8
1.1 Moştenirea priorităţii 8
1.2 Tavanele de prioritate 9
5 Concluzii 10
6 Bibliografie 11

1 SUMAR

Inversiunea prorităților este o problemă importantă a sistemelor de operare care poate cauza pagube în cazul unor aplicații în timp real pe sisteme critice și are la bază interacțiunea dintre două funcții de bază a sistemelor de operare. Tema de curs constă în explicarea conceptelor din spatele inversiunii necontrolate a priorităților proceselor din cadrul sitemelor de operare în timp real și prezentarea soluțiilor ce au fost elaborate pentru a rezolva acestă problemă.

2 CONCEPTE DE BAZĂ

2.1 PLANIFICAREA EXECUȚIEI

În zilele de început ale calculatoarelor, acestea rulau un singur program pe rînd care urma un singur fir de operații în secvență (care îi vom spune fir de execuție sau proces). Unitatea aștepta informații de la un dispozitiv de intrare apoi făcea calculele necesare apoi trimitea informațiile la un dispozitiv de ieșire și aștepta ca dispozitivul să termine ca să execute următoarele instrucțiuni.

Pentru a utiliza mai eficent timpul s-a inventat concepul de diviziune în timp (time sharing). rocesorul astfel are de executat mai multe programe în acelaş timp şi va alterna între programe executând o parte din instrucțiunile unui program apoi o parte din intrucțiunile altuia. Când la un program trebuie să aștepte pentru o operațiunie intrare/ieșire trece la alt program și în acest fel toate programele se vor executa mai repede decât dacă ar fi executate pe rînd [1].

Pentru ca procesorul să facă acest lucru însă trebuie ca sistemul de operare să ia mai multe decizii și anume cît timp să se ruleze fiecare proces pînă trece la următorul și în ce ordine să execute procesele. Componenta sistemului care se ocupă de aceasta este planificatorul de execuție. Funcția se numește planificarea execuției și este fundamentală pentru toate sistemele de operare moderne. Planificatorul trebuie să ia decizii în următoarele situații [2]:

1) Când un proces trece din starea de execuție în starea de așteptare (ca de exeplu după ce lansează o cerere de intrare/ieșire sau cănd programul îi cere să aștepte).
2) Când un proces iese din starea de execuție în starea de execuție (cum ar fi când este întrerupt).
3) Cînd un proces trece din starea de așteptare în starea ”pregătit de execuție” (cum ar fi când s-a terminat o operație de intrare/ieșire).
4) Când se termină un proces.

Pentru situațiile 2 și 3 planificatorul poate decide să ruleze același proces care executa înainte dar pentru 1 și 4 planificatorul trebuie să aleagă un alt proces decât cel curent. Dacă nu se schimbă procesele decât in cazurile 1 și 4 se spune că planificarea este nepreemptivă (sau cooperativă) altfel se spune că planificarea este preemptivă.

Un planificator cooperativ este mai simplu de realizat și unele arhitecturi hardware nu suportă decât acestă planificare. În schimb, planificarea preemptivă asigură un timp de răspuns bun în cazul în care se dorește un sistem interactiv dar pune mai multe probleme. De exemplu, dacă două

procese lucrează asupra aceleași structuri de date unul îl întrerupe pe celălalt atunci cînd scrie date în structură atunci al doilea proces poate să citească datele într-o stare eronată care ar da o eroare sau ar da un rezultat fals. Aceste probleme trebuie rezolvate prin sincronizarea proceselor ce se pot influenţa unul pe altul, funcţie explicată în secţiunea **Error! Reference source not found.**..

Pe lângă planificator mai este necesară o altă componentă numită dispacher pentru a asigura că tranziţiile între procese se fac corect şi rapid. Rolul dispacher-ului este de a salva şi a restaura contextul în care se află programul când a fost întrerupt şi de a seta privilegiile fiecărui proces.

Câţiva dintre algoritmi de planificare folosiţi în sistemele de operare sunt prezentaţi în paragrafele următoare.

1.1.1 PRIMUL VENIT, PRIMUL SERVIT

Cel mai simplu algoritm de planificare a execuţiei este algoritmul ”primul venit, primul servit” (FCFS, First-Come, First-Served). Când un proces intră în starea ”pregătit” identificatorul său este pus la sfârşitul unei structuri de tip coadă şi cînd procesorul este liber alege procesul din capul cozii pentru a-l executa până cere acces la un sistem de intrare/ieşire sau se termină, algoritmul fiind cooperativ.

Algoritmul este simplu de implementat dar are anumite dezavantaje. Dacă un proces are nevoie de foarte mult timp de execuţie se vor aduna multe procese la codă şi timpul de aştepatre pentru fiecare proces va fi mare, problemă numtă şi *efectul de convoi*. De asemenea dacă procesul cu timp de execuţie mare face o cerere de intrare/ieşiere şi toate celelalte procese au nevoie de un timp mic de execuţie acestea se pot termina înainte de terminarea operaţiunii de intrare/ ieşire lansate de primul proces şi procesorul va rămîne în aştepare. Din acestă cauză algoritmul este inefcient pentru sitemele cu time sharing.

1.1.2 PLANIFICAREA ROUND-ROBIN

Planificarea round-robin este fimilară cu planificarea ”primul venit, primul servit” dar fiecărui se alocă o cuantă de timp după care procesele sunt întrerupte şi se execută următorul proces din coadă într-o manieră circulară. Dacă un proces se termină sau întră în starea de aşteptare înaite de a expira cuanta de timp atunci planificatorul alege următorul proces din coadă ca în algoritmul FCFS. Planificarea evită problema efectului de convoi prin întreruperea proceselor cu timp de execuţie mare dând o şansă proceselor mici să se termine într-un timp mai scurt dar timpul de aşteptare rămîne destul de mare pentru toate procesele.

Performanţele acestui tip de planificare depinde foarte mult de cuanta de timp alocată. Dacă se alocă prea mult timp atunci planificrea este echivalentă cu FCFS, dacă se alocă prea puţin timp atunci pricesorul va pierde mult timp schimbând contextele de execuţie. În general trebuie ca majoritatea timpilor de execuţie ale proceselor să fie mai mici decât cuanta de timp alocată.

1.1.3 PLANIFICAREA CU PRIORITĂŢI

În planificarea cu priorităţi fiecare process are alocat un nivel de prioritate (adică un număr) şi procesorul va executa procesul cu cea mai mare prioritate în starea ”pregătit”, dacă două procese au acelaş nivel de prioritate atunci procesele se execută în ordinea FCFS. Prioritatea poate fi alocată

pe diverse criterii cum ar fi durara timpului de execuţie, importanţă sau resurse necesare. Panificarea poate de asemenea să fie preempdivă sau cooperativă. O probemă majoră este că un proces de prioritate mică poate fi blocat de procese de prioritate mai mare pe durată nelimitată.

2.2 SINCRONIZAREA PROCESELOR

Cum am menţionat şi în subcapitolul 1.1 dacă planificarea proceselor este preemptivă pot apărea probleme când mai multe procese vor să aceseze aceeaşi resursă în acelaş timp. Procesele pot fi întrerupte în orice punct din program şi acţiunile procesului pot rămîne pe jumătate făcute. Problemele ce pot apărea sunt ilustrate cel mai bine printr-un exemplu.

Presupunem două procese care vor să ruleze în acelaş timp pe un sistem cu planificare preemptivă şi ambele vor să adauge un element la o structură de tip listă din memorie. Lista are o succesiune de locaţii din memorie şi un numărător care stochează poziţia ultimului element. Fiecare proces trebuie să scrie în locaţia următoare şi să incrementeze numărătorul. Succesiunea de intrucţiuni pentru ambele procese este.

1. Registru = Numărător
2. Registru = Registru + 1
3. Vector[Registru]= element
4. Numărator= Registru

Fiecare proces are propriul registru şi propriul element dar vectorul şi numărătorul sunt comune. Dacă procesele se execută pe rând nu apare nici o problemă dar considerăm următoarea secvenţă de instrucţiuni.

Proces	**Instrucţiune**	**Variabile**
P1	Registru = Numărător	Registru1=10
P2	Registru = Numărător	Registru2=10
P2	Registru = Registru + 1	Registru2=11
P2	Vector[Registru]= element	Vectror[11]=element2
P2	Numărator= Registru	Numarator=11
P1	Registru = Registru + 1	Registru1=11
P1	Vector[Registru]= element	Vectror[11]=element1
P1	Numărator= Registru	Numarator=11

Elementul procesului P2 a dispărut fiind rescris de procesul P1 însemnând că datele sunt eronate. Pentru a preveni astfel de probleme trebuie să definim o **secţiune critică** a proceselor ce se referă la accesul unei resurse comune doar un singur proces poate să fie în secţiunea critică. Procesul trebuie să ceară permisiunea să intre în secţiunea critică, secţiunea în care se cere permisiunea este **secţiunea de intrare.** Secţiunea critică este urmată de o **secţiune de ieşire** în care procesul anunţă celelalte procesoare că a ieşit din seccţiunea critică, restul programului se numeşte **secţiunea necritică**. O soluţie de sinctronizare trebuie să indeplinească următoarele condiţii:

1. Excludere mutulă: Dacă procesul Pi execută în secţiunea critică nici un alt proces nu poate executa în secţiunea critică.

2. Progres: Doar procesele care nu execută în secţiunea necritică poate cere acces la secţiunea critică.

3. Aşteptare limitată: Thebuie să existe o limită pentru timpul de aşteptare al unui proces care a cerut intrarea într-o secţiune critică.

Un proces ce se află în secţiunea de intrare şi nu poate intra în secţiunea critică va intra în stare de aşteptare până când va putea intra în secţiunea critică.

2.2.1 MUTEX-URI

O soluţie pentru sincronizare este utilizarea unui mutex. Numele vine de la excluderea mutuală (MUTual EXclusion). Inainte de intrarea într-o secţiune critică o funcţie se închide mutex-ul folosind funcţia aquire() şi după ieşirea din secţiunea critică se eliberează mutexul folosind funcţia release() . Definiţiile funcţiilor sunt următoarele.

```
acquire() {
while (!available)
; /* busy wait */
available = false;;
}

release() {
available = true;
}
```

Unde available este o variabilă globală. În timpul închiderii mutexului procesul verifică constant dacă acesta este disponibil aşteptând de fapt să se elibereze. Aceestă ”învârtire în gol” sau ”busy wait ” consumă timp de execuţie care poate ar putea fi folosit în mod productiv de alte procese.

2.2.2 SEMAFOARE

Semafoarele reprezintă un sitem de sincronizare mai complex decât mutexurile. Acestea pot fi de două timpuri semafoare cu numărătoare şi semafoarele binare. Semafoarele cu numărătoare se folosesc dacă există mai multe copii ale unei resurse şi numără resursele disponibile. Semafoarele binare se comportă ca mutex-urile dar au o implementare diferită.

Semaforul constă dintr-un număr şi o listă de procese. Când un proces trebuie să intre în zona critică se uită să vadă dacă numărul e nenul. Dacă numărul este nenul atunci decrementeză numărul şi poate intra în zona dacă numărul este nul atunci se pune în listă şi intră în starea de aşteptare. Când un proces iese din zona critică incrementeză numărul, se uiă în listă şi dacă există procese îl porneşte pe primul din listă.

În acest fel procesele blocate nu se ”învîrt în gol” şi timpul de execuţie este folosit eficient.

2.3 SISTEME DE OPERARE ÎN TIMP REAL

Sistemele de operare în timp real (Real Time Operating Sistem, RTOS) sunt sisteme dedicate pentru aplicații în care răspunsul la un stimul de la o intrare trebuie să fie foarte scurt și previzibil. Ele se definesc ca ultimele două categorii din următoarea clasificare a sistemelor [3]:

- Sistemele **non-real-time** sunt sistemele pentru care un răspuns programat la un stimul este garantat să fie efectuat cândva în viitor.
- Sistemele **soft-real-time** sunt sistemele pentru care un răspuns programat la un stimul va fi efectuat într-un timp finit cunoscut aproape de fiecare dată.
- Sistemele **hard-real-time** sunt sistemele care garantează că un răspuns programat la un stimul va fi efectuat într-un timp finit cunoscut aproape de fiecare dată.

Ne vom concentra asupra ultimei categorii, aceste sisteme operează în medii critice unde este nevoie de un comportament previzibil și fiabil. De exemplu, sistemele RTOS sunt folosite în automobile pentru a controla sistemul anti-blocare (ABS) sau air-bag-ul. Fiecare stimul este asociat cu un proces ce are un punct de start și un deadline care nu trebuie depășit.

Procesele dintr-o aplicație real-time au anumite propprietăți speciale. Aceste procese sunt considerate periodice, adică au nevoie de timp de execuție la intervale de timp constante, numite perioade, au un timp de procesare constant și un deadline până la care trebuie să se termine. Depășirea deadline-ului este echivalentă cu a nu răspunde la stimul. Planificatorul de execuție trebuie să știe de acești parametri și să planifice în funcție de acestea.

Timpul de la simțirea stimului până la efectuarea răspunsului trebuie să fie scurt astfel dispecerul trebuie să fie rapid și eficient. De asemenea, un RTOS trebue să admită ca procesele cu prioritate mare să poată întrerupe procesele cu prioritate mai mică în plus procesele trebuie să fie executate în funcțiie de cerințele necesare pentru a termina procesul înainte de deadline.

3 INVERSIUNEA PRIORITĂȚII

Problema inversării de priorități are loc în contextul unui sistem de operare de tipul hard-real-time cu un algoritm de planificare a a execuției preemptiv bazat pe priorități. Sitemul de operare foloseste și un sitem de sincronizare pentru a bloca resurse fizice sau logice (cum ar fi o magistrală de comunicație sau o strucură de date).

Ca exemplu considerăm trei procese, L,M și H, ale căror priorități sunt în ordinea L < M< H. Presupunem că procesul H cere resursa R care este accesat de L. Procesul L este dejea în secțiunea critică și H intră în secțiunea de intrare și nu are de ales decât să aștepte ca L să ajungă în secțiunea de ieșire. Procesul M este pregătit de execuție și întrerupe procesul L. În mod indirect procesul M cu prioritate mai mică decât H și care nu are legătură cu acesta îl blochează și poate amâna execuția acestuia pe o perioadă nedeterminată.

Această problemă este foarte importantă în sisteme de operare în timp real. Procesul cu prioriate mare poate să își depășească deadline-ul și poate cauza alte pocese să își depășească deadline-ul lor stricând întregul sistem.

În 1997 sonda NASA, Mars Pathfinder a adus un robot pe Marte, Sojourner rover, care să facă experimente. Î timpul operării robotul făcea resetări frecvente ale sistemului de operare care îl împiedicau să funcţioneze. Un proces de prioritate mare ”bc_dist” dura mai mult decât era aşteptat deoarece era forţat să aştepte un proces de pritate mică pentru o resursă care la rîndul lui era întrerupt de alte procese. Procesul ”bc_dist” rămânea în aşteptare pînă când sistemul de operare descoperea acest lucru se reseta. Problema a fost rezolvată prin setare unui mod de planificare cu moştenirea priorităţii descris în capitolul următor.

1 SOLUȚII

Trebuie găsit un sistem prin care să se respecte cele trei condiţii ale sincronizării proceselor să păstreze planificarea preemptivă cu priorităţi şi să se evite situaţia de inversiune a priorităţii.

3.1 DEZACTIVAREA ÎNTRERUPERILOR

O soluţie simplă ar fi împiedicarea întreruperilor în timpul în care un proces este într-o secţiune critică. Un avantaj adiţional este că în sitemele monoprocesor nu mai este nevoie de semafoare pentru sincronizare. Acestă soluţie funcţionează doar dacă secţiunile critice sunt foarte mici deoarece un proces cu prioritate mică va bloca toate procesele îndiferent de prioritate sau dacă rolosesc resursa respectivă. Dacă secţiunile critice sunt mari sitemul se reduce la sistem cu planificare nepreemptivă şi nu poate lucra ca un sistem în timp real.

3.2 MOSTENIREA IMEDIATĂ A PRIORITĂȚII

O variantă mai moderată pentru dezactivarea întreruperior este alocarea unor priorităţi pentru resurse. O resursă va avea prioritatea cu unu mai mare decît toate procesele ce pot folosi resursa. Cînd un proces accesează resursa moşteneşte prioritatea resursei si poate intra în zona critică. Se reduce numărul de procese blocate dar tot se blochează procese fără să fie necesar [4].

1.1 MOŞTENIREA PRIORITĂȚII

Ideea principală a soluţiei este că atunci când un proces blochează alte procese prin accesarea unei resurse acel proces îşi va schimba nivelul de prioritate la cel mai înalt nivel al proceselor care sunt blocate. Atunci când se iese din secţiunea critică revine la prioritatea originală. Moştenirea priorităţii se defineşte astfel:

- Dacă un proces cu prioritatea cea mai mare încearcă să intre în secţiunea critică verifică dacă alt proces nu este dejea în secţiunea critică. Dacă există un proces în secţiunea critică atuci intră în starea de aşteptare. Dacă nu, atunci poate intra în secţiunea critică.

- Un proces foloseşte nivelul de prioritate alocat dacă nu blochează alte procese cu prioritate mai mare. Dacă blochează procese atunici el va moşteni nivelul de prioritate cel mai înalt din procesele blocate. La ieşirea din secţiunea critică procesul revine la prioritatea iniţială.

- Moştenirea de priorităţi este tranzitivă. Dacă un proces L blochează un proces cu prioritate mai mare M şi M blochează pe H atunci L va moşteni prioritatea cea mai mare dintre M şi H.

- Un poces H poate întrerupe un proces L dacă nivelul său de prioritate este mai mare decît nivelul de prioritate alocat sau moştenit a lui L.

În exemplul ilustrat în capitolui 2, dacă se moştenesc priorităţile L îl blochează pe H şi moşteneşte prioritatea sa. Atunci cînd M e pregătit de execuţie nu mai poate să îl întrerupă pe L dearece are prioritatea mai mică. L iese din secţiunea critică revine la prioritatea originală şi H urmează să execute apoi M şi L.

Algoritmul de planificare rezolvă inversiunea de prioritate dar are anumite probleme. El nu garantează o aşteptare limitată. Fie două procese M şi L care au nevoie de două resurse R1 şi R2. M poate intra în zona critică pentru R1 şi L poate să intre în zona critică pentru R2 ca apoi M şi L să-şi ceară reciproc permisiunea pentru celelalte resurse şi să rămînă în aşteptare definitiv. Situaţia se numeşte deadlock. Altă problemă ar fi că se pot forma lanţuri de blocări care deşi sunt limitate pot fi destul de lungi. Aceste probleme sunt rezolvate de următorul algoritm de planificare.

1.2 TAVANELE DE PRIORITATE

Protocolul tavanelor de prioritate foloseşte moştenirea priorităţilor din supcapitolul trecut şi adaugă un mecanism pentru prevenirea deadlock-urilor.

Fiecărei resurse i se alocă un tavan de prioritate egal cu prioritatea cea mai mare a proceselor care îl folosesc. Tavanul cel mai mare al resurselor blocate se numeşte tavanul sistemului. Dacă un proces care vrea să acceseze o resursă oarecare atunci tavanul resursei trebuie să fie mai mare decît tavanul sistemului pentru a putea intra în zona critică. Altfel procesul intră în aşteptare şi donează prioritatea sa procesului care ţine resursa cu tavanul cel mai mare. Procesul respectiv moşteneşte prioritatea dacă este mai mare şi revine la prioritatea originală după ce iese din zona critică.

Ilustrăm algoritmul cu un exemplu. Procesele A, B şi C ,ordonat descrescătoar după prioritate, vor să acceseze resursele S1, S2 şi S3 astfel:

- Procesul A accesează resursa S1 şi o eliberează.
- Procesul B accesează resursa S2 apoi resursa S3 şi le eliberează în ordine inversă.
- Procesul C accesează resursa S3 apoi resursa S2 şi le eliberează în ordine inversă.

Se observă că există potenţialul unui deadlock dacă se foloseşte algoritmul din subcapitolul 1.1. S1 are un tavan mai mare decît S2 şi S3 care au tavane egale. Procesele se derulează astfel:

1. Porneşte procesul C şi blochează resursa S3.
2. Procesul B întrerupe pe C.
3. Procesul B cere resursa S2 dar aceasta nu are tavan mai mare ca resursa S3. B Donează prioritatea lui A şi aşteaptă.
4. Intră procesul A, întrerupe pe C; blochează resursa S1 şi o foloseşte fiindcă are un tavan mai mare S3; deblochează resursa şi se termină.
5. Procesul C blochează resursa S2 şi apoi deblochează resursele S3 şi S3 şi revine la prioritatea iniţială.
6. Procesul B întrerupe pe C foloseşte resursele S2 , S3 şi se termină.

7. Procesul C reia calculele şi se termină.

Se observă că deadlockul dintre B şi C a fost evitat. De asemenea procesul cu prioritatea cea mai mare a putut să ruleze fără să fie blocat [5].

4 CONCLUZII

Inversiune priorităţilor este o situaţie problematică ce poate apărea în sistemele de operare cu o planificare preemptivă a execuţiei şi cu sisteme de sincronizare a proceselor. Sistemele de operare în timp real sunt cele mai vulnerabile în această situaţie deoarece situaţia duce la ratarea deadline-urilor. Un proces cu prioritate mare poate fi blocat de un proces de prioritate mică deoarece acesta a închis o resursă nesesară primului proces. În plus primul proces poate fi amânat pe perioadă nedeterminată deoarece procesul cu prioritate mică poate fi întrerupt de alte procese.

Am prezentat câteva soluţii la acestă problemă în oridinea crescătare a complexităţii. Se poate poate observa că algoritmii mai complecşi dau gestionează mai eficient procesele. Astfel un compromis trebuie realizat între complexitatea planificatorului şi eficienţa de gestiune a proceselor.

5 BIBLIOGRAFIE

[1] R. Radescu, "Soluții software de prelucrare paralelă," in *Arhitectura Sistemelor de Calcul*, București, Editura Politehnica Press, 2012, pp. 44-48.

[2] A. Silberschatz, P. B. Galvin and G. Gagne, Operating system concepts, Wiley, 2013.

[3] chibios.org, "RTOS Concepts," chibios.org, 03 10 2011. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts. [Accessed 25 1 25].

[4] Y. Wang, "Uppsala Universitet, Department of Information Technology," 18 9 2012. [Online]. Available: http://www.it.uu.se/edu/course/homepage/realtid/ht12/schedule/synchronization-resource-sharing.pdf. [Accessed 2015 2 4].

[5] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time," *IEEE Transactions on Computers,* vol. 39, no. 9, pp. 1175-1185, 1990.

[6] M. Herlihy, V. Luchangco and M. Moir, " Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *International Conference on Distributed Computing Systems*, 2003.

[7] E. L. Lamie, Real-time embedded multithreading: using ThreadX and ARM, CMP books, 2005.