Universitatea “Politehnica” din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Sincronizarea kernel-ului

Conducător ştiinţific
Dr.Ing. Ştefan STĂNCESCU

Masterand
Alexandru Nicolae SAVA

2017

Cuprins

Introducere……..……3

Capitolul 1: Concurenţă în kernel…….……6

Capitolul 2: Tehnici de sincronizare…….……8

2.1. Alocarea de date unice pentru fiecare procesor ……8

2.2. Operaţii atomice în sisteme multiprocesor……..……...9

2.3. Spin locks ……10

2.4. Read/Write Spin Locks……12

2.5. Seqlocks ……..13

2.6. Read-Copy Update..……...……..14

2.7. Excluderea mutuală……..……15

2.8. Excludere folosind *Sleep* și *Wakeup*…...…….16

2.9. Semafoare……...……...……..17

Concluzii…….19

Bibliografie…….20

Introducere

Fiecare sistem de calcul include un set de bază de programe numite sistem de operare. Cel mai important program din acest set este numit kernel. Acesta este încărcat în memoria RAM la pornirea sistemului și conține mai multe proceduri critice care sunt necesare pentru ca sistemul să funcționeze.

Unele sisteme de operare permit tuturor programelor utilizatorilor să aibă acces direct la componentele hardware (un exemplu tipic este MS-DOS). În schimb, un sistem de operare Unix, ascunde toate detaliile de nivel scăzut în ceea ce privește organizarea fizică a sistemului de calcul, în aplicațiile folosite de către utilizator. Atunci când un program dorește să utilizeze o resursă hardware, acesta trebuie să emită o cerere către sistemul de operare. Kernel-ul evaluează cererea și, în cazul în care alege să acorde resursa, interacționează cu componentele hardware corespunzătoare, în numele programului utilizator.[1]

Pentru a pune în aplicare acest mecanism, sistemele de operare moderne se bazează pe disponibilitatea caracteristicilor specifice hardware, care interzic programelor utilizatorului să interacționeze direct cu componentele hardware de nivel scăzut sau pentru a avea acces la locații de memorie partajate. Hardware-ul prezintă cel puțin două moduri de execuție diferite pentru CPU: un mod neprivilegiat pentru programele de utilizator și un mod privilegiat pentru kernel. Unix numește aceste moduri *user mode*, respectiv *kernel mode*.[1]

Kernel-ul este un program ce constituie nucleul central al sistemului de operare al unui computer. Acesta are control complet asupra tot ceea ce are loc în sistem. Ca atare, acesta este primul program încărcat la pornire, kernel-ul gestionând restul de pornire, precum cererile de intrare / ieșire software, traducându-le în instrucțiuni de procesare a datelor pentru unitatea centrală de procesare. De asemenea, este responsabil pentru gestionarea memoriei, precum și pentru gestionarea și comunicarea cu periferice de calcul, cum ar fi imprimante, difuzoare, s.a. (Figura 1). Kernel-ul este o parte fundamentală a sistemului de operare al unui computer.

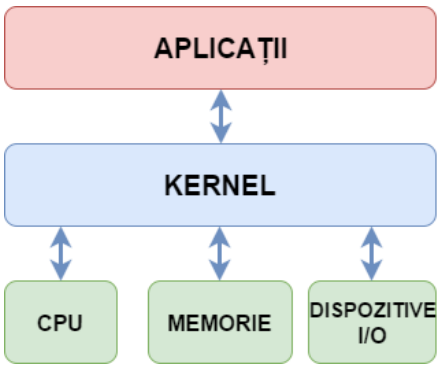

Figura 1. Kernel-ul coneactează aplicațiile la resursele computer-ului

Un proces este un program (cod obiect salvat într-un mediu de stocare) în execuție. Cu toate acestea, procesele sunt mai mult decât codul programului de executare. Procesul ține evidența resurselor folosite de program, inclusiv a fișierelor (stocare) și a rețelei (sockets). Fiecare resursă are un identificator unic (descriptor) și o stare asociată. Aceste resurse pot fi accesate unic de

procesul în execuție sau pot fi partajate cu alte procese. Practic, procesele sunt rezultatul executării codului programului. Kernel-ul trebuie să gestioneze toate aceste resurse în mod eficient și transparent. Un proces poate avea mai multe stări, asa cum arată diagrama din Figura 2.

Din punct de vedere al Kernel-ului, scopul unui proces este de a acționa ca o entitate în care resursele de sistem (timpul de utilizare al procesorului, memorie etc.) sunt alocate.

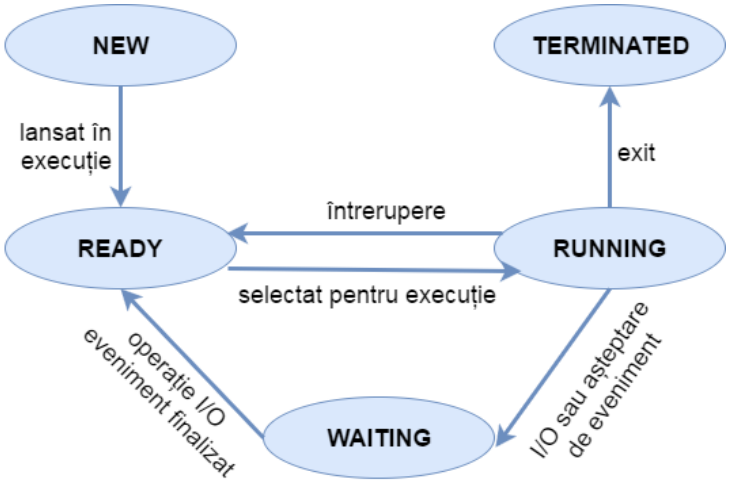

Figura 2. Diagrama de stări a unui proces

În timp ce un proces se execută, se schimbă și starea acestuia. Starea unui proces este definită de activitatea curentă a acestui proces. Fiecare proces poate fi într-una din următoarele stări:

• New: Procesul este creat.

• Running: Instrucțiunile sunt în curs de executare.

• Waiting: Procesul așteaptă ca un eveniment să aibă loc (cum ar fi finalizarea unei operații I/O sau primirea unui semnal).

• Ready: Procesul așteaptă să fie atribuit unui procesor.

• Terminated: Procesul și-a terminat execuția.

Firele de execuție, sau în engleză *threads*, sunt obiectele de activitate dintr-un proces. Fiecare thread include un contor de program unic, stivă de memorie și un set de regiștrii ai procesorului. Kernelul planifică thread-uri individuale, nu procese. În sistemele tradiționale Unix, fiecare proces consta dintr-un singur thread. Cu toate acestea, în sistemele moderne, programele multithread - care sunt formate din mai multe fire de execuție – sunt prezente în mod obișnuit. Linux are o implementare unică a thread-urilor: nu face diferența între thread-uri și procese. Pentru Linux, un thread este doar un tip special de proces.[2]

Cu toate acestea, o astfel de implementare a aplicațiilor multithread nu este foarte satisfăcătoare. De exemplu, să presupunem că un program de șah folosește două thread-uri: unul dintre ele controlează tabla de șah grafic, de așteptare pentru mișcările jucătorului uman și care arată mișcările calculatorului, în timp ce un al doliea thread pregătește următoarea mutare a jocului (jucatorului reprezentat de calculator). În timp ce primul thread așteaptă mutarea jucătorului uman, al doilea thread ar trebui să ruleze în mod continuu, exploatând astfel timpul de gândire al jucătorului uman. Cu toate acestea, în cazul în care programul de șah are doar un proces, primul thread nu poate emite un apel de blocare; în caz contrar, al doilea thread este blocat, de asemenea. În schimb, primul thread trebuie să folosească tehnici avansate neblocante, pentru a se asigura că procesul rămâne executabil.[1]

Capitolul 1: Concurenţă în kernel

Obiectivul programării multithreading este de a avea un proces care rulează în orice moment, pentru a maximiza utilizarea procesorului. Obiectivul este de partajare a timpului pentru a comuta între procese atât de frecvent încât utilizatorii să poată interacționa cu fiecare program în timp ce se execută. Pentru a îndeplini aceste obiective, planificatorul de procese selectează un proces disponibil (eventual dintr-un set de mai multe procese disponibile) pentru executarea acestuia pe CPU. Pentru un sistem cu un singur procesor, niciodată nu va fi mai mult de un singur proces care rulează. În cazul în care există mai multe procese, restul va trebui să aștepte până când procesorul este liber și poate fi reprogramat.

Fiecare proces are un identificator unic în sistemul de operare numit proces ID (PID). În sistemele Unix, fiecare proces are un proces părinte, iar procesele formează o ierarhie de procese. Identificatorul procesului parinte se numeste ppid (eng. Parent PID).

Un sistem de operare execută mai multe procese în același timp, gestionând accesul la resursele calculatorului pentru aceste procese și se asigură că niciun proces nu monopolizează aceste resurse un timp îndelungat. În practică, numarul de procese care rulează simultan este mai mic sau egal cu numărul de procesoare distincte din sistemul de calcul. Sistemul de operare creează iluzia execuției simultane a mai multor procese pe acelasi procesor, schimbând de mai multe ori pe secunda procesul care are acces la procesor.

Cuanta de timp este un parametru al sistemului de operare care decide cât poate un procesor rula neîntrerupt, și variază de la 0.1ms până la 100ms. O cuantă mai mică asigură o granularitate mai bună a împărțirii resurselor, însă rezultă un cost mai mare plătit pentru schimbările rapide de context între procese.

Un proces în execuție rulează până când:

• Îi expiră cuanta de timp alocată. În acest caz procesul va fi pus în starea „ready” și va astepta să fie planificat din nou.

• Asteaptă după un apel de I/O - procesul va fi trecut în coada „waiting” pentru resursa respectivă. Atunci cand cererea este finalizată, procesul revine în starea „ready”.

• Se termină.

Un kernel este preemptiv în cazul în care se poate produce o schimbare de proces în timp ce procesul înlocuit execută o funcție de kernel, în timp ce se execută în kernel mode. În realitate, în Linux (precum și în orice alt sistem de operare în timp real) lucrurile sunt mult mai complicate:

- atât în kernel-ul preemptiv, cât și în cel nonpreemptiv, un proces care rulează în kernel mode poate renunța în mod voluntar la accesul la CPU (de exemplu, pentru că trebuie să intre în așteptare pentru unele resurse). Acest tip de comutare între procese se numește o comutare planificată de procese (eng. planned process switch). Cu toate acestea, un nucleu preemptiv diferă de un nucleu nonpreemptiv prin modul în care un proces care rulează în kernel mode reacționează la evenimente asincrone care ar putea duce la comutarea unui proces în CPU (de exemplu, rutina

de tratare activează un proces cu prioritate mai mare. Acest tip de a comuta procesele se numește comutare forțată de proces (eng. forced process switch).

- atât în kernel-ul preemptiv, cât și în cel nonpreemptiv, o comutare de proces are loc atunci când un proces a terminat unele fire de execuție ale kernel-ului și programatorul de procese este invocat. Cu toate acestea, în nucleele nonpreemptive, procesul actual nu poate fi înlocuită dacă nu este pe cale de a trece la User Mode.

Prin urmare, principala caracteristică a unui Kernel preemptiv este aceea că un proces care rulează în Kernel Mode poate fi înlocuit cu un alt proces în timp ce acesta se află în mijlocul unei funcții Kernel.

O condiție de cursă poate apărea atunci când rezultatul unui calcul depinde de modul în care sunt imbricate două sau mai multe căi de control al kernel-ului intercalat. O regiune critică este o secțiune de cod care trebuie să fie complet executat de către calea de control al kernel-ului, înainte ca o altă cale de control al kernel-ului să poată fi setată.

Intercalarea căilor de control al kernel-ului complică dezvoltarea kernel-ului: trebuie să se aplice o atenție deosebită pentru a identifica regiunile critice din controlul execuției și firele de execuție ale kernel-ului. Odată ce o regiune critică a fost identificată, aceasta trebuie să fie protejată în mod corespunzător pentru a se asigura că în orice moment cel mult un thread al kernel-ului este în interiorul acelei regiuni.

Să presupunem, de exemplu, că două rutine de tratare diferite trebuie să acceseze aceeași structură de date care conține mai multe variabile, de exemplu, un buffer și un număr întreg care indică lungimea sa. Toate operațiile care afectează structura de date trebuie să fie puse într-o singură regiune critică. În cazul în care sistemul include un singur procesor, regiunea critică poate fi implementată prin dezactivarea întreruperilor în timpul accesării structurii de date partajate, deoarece îmbricarea căilor de execuție ale kernel-ului poate avea loc numai atunci când întreruperile sunt activate.

Pe de altă parte, în cazul în care aceeași structură de date este accesată numai de către rutina de serviciu de apeluri de sistem, iar sistemul include un singur procesor, regiunea critică poate fi protejată destul de simplu prin dezactivarea preemțiunii kernel-ului, în timp ce accesează structura de date partajate.

Lucrurile sunt mai complicate în sistemele multiprocesor. Mai multe procesoare pot executa cod kernel în același timp, astfel încât dezvoltatorii de kernel nu pot presupune că o structură de date este protejată deoarece kernel-ul de preemțiune este dezactivat, iar structura de date nu este niciodată accesată de o întrerupere sau o excepție.

Capitolul 2: Tehnici de sincronizare

Un proces cooperant reprezintă un proces care poate afecta sau poate fi afectat de alte procese executate în sistem. Procesele cooperante pot partaja fie în mod direct un spațiu de adrese logice (adică, atât codul și date), fie să li se permită să facă schimb de date numai prin fișiere sau mesaje. Primul caz se realizează prin utilizarea de thread-uri. Accesul concurent la date partajate poate duce la incoerența datelor. În continuare, vom prezenta diferite mecanisme pentru a asigura executarea ordonată a proceselor care au în comun un spațiu de adresă logică, astfel încât coerența datelor să fie menținută.

2.1. Alocarea de date unice pentru fiecare processor (Per-CPU variables)

Cea mai bună tehnică de sincronizare constă în proiectarea kernel-ului, astfel încât să se evite nevoia de sincronizare. Fiecare sincronizare explicită are un cost semnificativ de performanță.

Cea mai simplă și cea mai eficientă tehnică de sincronizare constă în a declara variabilele de kernel ca variabile per-CPU. Practic, o variabilă per-CPU este o matrice de date, cu un element pentru fiecare procesor din sistem.

Un procesor nu ar avea acces la elementele de matrice corespunzătoare celorlalte procesoare; pe de altă parte, poate citi și modifica în mod liber propriile sale elemente, fără teama de condițiile de cursă, deoarece acesta este singurul CPU care are dreptul să facă acest lucru. De asemenea, acest lucru înseamnă că variabilele per-CPU pot fi utilizate practic numai în cazuri particulare, atunci când are sens să se împartă datele între CPU-urile sistemului.

Elementele de matrice per CPU sunt aliniate în memoria principală, astfel încât fiecare structură de date este stocată într-o unitate diferită de cache hardware. Prin urmare, accese simultane la matricea per-CPU nu conduc la concutență și invalidare, operații costisitoare în ceea ce privește performanța sistemului.

Există mai multe beneficii la utilizarea variabilelor per-CPU. Un prim avantaj este reducerea cerințelor de blocare. În funcție de semantica prin care procesoarele accesează datele per-CPU, s-ar putea evita nevoia de metodelor de blocare. Regula ,,numai acest procesor accesează aceste date" este doar o convenție de programare. Este nevoie de a se verifica faptul că procesorul local accesează doar datele sale unice.[3]

În al doilea rând, datele per-CPU reduc șansele de invalidare a memoriei cache. Acest fenomen apare când procesoarele încerca să păstreze cache-urile în sincronizare. În cazul în care un procesor modifică datele deținute de memoria cache a unui alt procesor, acel procesor trebuie să actualizeze memeoria cache. Invalidarea constantă a cache-ul este numită *thrashing the cache* și afectează performanțele sistemului. Variabilele per-CPU păstrează efecte negative asupra cache-ului la un nivel minim, deoarece procesoarele, în mod ideal, accesează numai propria lor memorie de date.[3]

În timp ce variabilele per-CPU oferă o protecție împotriva accesări simultane de la mai multe procesoare, acestea nu oferă protecție împotriva căi de acces la funcțiile asincrone (întreruperi). În aceste cazuri, sunt necesare metode suplimentare de sincronizare.[1]

Prin urmare, utilizarea de date per-CPU elimină de multe ori (sau cel puțin încearcă să minimizeze) nevoia de blocare. Singura cerință de siguranță pentru utilizarea datelor per-CPU este dezactivarea preemțiunii kernel-ului, care este mai ușor de realizat decât blocarea, iar interfața sistemului face acest lucru în mod automat.

2.2. Operaţii atomice în sisteme multiprocessor

Mai multe tipuri de instrucțiuni ale limbajului de asamblare sunt de tip "citeste-modifică-scrie" (eng. read-modify-write), care accesează o locație de memorie de două ori, prima dată pentru a citi valoarea veche, iar a doua oară pentru a scrie o nouă valoare.

Să presupunem că două căi de control al execuției ale kernel-ului, care rulează pe două procesoare, încercă să "citiească-modifice-scrie" aceeași locație de memorie, în același timp, prin executarea operațiunilor non-atomice. La început, ambele procesoare încearcă să citească aceeași locație, dar arbitrul de memorie (un circuit hardware care serializează accesul la chip-uri RAM) întervine pentru a permite accesul unuia dintre procesoare și îl întârziere pe celălalt. Cu toate acestea, în cazul în care prima operațiune de citire s-a finalizat, procesorul întârziat citește exact aceeași valoare (veche) din locația de memorie. Apoi, ambele procesoare încercă să scrie același valoare (nouă) la locația de memorie; din nou, accesul de memorie pe magistrală este serializat de arbitrul de memorie, și în cele din urmă ambele operații de scriere vor reuși. Cu toate acestea, rezultatul la finalul operațiilor este incorect deoarece ambele procesoare scriu același valoare (nouă). Astfel, cele două operații "citire-modificare-scriere", acționează ca una singură.[1]

Thread 1	Thread 2
citește i(5)	citește i(5)
incrementează i(5→6)	----------
----------	incrementează i(5→6)
scrie i(6)	----------
----------	scrie i(6)

Cel mai simplu mod de a preveni condițiile de cursă din cauza instrucțiunilor "citire-modificare-scriere" este prin asigurarea faptului că astfel de operațiuni sunt atomice, la nivel hardware. Orice astfel de operații trebuiesc executate într-o singură instrucțiune, fără a fi întreruptă în mijlocul execuției și evitând accesul la aceeași locație de memorie de alte procesoare.

Thread 1	Thread 2
citește, incrementează și scrie i(5→6)	----------
----------	citește, incrementează și scrie i(5→6)

sau

Thread 1	Thread 2
----------	citește, incrementează și scrie i(5→6)
citește, incrementează și scrie i(5→6)	----------

În exemplul de mai sus, valoarea finală, mereu șase, este corectă. Niciodată nu este posibil ca cele două operații atomice să folosească aceeași variabilă, în aceeași timp, concurent .Prin urmare, condițiile de cursă nu sunt posibile pentru operația de incrementare.

Kernel-ul oferă două seturi de interfețe pentru operațiile atomice: una care operează pe numere întregi și una care operează pe interfețele cu biți. Aceste interfețe sunt puse în aplicare pe orice arhitectură care suportă Linux. Cele mai multe arhitecturi conțin instrucțiuni care furnizează versiuni atomice ale operații aritmetice simple. Alte arhitecturi, lipsite de operații atomice directe, furnizează o operație pentru a bloca magistrala de memorie pentru o singură operație, garantând astfel că o altă operațiune care afectează memoria nu poate avea loc simultan cu o altă operație.[1]

2.3. Spin locks

O tehnică de sincronizare utilizată pe scară largă este tehnica de blocare. Atunci când o cale de execuție a kernel-ului trebuie să acceseze o structură de date partajată sau o regiune critică, aceasta trebuie să achiziționeze o "cheie" pentru ea. O resursă protejată printr-un mecanism de blocare este destul de similar cu o resursă limitată într-o cameră a cărei ușă este blocată, când cineva este înăuntru. În cazul în care o cale de execuție a kernel-ului dorește să acceseze resursa, încearcă să "deschidă ușa" prin achiziționarea unei “chei”. Reușește numai în cazul în care resursa este liberă. Apoi, cât timp dorește să folosească resursa, ușa rămâne blocată. În cazul în care calea de execuție a kernel-ului eliberează dispozitivul de blocare, ușa este deblocată și o altă cale de execuție al kernel-ului poate intra în cameră. [1]

Presupunem situația în care cinci căi de execuție ale kernel-ului (P0, P1, P2, P3, P4) încearcă să acceseze două regiuni critice (C1 și C2). Calea de execuție P0 este în interiorul regiunii C1, în timp ce P2 și P4 așteaptă să acceseze C1. În același timp, P1 este în interiorul regiunii C2, în timp ce P3 așteaptă să acceseze C2. Se observă că P0 și P1 ar putea rula simultan. Dispozitivul

de blocare pentru regiunea critică C3 este "deschisă", deoarece nicio cale de execuție a kernel-ului nu o accesează (Figura 2.1).

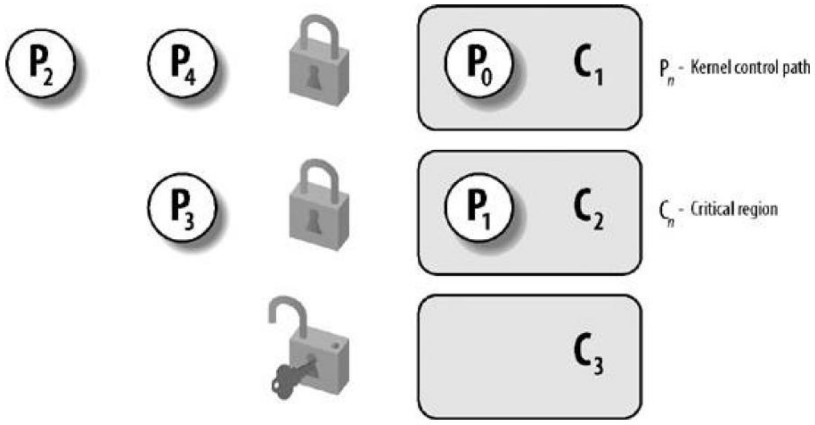

Figura 2.1. Protejarea regiunilor critice, cu mai multe încuietori[1]

Spin locks sunt un tip special de blocare proiectat pentru a lucra într-un mediu multiprocesor. În cazul în care calea de execuție a kernel-ului găsește spin lock-ul "deschis", acesta dobândește cheia de blocare și continuă execuția acestuia. Pe de altă parte, în cazul în care calea de execuție a kernel-ului găsește dispozitivul de blocare "închis", printr-un traseu de control al kernel-ului care rulează pe un alt procesor, acesta "se învârte" în jurul valorii, executând în mod repetat, o buclă de instrucțiuni, până când dispozitivul de blocare este eliberat.

Faptul că o blocare de tipul spin lock provoacă căilor de execuție să se , "rotească" (în esență, pierdem timpul procesorului) în timp de așteptare pentru blocare este remarcabil. Acest comportament reprezintă punctul de blocare de spin. Nu este recomandat să se folosească un sistem de blocare de spin pentru o lungă perioadă de timp. Aceasta este natura de blocare de spin: un sistem de blocare ușor cu un singur titular, care ar trebui să aibă loc pentru durate scurte. Un comportament alternativ când este susținut de blocare, este de a pune thread-ul curent în starea , "sleep" și de al trece în starea "awake" atunci când acesta devine disponibil. Apoi procesorul poate executa alt cod. Acest lucru are nevoie de coordonare, pentru a comuta din "awake" în "sleep" și înapoi thread-ul blocat, ceea ce necesită mult mai mult cod decât cele câteva linii utilizate pentru a pune în aplicare un sistem de blocare de spin. Prin urmare, este recomandat să se folosească încuietori de spin pentru o durată mai mică decât cea de comutare între cele 2 stări. [2]

Mecanismul de blocare poate avea loc simultan cu cel mult un singur fir de execuție. În consecință, doar un singur thread este acceptat în zona critică, la un moment dat. Acest lucru asigură protecția necesară de pe sistemele de calcul cu concurență datorată procesoarelor multiple. Pe sistemele uniprocesor, încuietorile nu există; acestea acționează ca markeri pentru a dezactiva și activa preemțiunea kernel-ului. În cazul în care preemțiunea kernel-ului este oprită, mecanismele de blocare dispar în întregime.

Spin locks pot fi folosite în rutine de întreruperi, în timpul în care semafoarele nu poate fi utilizate, deoarece sunt în modul "sleep". În cazul în care un sistem de blocare este utilizat într-o rutină de tratare a întreruperii, trebuie să dezactivați, de asemenea, întreruperile locale (cereri de întrerupere pe procesorul curent) înainte de a obține cheia de blocare. În caz contrar, este posibil ca o rutină de tratare a întreruperii să întrerupă codul kernel-ului în timp ce mecanismul de blocare are loc și încercă să redobândească imeidat cheia de blocare. Rutina de întrerupere se "învârte", așteptând ca accesul la o zonă critică să devină disponibil. Acesta este un exemplu de dublu blocaj. Trebuie dezactivată întreruperea numai pe procesor curent. Dacă apare o întrerupere pe un alt procesor, și se "rotește" în aceeași blocare, aceasta nu împiedică titularul de blocare (care se află pe un alt procesor) de la eliberarea cheii de blocare.

2.4. Read/Write Spin Locks

Blocările spin de tip citire/scriere (eng. Read/Write Spin Locks) au fost introduse pentru a crește concurența în interiorul kernel-ului. Acestea permit mai multor căi de execuție ale kernel-ului pentru a citi simultan aceeași structură de date, atâta timp cât nicio cale de execuție a kernel-ului nu modifică. În cazul în care un fir de execuție al kernel-ului dorește să scrie în structura de date, aceasta trebuie să achiziționeze cheia de scriere de blocare, care oferă acces exclusiv la resursă. Permițând citirea în paralel a structurilor de date se îmbunătățește performanța sistemului.[1]

Figura 2.2 ilustrează două regiuni critice (C1 și C2), protejate cu încuietori de tip citire/scriere. Căile de execuție ale kernel-ului, . În timp ce calea de execuție a kernel-ului W1 scrie în structura de date C2, atât R2 și W2 sunt în așteptare pentru a dobândi cheia de blocare pentru citire, respectiv scriere.

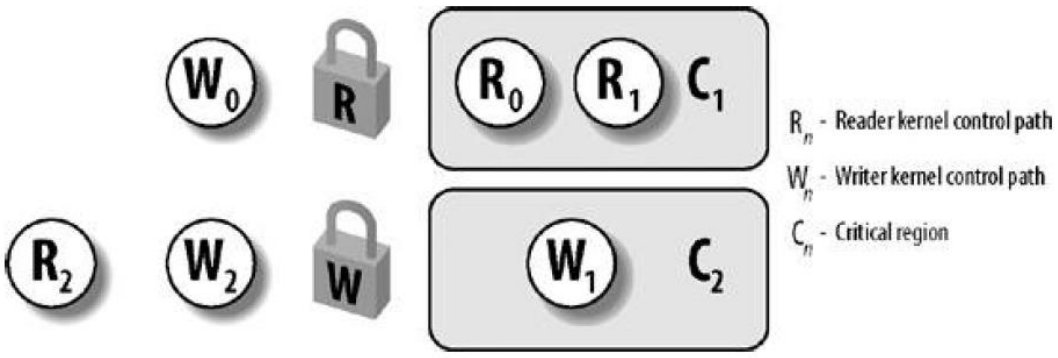

Figura 2.2. Read/write spin locks[1]

Un lucru important în utilizarea încuietorilor spin de tipul citire/scriere este că acestea favorizează cititorii, în detrimentul scriitorilor. În cazul în care dispozitivul de blocare de citire este reținut și un scriitor așteaptă pentru acces exclusiv, cititorii care încearcă să achiziționeze

cheia de blocare, continuă să reușească. Scriitorul nu dobândește cheia de blocare până când toți cititorii nu eliberează dispozitivul de blocare. Prin urmare, un număr suficient de mare de scriitori pot aștepta un timp îndelungat după cititori. Uneori, acest comportament este benefic, uneori reprezintă o problemă.

Spin locks asigură o blocare rapidă și simplă. Comportamentul reprezintă o soluție optimă pentru durate scurte de așteptare și un cod care nu poate intra în sleep. În cazurile în care timpul de sleep ar putea fi prea mare, semaforul reprezintă o soluție.

2.5. Seqlocks

Atunci când se utilizează încuietori spin de tip citire/scriere, cererile emise de căile de execuție ale kernel-ului pentru a efectua o read_lock sau o operațiune write_lock au aceeași prioritate: cititorii trebuie să aștepte până când scriitorul a terminat și, în mod similar, un scriitor trebuie să aștepte până când toți cititorii au terminat.

Seqlocks sunt similare cu încuietorile spin de citire/scriere, cu excepția faptului că acestea dau o prioritate mult mai mare pentru scriitori: de fapt, unui scriitor îi este permis să procedeze chiar și atunci când cititorii sunt activi. Avantajul acestei strategii este că un scriitor nu așteaptă (cu excepția cazului în alt scriitor este activ); partea negativă este că un cititor poate fi uneori forțat să citească aceleași date de mai multe ori, până când acestea devin o copie validă.

Fiecare seqlock reprezintă o structură *seqlock_t* formată din două câmpuri: un câmp de blocare de tip *spinlock_t* și un câmp de *secvență* (eng. sequence) întreg. Cel de-al doilea câmp joacă rolul unui contor de secvență. Fiecare cititor trebuie să citească acest contor secvență de două ori, înainte și după citirea datelor, și să verifice dacă cele două valori coincid. În caz contrar, un nou scriitor a devenit activ și a crescut contorul de secvență, astfel, spune implicit cititorului că datele citite anterior nu mai sunt valide.

O variabilă *seqlock_t* este inițializată cu "deblocat", fie prin atribuirea valorii *SEQLOCK_UNLOCKED*, fie prin executarea macro-ului *seqlock_init*. Scriitorii dobândesc și eliberează un *seqlock* prin execuția *write_seqlock()* și *write_sequnlock()*. Prima funcție dobândește blocarea de tip spin în structura de date *seqlock_t*, apoi incrementează contorul de secvență cu o unitate. Cea de-a doua funcție incrementează contorul de secvență încă o dată, apoi eliberează blocarea de tip spin. Acest lucru asigură faptul că, atunci când scriitorul este în mijlocul scrierii, contorul este impar, și că, atunci când niciun scriitor nu modifică date, contorul este par.

Atunci când un cititor intră într-o regiune critică, acesta nu are nevoie să dezactiveze kernel-ul de preemțiune; pe de altă parte, scriitorul dezactivează automat kernel-ul de preemțiune atunci când intră în regiunea critică, deoarece acesta dobândește blocare de spin.

Nu orice fel de structură de date poate fi protejată printr-un seqlock. Ca regulă generală, următoarele condiții trebuiesc îndeplinite:

- Structura de date care urmează să fie protejată nu include *pointers* care sunt modificați de către scriitorii și dereferențiați de către cititori (în caz contrar, un scriitor ar putea schimba pointer-ul fără ca un cititor să-și dea seama).
- Codul din regiunile critice ale cititorilor nu se modifică (în caz contrar, citiri multiple ar avea soluții diferite, de la o citire la alta)

Regiunile critice ale cititorilor ar trebui să fie scurte, iar scriitorii ar trebui să dobândească rareori *seqlock*, altfel accese repetate de citire ar provoca un blocaj sever. O utilizare tipică a *seqlocks* în Linux constă în protejarea unor structuri de date legate de manipularea timpului de sistem.

2.6. Read-Copy Update

Read-Copy Update (RCU) reprezintă o tehnică de sincronizare concepută pentru a proteja structurile de date care sunt, în general, accesate pentru citirea de către mai multe procesoare. RCU permite mai multor cititori și mai multor scriitori să execute în același timp (o îmbunătățire față de seqlocks, care permit unui singur scriitor pentru a continua). Mai mult, RCU este *block free*, care nu folosește nicio tehnică de blocare sau contor partajat de către toate procesoarele. Acest lucru reprezintă un mare avantaj față de încuietori de spin și seqlocks citire/scriere, care provoacă probleme de sincronizare din cauza invalidării cache-ului.[1]

RCU reușește să sincronizeze mai multe procesoare, fără structuri de date comune, prin limitarea sferei de Read-Copy Update după cum urmează:

- Numai structurile de date care sunt alocate în mod dinamic și sunt referențiate prin intermediul unor *pointers* pot fi protejate prin RCU.
- Nicio cale de execuție a kernel-ului nu poate intra în modul , "sleep" într-o regiune critică protejată de Read-Copy Update.

Atunci când o cale de execuție a kernel-ului vrea să citească o structură de date protejate cu RCU, execută macro-ul *rcu_read_lock()*, care este echivalent cu *preempt_disable()*. În continuare, cititorul dereferențiază pointer-ul către structura de date și începe să citească. După cum s-a precizat anterior, cititorul nu poate intra în , "sleep" până când nu se termină citirea structurii de date; la finalul accesării regiunii critice se execută macro-ul *rcu_read_unlock()*, care este echivalent cu *preempt_enable()*.

Deoarece cititorul face foarte puțin pentru a preveni condițiile de cursă, ne-am putea aștepta ca scriitorul să aștepte putin mai mult. De fapt, atunci când un scriitor dorește să actualizeze structura de date, acesta dereferențiază pointer-ul și face o copie a întregii structuri de date. În continuare, scriitorul modifică acea copie. Odată ce a terminat, scriitorul schimbă pointer-ul la structura de date, astfel încât să indice către copia actualizată. Pentru că modificarea valorii pointer-ului este o operație atomică, fiecare cititor sau scriitor vede fie copia veche sau pe cel nouă: nu poate să apară o invalidare în structura de date. Cu toate acestea, este necesară o barieră de memorie pentru a se asigura că pointer-ul actualizat este văzut de către celelalte CPU-uri numai

după ce structura de date a fost modificată. O astfel de barieră de memorie este introdusă în mod implicit în cazul în care un sistem de blocare de spin este cuplat cu RCU pentru a interzice executarea concomitentă a scriitorilor.

Problema care apare la tehnica RCU este faptul că vechea copie a structurii de date nu poate fi eliberată imediat ce scriitorul actualizează pointer-ul. De fapt, cititorii care au acces la structura de date atunci când scriitorul a început actualizarea acestuia încă pot citi vechea copie. Aceasta poate fi eliberată numai după ce toți (potențialii) cititori au executat instrucțiunea *rcu_read_unlock()*. Kernel-ul cere fiecărui potențial cititor de a executa această instrucțiune înainte ca: • Procesorul să efectuează o comutare de proces.

• Procesorul să înceapă să execute în User Mode.

• Procesorul să execute în buclă.

Read-Copy Update este utilizat la nivelul de rețea și în Virtual Filesystem.

2.7. Excluderea mutuală

Dezactivarea întreruperilor

Pe un sistem cu un singur procesor, cea mai simplă soluție este aceea ca fiecare proces să dezactiveze toate întrerupe imediat după intrarea într-o regiune critică și să le reactiveze chiar înainte de a părăsi regiunea critică. Cu întreruperile dezactivate, nu pot avae loc întreruperi de ceas. CPU este pornit doar de la proces la proces, ca urmare a ceasului sau a altor întreruperi, ținând cont că are întreruperile oprite, procesorul nu va trece la un alt proces. Astfel, odată ce un proces a dezactivat întreruperile, poate examina și actualiza memoria partajată, fără teama că orice alt proces va interveni. [4]

Această abordare este evitată, în general, deoarece nu este recomandat ca proceselor din zona utilizatorilor să li se ofere posibilitatea de a utiliza întreruperi. Mai mult, în cazul în care sistemul este unul multiprocesor (cu două sau mai multe procesoare) dezactivarea întreruperilor afectează doar CPU-ul care execută instrucțiunea *disable*. Celelalte vor continua să ruleze și pot accesa memoria partajată.

Pe de altă parte, este adesea convenabil pentru kernel în sine de a dezactiva întreruperile pentru câteva instrucțiuni în timp ce scrie variabile sau, mai ales, liste. În cazul în care o întrerupere apare în timp ce , spre exemplu, este modificată lista de procese în strarea *ready*, aceasta este într-o stare inconsistentă, lucru care ar putea duce la condițiile de cursă. [4]

Concluzia este: dezactivarea întreruperilor este adesea o tehnică utilă în cadrul sistemului de operare în sine, dar nu este adecvat să fie folosită ca un mecanism general de excludere reciprocă pentru procesele utilizator.

Variabile de blocare

Fie o singură variabilă partajată (blocată), inițial 0. Când un proces dorește să intre în regiunea sa critică, ea testează mai întâi dacă regiunea este blocată sau nu. Dacă blocarea este 0, procesul o modifică la 1 și intră în zona critică. Dacă blocarea este deja 1, procesul așteaptă până când acesta devine 0. Astfel, 0 înseamnă că niciun proces nu accesează regiunea sa critică, iar 1 înseamnă că un proces este în zona critică.

Din păcate, această idee conține exact același defect fatală pe care am văzut în directorul spoolerul. Să presupunem că un singur proces vrea să acceseze regiunea critică și vede că este 0. Înainte de a putea seta blocarea la 1, un alt proces este programat, execută și setează blocarea la 1. Când primul proces execută din nou, se va observa, de asemenea, blocarea 1,iar două procese vor fi în regiuni critice în același timp.

2.8. Excludere folosind *Sleep* și *Wakeup*

Se consideră un calculator cu două procese, *H*, cu prioritate ridicată, și *L*, cu prioritate scăzută. Regulile de planificare sunt de așa natură încât *H* este rulat ori de câte ori este în starea *ready*. La un moment dat, cu *L* în regiunea critică, *H* devine gata pentru a rula (de exemplu, o operație I/O se finalizează). *H* intră acum în așteptare ocupat, dar din moment ce *L* nu este programat să ruleze în timp ce se execută *H*, *L* nu are șansa de a părăsi regiunea critică, astfel încât *H* așteaptă în buclă pentru totdeauna. Această situație este uneori menționată ca problema priorității inverse.

Una dintre cele mai simple soluții o reprezintă perechea *sleep* și *wakeup*. *Sleep* este un apel de sistem care determină prosesul apelat să fie blochat, adică să fie suspendat, până când un alt proces îl trezește. Apelul de trezire are un singur parametru, procesul care urmează să fie trezit. În mod alternativ, atât *sleep* cât și *wakeup* un singur parametru, o adresă de memorie utilizată pentru a se compara cu *sleeps* și *wakeups*.

2.9. Semafoare

În 1965 E. W. Dijkstra a sugerat utilizarea unei variabile de tip întreg pentru a contoriza numărul de *wakeups* salvate, pentru o utilizare ulterioară. În propunerea sa, un nou tip de variabilă, pe care el a numit-o *semafor*, a fost introdusă. Un semafor ar putea avea valoarea 0, indicând faptul că nu au fost salvate *wakeups*, sau o valoare pozitivă (>0) în cazul în care una sau mai multe *wakeups* au fost în așteptare.

Dijkstra a propus realizarea a două operații asupra semafoarelor, ulterior numite *down* și *up* (generalizări pentru *sleep*, respective *awake*). Operația *down* realizată asupra semafoarelor verifică dacă valoarea este mai mare decât 0. Dacă da, decrementează valoarea și continuă. În cazul în care valoarea este 0, procesul este pus în modul "sleep" fără a realiza operația *down*, pentru moment. Verificarea valorii, modificarea acesteia și, eventual, modul "sleep", toate acestea sunt făcute ca o singură acțiune atomică, indivizibilă. Este garantat că, odată ce a început o operațiune a semaforului, nici un alt proces nu poate avea acces la semafor până când operația nu s-a încheiat sau blocat. Operațiile atomice sunt absolut esențiale pentru rezolvarea problemelor de sincronizare și pentru evitarea condițiilor de cursă. [4]

Operația *up* incrementează valoarea semaforului adresat. În cazul în care unul sau mai multe procese au fost în modul "sleep" pe acel semafor, adică în imposibilitatea de a finaliza o operație anterioară de tipul *down*, unul dintre ele este ales de către sistem (de exemplu, la întâmplare) și este lăsat să finalizeze operația sa *down*. Astfel, după o pe un semafor cu procesele de dormit pe ea, semaforul va fi în continuare 0, dar va exista un singur proces de dormit mai puțin pe ea. Operarea de incrementare a semaforului și trecerea în modul "awake" la un proces sunt, de asemenea, indivizibile. Niciun proces nu poate bloca operația *up*, la fel cum niciun proces nu poate bloca trecerea în modul "awake", în modelul anterior.[4]

Rezolvarea problemei producător-consumator folosind semafoare

Această soluție utilizează trei semafoarele: unul numit *full* pentru contorizarea numărului de sloturi care sunt pline, unul numit *empty* pentru contorizarea numărului de sloturi care sunt goale și unul numit *mutex* pentru a se asigura că producătorul și consumatorul nu avea acces la "depozit" în acelasi timp. Semaforul *ful* este inițial 0, *emplty* este inițial egal cu numărul de sloturi din buffer ("depozit"), iar *mutex* este inițial 1. Semafoarelor care sunt inițializate cu valoarea 1 și sunt utilizate de două sau mai multe procese pentru a se asigura că numai unul dintre ele poate intra în regiunea critică, la un moment de timp, se numesc *semafoare binare*. În cazul în care fiecare proces are un *down* chiar înainte de a intra în regiunea critică și un *up* imediat după părăsirea acesteia, excludere reciprocă este garantată.[4]

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

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

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

Figura 2.3. Problema producător-consumator folosind semafoare[4]

Într-un sistem care utilizează semafoare, modul natural de a ascunde întreruperile este acela de a avea un semafor stabilit inițial la 0, asociat cu fiecare dispozitiv I/O. Imediat după începerea utilizării unui dispozitiv I/O, procesul de gestionare realizează operația *down* pe semaforul asociat, blocând astfel imediat zona de memorie. În cazul în care vine o întrerupere, rutina de tratare a întreruperii realizează operația *up* pe semaforul asociat, ceea ce face ca procesul să fie gata pentru a rula din nou.

În exemplul din Figura 2.3. s-au folosit semafoarele în două moduri diferite. Semaforul *mutex* este folosit pentru excludere reciprocă. Acesta este conceput pentru a garanta că doar un singur proces, la un moment dat, va fi citit sau scris în buffer și variabilele asociate. Această excludere reciprocă este necesară pentru a preveni un haos. Cealaltă utilizarea a semafoarelor este pentru sincronizare. Sunt necesare semafoarelor *full* și *empty* pentru a garanta că anumite secvențe de evenimente apar sau nu. În acest caz, se asigură că producătorul se oprește atunci când buffer-ul ("depozitul") este plin, iar consumatorul încetează să mai consume atunci când acesta este gol. Această utilizare este diferită de excludere reciprocă.[4]

Concluzii

Având în vedere o colecție de procese secvențiale care au în comun date, trebuie să se execute excluderea reciprocă pentru a se asigura că o secțiune critică de cod este utilizat de către un singur proces sau thread la un moment dat. De obicei, hardware-ul de calculator oferă mai multe soluții care asigură excluderea reciprocă. Cu toate acestea, astfel de soluții bazate pe hardware sunt prea complicate pentru majoritatea dezvoltatorilor pentru a le implementa.

Diverse probleme de sincronizare (cum ar fi problema memoriei cache, problema cititori-scriitori) sunt importante, în principal, deoarece acestea reprezintă exemple ale unui mari clase de probleme de control a concurenței. Aceste probleme sunt folosite pentru a testa aproape fiecare metodă de sincronizare recent propusă.

Sistemul de operare trebuie să ofere suport pentru sincronizare. De exemplu, majoritatea dezvolatorilor de sisteme de operare oferă mecanisme, cum ar fi, mutex, semafoare și spinlocks pentru a controla accesul la datele partajate.

O tranzacție reprezintă o unitate de program, care trebuie executată atomic: fie toate operațiile asociate acesteia sunt executate până la finalizare, fie niciuna nu este efectuată. Pentru a asigura atomicitatea cu scopul de a evita eșecul în sistem, putem folosi un set de log-uri modificate înaintea unor operații atomice. Toate actualizările sunt înregistrate în log-uri, care sunt păstrate într-o zonă de memorie stabilă. În cazul în care se produce un eroare de sistem, informațiile din log-uri sunt utilizate în restabilirea stării elementelor de date actualizate, care se realizează prin utilizarea operațiilor *undo* și *redo*. Pentru a reduce timpul pentru căutarea în log-uri după ce s-a produs o eroare de sistem, putem folosi un sistem de verificare.

Bibliografie

[1]. Understanding the Linux Kernel, 3rd Edition, Daniel P. Bovet, Marco Cesati, O'Reilly, 2005;

[2]. *Linux Kernel Development*, Third Edition, Robert Love, Pearson Education, 2010;

[3]. *Operating System*, 8th Edition, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, John Wiley & Sons Inc., 2012;

[4]. *MODERN OPERATING SYSTEMS*, 4th Edition, Andrew S. Tanenbaum, Herbert Bos, Vrije Universiteit, Amsterdam, The Netherlands, Pearson Education, 2015;

http://www.cs.rpi.edu/~goldsd/fall2014-csci4210.php