Mecanisme de sincronizare

TRIFAN ALEXANDRU

Cuprins

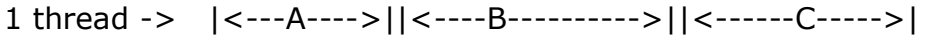
1. Motivație

Se va descrie cum sistemul de operare incearca rezolvarea problemelor de sincronizare la nivelul task-urilor, rezolvarea problemelor de consum de curent, de viteza, de context switching, probleme de memorie generate de lucru pe multiple fire de executie.

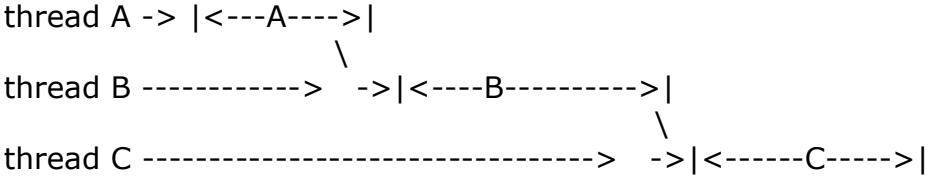
Prezentăm mecanismele curente ale kernelului de Linux, ce mecansime de sincronizare are cum rezolvă problemele de multi tasking, ce aduce si ce implicații au aceste mecanisme de sincronizare și de switching.

2. Notiuni despre ideeea de sincron, asincron

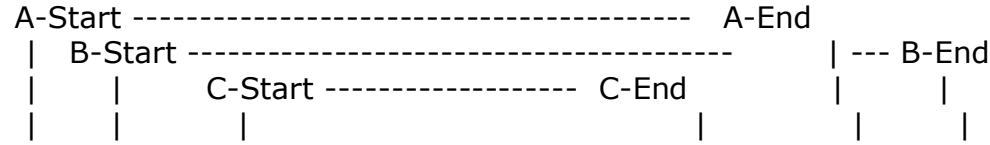
Sincron (un singur fir de execuție):



Sincron (mai multe fire de execuție):



Asincron (un singur fir de execuție):



```
              V     V          V                                  V           V       V
1 thread->|<-A-|<--B---|<-C-|-A-|-C-|--A--|-B-|--C-->|---A---->|--B-->|
```

Asincron (mai multe fire de execuție):

```
 thread A ->     |<---A---->|
 thread B ----->     |<----B---------->|
 thread C --------->     |<------C--------->|
```

Punctele de început și de capăt ale sarcinilor A, B, C sunt reprezentate de caracterele <,>. Feliile de timp ale CPU-ului sunt reprezentate prin bare verticale |

Sincronizat înseamnă "conectat" sau "dependent" într-un anumit fel. Cu alte cuvinte, cele două sarcini sincrone trebuie să se cunoască una pe cealaltă, și trebuie să se execute într-un mod care este dependentă de cealaltă. În cele mai multe cazuri, acest lucru înseamnă că un task nu poate începe până când celălalt nu a terminat. Asincron înseamnă că sunt total independente și nici unul nu trebuie să ia în considerare celălalt task în nici un fel, nici în inițierea și nici în execuție.

Ca o paranteză, trebuie menționat că din punct de vedere tehnic, conceptul de sincron /asincron nu are nimic de a face cu firele de execuție. Cu toate că, în general, ar fi neobișnuit pentru a găsi sarcini asincrone care rulează pe același fir, este posibil (a se vedea în exemplul de mai sus) dar este comun să întâlnim două sau mai multe sarcini de executat sincron pe fire separate.

Conceptul de sincron / asincron are de a face numai cu faptul dacă este sau nu posibil ca un al doilea sau ulterior task să poate fi inițiat înainte de cealalt (primul task) înainte ca acesta să fi finalizat, sau dacă acesta trebuie să aștepte. Nu este relevant pe ce fir de execuție sau ce proces sau ce procesoare sau hardware se execută taskul.

Prelucrarea asincronă: În rezolvarea multor probleme de inginerie, software-ul este proiectat pentru a împărți problema generală în sarcini individuale multiple, și apoi să le executăm asincron. Inversarea unei matrice, sau o problemă de analiză cu element finit, sunt exemple bune. În calcul, sortarea unei liste este un exemplu. Rutina rapidă de sortare, de exemplu, împarte lista în două liste, și sortează fiecare dintre ele într-un mod recursiv. În ambele exemple de mai sus, cele două sarcini pot (și de multe ori au fost) executate asincron. Task-urile nu au avut nevoie să fie executate pe fire separate. Chiar și o mașină cu un singur procesor, și doar un singur fir de execuție poate fi programată pentru a iniția procesarea unei a doua sarcini înainte ca prima să fie gata. Singurul criteriu este că rezultatele unei sarcini să nu necesar intrării în cealaltă sarcină. Atâta timp cât începerea și de terminarea sarcinilor se suprapun, (posibil numai în cazul în care este necesară ieșirea unui task ca intrare a celuilalt), ele sunt executate asincron, indiferent cât de multe fire de execuție sunt în uz.

Prelucrarea sincronă: Orice proces care constă din mai multe sarcini, în cazul în care sarcinile trebuie să fie executate în ordine, dar unul dintre taskuri trebuie să fie executate pe o altă mașină(request HTTP). Dacă acesta din urmă are loc pe o mașină separată, atunci el va fi executat pe un fir separat, fie sincron sau asincron

3. Thread-uri

Thread-urile reprezintă varianta ușoară a proceselor LWP(Light Weight Processes). Un proces are 5 părți fundamentale: cod, date, stivă de memorie, adresare I/O și tabele de semnale. În cazul proceselor standard HWP în cazurilor switch-urilor de context au loc un set foarte complicat de operații care necesită mult timp (toate tabelele trebuie golite din procesor pentru a face o schimbare de context pentru diferite taskuri) în plus singura metoda de a partaja informație între procese este prin tunele de comunicație (pipes) și prin memorie partajată. Dacă un proces creează un proces copil cu ajutorul rutinei fork() singura parte partajată este codul „textul”.

Thread-urile reduc latențele prin partajarea părților fundamentale. Datorită acestei partajare, comutarea se poate face mult mai des și mai eficient în cazul firelor de execuție. De asemeni partajarea informației nu mai este atât de dificilă. În sistemele de operare există două tipuri de thread-uri cele din „spațiu utilizator” și cele din „spațiu kernel”.

3.1. Thread-uri din spațiu utilizator

Aceste fire de execuție nu fac apeluri la kernel și își gestionează în mod independent tabelele. Adesea aceste fire de execuție se mai numesc și „multitasking cooperativ” unde task-ul definește un set de rutine care ajung să fie executate prin manipularea registrului SP (stack pointer). În mod standard fiecare thread cedează unitatea centrală de calcul CPU prin apelul unui comutator explicit prin care se trimite un semnal sa se execută o operație care implică acest comutator. De asemeni un semnal de ceas poate forța comutarea. Thread-urile din spațiu utiliator comuta mai repede decât cele din spațiu kernel (totuși în Linux comutarea thread-urilor de kernel are chiar timpi comparativi).

Kernel-ul nu este contient de existenţa lor, şi managementul lor este făcut de procesul în care ele există, folosind de obicei o bibliotecă. Astfel, schimbarea contextului nu necesită intervenţia kernel-ului, iar algoritmul de planificare depinde de aplicaţie.

3.1.1. Avantaje :

- schimbarea de context nu implică kernelul, deci avem o comutare rapidă planificarea poate fi aleasă de aplicaţie si deci se poate alege una care să favorizeze creşterea vitezei aplicaţiei noastre
- thread-urile pot rula pe orice SO, deci şi pe cele care nu suportă thread-uri (au nevoie doar de biblioteca ce le implementează)

3.1.2. Dezavantaje :

- kernel-ul nu tie de thread-uri, deci dacă un thread apelează ceva blocant toate thread-urile planificate de aplicaţie vor fi blocate. Cele mai multe apeluri de sistem sunt blocante
- kernel-ul planifică thread-urile de care ştie, fiecare pe un singur procesor la un moment dat. În cazul user-level threads, el va vedea un singur thread. Astfel, chiar dacă 2 thread-uri user-level sunt implementate folosind un singur thread "văzut" de kernel, ele nu vor putea folosi eficient resursele sistemului (vor împărţi amândouă un acelaşi procesor).

3.2. Thread-uri din spațiu kernel

Thread-urile din acest spațiu sunt implementate în kernel folsoind tabele multiple(fiecare task primește un tabel de thread-uri). În acest caz, kernelul gestionează/programează execuție fiecărui thread în cuanta de timp al fiecărui proces. Există un mic overhead datorat comutării între spațiile user-»kernel-» și încărcarea contextelor mari, dar modificarea perfromanței în timp este neglijabilă.

Managementul thread-urilor este făcut de kernel, şi programele user-space pot crea/distruge thread-uri printr-un set de apeluri de sistem. Kernel-ul menţine informaţii de context atât pentru procese cât i pentru thread-urile din cadrul proceselor, iar planificarea pentru execuţie se face la nivel de thread.

3.2.1. Avantaje :

- dacă avem mai multe procesoare putem lansa în execuţie simultană mai multe thread-uri ale aceluiai proces; blocarea unui fir nu înseamnă blocarea întregului proces.
- putem scrie cod în kernel care să se bazeze pe thread-uri.

3.2.2. Dezavantaje :

- comutarea de context o face kernelul, deci pentru fiecare schimbare de context se trece din firul de execuţie în kernel și apoi se mai face încă o schimbare din kernel în alt fir de execuţie, deci viteza de comutare este mică.

3.3. Sincronizare

3.3.1. Mutex

Mutexurile sunt obiecte de sincronizare utilizate pentru a asigura accesul exclusiv la o secțiune de cod în care se accesează date partajate între două sau mai multe fire de execuie.
Un mutex are două stări posibile: ocupat și liber. Un mutex poate fi ocupat de un singur fir de execuie la un moment dat. Atunci când un mutex este ocupat de un fir de execuție, el nu mai poate fi ocupat de niciun altul. În acest caz, o cerere de ocupare venită din partea unui alt fir, în general va bloca firul până în momentul în care mutexul devine liber.

3.3.2. Futex

Mutexurile din firele de execuie POSIX sunt implementate cu ajutorul futexurilor, din considerente de performanță. Numele de futex vine de la Fast User-space muTEX. Ideea de la care a plecat implementarea futexurilor a fost aceea de a optimiza operația de ocupare a unui mutex în cazul în care acesta nu este deja ocupat. Dacă mutexul nu este ocupat, el va fi ocupat fără ca procesul care îl ocupă să se blocheze. În acest caz, nefiind necesară blocarea, nu este necesar ca procesul să intre în kernel-mode (pentru a intra într-o stare de ateptare). Optimizarea constă în testarea și setarea atomică a valorii mutexului (printr-o instrucțiune de tip test-and-set-lock) în user-space, eliminându-se trap-ul în kernel în cazul în care nu este necesară blocarea.
Futexul poate fi orice variabilă dintr-o zonă de memorie partajată între mai multe fire de execuție sau procese. Așadar, operațiile efective cu futexurile se fac prin intermediul funcției *do_futex*, disponibilă prin includerea headerului *linux/futex.h*. Signatura ei arată astfel:

long do_futex(unsigned long uaddr, int op, int val, unsigned long timeout, unsigned long uaddr2, int val2);

În cazul în care este necesară blocarea, *do_futex* va face un apel de sistem - *sys_futex*. Futexurile pot fi utile (și poate fi necesară utilizarea lor explicită) în cazul sincronizării proceselor, alocate în variabile din zone de memorie partajată între procesele respective.

3.3.3. Semafor

Semafoarele sunt obiecte de sincronizare ce reprezintă o generalizare a mutexurilor prin aceea că salvează numărul de operaii de eliberare (incrementare) efectuate asupra lor.
Practic, un semafor reprezintă un întreg care se incrementează/decrementează atomic. Valoarea unui semafor nu poate scădea sub 0. Dacă semaforul are valoarea 0, operaia de decrementare se va bloca până când valoarea semaforului devine strict pozitivă. Mutexurile pot fi privite, aadar, ca nite semafoare binare.

3.3.4. Variabilă de condiție

Variabilele condiție pun la dispoziție un sistem de notificare pentru fire de execuție, permițându-i unui fir să se blocheze în ateptarea unui semnal din partea unui alt fir.
Folosirea corectă a variabilelor condiție presupune un protocol cooperativ între firele de execuție. Mutexurile (mutual exclusion locks) și semafoarele permit blocarea altor fire de execuție. Variabilele de condiție se folosesc pentru a bloca firul curent de execuie până la îndeplinirea unei condiii. Variabilele condiie sunt obiecte de sincronizare care-i permit unui fir de execuție să-i suspende execuția până când o condiție (predicat logic) devine adevărată. Când un fir de execuție determină că predicatul a devenit adevărat, va semnala variabila de condiție, deblocând astfel unul sau toate firele de execuție blocate la acea variabilă de condiție (în funcițe de cum se dorește). O variabilă de condiție trebuie întotdeauna folosită împreună cu un mutex pentru evitarea race-ului care se produce când un fir se pregătește să aștepte la variabila de condiție în urma evaluării predicatului logic, iar alt fir semnalizează variabila de condiție chiar înainte ca primul fir să se blocheze, pierzându-se astfel semnalul.
Așadar, operațiile de semnalizare, testare a condiției logice și blocare la variabila de condiție trebuie efectuate având ocupat mutexul asociat variabilei de condiție. Condiția logică este testată sub protecția mutexului, iar dacă nu este îndeplinită, firul apelant se blochează la variabila de condiție, eliberând atomic mutexul. În momentul deblocării, un fir de execuție va încerca să ocupe mutexul asociat variabilei de condiție. De asemenea, testarea predicatului logic trebuie făcută într-o buclă, pentru că dacă sunt eliberate mai multe fire deodată, doar unul va reuși să ocupe mutexul asociat condiției. Restul vor atepta ca acesta să-l elibereze, însă este posibil ca firul care a ocupat mutexul să schimbe valoarea predicatului logic pe durata deținerii mutexului. Din acest motiv celelalte fire trebuie să testeze din nou predicatul pentru că altfel și-ar începe execuția presupunând predicatul adevărat, când el este, de fapt, fals.

Dezavantaje

Lucrul cu firele de execuție poate fi avantajos și ușor de înteles dar în majoritatea cazurilor aduce un set neplăcut de dezavantaje precum:7

- Complexitate crescută
- Sincronizarea resurselor partajate (obiecte, date)
- Dificultate în depanare iar unele rezultate pot fi impredictibile
- Posibile „deadlock”-uri
- Din cauza unui design defectuos poate apărea efectul de „înfometare”
- Creearea și sincronizarea firelor de execuție este intensivă din punct de vedere CPU/memorie lucru care reprezintă în multe cazuri consum crescut de energie – factor foarte important în cazul dispozitivelor portabile

Crearea unui nou thread este eficientă din punct de vedere al costului și overhead-ului implicat. În particular, în Linux se creează foarte rapid chiar și procese și diferența nu este foarte mare.

De multe ori, este de dorit ca un program să facă (sau cel putin să pară că face) două lucruri în același timp. Un caz este editarea text-ului unui document cu menținerea simultană a informațiilor despre acesta (număr de cuvinte). Un thread se poate ocupa de intrarea utilizatorului și de editare. Altul se ocupă de update-ul numărului de cuvinte. Un exemplu mai realist este utilizarea unui sistem de bază de date multithreaded. Aici un proces server deservește mai mulți clienți, și mărește productivitatea prin deserverirea unor cereri în timp ce altele sunt blocate.

Thread-urile au și dezavantaje. Deoarece ele partajează o mare parte din resursele unui proces, scrierea de programe multithreaded solicită un efort de planificare suplimentar. Pot apărea inconsistențe în situația în care nu am realizat sincronizarea anumitor variabile. Depanarea programelor multithreaded este, de asemenea, mai dificilă.

Un program care folosește thread-uri pentru rezolvarea unei probleme de calcul intens nu va rula mai rapid pe un sistem uniprocesor.

Totuși, scrierea unei aplicatii care realizează o mixare a intrării, calculului și ieșirii poate fi imbunătățită prin rularea unor thread-uri separate. În timp ce un thread este blocat in așteptarea unor informații, altul poate să își continue execuția.

Schimbarea de context în cazul thread-urilor este mai puțin costisitoare. Thread-urile cer mai puține resurse și devine o solutie practică rularea de programe multithreaded pe sisteme uniprocesor. Totuși pe unele sisteme, cum este Linux, această soluție nu este atât de eficientă pe cât pare. Linux-ul tratează procesele și thread-urile în mod identic din punct de vedere al planificării și execuției.

4. Procese

Procesele efectuează sarcini în cadrul sistemului de operare. Un program conține un set de instrucțiuni de cod mașină și datele stocate într-o imagine executabilă pe disc și este, ca atare, o entitate pasivă; un proces poate fi gândit ca un program de calculator în acțiune.

Un proces este o entitate dinamică, în continuă schimbare datorită instrucțiunilor de tip cod mașină care sunt executate de către procesor. Împreună cu instrucțiunile și datele programului, procesul include, de asemenea, contorul de program și toate registrele procesorului precum și stack-urile de proces care conțin date temporare, cum ar fi parametrii de rutină, adrese de revenire și variabile salvate. Actualul program de executat, sau proces, include toată activitatea curentă în microprocesor.

Linux este un sistem de operare multiprocesor. Procesele sunt sarcini separate, fiecare cu propriile lor drepturi și responsabilități. În cazul în care un singur proces se blochează, nu va provoca un alt proces în sistem să se prăbușească. Fiecare proces individual se execută în propriul său spațiu de adrese virtuale și nu este capabilă să interacționeze cu un alt proces decât prin mecanisme sigure gestionate de către kernel.

Pe parcursul duratei de viață un proces va folosi mai multe resurse de sistem. Acesta va folounitatea de calcul din sistem pentru a rula instrucțiunile sale și memoria fizică a sistemului pentru a-și stoca datele. Va deschide și utiliza fișiere în sistemele de fișiere și poate utiliza în mod direct sau indirect, dispozitivele fizice în sistem. Linux trebuie să țină evidența procesului în sine și a resurselor de sistem pe care le are, astfel încât să poată arbitra corect și celelalte procese din sistem. Nu ar fi corect față de celelalte procese din sistem în cazul în care un singur proces ar monopoliza cele mai multe resurse de memorie fizică a sistemului sau CPU-ului.

Resursa cea mai prețioasă în sistem este CPU, de obicei, există doar unul. Linux este un sistem de operare multiprocesor, obiectivul este de a avea un proces care rulează pe fiecare CPU în sistem, în orice moment, pentru a maximiza utilizarea procesorului. În cazul în care există mai multe procese decât procesoare(și deobicei acesta este cazul), restul proceselor trebuie să aștepte ca CPU-ul să fie liber înainte pentru a putea fi rulate. Multiprocessing este o idee simplă; un proces este executat până când trebuie să aștepte, de obicei, pentru unele resurse de sistem; când are această resursă, procesul poate rula din nou. Într-un sistem uniprocessing, de exemplu DOS, procesorul ar sta pur și simplu inactiv, iar timpul de așteptare s-ar fi irosit. Într-un sistem de multiprocesare multe procese sunt păstrate în memorie în același timp. De fiecare dată când un proces trebuie să aștepte, sistemul de operare preia CPU de la procesul în cauză și îl dă la alt proces care merită mai mult. Scheduler-ul este cel care alege care este procesul cel mai potrivit pentru a rula următorul și Linux utilizează o serie de strategii de planificare pentru a asigura corectitudinea.

Linux suportă un număr de diferite formate de fișiere executabile, ELF este una, Java este o alta, iar acestea trebuie să fie gestionate în mod transparent pentru ca procesele să poată utiliza bliotecile partajate ale sistemului.

Dezavantaje:

Switch-urile de context în cazul proceselor sunt de obicei computațional intensive și o mare parte din proiectarea sistemelor de operare este de a optimiza utilizarea de switch-uri de context. Trecerea de la un proces la altul necesită o anumită perioadă de timp pentru a face administrarea - salvarea și încărcarea registrelor și hărților de memorie, actualizarea diverselor tabele și liste, etc. De exemplu, în comutarea contextului kernel-ul Linux-ului implică comutarea de registre, stivă, pointeri și contorul de program, dar este independent de comutare spațiu de adrese, deși în cadrul comutării spațiul de adrese se schimbă.

5. Pipeline

Pipeline este o tehnică de creștere a vitezei de execuție totale a procesoarelor, fără a ridica tactul. Ea constă în subdivizarea fiecărei instrucțiuni într-un număr de etape sau segmente, fiecare etapă fiind executată de câte o unitate funcțională separată a procesorului (segment *pipeline*). Segmentele *pipeline* sunt conectate între ele într-un mod analog asamblării unei conducte din segmente de țeavă. Segmentele tipice de execuție ale unei instrucțiuni mașină pe un procesor sunt:

- FI - aducere instrucțiune (engleză: *fetch instruction*). Se referă la un ciclu special în care procesorul citește din memorie instrucțiunea următoare ce trebuie executată.
- DI - decodare instrucțiune (*decode instruction*). În această etapă instrucțiunea este recunoscută și procesorul colectează toate informațiile necesare pentru executarea ei.
- CO - calculare adresă operand (*calculate operand address*). Se calculează adresa de memorie de unde se aduce primul operand.
- FO - aducere operand (*fetch operand*). Citește operandul aflat în memorie la adresa calculată în pasul anterior.
- Dacă e nevoie, cele 2 etape anterioare se repetă și pentru al doilea operand etc.
- EI - execută instrucțiune (*execute instruction*). Execuția propriu-zisă a instrucțiunii inclusiv calculul rezultatului ei.
- WO - scriere operand (*write operand*). Scrierea rezultatului înapoi în memorie.

Principiul se bazează pe faptul că fiecare din aceste operații lucrează în principiu cu alte resurse, deci, cu ajutorul tehnicii potrivite, 2 până la 6 operații se pot executa și în paralel. La un moment dat ele se află în execuție în etape diferite; în cazul cel mai fericit la un moment dat există câte o instrucțiune tratată în fiecare etapă a *pipeline*-ului. Deși execuția unei instrucțiuni de la început până la sfârșit necesită de obicei mai multe etape, instrucțiunea următoare poate începe să fie executată imediat ce instrucțiunea anterioară a trecut de prima etapă (nu e nevoie să se aștepte până la terminarea completă a instrucțiunii anterioare).

5.1. Dezavantaje:

Apar diferite tipuri de hazarde:

5.1.1. Hazarde structurale

Apar când o anumită resursă (memorie, unitate funcțională) este cerută de mai multe instrucțiuni în același moment. Anumite resurse sunt duplicate tocmai pentru a fi evitate hazardele structurale. Unitățile funcționale pot fi și ele de tip *pipeline* pentru a suporta mai multe instrucțiuni în același timp. O metodă clasică de a evita hazardele ce implică accesul la memorie este aceea de a avea câte o memorie de tip *cache* separat pentru instrucțiuni și pentru date. În imaginea 1 (hazard structural) se observă că instrucțiunea ADD R4, X aduce în stagiul FO operandul X din memorie. În timpul acestui ciclu memoria nu acceptă altă accesare.

5.1.2. Hazarde de date

Un scenariu posibil pentru un hazard de date este următorul: trebuie executate 2 instrucțiuni, I1 și I2. Într-un *pipeline* execuția lui I2 poate începe înainte ca I1 să se termine. Dacă într-un anumit stagiu al *pipeline*-ului I2 are nevoie de rezultatul produs de I1, dar acest rezultat nu a fost încă generat, se produce un hazard de date. în imaginea 2 (hazard de date), înaintea executării stagiului FO, instrucțiunea ADD este blocată, pînă cînd instrucțiunea MUL a scris rezultatul în R2.

5.1.3. Hazarde de control

Sunt produse de instrucțiuni de salt. Atunci când se execută un salt, *pipeline*-ul trebuie golit și umplut cu instrucțiunea următoare de la adresa de memorie la care se

face saltul. În imaginea 3 (hazard de control ; salt necondiționat), după stagiul FO al instrucțiunii de salt adresa destinației este cunoscută și instrucțiunea următoare poate fi "adusă" (*fetch*). Înainte ca stagiul DI să se termine nu se știe dacă s-a executat saltul. Mai târziu, instrucțiunea adusă este abandonată. În imaginile 4 și 5 sunt prezentate situații de hazard de control cu instrucțiuni de tip salt condiționat, efectuat sau neefectuat.

6. Întreruperi

Sistemul de intreruperi este acea parte a unui sistem de calcul care permite detectia unor evenimente externe sau interne si declansarea unor actiuni pentru tratarea lor. Astfel de evenimente pot fi: receptia unui caracter pe un canal serial, golirea unui registru de transmisie, impuls generat de un contor de timp, tentativa de executie a unui cod de instructiune nepermis (inexistent sau protejat), terminarea unei anumite operatii de catre o interfata, eroare in timpul executiei unei operatii aritmetice (impartire ci zero) si multe altele. Intreruperile permit calculatorului sa reactioneze rapid la aceste evenimente, sa se sincronizeze cu ele si sa le trateze in timp util.

O alternativa la sistemul de intreruperi ar fi testarea periodica prin program (prin polling) a tuturor indicatorilor de stare si a semnalelor de intrare. Aceasta solutie este ineficienta in cazul in care numarul de elemente care trebuie testate este mare.

Pentru un sistem de calcul, politica de solutionare a intreruperilor caracterizeaza adaptabilitatea sistemului la stimuli interni si externi. Majoritatea sistemelor de intrerupere utilizeaza un sistem de prioritati pentru a stabili ordinea de deservire a cererilor concurente de intrerupere. Prioritatea se stabileste pe baza importantei acordate evenimentului tratat si a restrictiilor de timp in solutionarea intreruperii. Politica de prioritati trebuie sa asigure solutionarea echitabila si in timp util a tuturor cererilor.

Un calculator poate să identifice mai multe tipuri de intrerupere (numite nivele de intrerupere). Pentru fiecare nivel se poate defini cite o rutina de tratare a intreruperii respective. Adresele de inceput ale acestor rutine se pastreaza intr-o tabela de pointeri denumita tabela de intreruperi. Aceste rutine sunt activate la aparitia si acceptarea de catre calculator a intreruperii corespunzator. Functie de cerintele aplicatiei executate, anumite nivele de intrerupere pot fi invalidate, temporar sau pe toata durata aplicatiei. O intrerupere invalidata (sau mascata) nu este recunoscuta de catre calculator.

La activarea unui semnal de intrerupere se testeaza daca nivelul corespunzator este validat si daca nu sunt in curs de deservire alte intreruperi mai prioritare; in caz afirmativ are loc intreruperea temporara a secventei curente de executie, se salveaza pe stiva adresa instructiunii urmatoare si se face salt la rutina de tratare a intreruperii. Dupa executia rutinei de intrerupere se revine la secventa intrerupta prin incarcarea adresei salvate pe stiva.

Adesea, pentru controlul intreruperilor externe se utilizeaza un circuit specializat denumit controlor de intreruperi. Un astfel de circuit deserveste un set de semnale de intrerupere. Functiile tipice ale unui astfel de controlor sunt: detectia conditiei

de producere a unei intreruperi (ex: front crescator al semnalului de intrerupere), arbitrarea cererilor multiple, mascarea unor intreruperi, evidenta intreruperilor deservite, etc.

In cadrul setului de instructiuni al unui procesor pot sa existe instructiuni dedicate pentru tratarea intreruperilor (ex: validarea/invalidarea intreruperilor, revenirea din rutina de intrerupere, simularea software a unor intreruperi hardware, etc.).

Un sistem de intreruperi poate sa aiba mai multe moduri de functionare; aceste moduri difera prin: modul de alocare a prioritatilor (fixa, rotativa), modul de detectie a semnalului de intrerupere (pe front sau pe nivel), acceptarea sau nu a intreruperilor imbricate (intrerupere 3-2 imbricata = intrerupere accesptata in timpul executiei unei alte rutine de intrerupere), numarul de nivele de imbricare, etc.

Intreruperile sunt o solutie de implementare a unor activitati concurente si joaca un rol important in realizarea sistemele de operare multitasking si de timp-real.

Întreruperile microprocesorului 8086 se mai pot clasifica în trei grupe: întreruperi predefinite generate de funcţii speciale, întreruperi hardware definite de utilizator şi întreruperi software definite de utilizator. Vom descrie pe scurt aceste tipuri de întreruperi.

6.1. Hardware

Aşa cum am mai spus, întreruperile hardware definite de utilizator sunt iniţiate de circuite speciale prin activarea, trecerea pe "1", a semnalului de la intrarea INTR a procesorului. Aceste întreruperi sunt mascabile cu ajutorul bitului I din registrul de stare. Starea intrării INTR este testată în timpul ultimului ciclu de ceas al fiecărei instrucţiuni. Excepţie de la această regulă fac instrucţiunile MOV şi POP cu un registru de segment, instrucţiunea WAIT, instrucţiunile pe şiruri precedate de prefixul de repetare REP.

În cazul instrucţiunilor MOV şi POP cu un registru de segment microprocesorul va testa starea conexiunii INTR după executarea instrucţiunii următoare instrucţiunilor MOV şi POP menţionate. Astfel, se permite încărcarea unui pointer de stivă de 32 de biţi în registrele SS şi SP fără pericolul apariţiei unei întreruperi între cele două încărcări. O secvenţă cum este şi cea de mai jos nu va fi deci interuptibilă:

```
mov ss, segment_stiva_nou
mov sp, pointer_stiva_nou
```

În timpul instrucţiunii WAIT care aşteaptă trecerea pe "0" a intrării TEST a microprocesorului se testează şi starea semnalului la conexiunea INTR pentru a se permite executarea rutinelor de întrerupere în timpul aşteptării. Particularitea în această situaţie constă în faptul că atunci când se dectectează o întrerupere, 8086 mai extrage încă o dată instrucţiunea WAIT înainte de a transfera controlul rutinei de serviciu. Aceasta pentru a garanta revenirea din rutină tot la instrucţiunea de aşteptare WAIT.

6.2. Predefinite

La invocarea din hardware sau din software a unei întreruperi predefinite microprocesorul va transfera controlul rutinei a cărei adresă este specificată de vectorul asociat tipului de întrerupere. Utilizatorul are sarcina să scrie rutinele de serviciu şi să iniţializeze tabela vectorilor cu adresele corespunzătoare.

Tipul 0 – împărţire la 0. Această întrerupere este invocată la orice încercare de împărţire pentru care câtul depăşeşte valoarea maximă, cum este, de exemplu, cazul împărţirii la zero. Întreruperea nu este mascabilă şi poate fi considerată ca o secvenţă aparţinând operaţiei de împărţire.

Tipul 1 – pas-cu-pas. Întreruperea apare după o instrucţiune de la poziţionarea indicatorului de condiţie Derută, T, în registrul de stare al microprocesorului. Se permite astfel introducerea software a funcţionării pas-cu-pas în cadrul unei secvenţe de program. Secvenţa de întrerupere începe cu salvarea registrului de stare şi a numărătorului de program şi ştergerea indicatorului T permiţându-se astfel execuţia normală, nu pas-cu-pas, a rutinei de serviciu specifice. La revenire în programul principal, modul de execuţie pas-cu-pas, este asigurat prin restaurarea IP, CS şi a indicatorilor de condiţii, deci şi a lui T. Nici această întrerupere nu poate fi mascată.

Tipul 2 – întrerupere nemascabilă, NMI. Întreruperea este de tip hardware şi are prioritatea cea mai înaltă. Intrarea corespunzătoare a microprocesorului, conexiunea NMI, este comutată pe front şi apoi sincronizată în interiorul procesorului cu ceasul CLK. Pentru ca o întrerupere NMI să fie recunoscută, semnalul intern sincronizat trebuie să fie activ cel puţin două perioade ale ceasului. De asemenea, dacă intrarea NMI rămâne pe "1" mai multă vreme, revenirea pe "0", timpul cât stă pe "0", înainte ca o nouă întrerupere să fie declanşată este de minimum două perioade CLK. Semnalul de la intrarea lui 8086 poate fi dezactivat înainte de intrarea în rutina de serviciu. O atenţie deosebită trebuie acordată eliminarii spikeurilor, impulsurilor parazite, care pot genera întreruperi. Întreruperea NMI este rezervată, de obicei, evenimentelor catastrofale cum sunt căderile de tensiune sau semnalizările de la un sistem de supraveghere de tip ceas de gardă, watchdog.

Tipul 3 – întrerupere pe un octet. Este o întrerupere software nemascabilă invocată de o instrucţiune reprezentată pe un octet, INT 3, destinată în primul rând ca întrerupere de tip breakpoint pe parcursul punerii la punct a programelor.

Tipul 4 – întrerupere la depăşire. Este o întrerupere software nemascabilă care apare la execuţia instrucţiunii INTO dacă indicatorul de condiţie Depăşire, O, este poziţionat. Instrucţiunea permite derutarea programului la o rutină de serviciu în cazul apariţiei unei erori de depăşire. Întreruperile de tip 0 sau 2 pot apărea fără ca programatorul să acţioneze în vreun fel specific, cu excepţia unei împărţiri la 0 care ar putea fi o greşeală de programare.

Întreruperile de tip 1, 3 şi 4 necesită pentru a fi generate acte conştiente din partea programatorului. Toate tipurile de întreruperi prezentate mai sus, cu excepţia NMI, sunt invocate software şi sunt asociate direct cu câte o instrucţiune specifică.

6.3. Software

Întreruperile software sunt generate de instrucţiunea INT nn unde nn reprezintă numărul tipului de întrerupere definit de utilizator. Instrucţiunile INT, deci întreruperile software, nu sunt mascabile cu ajutorul indicatorului de condiţii Validare/Invalidare Întrerupere, I. Revenirea din rutina de serviciu apelată printr-o întrerupere software se face cu instrucţiunea IRET. Transferul controlului la o întrerupere software se face la sfârşitul instrucţiunii INT nn fără ca microprocesorul să iniţieze pe magistrală un ciclu de achitare INTA. De asemenea, întreruperile software vor invalida întreruperile mascabile prin punerea pe "0" a indicatorilor I şi T.

Avantaje

În mare parte o întrerupere împreună cu rutina de tratare seamănă ca și detaliu de implementare cu un context switch al thread-urilor respectiv proceselor, diferența ar fii că în al doilea caz acest switch se gestionează de către kernel și implică diferiți algoritmi de restabilire etc, care costa timp si putere de procesare + defragmentarea resurselor.

În cazul întreruperilor switch-ul se face la nivel mult mai rapid direct in hardware și în general acesta dispune de aceleași resurse

Dezavantaje

Cel mai mare dezavantaj al întreruperilor este programatorul + designul taskului întrucât există șansa ca din cauza priorităților întreruperilor programul principal să numai fie rulat niciodată.

7. Procese asincrone ale sistemului de operare

7.1. I/O – epolling

Epoll este un apel de sistem al kernelului de Linux pentru a oferii un mecanism scalabil de notificări de tip I/O. A fost introdus prima oară în versiunea 2.5.44 de „mainline-ul” celor de la Linux kernel. Funcția lui este de a monitoriza descriptori multipli de fișiere pentru a vedea dacă operațiile de tip I/O sunt posibile pe oricare din ei. A fost introdus pentru a înlocuii mai vechiul **select** și **poll** din apelurile de sisteme ale POSIX și pentru a obține o performanță mai bună atunci când numărul de descriptori urmăriți este foarte marte (spre deosebire de predecesorul său care opera într-o complexitate de O(n) epoll operează în O(1)^2).

Epoll este foarte similar cu **kqueue** al celor de la FreeBSD la modul că operează asupra unui obiect de tip kernel configurabil, expus în spațiu utilizator ca un descriptor de fișier.

Un alt lucru important la epoll este faptul că se primesc „sugestii” când kernell crede că un descriptor de fișier a devenit disponibil pentru I/O. Astfel avantajul primordial este acela că spațiu kernel nu trebuie să țină cont de starea descriptorului de fișier.

Acest mecanism este folosit pentru comunicarea cu perifericele sistemului respectiv placa de retea.

7.2. Event loop

În informatică, event loop, dispecer de mesaje, bucla de mesaje, pompa de mesaj sau buclă de rulare este o metodologie, care așteaptă si expediaza evenimente sau mesaje într-un program. Functioneaza prin trimiterea unei solicitări la "furnizor de evenimente", intern sau extern (care, în general, blochează cererea până când un eveniment a sosit) iar apoi solicită tratarea evenimentului în cauză. Event loop-ul poate fi utilizat împreună cu un „rezolvator”, în cazul în care furnizorul de eveniment urmează „interfața fișier”, care poate fi selectat sau interogat(pollat). Event loop-ul comunica cu initiatorul mesajului aproape întotdeauna in mod asincron.

Când bucla de evenimente formează fluxul de control central al unui program, așa cum se întâmplă adesea, acesta poate fi numit bucla principală sau bucla de evenimente principala. Acest titlu este adecvat, deoarece o astfel de buclă de eveniment este la cel mai înalt nivel de control în cadrul programului.

Tratarea semanlelor

Unul dintre puținele lucruri din Unix, care nu sunt conforme cu interfața de fișiere sunt evenimente asincrone (semnale). Semnalele sunt recepționate în capcana de semnal (signal handler), bucăți mici limitate de cod care se execută în timp ce restul de sarcina este suspendată; în cazul în care un semnal este recepționat și manipulat în timp ce sarcina se blochează în select (), select-ul va reveni mai devreme cu interuperea EINTR; în cazul în care un semnal este recepționat în timp ce sarcina este de tip CPU, sarcina va fi suspendat între instrucțiuni până când revine handler-ul de semnal.

Solutia propusa de POSIX este apelul rutinei pselect() similar cu select() dar care primeste un parametru aditional care descrie masca semnalului(sigmask). Acest lucru confera aplicatiei posibilitatea de a masca semnale in cadrul task-ului principal apoi sa scoata masca pe durata instructiunii select() astfel incat rutinele de tratare a semanlului sa fie invocate doar cand aplicatia este implicata in operatii de tip I/O. Aceasta solutie in schimb se aplica de la versiuni de Linux mai mari de 2.6.16 intrucat pselect() a devenit recent stabil.

O solutie alternativa mai portabila este aceea de a convertii evenimentele asincrone la evenimente de tip „fisier” folosit tucuri de tip „self-piped” (o rutina de tratare a semnalul scrie un byte pe un pipe monitorizat pe cealalta parte de select() in cadrul programului principal). In versiunea 2.6.22 din Linux un alt apel de sistem signalfd() a fost adaugat care ne ajuta sa primim semnale printr-un descriptor de fisier special.

7.3. AIO

Aceasta reprezinta o metoda de a face operatii de tip I/O introdusa in versiuni mai recente de Linux prin care un proces care face o cerere de tip I/O nu este blocat

pana cand datele sunt disponibile. Astfel un proces care trimite o cerere de tip I/O poate continua sa isi execute codul si apoi sa verifice status-ul cererii sale.

Kernelul de Linux ofera in acest moment doar 5 apeluri de sistem executa procesari I/O asincrone. Alte functii de tip AIO sunt deobicei implementate in librarii din spatiu utilizator si folosesc intern apelurile de sistem. Exista cateva librarii care pot emula functionalitatea AIO-ului in totalitate in spatiu utilizator fara a avea nevoie de suportul kernel-ului.

Metodele disponibile in kernel sunt:

- io_setup
- io_destroy
- io_submit
- io_cancel
- io_getevents

In versiunile anterioare de UNIX un apel de tip citire era considera ca fiind logic sincron pentru ca apelul nu putea fi intors la utilizator pana cand cererea de rezolvare a datelor nu era citit in spatiul de adrese ale procesului adresator. Deci un proces din user space era blocat pana cand datele erau disponibile.

Era de asemeni si o problema de performanta in cadrul copierii din spatiu kernel in spatiu utilizator a datelor. Astfel cu AIO o aplicatie poate trimite o cerere de tip I/O si poate executa alte taskuri orientate pe unitatea de calcul fie pana cand este notificat in mod sincron de finalizarea operatiei I/O (pe baza de functii de tip callback) sau pana cand doreste sa interogheze asupra starii operatiei.

Acest lucru este un avantaj pentru aplicatii care au nevoie de multe date I/O.

8. Concluzii

Nu exista un mod „perfect” de a executa multitasking-ul si nici just din punct de vedere al „colaborarii”. Exista o multime de mecanisme puse la dispozitie de catre kernel-ul de linux pentru a gestiona task-urile colaborative.

Sistemul de operare Linux incearca imbinarea acestor mecanisme software de sincronizare (unele simple altele foarte avansate) cu mecanisme hardware de sincronizare pentru a gestiona problemele de lifecycle ale unitatilor fizice(durata de viata a memoriilor) cat si pentru a rezolva probleme de consum de curent si memorie pe diverse dispozitive.

9. Studiu de caz

Într-o lume dominată de sisteme de operare în timp real cu matrici enorme de functionalitate si de sisteme de operare la scară largă precum Linux, poate ar fi util să luam în considerare elementele de bază. Ce se întâmplă dacă ai nevoie doar de un multi-threading de baza? Cât de complex trebuie acest lucru să fie?

De ce multitasking?

Aproape toate sistemele embeded ruleaza software specific de indata ce sunt alimentate si continua sa faca asta pana cand sunt oprite. Vorbind intr-un sens mai extins, codul poate avea 3 forme arhitecturale:

1) O singura bucla infinita. Codul efectueaza pur si simplu o secventa de actiuni apoi se intoarce si face totul din nou pe termen nelimitat. Aceasta structura are avantajul simplitatii dar are si dezavantaje. Dezavantajele sunt foarte mari: codul nu este scalabil, adaugarea de cod aditional are sanse sa afecteze (foarte probabil negativ) functionalitatea curenta; daca o parte din cod nu functioneaza cum trebuie(ramane blocat asteptand un eveniment extern) intreaga aplicatie este afectata.
2) O bucla care este augmentata de catre rutine de intreruperi (ISR-uri – interrupt service routines). Bucla infinita principala se ocupa doar de operatiile principale ale dispozitivului, iar ISR-urile se ocupa de evenimente externe. Rutinele de intrerupere ar trebuii sa fie scurte, simple si doar sa ofere date pentru procesare in bucla principala cu minim de intruziune. Arhitectura este mult mai complexa dar cumva poate fi mai flexibila si scalabila cu anumite limitari.
3) Un numar de bucle independente augmentate de ISR-uri. Multe aplicatii abordeaza acest tip de arhitectura care poate fi implementat cu un anumit tip de kernel multitatsking. Implementarea este mai complexa decat a celorlalte 2 arhitecturi de mai sus dar ofera multa flexibilitate si scalabilitate.

Fiecare dintre aceste arhitecturi ar trebuii sa fie luate in considerare pe rand in functie de cerintele implementarii curente dar si de viitoare imbunatatiri si extensii ale aplicatiei. Nu este tot timpul o solutie optima sa algem arhitectura cu numarul 3 cand una dintre celelalte 2 arhitecturi poate fi satisfacatoare.

Totusi este comun din partea dezvoltatorilor sa concluzioneze ca un model multitasking este cea mai buna optiune. Fiind acesta cazul, detaliile de implementare trebuie adresate si in mod normal aceasta implica si instalarea acestui kernel de tip multitasking. Majoritatea unitatilor de calcul de tip embeded sunt capabile sa suporte un singur thread pentru executia codului la un anumit moment de timp. Scopul unui kernel de tip multitasking este acela de a ajuta un programator sa programeze ca si cum unitatea de calcul poate executa taskuri multiple simultan.

Metode de gestionare a multitaskingului.
Pentru a efectua magia de a face un procesor sa para sa ruleze mai multe fire de executie un kernel trebuie sa faca swaping continu intre executia tuturor taskurilor astfel dand impresia de simultaneitate. In realitate taskurile impart acelasi ceas. Managementul procesului de partajare se numeste „scheduling”.

Cea mai simpla metoda de „scheduling” folsesti „run to completion”. Odata ce un task este pornit el ruleaza pana cand se termina, apoi controlul este predat kernelului. Este o cerinta ca fiecare task sa coopereze si sa nu acapareze timpul CPU-ului.