Sincronizare

SO: Curs 09

Cuprins

- Recapitulare thread-uri
- Nevoia de sincronizare
- Sincronizarea la procese și thread-uri
- Moduri de sincronizare
- Implementarea sincronizării
- Problematica sincronizării

Suport de curs

- Operating Systems Concepts
  - Capitolul 6 - Process Synchronization
- Modern Operating Systems
  - Capitolul 2 - Processes and Threads
    - Secțiunea 2.3 - Interprocess Communication
- Allen B. Downey - The Little Book of Semaphores

RECAPITULARE THREAD-URI

Ce este un thread?

- Instanță de execuție (planificabilă)
- Instruction Pointer + Stack Pointer + stare (registre)
- Partajează cu alte thread-uri resursele procesului: fișiere, memorie
- Permite paralelism
- Necesită sincronizare (partajare date)
- User-level threads / kernel-level threads

De ce thread-uri?

- Lightweight: timp de creare scurt, timp de comutare scurt
  - Nu se face schimbare de tabelă de pagini (+TLB flush) la context switch
- Comunicare rapidă (memorie partajată)
- Adecvate pentru calcule multiprocesor cu date comune
- Thread pool pentru prelucrarea de task-uri
  - Mai ușor de programat față de un model asincron

De ce nu thread-uri?

- O problemă a unui thread afectează întregul proces
- Nevoie de sincronizare
  - Date incoerente, comportament incoerent
  - Probleme ale sincronizării
  - Reentranță
- Mai puțin relevante pentru probleme seriale

NEVOIA DE SINCRONIZARE

Nevoia de sincronizare

```
unsigned long sum = 0;

void thread_func(size_t i)
{
    sum += i * i * i;
    print("sum3(%zu): %lu\n", i, sum);
}

int main(void)
{
    size_t i;
    for (i = 0; i < NUM_THREADS; i++)
        create_thread(thread_func, i);
    [...]
}
```

Ce probleme sunt în codul de mai sus?
Cum le-am rezolva?

Nevoia de sincronizare (2)

- Acces exclusiv / serializare / atomizare
  - Regiune critică
- Secvențiere / ordonare
  - Read after write, write after write, use after create
- Nedeterminism
  - Poate să meargă adesea deși sunt probleme

Interacțiune în sistemele de calcul

- Între componente hardware
- Între procese/thread-uri
- Între user space și kernel space
- Între aplicații și hardware
- Metode
  - Notificări
  - Transfer de date (e.g. message passing)
  - Date partajate (e.g. shared memory)

Exemple

1. Placa de rețea primește pachete pe care să le transfere procesorul sau DMA-ul în memorie
2. Două procesoare au nevoie de acces la aceleași date din memoria RAM.
3. Un proces solicită sistemului de operare aducerea de date de pe disc.
4. Un proces așteaptă încheierea unei acțiuni a altui proces.
5. Mai multe thread-uri execută operații pe o structură de date comună (partajată).

Probleme

1. Placa de rețea primește mai multe informații decât poate procesorul prelucra; se pierd informații.
2. Ambele procesoare încearcă să modifice aceeași dată. Data este acum incoerentă/coruptă.
3. Bufferul de memorie unde sistemul de operare stochează datele este suprascris cu alte date (cerute de alt proces).
4. Procesul “pierde” notificarea din partea celuilalt proces și așteaptă nedefinit.
5. Thread-urile modifică în mod concurent structurile de date (vectori, liste, matrice, întregi, pointeri) și acum sunt incoerente/corupte.

Producător-consumator cu probleme

producer

```
wait(buffer_not_full);
produce_item();
signal(buffer_not_empty);
```

consumer

```
wait(buffer_not_empty);
consume_item();
signal(buffer_not_full);
```

Producător-consumator corect

producer

```
lock(mutex);
if (is_buffer_full())
   wait(buffer_not_full,mutex);
produce_item();
signal(buffer_not_empty);
unlock(mutex);
```

consumer

```
lock(mutex);
if (is_buffer_empty())
   wait(buffer_not_empty, mutex);
consume_item();
signal(buffer_not_full);
unlock(mutex);
```

Cod pe un server web

```
def add_quest_points(quest_level):
     points = get_current_player_points()
     points += points_for_level(quest_level)
     save_player_points(points)
     increase_player_quest_level()

def add_challenge_points(points):
     ...  /* something similar to above */
```

Scenarii la codul de pe serverul web

- Se apelează simultan funcția add_quest_points și funcția add_challenge_points
  - Jucătorul rezolvă quest-ul și altceva generează apelul pentru challenge
  - Poate există quest-uri pe echipă și simultan doi jucători rezolvă părți din același quest (sau quest-uri diferite)
- Jucătorul apasă (conștient sau nu) de mai multe ori în browser pe butonul “Submit quest answer” (ajung două cereri identice la server). Care va fi noul nivel în quest al jucătorului? Ce punctaj va avea?
- Serverul web poate fi multi-proces sau multi-thread. Sincronizarea se face diferit în acele cazuri.

Sincronizare

- Pot apărea probleme din interacțiune
  - Pierdere notificări, pierdere date
  - Date corupte, date incoerente
- Sincronizarea asigură
  - Comportament determinist
  - Date coerente
- Moduri
  - Secvențiere/ordonare operații (A după B)
  - Acces exclusiv la resurse (doar A sau doar B)

Sincronizarea prin secvențiere

- O acțiune înaintea alte acțiuni
- Exemplu: operația wait/waitpid
- Este nevoie ca un proces/thread să execute o acțiune și să producă un efect care să declanșeze acțiunea unui alt proces
- Exemplu: Thread-ul boss trebuie să producă “work item-ul” și să îl adauge în coadă înainte ca un thread worker să îl preia
- În absența sincronizării, se vor prelua date nevalide și programul va avea comportament arbitrar

Sincronizarea prin acces exclusiv

- Două sau mai multe procese/thread-uri au nevoie la aceleași date
- Exemplu: Mai multe thread-uri lucrează pe o matrice partajată
- Accesul exclusiv permite unuia singur accesul
  - Serializare
  - Regiune critică
- Dacă au nevoie toate să citească nu e nevoie de sincronizare
- Dacă unul scrie e nevoie de sincronizare prin acces exclusiv
  - E posibil ca atunci când unul scrie altul să citească/parcurgă (de exemplu, într-o listă înlănțuită)
- În absența accesului exclusiv, cititorul va folosi structuri în stare incoerentă sau corupte

SINCRONIZAREA LA PROCESE ȘI THREAD-URI

Reminder: Procese și thread-uri

Procese

- Resurse dedicate
- Spațiu de adresă propriu
- Planificabile
- Schimbare de context costisitoare (TLB misses)
- Single-threaded sau multi-threaded

Thread-uri

- Partajează resursele procesului
- Pot accesa (concurent) date comune
- Planificabile (cu suport la nivelul nucleului)
- Schimbări de context rapide
- exit() sau primirea unui semnal încheie întreg procesul

De ce procese și de ce thread-uri?

Procese

- Izolare: un proces “stricat” nu “strică” pe altul
- Programare facilă
- De obicei, nu necesită sincronizare

Thread-uri

- Comunicare rapidă
- Aplicații pentru sisteme multi-core
- Mai “light” decât procesele (timpi de creare, schimbare de context)

Când folosim thread-uri

- Multi-core programming
  - CPU intensive: libx264 (encoding), comprimare, criptare
- În medii ce indică folosirea thread-urilor
  - Java
  - kernel development
  - OpenMP
- Pentru HPC (High Performance Computing)

Când nu folosim thread-uri

- Pe sisteme single core (dar există cazuri de utilizare și acolo)
- Atunci când nu suntem confortabili cu API-ul
- Atunci când avem programe seriale (nu paralelizăm ce nu este nevoie)
- Atunci când ținem foarte mult la robustețea programului

Sincronizarea proceselor

- În general implicită
  - socketi
  - message passing, message queues
- Explicită
  - mai rar primitive de sincronizare “clasice” (mutex-uri, semafoare, file locks)
  - memorie partajată (acces exclusiv)
  - semnale (secvențiere)
  - API de lucru cu procese: wait

Sincronizarea thread-urilor

- În general explicită
- Date partajate, concurență: acces exclusiv
  - operații atomice
  - mutex-uri
  - spinlock-uri
- Operații secvențiale: ordonare operații
  - semafoare
  - variabile condiție
  - bariere

MODURI DE SINCRONIZARE

Date coerente și date corupte

- Datele coerente “au sens”, determină comportament determinist pentru aplicație
- Interacțiunea necorespunzătoare (nesincronizată) produce date incoerente
- Exemplu: read before write
  - Dacă un pointer a fost citit dar neinițializat ...
- Nevoie de sincronizare

Sincronizare implicită

- Primitivele folosite nu au ca obiectiv sincronizarea
  - Sincronizarea este un efect secundar (implicită)
- Transfer de date/mesaje: socketi, pipe-uri, cozi de mesaje, MPI
- Date separate/partiționate
- Avantajos: ușor de folosit de programator

Transferul datelor

- Metode: read/write
  - read: în general blocant, așteaptă date scrise
  - write: poate fi blocant dacă e bufferul plin
- Sincronizare implicită
  - nu există date comune, datele sunt copiate/duplicate
  - secvențiere, ordonare: read-after-write
- E nevoie de sincronizare explicită pentru mai mulți transmițători sau receptori
- Avantaje: sincronizare implicită, ușor de programat, sisteme distribuite
- Dezavantaje: overhead de comunicare, blocarea operațiilor

Partiționarea datelor

- Se împart datele de prelucrat
- Fiecare entitate lucrează pe date proprii
- În mod ideal nu există comunicare între entități
  - realist, e nevoie de comunicare (cel puțin agregarea datelor finale)
- Avantaje: reducerea zonelor de sincronizare, programare facilă
- Dezavantaje: date duplicate, posibil overhead de partiționare și agregare, aplicabilitate redusă

Sincronizare explicită

- Referită doar ca “sincronizare”
- Metode/mecanisme specifice de sincronizare
- Două forme de sincronizare
  - pentru date partajate: primitive de acces exclusiv
  - pentru operații secvențiale: primitive de secvențiere, ordonare
- Folosită atunci când este nevoie
- Menține coerența datelor

Acces exclusiv

- În cazul datelor partajate
- Thread-urile/procesele concurează la accesul la date
  - thread-urile modifică simultan date -> date incoerente
  - race condition (condiție de cursă)
- Acces exclusiv
  - delimitarea unei zone în care un singur thread are acces
  - regiune critică (critical section)
  - e vorba de date, nu cod
- Metode: lock și unlock, acquire și release
- Primitive: operații atomice, mutex-uri, spinlock-uri
- Avantaje: date coerente
- Dezavantaje: overhead de așteptare, cod serializat

Atomicitate

- Accesul la date să fie realizat de un singur thread
- Accesul exclusiv se referă la metodă
- Atomicitatea este o condiție a zonei/datei protejate
- Datele/zonele neatomice partajate sunt susceptibile la condiții de cursă
- Atomicitate: variabile atomice, primitive de acces exclusiv

Secvențierea operațiilor

- În cazul în care operațiile trebuie ordonate
- Un thread scrie, un alt thread citește
  - e nevoie ca un thread să aștepte după altul
- Un thread produce, alt thread consumă
- Secvențiere
  - un thread B se blochează în așteptarea unui eveniment produs de un thread A
  - thread-ul B execută acțiunea după thread-ul A
- Metode: signal/notify și wait
- Primitive: semafoare, variabile condiție, bariere
- Avantaje: comportament determinist
- Dezavantaje: overhead de așteptare

Folosire acces exclusiv

- Oricând avem date partajate
- Garantăm că un singur thread poate avea acces la acele date
- Codul este serializat (probleme de paralelism)
- Între lock și unlock definim o regiune critică (serială)
- Pe cât posibil folosim operații atomice (rapide)
- Dacă regiunea critică este mică folosim spinlock-uri
- Dacă regiunea critică este mai mare folosim mutex-uri

Folosire secvențiere

- În general în probleme care se reduc la producător-consumator
  - Un thread “produce” informații/date/structuri/pachete, iar altul “consumă”
- De obicei există un obiect de sincronizare și o variabilă (flag) care indică producerea informației
- Producătorul produce, pune flag-ul pe activ și notifică obiectul de sincronizare
- Consumatorul verifică flag-ul, dacă e inactiv așteaptă la obiectul de sincronizare până când este notificat și apoi consumă informația

IMPLEMENTAREA SINCRONIZĂRII

Cadrul pentru sincronizare

- Mai multe entități active (procese/thread-uri)
- Puncte de interacțiune (date comune, evenimente)
- Preemptivitate
  - O entitate activă poate fi întreruptă oricând de o altă entitate activă
    - oricând = atunci când vine o întrerupere de ceas, se invocă planificatorul care poate decide trecerea entității active curente din RUNNING în READY și planificarea alteia

Implementare simplă

- Un singur proces/thread (doar pentru medii specializate)
- Date complet partiționate (izolare)
- Dezactivarea preemptivității
  - configurare planificator
  - dezactivarea întreruperilor (întreruperii de ceas)
  - doar pentru sisteme uniprocesor
- Folosirea de operații atomice

Implementare mutex

- Un boolean pentru starea internă (locked/unlocked)
- O coadă cu thread-urile care așteaptă eliberarea mutexului

```
struct mutex {
    bool state;
    queue_t *queue;
};
```

```
void lock(struct mutex *m)
{
    while (m->state == LOCKED) {
        add_to_queue(m->queue);
        wait();
    }
    m->state = LOCKED;
}
```

```
void unlock(struct mutex *m)
{
    m->state = UNLOCKED;
    t = remove_from_queue(m->queue);
    wake_up(t);
}
```

Nevoia de spinlock

- Funcțiile lock și unlock pe mutex trebuie să fie atomice
- Dezactivarea preemptivității
- Folosirea unui spinlock
- Spinlock-ul este o variabilă simplă cu acces atomic
- Operațiile folosesc busy waiting
- Pentru atomicitate: suport hardware
  - TSL (test and set lock)
  - cmpxchg (compare and exchange)

Implementare spinlock

- Un boolean/întreg cu acces atomic
- Atomic compare and exchange pentru lock

```
int atomic_cmpxchg(int val, int compare, int replace)
{   /* This is an equivalent implementation. */
    /* The entire operation is atomic. */
    int init = val;
    if (val == compare)
        val = replace;
    return init;
}
```

```
0 – locked
1 – unlocked
```

```
void lock(struct spinlock *s)
{
    while (atomic_cmpxchg(s->val, 1, 0) == 0)
        ;  /* do nothing */
}
```

```
void unlock(struct spinlock *s)
{
    atomic_set(s->val, 1);
}
```

Spinlock vs. mutex

Spinlock

- Busy-waiting
- Simplu
- Pentru regiuni critice scurte

Mutex

- Blocant
- Coadă de așteptare
- Pentru regiuni critice mari sau în care thread-ul se blochează

PROBLEMATICA SINCRONIZĂRII

Dacă nu folosim sincronizare ...

- Race conditions
  - read-before-write
  - write-after-write
  - Time Of Check To Time Of Use
- Date corupte, incoerente
- Comportament nedeterminist, arbitrar
- Procesul/thread-ul poate provoca un crash (acces nevalid la memorie)

Dacă folosim sincronizare ...

- Implementare incorectă
  - deadlock
  - așteptare nedefinită (pierderea notificării)
- Implementare ineficientă
  - granularitate regiune critică
  - cod serial
  - thundering heard
  - lock contention
- Probleme implicite
  - overhead de așteptare
  - overhead de apel

Deadlock

- Deadly embrace
- Două sau mai multe thread-uri așteaptă simultan unul după altul
  - A obține lock-ul L1 și așteaptă după lock-ul L2
  - B obține lock-ul L2 și așteaptă după lock-ul L1
- Lock-urile trebuie luate în ordine ca să prevină deadlock
- Forma de deadlock cu spinlock-uri: livelock

Așteptare nedefinită

- A așteaptă un eveniment
- B nu produce evenimentul (sau A pierde notificarea)
- A așteaptă nedefinit (evenimentul nu se produce)

Granularitatea regiunii critice

- Regiune critică mare
  - mult cod serial, paralelism redus
  - legea lui Amdahl
- Regiune critică mică
  - overhead semnificativ de lock și unlock față de lucrul efectiv în zonă
  - lock contention (încărcare pe lock)

Thundering herd

- În momentul apelului unlock() sau notify() sunt trezite toate thread-urile
- Toate thread-urile încearcă achiziționarea lock-ului
- Doar unul reușește
- Restul se blochează
- Operația este reluată la următorul apel unlock() sau notify()

Overhead de apel

- Apelurile de lock, unlock, wait, notify sunt costisitoare
- În general înseamnă apel de sistem
- De multe ori invocă planificatorul
- Mai multe apeluri, mai mult overhead
- Lock contention generează mai mult overhead
- De preferat, unde se poate, operații atomice

CONCLUZII

Când folosim thread-uri

- Multi-core programming
  - CPU intensive: libx264 (encoding), comprimare, criptare
- În medii ce indică folosirea thread-urilor
  - Java
  - kernel development
  - OpenMP
- Pentru HPC (High Performance Computing)

Când folosim sincronizare explicită

- Când e neapărat nevoie
- În cazul în care avem date comune (partajate)
- În cazul în care avem nevoie de secvențiere de operații
- Nu am putut separa/partiționa datele
- Nu am putut folosi transfer de date/mesaje (sincronizare implicită)
- Sincronizarea este un rău necesar
  - rău: overhead, regiuni seriale, deadlock la folosire necorespunzătoare
  - necesar: fără sincronizare obținem date corupte, comportament arbitrar, crash-uri, vulnerabilități de securitate

Dacă folosim sincronizare explicită ...

- Ne referim la date (nu la cod)
- Gândim bine soluția
- Avem grijă la dimensiunea regiunilor critice
- Folosim variabile atomice unde se poate
- Pentru regiuni critice mici folosim spinlock-uri
- Pentru regiuni critice mari folosim mutexuri
- Nu folosim mai multe thread-uri decât avem nevoie
  - pool de thread-uri
  - modelul boss/workers
  - modelul worker threads

Cuvinte cheie

- sincronizare
- coerență
- date partajate
- comportament arbitrar
- date corupte
- acces exclusiv
- atomicitate
- date partiționate
- secvențiere
- preemptivitate
- operații atomice
- mutex
- spinlock
- cmpxchg()
- lock()
- unlock()
- signal()
- wait()
- deadlock
- așteptare nedefinită
- granularitate
- cod serial
- race condition
- lock contention
- thundering herd