Sincronizare

SO: Curs 09

Cuprins

- Nevoia de sincronizare
- Asigurarea coerenței
- Sincronizarea la procese și thread-uri
- Implementarea sincronizării
- Problematica sincronizării
- Alternative la sincronizare

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

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)

Sincronizare

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

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

Când folosim thread-uri

- Când e neapărat nevoie
- În medii ce impun folosirea thread-urilor
  - Java
  - kernel development
  - OpenMP
- Pentru HPC
- Multi-core programming
  - CPU intensive: libx264 (encoding), comprimare, criptare
- Nu pentru perfomanță: event I/O vs. threaded I/O
- De minimizat cazurile de folosire a thread-urilor

Sincronizarea proceselor

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

Sincronizarea thread-urilor

- În general explicită
- Date partajate, concurență: acces exclusiv
  - mutex-uri
  - spinlock-uri
- Operații secvențiale: ordonare operații
  - semafoare
  - variabile condiție
  - bariere
  - monitoare
  - evenimente
  - cozi de așteptare

ASIGURAREA COERENȚEI DATELOR

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
- 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: 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

IMPLEMENTAREA SINCRONIZĂRII

Cadrul pentru sincronizare

- Mai multe entități active (procese/thread-uri)
- Puncte de interacțiune (date comune, evenimente)
- Preemptivitate

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)
        ;
}
```

```
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
- 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, multithreading 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

ALTERNATIVE LA SINCRONIZARE

Prevenirea concurenței

- Dezactivarea preemptivității
  - comportament al planificatorului
  - dezactivarea întreruperilor
  - doar pentru sisteme uniprocesor
- Partiționarea datelor

Regândirea soluției

- Trecerea de la sincronizare explicită la implicită
- Folosirea de procese în loc de thread-uri
- Date partiționate/algoritmi paralelizabili
  - reducerea “proporției” de sincronizare

Lockless synchronization

- RCU (Read-Copy-Update)
- Memorie tranzacțională
- De forma Commit/Abort

Event-driven I/O vs. Thread I/O

Event-driven I/O

- Operații asincrone
- Un singur thread
- Automat de stări
- Scalabilitate
- Mai dificil de programat

Thread I/O

- Thread-uri per “conexiuni”
- Ușor de programat
- Necesită sincronizare/interacțiune

CONCLUZII

Când folosim thread-uri?

- Când e neapărat nevoie
- În framework-uri/medii ce impun folosirea thread-urilor
  - Java
  - kernel development
  - OpenMP
- Pentru HPC (High Performance Computing)
- Multi-core programming
  - CPU intensive: libx264 (encoding), comprimare, criptare
- Procesele au timp de creare foarte bun și separație bună a datelor
  - un proces “stricat” nu le “strică” pe celelalte
- Event-based I/O este mai scalabil
- De minimizat cazurile de folosire a thread-urilor

Când folosim sincronizare explicită?

- Când e neapărat nevoie
- În cazul în care folosim thread-uri
- Nu am putut separa/partiționa datele
- Nu am putut folosi transfer de date/mesaje (sincronizare implicită)

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
  - boss-workers
  - worker threads

Cuvinte cheie

- sincronizare
- coerență
- date partajate
- acces exclusiv
- atomicitate
- date partiționate
- secvențiere
- preemptivitate
- mutex
- spinlock
- cmpxchg()
- lock()
- unlock()
- signal()
- wait()
- deadlock
- așteptare nedefinită
- granularitate
- cod serial
- race condition
- lock contention
- thundering herd
- dvent-based I/O