

# Sincronizare

SO: Curs 09

# Cuprins

- Recapitulare thread-uri
- Contextul sincronizării
- Implementarea mecanismelor de sincronizare
- Sincronizarea multi core
- Structuri de 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**

# 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)
- 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

# CONTEXTUL SINCRONIZĂRII



# Context

- **Sisteme multiproces**
  - procese, thread-uri date comune
- **Sisteme multicore**
  - magistrală comună, memorie comună
- **Nevoie de comunicare**
  - canale de comunicație, date partajate

# Situații posibile

- read before write
- time of check to time of use (TOCTOU)
- interleaved access

# Condiții de cursă

- race conditions
- output-ul unui sistem depinde de timp sau de evenimente
- dacă ordinea nu e cea dorită e un bug
- nedeterminism în execuție
- date inconsecvente/neintegre
- poate fi vulnerabilitate exploatabilă

# Date inconsecvente

```
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?**

# TOCTOU

```
if (access("file", W_OK) != 0) {  
    exit(1);  
}  
  
/* done by another process */  
. . . . .  
symlink("/etc/passwd", "file");  
. . . . .  
  
fd = open("file", O_WRONLY);  
write(fd, buffer, sizeof(buffer));
```

[https://en.wikipedia.org/wiki/Time\\_of\\_check\\_to\\_time\\_of\\_use#Examples](https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use#Examples)

# Sincronizare

- Asigurarea accesului corect la date
  - date integre
  - determinism
- Acces exclusiv / serializare / atomizare
  - regiune critică
- Secvențiere / ordonare
  - read after write, write after write, use after create

# Primitive de sincronizare

- Ordonare
  - wait()
  - notify()
- Acces exclusiv
  - lock()
  - unlock()

# Mecanisme de sincronizare

- Ordonare
  - semafoare
  - cozi de așteptare
  - variabile condiție
  - monitoare
- Acces exclusiv
  - variabile atomice
  - spinlock-uri
  - mutex-uri



# IMPLEMENTAREA MECANISMELOR DE SINCRONIZARE

# Atomicitatea unei operații

- Atomicitate: operația se execută fără intervenția unui alt proces sau core
- Este `a += 5` atomică în C?

# Single core / multi core

- `a += 5`
- x86: `add [ebp-12], 5`
  - atomică single core
  - înseamnă read-update-write
  - poate fi “întreruptă” de alt core
  - neatomică multi core
- ARM: `load, add, store`
  - neatomică nici pe single core

# Adunare atomică

- `__sync_fetch_and_add` (GCC)
  - <https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html>
- pe x86 multi core pune lock pe magistrală
- pe ARM leagă face operații tranzacționale
  - tranzațional: totul sau nimic
  - dacă nu iese, încearcă iar

# Atomic pe multi core

- un singur core să poată folosi exclusiv magistrala
- prefixul lock pe x86
- ldrex, strex pe ARM

# Locking (implementare naivă)

```
lock = 0; /* init */
```

```
while (lock == 1)
```

```
    ; /* do nothing */
```

```
lock = 1; /* get lock */
```

# Compare and Swap (CAS)

- Compare and Exchange
- Prevenirea TOCTOU pentru locking naiv
- Operație atomică simultană de verificare și actualizare
- `compare_and_exchange(lock, 0, 1);`

# Compare and Swap (CAS) (2)

- `compare_and_exchange(lock, 0, 1);`
- atomic for:

```
if (value == to_compare)
    value = to_update;
return value;
```



# Locking using CAS

- show demo

# Spinlock

- basic exclusive access primitive
- CAS-based, în general optimizat
- suport multi core, access exclusiv la magistrală
- uses busy waiting
- `spin_lock()`, `spin_unlock()`
- show demo

# Mutex

- primitivă de acces exclusiv
- blocantă, are o coadă de așteptare pentru procese
- adecvat pentru regiuni critice mai mari

# Implementare mutex

- o structură
- un câmp stare internă
- o coadă de așteptare de procese
- un spinlock pentru protejarea structurii interne
- dacă regiunea critică nu e ocupată (not contended) nu intră în coada de așteptare (fast path)
- futex: implementare cu suport user-space în Linux
  - evită apeluri de sistem pentru not contended

# 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

# SINCRONIZAREA MULTI CORE

# SMP

- Symmetric Multi Processing
  - memorie comună
  - magistrală comună de acces la memorie
- sincronizarea pe un cor nu ține cont de magistrală
  - de exemplu: dezactivare întreruperi
- pe multi core: acces exclusiv la magistrală

# Acces exclusiv pe magistrală

- prefixul lock pe x86

lock add [ebp-12], 5

- ldrex, strex pe ARM

ldrex ...

...

strex ...

cmp ... ; if not transactional, try again

- overhead de serializare a accesului



# Sincronizarea cu lock multi core

- lock-ul este o variabilă comună
- este încăcată în memoria cache a fiecărei core
- modificările unui core (lock, unlock) se fac în cache-ul local
- modificările se propagă la celelalte cache-uri

# Cache thrashing

- modificări într-un cache se propagă în celelalte cache-uri
- intrările în celelalte cache-uri se invalidează
- spinlock-urile pe core-uri diferite duc la modificări pe un core și invalidări pe celelalte
- are loc cache thrashing
  - invalidări frecvente
  - citiri din memoria principală
  - overhead

# Variabile per-CPU

- câte o variabilă per procesor
- nu e nevoie de sincronizare inter-core
- exemplu: procesul curent care rulează
- partiționare a datelor, eliminarea overhead-ului de sincronizare

# STRUCTURI DE SINCRONIZARE

# Acces la date comune

- **acces concurent**
  - nevoie de acces exclusiv
- **acces comunicativ/colaborativ**
  - nevoie de ordonare citire/scriere
  - producător-consumator

# Producător consumator

- zonă comună partajată
- unul sau mai mulți scriitori (producători)
- unul sau mai mulți cititori (consumatori)
- buffer/zonă cu mai multe celule
  - diferență de viteză producători/consumatori
  - networking, I/O, message passing

# Implementare producător consumator

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);
```

# Buffer circular

- circular buffer, ring buffer



<https://blog.grijjy.com/2017/01/12/expand-your-collections-collection-part-2-a-generic-ring-buffer/>



# Buffer circular

- buffer obișnuit cu “wrap around”
- consume() / produce() sau get() / put()
- $\text{index} = (\text{index} + 1) \% \text{BUFFER\_SIZE}$
- read\_index, write\_index, capacity, size
- demo
- exemplu implementare: kfifo în nucleul Linux

# CONCLUZII

# Contextul sincronizării

- **Sisteme multiproces**
  - procese, thread-uri date comune
- **Sisteme multicores**
  - magistrală comună, memorie comună
- **Nevoie de comunicare**
  - canale de comunicație, date partajate
- **Nevoie de ordonare, determinism, date integre**

# Mecanisme de sincronizare

- necesită suport hardware
- compare and swap (CAS)
- acces exclusiv/serial
  - variabile atomice
  - spinlock
  - mutex
- secvențiere/determinism/ordonare
  - semafoare
  - cozi de așteptare
  - variabile condiție
  - monitoare

# Sincronizare SMP

- acces exclusiv pe magistrală (lock)
- cache thrashing
- variabile per-CPU

# Buffer circular

- folosit pentru producător-consumator
- viteze diferite de scriere și citire
- buffer obișnuit cu “wrap around”

# Probleme de sincronizare

- race conditions
- TOCTOU
- deadlock: așteptare mutuală
- livelock: “așteptare” cu spinlock-uri

# 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



# Cuvinte cheie

- sincronizare
- multi core
- determinism
- condiție de cursă
- inconsecvență
- TOCTOU
- acces exclusiv
- atomicitate
- secvențiere
- operații atomice
- mutex
- spinlock
- CAS (compare and swap)
- cmpxchg()
- lock()
- unlock()
- signal()
- wait()
- locking pe magistrală
- cache thrashing
- per-cpu variables
- ring buffer
- producător-consumator
- overhead de sincronizare