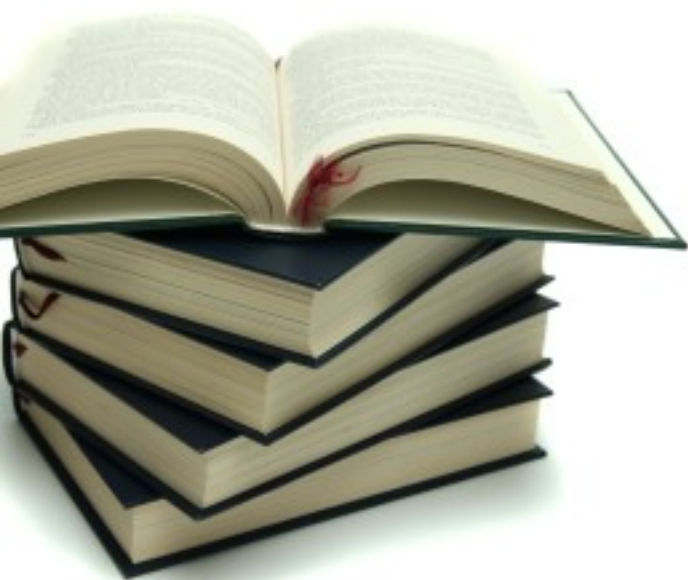


- De ce rularea unei întreruperi cu întreruperile activate poate duce la stack overflow?
- Se poate partaja aceeași întrerupere de către două dispozitive?  
Explicați.
- În Linux, pot întreruperile software duce la „process starvation”?  
Explicați.



# 5

## Sincronizare

24 martie 2010

- Concurență în kernel
- Operații atomice
- Spinlock-uri
- Cache trashing
- Primitive de sincronizare blocante
- Per-cpu data
- Bariere
- RCU (Read-Copy-Update)
- Bibliografie
  - UTLK: capitolul 5
  - LKD: capitolele 8, 9
  - <http://lwn.net/Articles/262464/>

- Avem nevoie de sincronizare atunci când
  - Există concurență
  - „Concurenții” accesează aceeași zonă de memorie
  - Se fac atât accese RO cât și RW
- Lipsa sincronizării duce la condiții de cursă

- Două fire de execuție decrementează variabila  $V$  până la 0

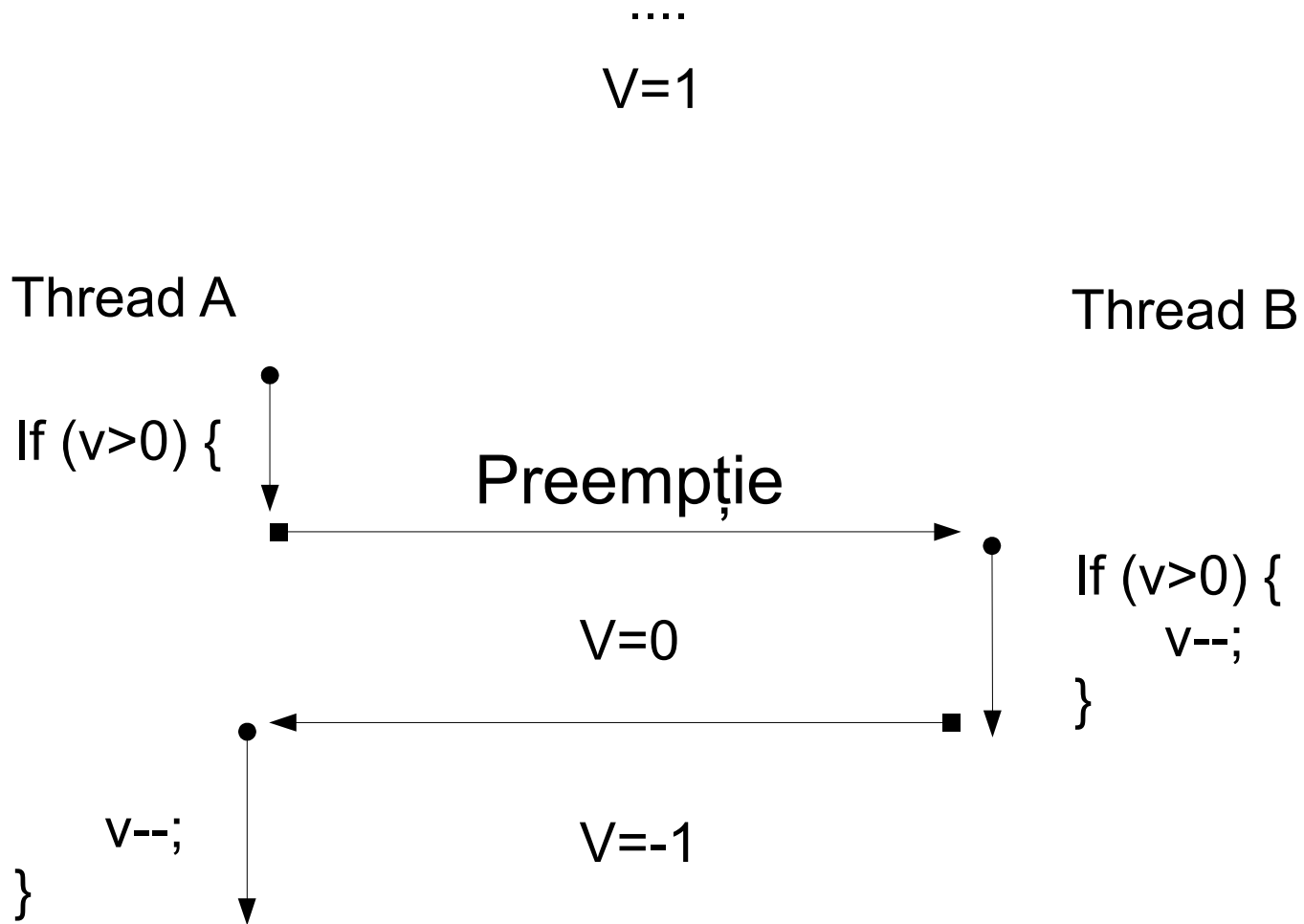
```
while (v>0)  
    v--;
```

Thread A

```
while (v>0)  
    v--;
```

Thread B

- Ce valoare va avea variabila  $V$  după terminarea execuției celor două fire de execuție?



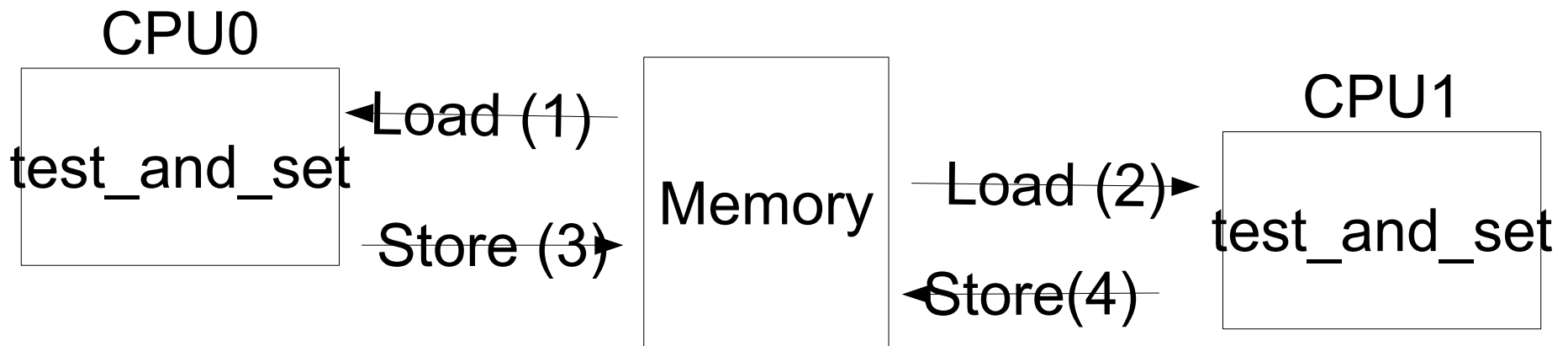
- Zona critică = zona de cod ce accesează datele partajate din mai multe contexte de execuție
- Accesul la date în zona critică făcut într-un mod ordonat, astfel încât rezultatele să fie predictibile
  - În exemplul precedent, thread-ul B ar trebui să aștepte până thread-ul A iese din if
- Soluție: atomicizarea zonei critice, dezactivarea preempției în zona critică, secvențializarea accesului la zona critică

- Sisteme uniprocessor, kernel nepreemptiv: procesul curent poate fi preemptat de către întreruperi
- Sisteme uniprocessor, kernel preemptiv: procesul curent poate fi preemptat de către alte procese
- Sisteme multiprocessor: procesul curent se poate executa în paralel cu un alt proces sau întrerupere ce rulează pe alt procesor

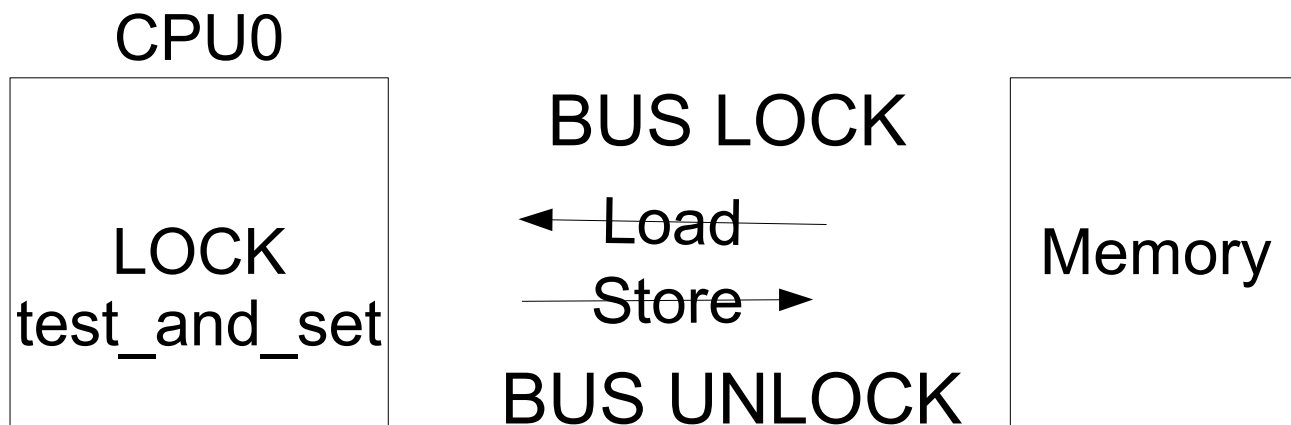


- Operații cu întregi:
  - incrementare, decrementare, adunare, scădere
  - `add_and_test`, `sub_and_test`, `dec_and_test`, `inc_and_test`: efectuează operație și întoarce valoarea după operație
- Operații cu biți:
  - `test/set/clear/change bit`
  - `test_and_set/clear/change bit`: efectuează operația și întoarce valoarea dinaintea operației

- Chiar dacă procesoarele pot rula cod concurent, memoria este partajată, deci accesul este secvențializat
- Dar, operațiile atomice la nivel de procesor nu mai sunt atomice la nivel de sistem



- Soluția: lock pe magistrală pe durata execuție instrucțiunii



- Pe sistemele uniprocessor, dezactivarea întreruperilor elimină sursa de concurență
- Operația e suficientă pentru a proteja zonele critice
- Funcționează atât pentru kernel preemptive cât și pentru kernel nepreemptive

```
local_irq_disable() \  
    asm volatile („cli” : : : „memory”);  
  
local_irq_enable() \  
    asm volatile („sti” : : : „memory”);  
  
local_irq_save(flags) \  
    asm volatile ("pushf ; pop  
    %0" : "=g" (flags): /* no  
    input */: "memory")  
  
local_irq_restore(flags) \  
    asm volatile ("push %0 ; popf"  
    : /* no output */  
    : "g" (flags) : "memory", "cc");
```

- Necesare pe sisteme multiprocesor
- Sunt folosite pentru a secvențializa accesul la zona critică
- Implementare tipică

```
spin_lock:  
    LOCK BTS [LOCK],0  
    JC spin_lock
```

```
unlock:  
    MOV LOCK, 0
```

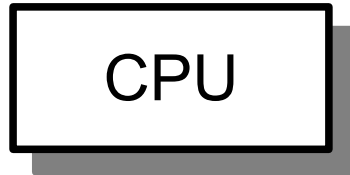
- Usage: `BTS dest,src`
- Modifies Flags: CF
- The destination bit indexed by the source value is copied into the Carry Flag and then set in the destination.

- Există lock contention atunci când avem cel puțin un procesor care așteaptă să între într-o zonă critică
- Cu cât zona critică este mai mare, cu atât crește probabilitatea și timpul petrecut în lock contention
- Cu cât avem un număr mai mare de procesoare cu atât crește probabilitatea și timpul petrecut în lock contention

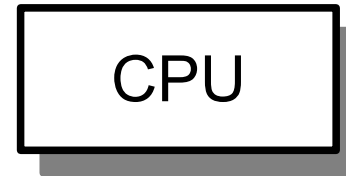
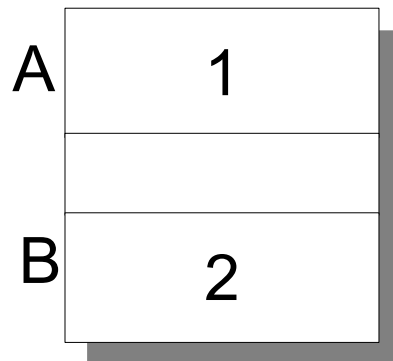


- Datorită faptului că spinlock-urile accesează foarte des memoria atunci când există contention, există o probabilitate crescută pentru a induce fenomenul de cache trashing
- Fenomenul de cache trashing apare la sistemele multiprocesor atunci când o line de cache se „plimbă de la un procesor la altul”
- Consecință a modului în care e implementată sincronizarea cache-urilor (L1) în sisteme multiprocesor

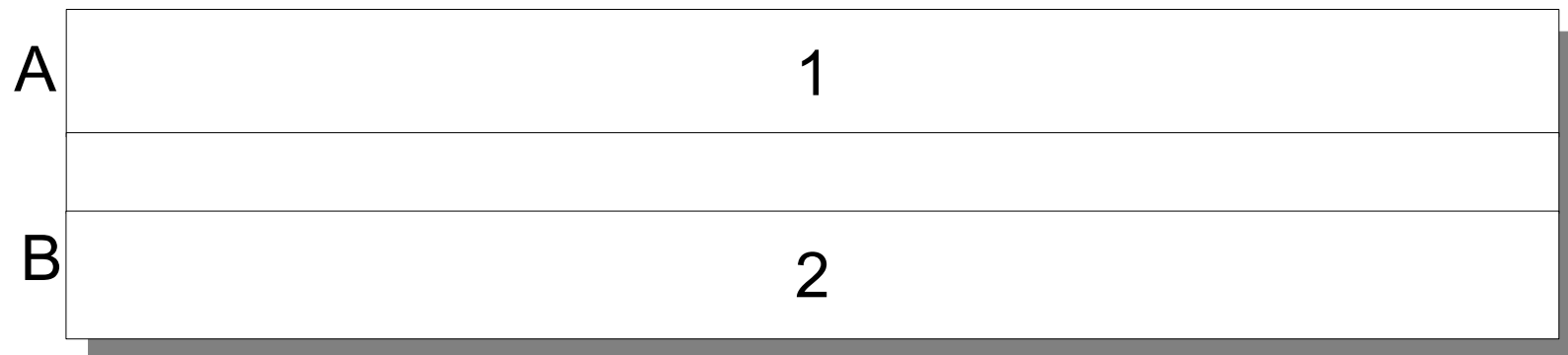
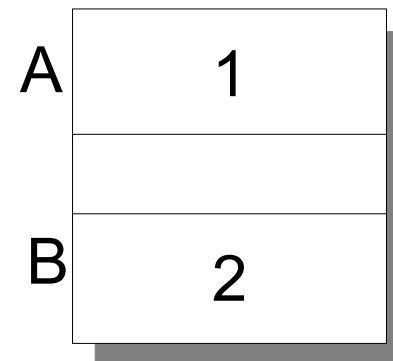
# De ce trebuie să sincronizăm cache-urile?



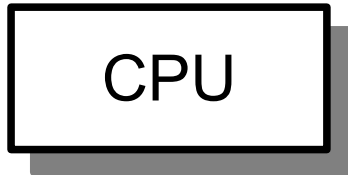
Write-back Cache



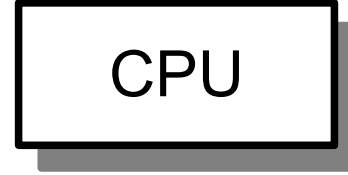
Write-back Cache



# De ce trebuie să sincronizăm cache-urile? (2)

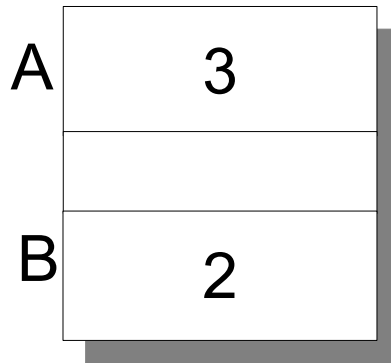


NOP  
A=A+B

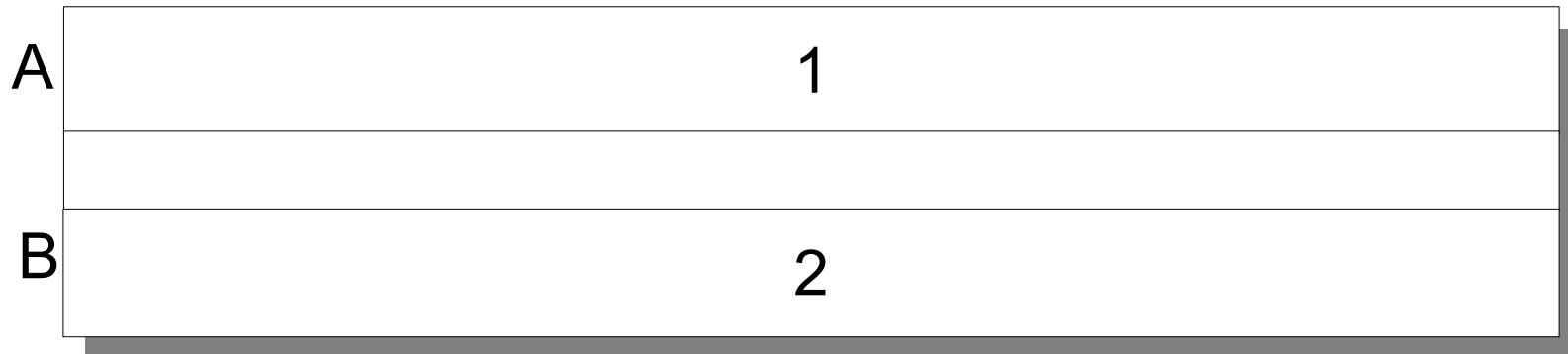
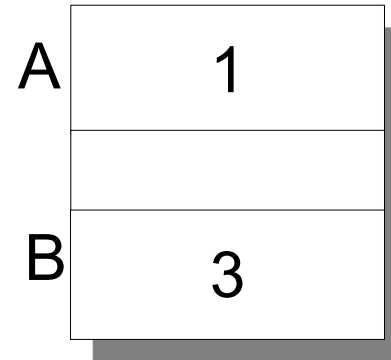


B=A+B  
NOP

Write-back Cache

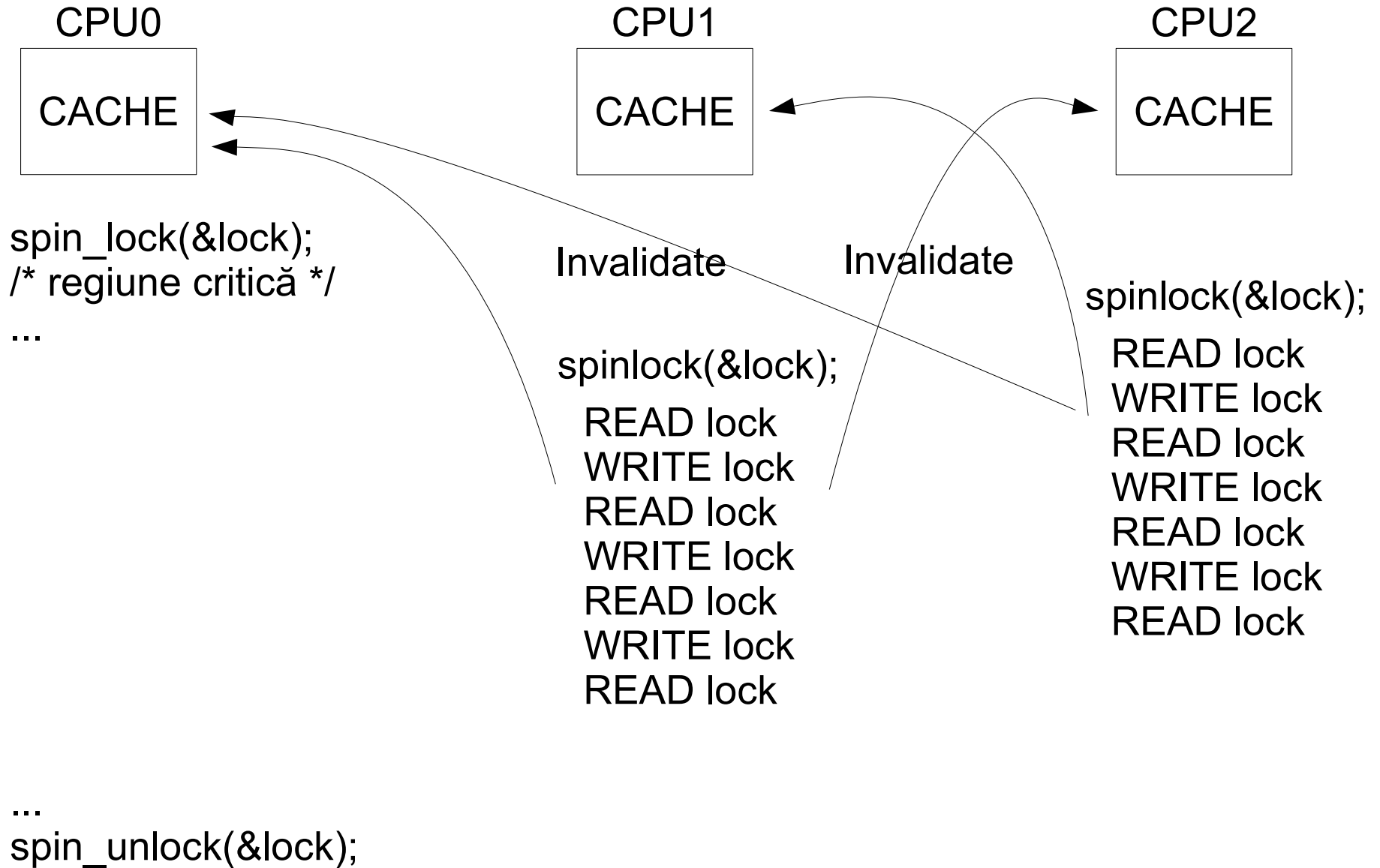


Write-back Cache



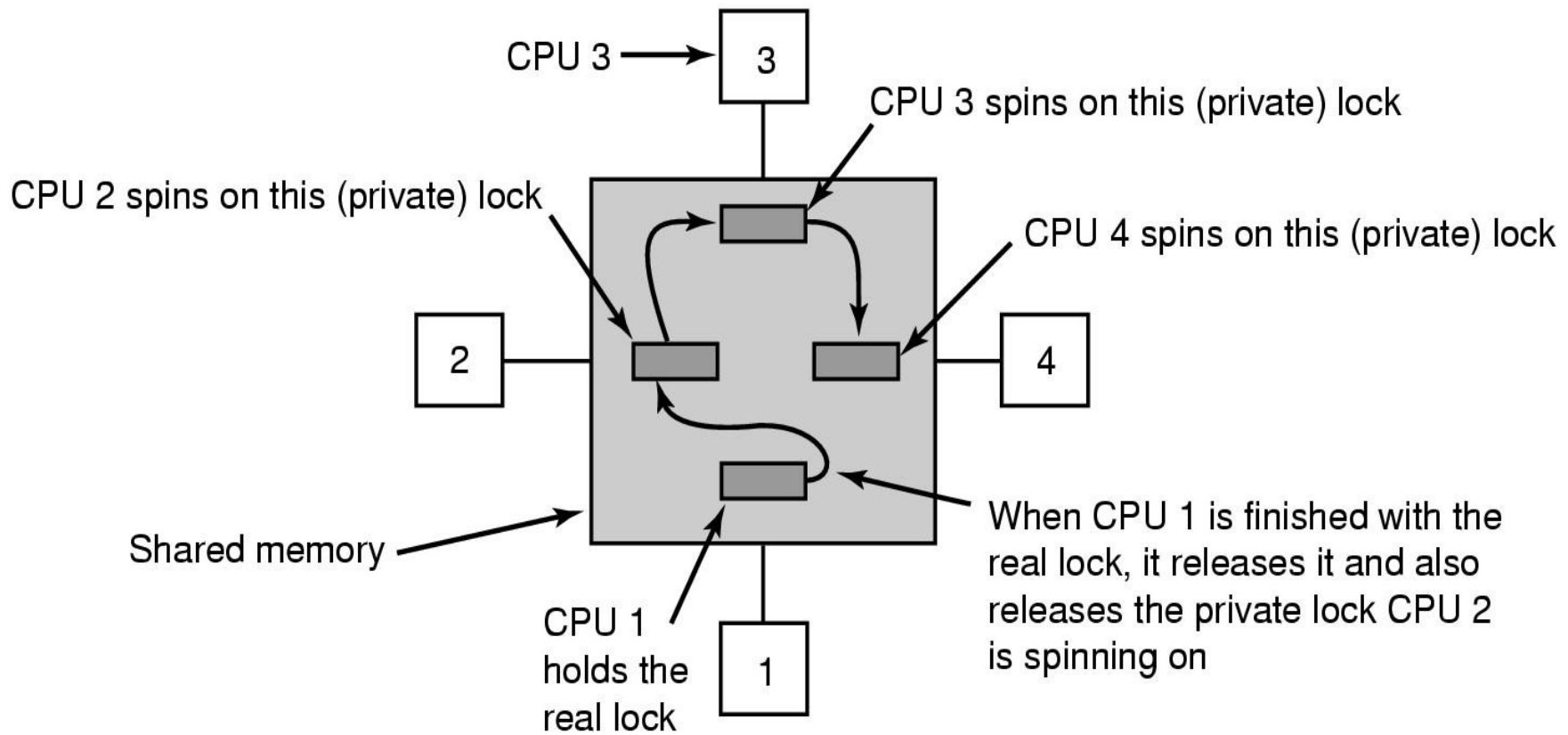
- În hardware, cu protocoale de sincronizare
- Snooping: cache-ul monitorizează accesul la memorie și în cazul în care se detectează o operație de scriere se invalidează linia de cache
- Directory: se mențin global informații despre unde (în ce cache) se găsesc cele mai actuale modificări
- Snooping vs Directory: directory scalează mai bine (este folosit pentru  $\geq 64$  procesoare), snooping are latență mai mică

- Politica de caching: write-back
- O linie de cache are 4 stări: Modified, Exclusive, Shared, Invalid
- Operațiile de citire pot fi servite din orice stare, mai puțin Invalid
- Operațiile de scriere pot fi servite doar din stările Modified și Exclusive
- Dacă linia este în starea Shared, un Write va determina un cache Invalidat pe toate celelalte procesoare
- Procesorul care ține linia în Modified, trebuie să intercepteze toate operațiile de citire (snoop) și să:
  - Pună pe hold operația de citire
  - Să facă flush la linie modifică
  - Să reia operația de citire



```
;
; implementarea
; KeAcquireSpinLock
;
spin_lock:
    rep ; nop
    test lock_addr, 1
    jnz spin_lock
    lock bts lock_addr
    jc spin_lock
```

- Majoritatea acceselor vor fi de tip READ astfel încât se va reduce frecvența invalidărilor cacheului
- Rep; nop delay proporțional cu viteza memoriei





- Scenariu: date partajate între cod ce rulează în process context și cod ce rulează într-o întrerupere
- În context proces, se ia spinlock-ul
- Pe același procesor vine întreruperea, și se rulează rutina de tratare
- Rutine de tratare încearcă să ia spinlock-ul
- Deadlock

- Context process:
  - dezactivare întreruperi: ne protejează împotriva preempției pe procesorul curent +
  - luare spinlock: ne protejează împotriva rulării codului din al proces context sau întrerupere, pe alt procesor
- Context întrerupere:
  - luare spinlock: ne protejează împotriva rulării codului în context proces pe alt procesor
- Linux:
  - `spin_lock_irqsave`
  - `spin_unlock_irqrestore`
- Windows
  - `KeSynchronizeExecution`
  - `KeAcquireInterruptSpinLock`

- Același scenariu se aplică și la BH-uri, pentru că BH-urile pot preempta contextul process
- Linux:
  - `spin_lock_bh`: `local_bh_disable` + `spin_lock`
  - `spin_unlock_bh`: `spin_unlock` + `local_bh_enable`
- Windows
  - `KeAcquireSpinLock` ridică automat nivelul IRQL la `DISPATCH_LEVEL`

```
#define local_bh_disable() \
    add_preempt_count(SOFTIRQ_OFFSET); \
    barrier();

#define local_bh_enable() \
    sub_preempt_count(SOFTIRQ_OFFSET);

#define irq_count() \
    (preempt_count() & (HARDIRQ_MASK | SOFTIRQ_MASK))

#define in_interrupt() irq_count()

asmlinkage void do_softirq(void)
{
    if (in_interrupt()) return;
    ...
```

- Poate fi configurat
  - Activat – timp de răspuns mai bun
  - Dezactivat – throughput mai bun
- Este dezactivat la luarea unui spinlock, nu avem nevoie de protecție specială
- Dezactivarea poate fi imbricată, se ține un contor de preempție în `current_thread_info()`

- Dacă regiunea critică este protejată, se pune procesul curent în așteptare
- Nu pot fi chemate din context întrerupere
- Latența mai mare decât spinlock-urile
- Throughput mai bun dacă overhead-ul schimbării de context e mai mic decât timpul de așteptare mediu
- Semafoare, Mutex-uri

```
static inline void down(struct semaphore * sem)
{
    might_sleep();
    __asm__ __volatile__ (
        "LOCK decl %0\n\t"
        "jns 2f\n"
        "lea %0,%%eax\n\t"
        "call __down_failed\n"
        "2:"
        : "+m" (sem->count) : : "memory", "ax" );
}
```

```
void __sched __down(struct semaphore *sem)
{
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);
    unsigned long flags;

    tsk->state = TASK_UNINTERRUPTIBLE;
    spin_lock_irqsave(&sem->wait.lock, flags);
    add_wait_queue_exclusive_locked(&sem->wait, &wait);

    sem->sleepers++;
    for (;;) {
        int sleepers = sem->sleepers;
        if (!atomic_add_negative(sleepers - 1, &sem->count)) {
            sem->sleepers = 0;
            break;
        }
        sem->sleepers = 1;          /* us - see -1 above */
        spin_unlock_irqrestore(&sem->wait.lock, flags);
    }
}
```



```
schedule();
```

```
spin_lock_irqsave(&sem->wait.lock, flags);  
tsk->state = TASK_UNINTERRUPTIBLE;
```

```
}
```

```
remove_wait_queue_locked(&sem->wait, &wait);  
wake_up_locked(&sem->wait);  
spin_unlock_irqrestore(&sem->wait.lock, flags);
```

```
tsk->state = TASK_RUNNING;
```

```
}
```

- Primitivele de sincronizare serializează codul
- Scalabilitatea sistemului este dată de cât de fină este serializarea
  - De exemplu, în Linux 2.0 tot kernelul era protejat de un singur lock
- Chiar dacă granularitatea locking-ului este fină, în anumite workload-uri poate apărea contention
- Chiar și operațiile atomice pot cauza probleme de performanță (cache trashing)

- Două tipuri de lock-uri: pentru readeri și pentru writeri
- Într-o zonă critică pot intra oricât de mulți readeri dar un singur writer
- Reduce contention-ul dacă majoritatea operațiilor efectuate sunt de tip read

- Metoda ideală de „sincronizare”
- Datele se țin distribuit, per procesor
- Nu avem nevoie de sincronizare
- Nu există contention, crește scalabilitatea
- Scade probabilitatea de cache trashing
- Abordare hibridă: folosirea per-cpu-data most of the time cu agregarea per-cpu-data some of the time

- Procesoarele si/sau compilatoarele pot genera cod out of order
- Exemplu:

```
a=1;  
b=2;
```

```
MOV R10, 1  
MOV R11, 2  
STORE R11, b  
STORE R10, a
```

- Barierele forțază compilatorul și procesorul să ordoneze operațiile de citire / scriere
- Read memory barrier – operațiile de citire nu sunt reordonate „peste” barieră
- Write memory barrier – operațiile de scriere nu sunt reordonate „peste” barieră

- Read-Copy-Update
- Acces read-only fără locking la structuri modificate concurent
- Nu necesită lock-uri sau instrucțiuni atomice
- Este folosit de obicei cu structurile de date înlănțuite (liste, cozi) traversate unidirecțional cu operații dese de citire (read-mostly)

- Actualizările (modificările) se împart în două faze
  - eliminare (removal)
  - reclamarea spațiului eliberat (reclamation)
- Faza de eliminare are loc imediat
- Faza de reclamare este amânată până când toți cititorii prezenți în faza de eliminare și-au încheiat activitatea
  - se înregistrează o funcție tip callback



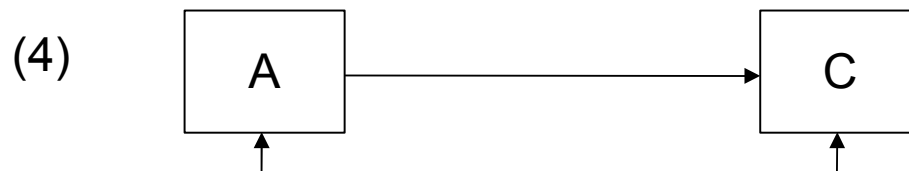
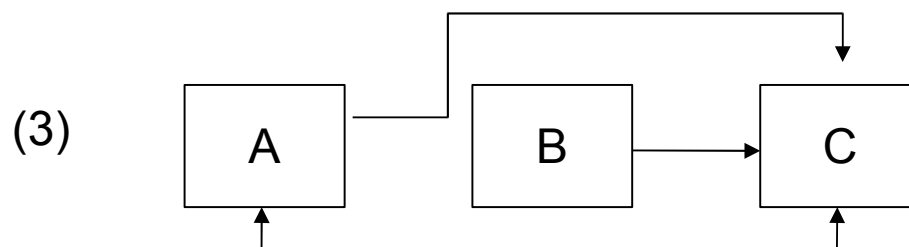
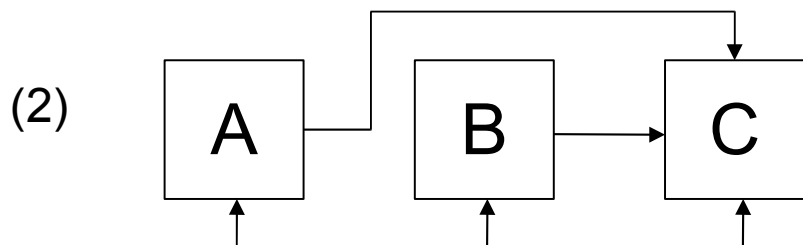
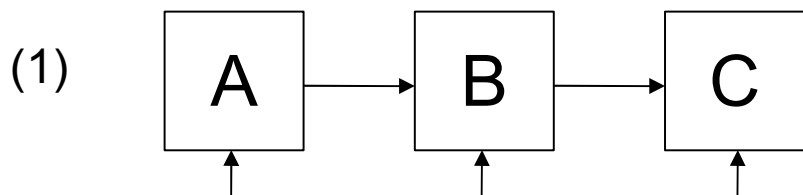
- **eliminare**
  - eliminarea tuturor referințelor către structura curentă
  - referințele pot fi înlocuite cu referințe către o versiune nouă a structurii (elementul următor dintr-o listă)
  - cititorii care refereau structura vor lucra în continuare pe această versiune
  - noii cititori vor referi structura nouă
- **reclamare**
  - eliberarea spațiului ocupat de structura eliminată
  - începe doar după ce nu mai există cititori cu referință la structură

(1) – parcurgere,

(2) – eliminare (B este încă referit)

(3) – nu se mai referă

(4) – eliminare efectivă



- pentru o regiune critică RCU constrângerile sunt asemănătoare cu cele pentru utilizarea unui spinlock
  - nu se poate realiza schimbare de context (nu se poate dormi, nu se poate folosi un lock blocant, etc.)
- dacă pe un procesor se realizează o schimbare de context, procesul asociat nu mai deține nici o referință la un element eliminat -> se poate reclama
- așteptarea tuturor cititorilor (pasul 2 din secvența tipică) înseamnă rularea pe fiecare procesor până la o schimbare de context

?

