Sisteme de operare avansate

RCU - Read-Copy-Update

Paul McKenney, Andrea Arcangeli et al.

Locking vs. lockless synchronization

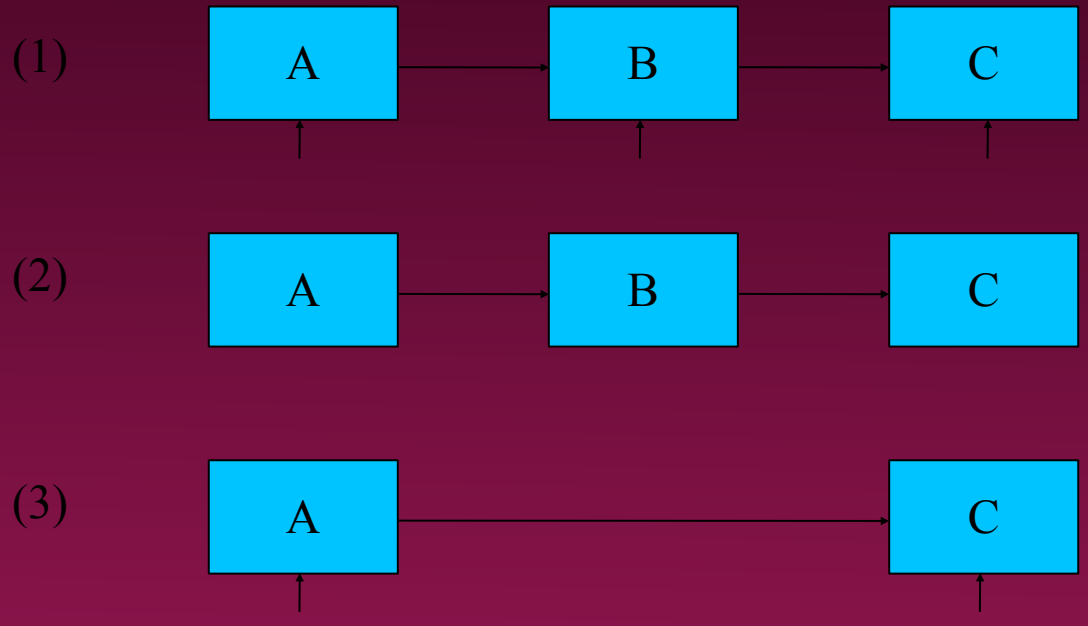
- sincronizarea este necesară pentru aplicaţiile multithreaded şi sisteme multiprocesor
- sincronizarea thread-urilor presupune overhead
- alte probleme apărute prin folosirea primitivelor de sincronizare
  - deadlocks (aşteptare mutuală blocantă)
  - inversarea priorităţii (un thread mai puţin prioritar “ţine” lock-ul în faţa unui mai prioritar)
  - convoying (se “strâng” multe thread-uri la un singur lock)
  - sincronizarea este scumpă (achiziţionarea şi eliberarea unui spinlock uncontended necesită sute de cicluri de procesor)
- soluţii
  - evitarea sincronizării (pe cât posibil)
  - folosirea de sincronizare fără locking
  - este nevoie de structuri specializate care implică folosirea unei scheme de reclamare a memoriei

RCU

- read-copy update
- acces read-only fără locking la structuri modificate concurent
- nu necesită lock-uri sau instrucţiuni atomice
- este folosit de obicei cu structurile de date înlănţuite (liste, cozi) traversate unidirecţional cu operaţii dese de citire (read-mostly)
- rcu-poll
- rcu-ltimer

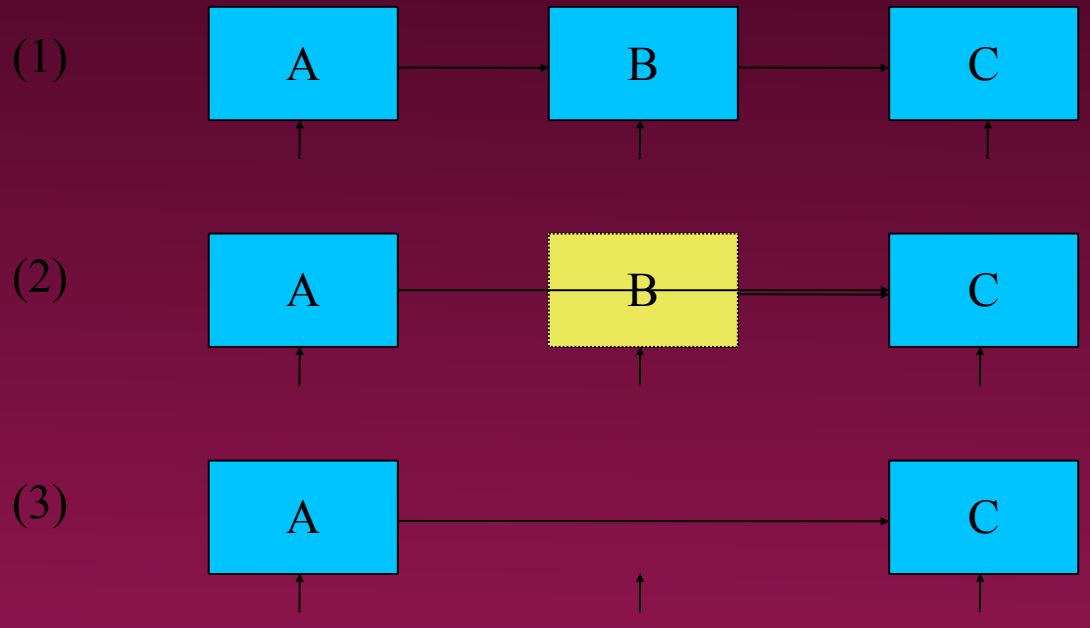
Acces sincronizat la o listă

- parcurgere şi ştergere - readers/writers lock
- (1) – parcurgere listă (read lock held)
- (2) – eliminare nod B (write lock held)
- (3) – parcurgere listă după eliminare (read lock held)

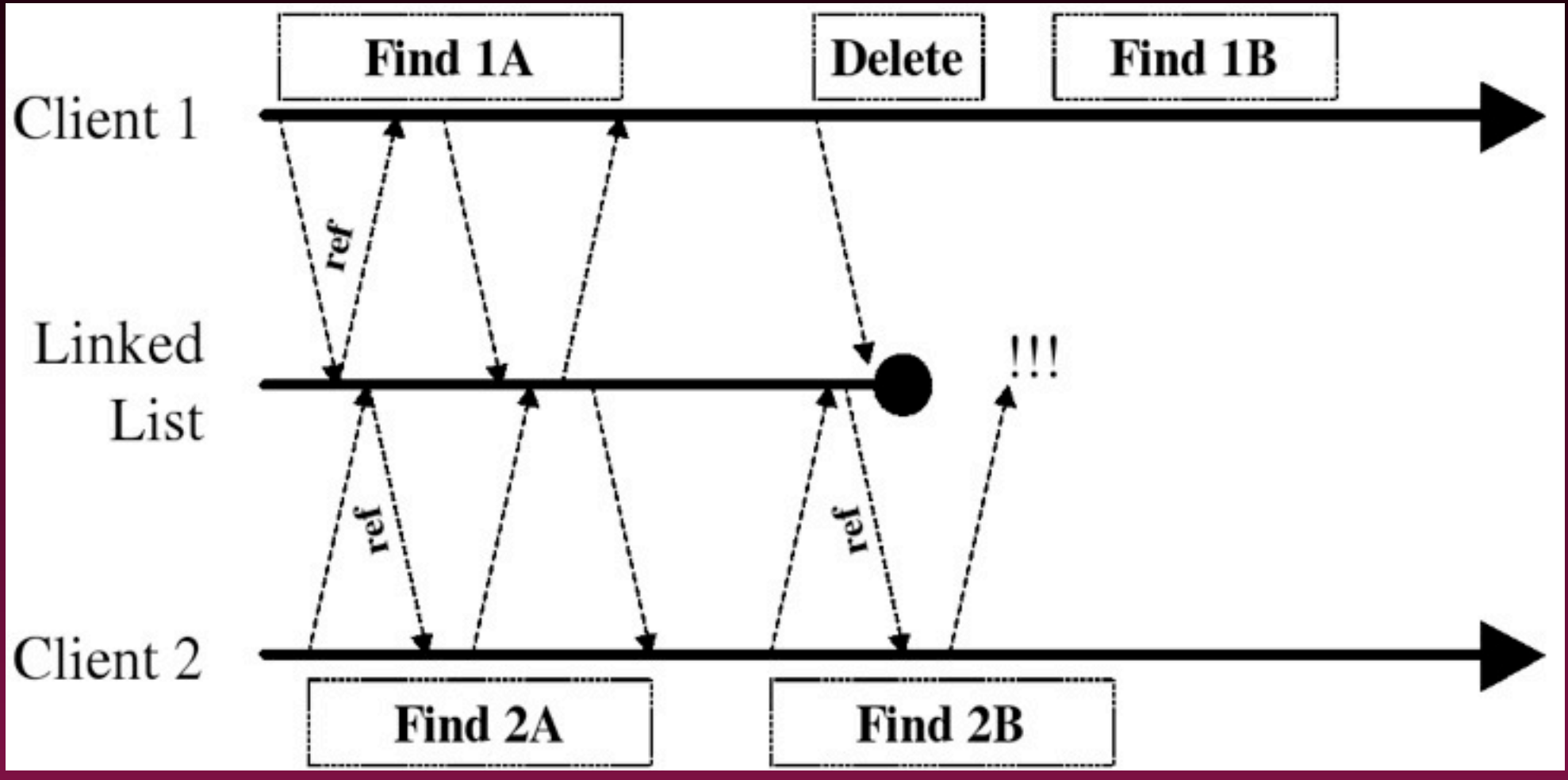


Acces fără locking la o listă

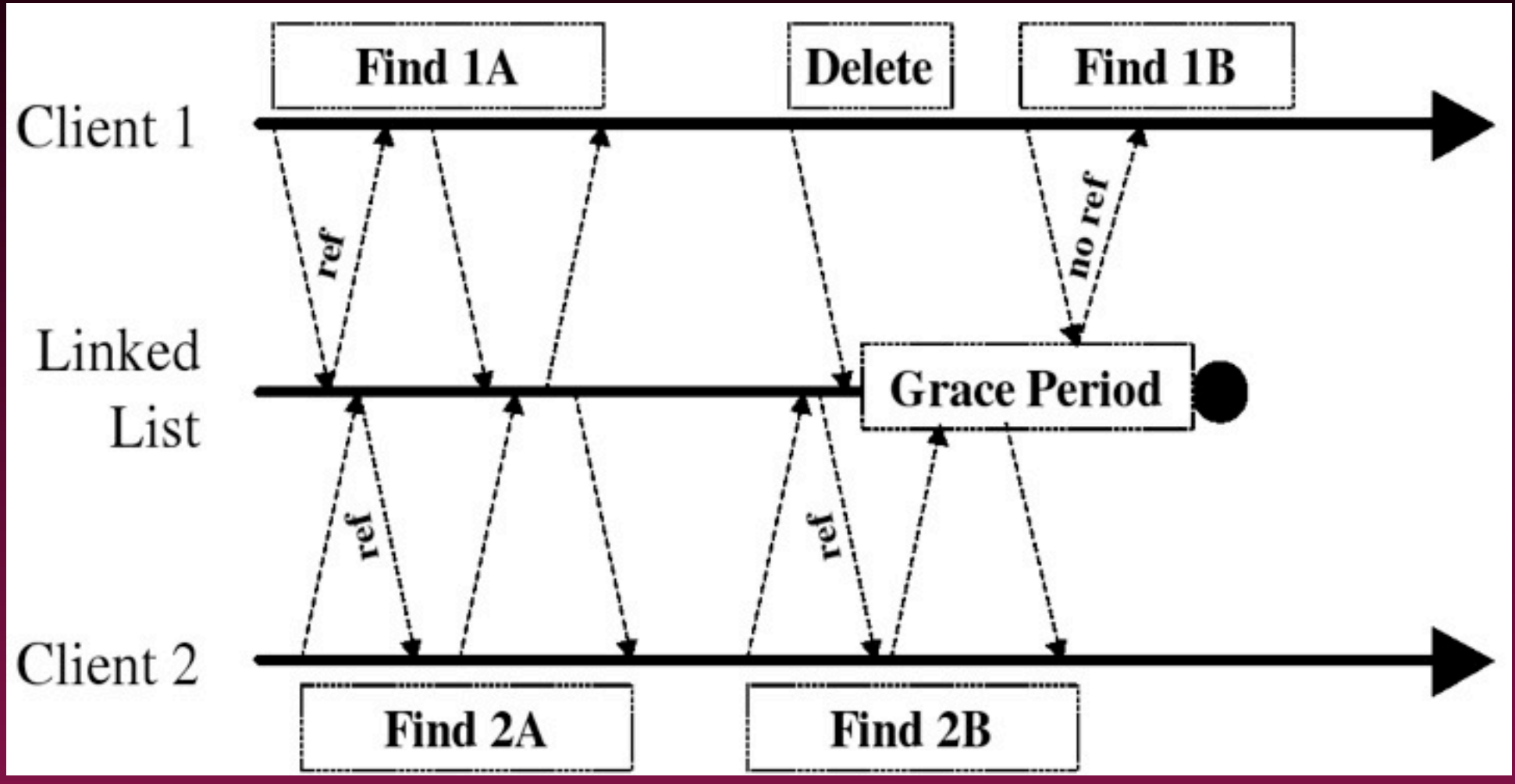
- de ce?
  - de multe ori o listă este accesată (parcursă) mult mai des decât este modificată (tabela de rutare)
  - obţinerea unui lock conduce la overhead important
- probleme
  - modificarea (update) nu mai este atomică



Acces fără locking

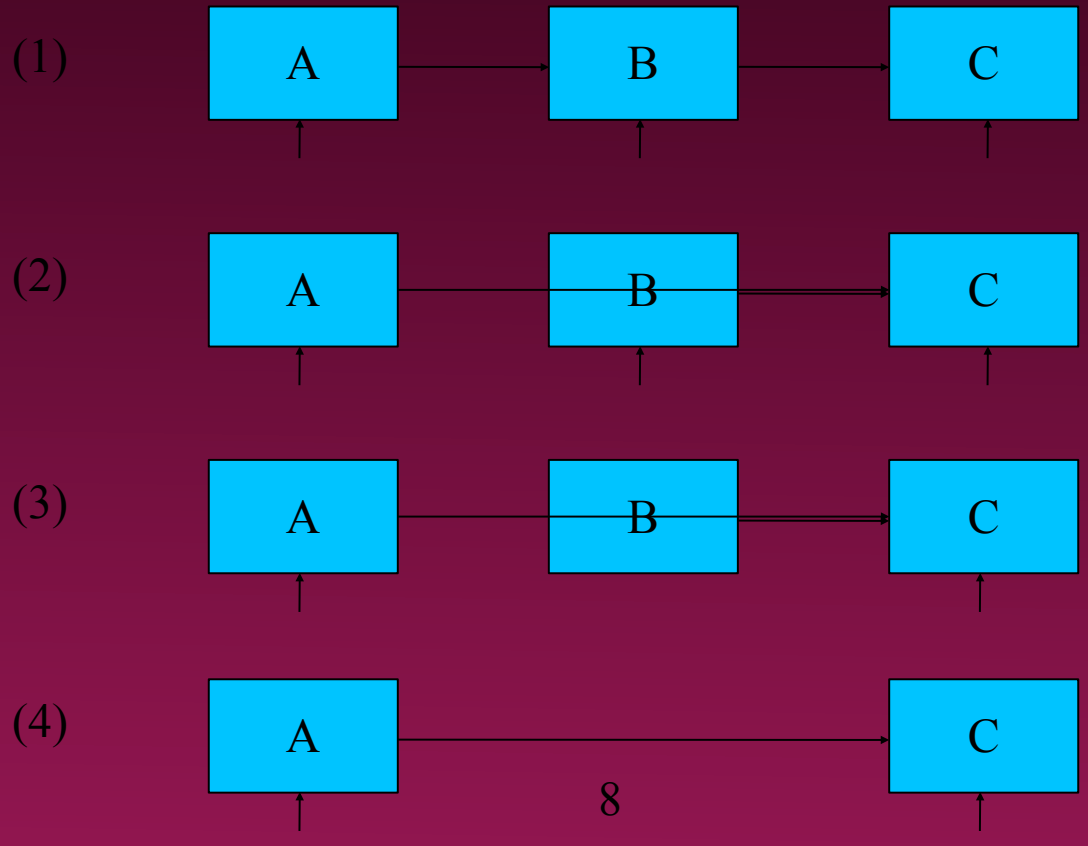


Acces fără locking cu perioadă de grație



Funcţionare RCU

- se amână eliberarea elementului eliminat până când nici un thread care să refere acel element
- (1) – parcurgere, (2) – eliminare (B este încă referit), (3) – nu se mai referă, (4) – eliminare efectivă



Funcţionare RCU (2)

- actualizările (modificările) se împart în două faze
  - eliminare (removal)
  - reclamarea spaţiului eliberat (reclaim)
- eliminare
  - eliminarea tuturor referinţelor către structura curentă
  - referinţele pot fi înlocuite cu referinţe 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 spaţiului ocupat de structura eliminată
  - începe doar după ce nu mai există cititori cu referinţă la structură

Funcţionare RCU (3)

- faza de eliminare are loc imediat
- faza de reclamare este amânată până când toţi cititorii prezenţi în faza de eliminare şi-au încheiat activitatea
  - se blochează până la încheiere
  - se înregistrează o funcţie tip callback
- secvenţa tipică este
  1) eliminarea pointer-ilor la o structură pentru a nu putea fi referiţi de alţi cititori
  2) aşteptarea tuturor cititorilor pentru încheierea regiunilor critice RCU (read-side)
  3) reclamarea spaţiului ocupat de structură (kfree)

Constrângeri RCU

- pentru o regiune critică RCU costrângerile 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 deţine nici o referinţă la un element eliminat -> se poate reclama
- aşteptarea tuturor cititorilor (pasul 2 din secvenţa tipică) înseamnă rularea pe fiecare procesor până la o schimbare de context

Glosar

- live variable: variabilă care poate fi accesată înainte de a fi modificată; valoarea curentă poate afecta execuţia
- dead variable: o variabilă care va fi modificată înainte de a fi accesată; valoarea curentă nu poate avea nici o influenţă asupra execuţiei
- regiune critică: regiune protejată de fluxul extern de execuţie prin mecanisme de sincronizare
- regiune critică read-side: regiune protejată de modificările altor procesoare, dar care permite citire simultană de către mai multe procesoare
- variabilă temporară: o variabilă care este în live doar în cadrul unei regiuni critice (un pointer folosit pentru parcurgerea unei liste)
- variabilă permanentă: o variabilă care este live în afara unei

Glosar (2)

- stare latentă (quiescent state): un punct în cod unde toate variabilele temporare folosite într-o regiune critică sunt dead
  - într-un kernel non-preemptiv o schimbare de context este o stare latentă pentru un procesor
  - într-un kernel preemptiv se folosesc rcu_read_lock, rcu_read_unlock care dezactivează preemptivitatea kernel-ului
- perioadă de grație: intervalul de timp în care toate procesoarele trec prin cel puțin o stare latentă
  - nu este o perioadă fixă; este obținută prin verificarea stărilor latente ale procesoarelor
  - determinarea duratei perioadei de grație este esențială pentru eficiența folosirii RCU

RCU API

void synchronize_kernel (void);

- funcţie care se blochează pentru o întreagă perioadă de graţie
- uşor de folosit
- overhead de schimbare de context important
- nu poate fi apelată din context întrerupere

void call_rcu (struct rcu_head *head,

void (*func) (void *arg),
void *arg);

- **se planifică o funcţie tip callback care va fi rulată la sfârşitul perioadei de graţie**
- **poate fi apelată din context întrerupere sau cu spinlock obţinut**

```
struct rcu_head {
struct list_head list;
void (*func) (void *obj);
void *arg;
};
```

- **structura memorează funcţia callback de apelat la sfârşitul perioadei de graţie**

RCU API (2)

- void rcu_read_lock (void);
- void rcu_read_unlock (void);
  - delimitează o secţiune critică RCU read-side
  - nu generează cod în kernel non-preemptiv
  - în kernel preemptiv dezactivează preemptivitatea

Analogie reader-writer-lock/RCU

- RCU este folosit, de obicei, ca substituent pentru reader-writer locking

RWL

- rwlock_t
- read_lock ()
- read_unlock ()
- write_lock ()
- write_unlock ()
- list_add ()
- list_add_tail ()
- list_del ()
- list_for_each ()

RCU

- spinlock_t
- **rcu_read_lock ()**
- **rcu_read_unlock ()**
- spin_lock ()
- spin_unlock ()
- list_add_rcu ()
- list_add_tail_rcu ()
- list_del_rcu ()
- list_for_each_rcu ()

RWL vs. RCU

```
void delete(long mykey)
{
        struct el *p;
        write_lock(&list_lock);
        p = search(mykey);
        if (p != NULL) {
                list_del(p);
        }
        write_unlock(&list_lock);
        my_free(p);
}

void delete(long mykey)
{
        struct el *p;
        spin_lock(&list_lock);
        p = search(mykey);
        if (p != NULL) {
                list_del_rcu(p);
        }
        spin_unlock(&list_lock);
        call_rcu(&p->rcuhead,
                 (void (*)(void *))my_free, p);
}
```

RWL vs. RCU (2)

```
void insert(long key, long data)
{
    struct el *p;
    p = kmalloc(sizeof(*p), GPF_ATOMIC);
    p->key = key;
    p->data = data;
    write_lock(&list_lock);
    list_add_tail(&(p->list), &head);
    write_unlock(&list_lock);
}

void insert(long key, long data)
{
    struct el *p;
    p = kmalloc(sizeof(*p), GPF_ATOMIC);
    p->key = key;
    p->data = data;
    spin_lock(&list_lock);
    list_add_tail_rcu(&(p->list), &head);
    spin_unlock(&list_lock);
}
```

RWL vs. RCU (3)

```
struct el *search(long mykey)
{
    struct el *p;
    list_for_each(p, &head) {
        if (p->key == mykey) {
            return (p);
        }
    }
    return (NULL);
}

struct el *search(long mykey)
{
    struct el *p;
    list_for_each_rcu(p, &head) {
        if (p->key == mykey) {
            return (p);
        }
     }
    return (NULL);
}
```

RWL vs. RCU (4)

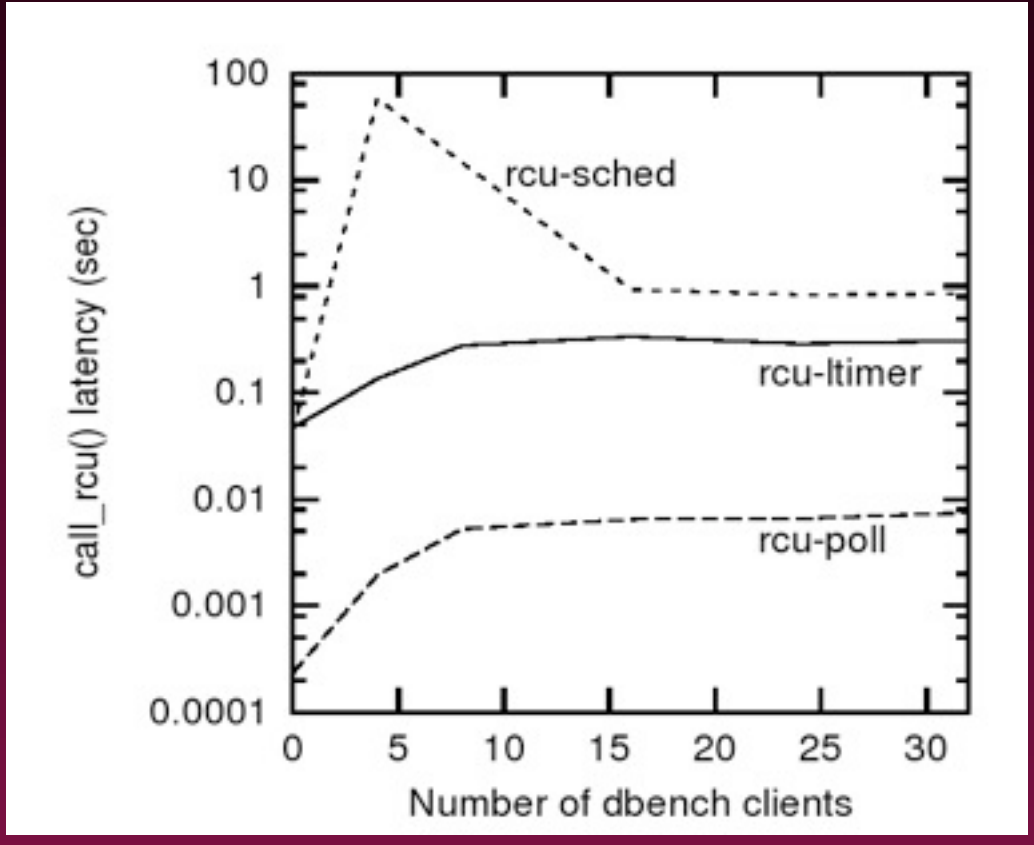
```
/* Read-only search */
struct el *p;
read_lock(&list_lock);
p = search(mykey);
if (p == NULL) {
    /* handle error condition */
} else {
    /* access *p w/out modifying */
}
read_unlock(&list_lock);
```

```
/* Read-only search */
struct el *p;
rcu_read_lock(); /* nop unless CONFIG_PREEMPT */
p = search(mykey);
if (p == NULL) {
    /* handle error condition */
} else {
    /* access *p w/out modifying */
}
rcu_read_unlock(); /* nop unless CONFIG_PREEMPT */
```

Implementare RCU

- performanţa RCU este dată de eficienţa implementării funcţiei call_rcu
- un factor important este latenţa perioadei de graţie
  - perioadă de graţie mică -> RCU eficient, overhead la schimbare de context
  - perioadă de graţie mare -> RCU mai puţin eficient, mai puţin overhead, interactivitate scăzută
- rcu_poll
  - fiecare procesor este forţat să intre într-o stare latentă -> perioade de graţie mici
- rcu_ltimer
  - instrumentează funcţia scheduler_tick -> overhead scăzut, perioade de graţie de până la 100 ms
- rcu_sched
  - foloseşte un mecanism de tip token-passing
  - overhead scăzut dar perioade de graţie de până la 1 minut

Latenţa RCU



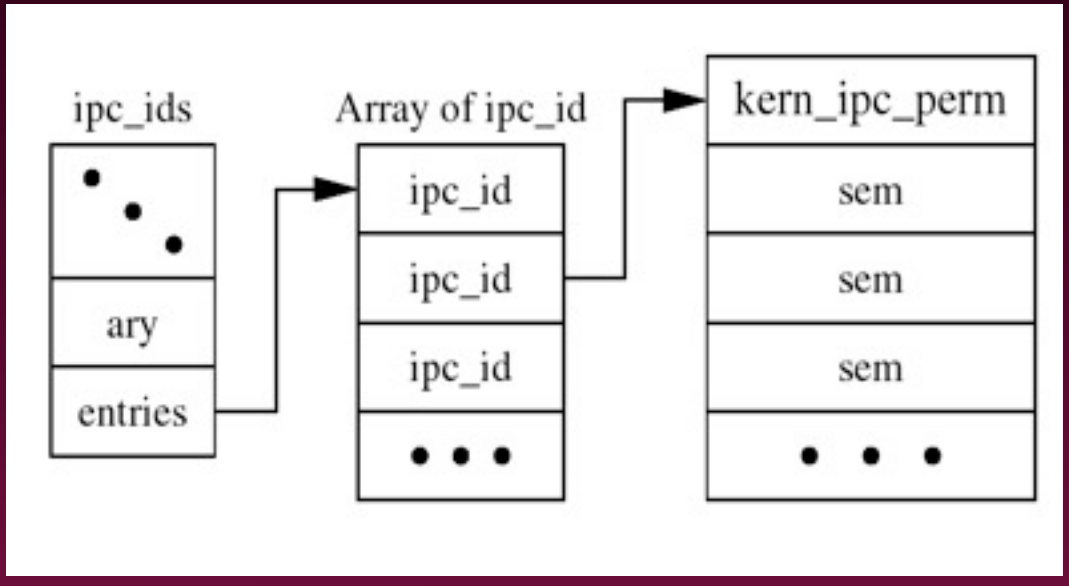
- rcu_poll are latenţa cea mai mică
- se apelează reschedule_task pe fiecare procesor
- overhead important din invocarea scheduler-ului
- cache thrashing
  - folosirea unei singure liste de callback pentru toate procesoarele
  - callback-urile rulează pe un alt procesor
  - soluţie: folosirea de liste per procesor

Performanţa RCU

	CPU Utilization		ms/Iteration	
	Avg	Std	Avg	Std
rcu-ltimer (2.5)	77.52%	0.05%	22.47	0.01
rcu-poll	84.20%	0.13%	22.95	0.03
rcu-sched	81.95%	0.20%	22.46	0.02

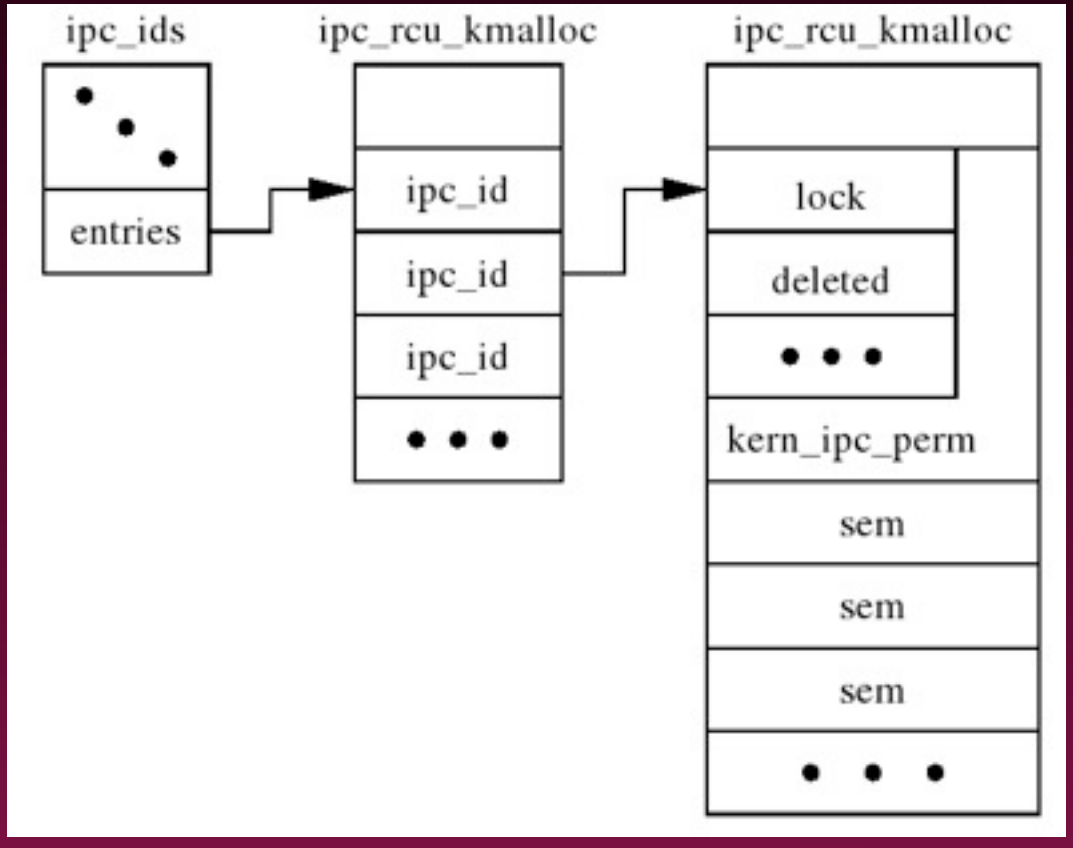
- rcu_ltimer este mai rapid şi foloseşte mai puţin procesorul decât rcu_poll
- deşi rcu_poll are o latenţă excelentă overhead-ul impus de schimbarea de context este considerabil
- se recomandă folosire rcu_ltimer (doar dacă nu este foarte importantă latenţa)

Studiu de caz RCU – System V IPC



- ipc_ids monitorizează semafoarele folosite
- ary -> spinlock
- entry -> pointer la un vector de structuri ipc_id (creşte dinamic)
- fiecare intrare din vector punctează către o structură sem_array (vector de semafoare)
- vectorul de semafoare este creat cu sem_get
- ary menţine un lock al întregii structuri (ipc_ids) -> nu paralelizează

System V IPC - RCU



- RCU permite operaţii complet paralele pe un sistem multiprocesor
- structura ipc_rcu_kmalloc conţine rcu_head
- nu mai există ary ci un lock per structură sem_array

System V IPC – RCU – performanţe

Kernel	Average	Standard Deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

Table 4: DBT1 Database Benchmark Results (TPS)

Kernel	Run 1	Run 2	Avg
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

Table 5: semopbench Microbenchmark Results (seconds)

Resurse utile

- http://www.rdrop.com/users/paulmck/rclock/rcu.FREENIX.2003.06.14.pdf
- http://www.rdrop.com/users/paulmck/RCU/rclock.OLS.2002.07.08a.pdf
- http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf
- http://lse.sourceforge.net/locking/rcupdate.html
- http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html
- http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms