

Curs 3

RCU – Read-Copy-Update

Andrea Arcangeli, Mingming Cao, Paul E. McKenney and
Dipankar Sarma

Operating Systems Practical

16 Octombrie 2013

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

- ▶ sincronizarea este necesară pentru aplicațiile multithreaded și sisteme multiprocesor
- ▶ sincronizarea thread-urilor presupune overhead
- ▶ alte probleme apărute prin folosirea primitivelor de sincronizare
 - ▶ deadlocks (așteptare mutuală blocantă)
 - ▶ inversarea priorității (un thread mai puțin prioritar "ține" lock-ul în fața unui mai prioritar)
 - ▶ convoying (se "strâng" multe thread-uri la un singur lock)
 - ▶ sincronizarea este scumpă (achiziționarea și eliberarea unui spinlock uncontended necesită sute de cicluri de procesor)
- ▶ soluții
 - ▶ evitarea sincronizării (pe cât posibil)
 - ▶ folosirea de sincronizare fără locking
 - ▶ este nevoie de structuri specializate care implică folosirea unei scheme de reclamare a memoriei

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

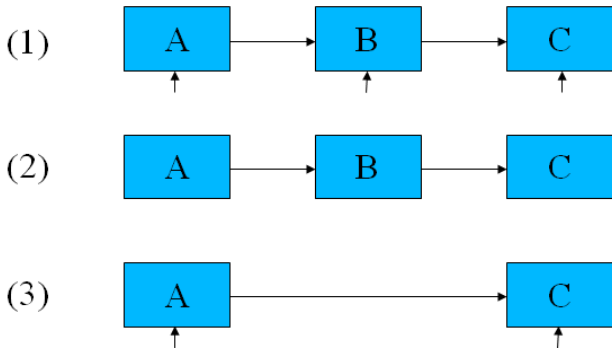
Cuvinte cheie

Resurse

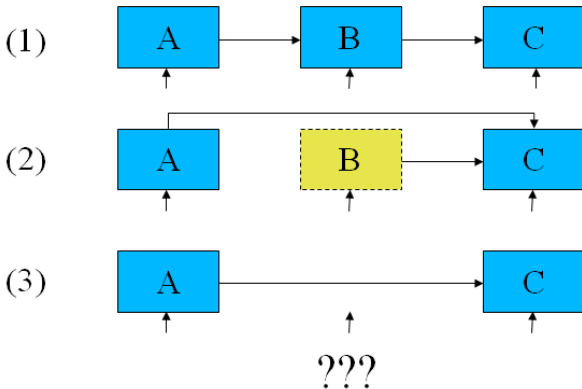
Întrebări

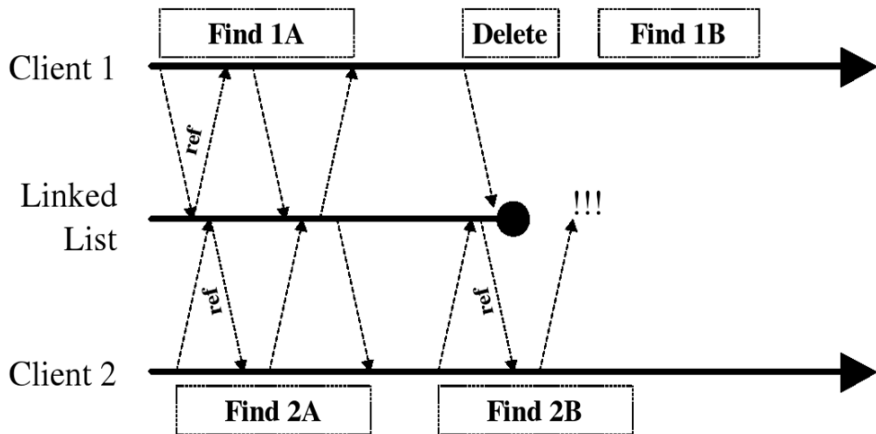
- ▶ read-copy update
- ▶ acces read-only fără locking la structuri modificate concurent
- ▶ nu necesită lock-uri sau instrucțiuni atomice
- ▶ este folosit de obicei cu structurile de date înlănțuite (liste, cozi) traversate unidirecțional cu operații dese de citire (read-mostly)
- ▶ rcu-poll
- ▶ rcu-ltimer

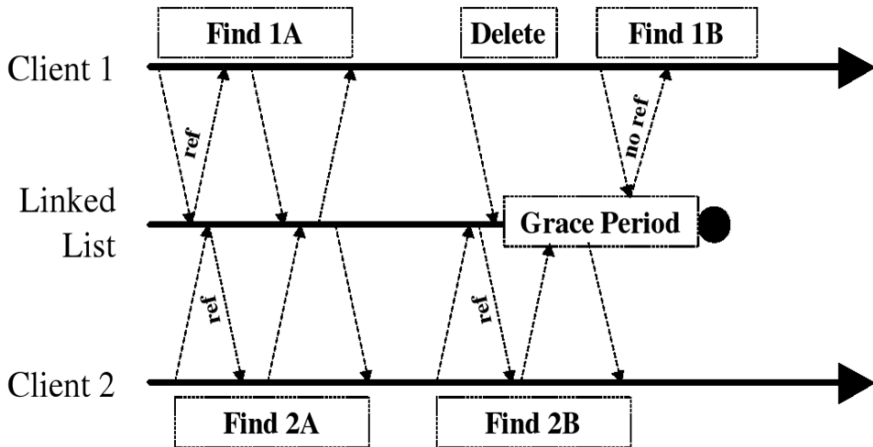
- ▶ parcurgere și ștergere - readers/writers lock
- ▶ (1) – parcurgere listă (read lock held)
- ▶ (2) – eliminare nod B (write lock held)
- ▶ (3) – parcurgere listă după eliminare (read lock held)



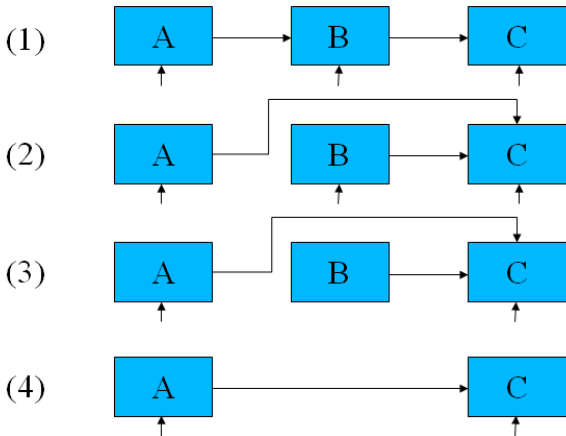
- ▶ de ce?
 - ▶ de multe ori o listă este accesată (parcursă) mult mai des decât este modificată (tabela de rutare)
 - ▶ obținerea unui lock conduce la overhead important
- ▶ probleme
 - ▶ modificarea (update) nu mai este atomică







- ▶ 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



- ▶ actualizările (modificările) se împart în două faze
 - ▶ eliminare (removal)
 - ▶ reclamarea spațiului eliberat (reclamation)
- ▶ 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

- ▶ 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 blochează până la încheiere
 - ▶ se înregistrează o funcție tip callback
- ▶ secvența tipică este
 1. eliminarea pointer-ilor la o structură pentru a nu putea fi referiți de alți cititori
 2. așteptarea tuturor cititorilor pentru încheierea regiunilor critice RCU (read-side)
 3. reclamarea spațiului ocupat de structură (kfree)

- ▶ 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

- ▶ live variable: variabilă care poate fi accesată înainte de a fi modificată; valoarea curentă poate afecta execuția
- ▶ dead variable: o variabilă care va fi modificată înainte de a fi accesată; valoarea curentă nu poate avea nici o influență asupra execuției
- ▶ regiune critică: regiune protejată de fluxul extern de execuție 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 regiuni critice (capătul unei liste)

- ▶ 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


```
void synchronize_kernel (void);
```

- ▶ funcție care se blochează pentru o întreagă perioadă de grație
- ▶ ușor 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 funcție tip callback care va fi rulată la sfârșitul perioadei de grație
- ▶ poate fi apelată din context întrerupere sau cu spinlock obținut

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

- ▶ structura memorează funcția callback de apelat la sfârșitul perioadei de grație

```
void rcu_read_lock (void);
```

```
void rcu_read_unlock (void);
```

- ▶ delimitează o secțiune critică RCU read-side
- ▶ nu generează cod în kernel non-preemptiv
- ▶ în kernel preemptiv dezactivează preemptivitatea

- ▶ 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 ()`

```
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);
}
```

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

```
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);
}
```



```
/* 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 */
```

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

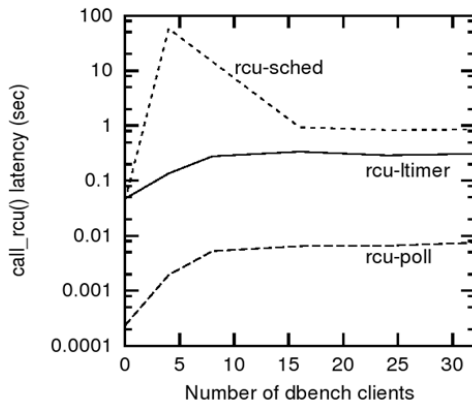
System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

- ▶ performanța RCU este dată de eficiența implementării funcției `call_rcu`
- ▶ un factor important este latența perioadei de grație
 - ▶ perioadă de grație mică -> RCU eficient, overhead la schimbare de context
 - ▶ perioadă de grație mare -> RCU mai puțin eficient, mai puțin overhead, interactivitate scăzută
- ▶ `rcu_poll`
 - ▶ fiecare procesor este forțat să intre într-o stare latentă -> perioade de grație mici
- ▶ `rcu_ltimer`
 - ▶ instrumentează funcția `scheduler_tick` -> overhead scăzut, perioade de grație de până la 100 ms
- ▶ `rcu_sched`
 - ▶ folosește un mecanism de tip token-passing
 - ▶ overhead scăzut dar perioade de grație de până la 1 minut



- ▶ rcu_poll are latența 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
 - ▶ soluție: folosirea de liste per procesor

- ▶ rcu_ltimer este mai rapid și folosește mai puțin procesorul decât rcu_poll
- ▶ deși rcu_poll are o latență excelentă overhead-ul impus de schimbarea de context este considerabil
- ▶ se recomandă folosire rcu_ltimer (doar dacă nu este foarte importantă latența)

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

Locking

RCU

RCU API

Analogie RWL - RCU

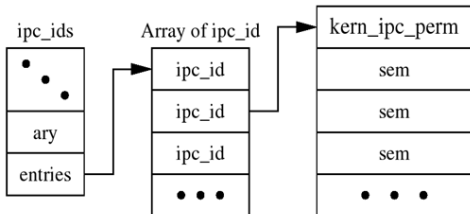
Performanță RCU

System V IPC - RCU

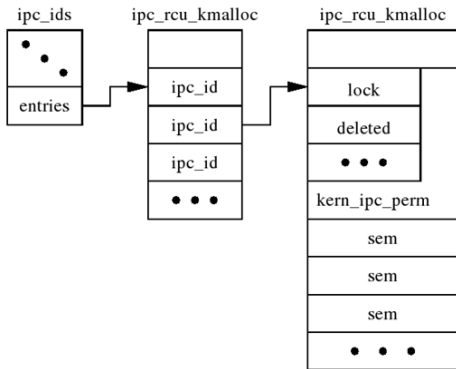
Cuvinte cheie

Resurse

Întrebări



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



- ▶ RCU permite operații complet paralele pe un sistem multiprocesor
- ▶ structura `ipc_rcu_kmalloc` conține `rcu_head`
- ▶ nu mai există ary ci un lock per structură `sem_array`

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)

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

- ▶ sincronizare
- ▶ lockless
- ▶ RWL
- ▶ RCU
- ▶ IPC
- ▶ semafor

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

- ▶ `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`

Locking

RCU

RCU API

Analogie RWL - RCU

Performanță RCU

System V IPC - RCU

Cuvinte cheie

Resurse

Întrebări

?