Assignment 4

- File system integrity in Linux

When should we use spin locks?

- When the critical section is small

Does a spin lock make sense on uniprocessor?

The current process will not get the lock until the process which is holding the lock gets scheduled.

It’s better to yield instead of spinning.

Readers-writer lock

- Multiple readers can concurrently execute in the critical section
- Only a single writer is allowed in the critical section

Readers-writer lock

```
struct node {
    int key, val;
    struct node *next, *prev;
}

void delete (struct node *node) {
    node->next->prev = node->prev;
    node->prev->next = node->next;
    free (node);
}
```

```
int search (struct node *list, int key) {
    int ret = -1;
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        list = list->next;
    }
    return ret;
}
```

Readers-writer lock

```
volatile unsigned lockvar = 10000;
read_acquire:
while (atomic_sub(&lockvar, 1) < 0) {
    atomic_add (&lockvar, 1);
    while (lockvar <= 0);
}
read_release:
atomic_add (&lockvar, 1);
```

```
write_acquire:
while (atomic_sub (&lockvar, 10000) != 0) {
   atomic_add (&lockvar, 10000);
   while (lockvar != 10000);
}
write_release:
atomic_add (&lockvar, 10000);
```

Readers-writer lock

- atomic_sub: atomically subtracts the input value from an input memory location and return the updated value

- atomic_add: atomically adds the input value to an input memory location and return the updated value

Readers-writer lock

```
struct node {
    int key, val;
    struct node *next, *prev;
}

void delete (struct node *node) {
    write_acquire (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    write_release (&lock);
    free (node);
}
```

```
int search (struct node *list, int key) {
    int ret = -1;
    read_acquire (&lock);
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        list = list->next;
    }
    read_release (&lock);
    return ret;
}
```

Problem with reader-writer lock

- Starvation

- Both reader and writer execute an atomic instruction on a shared lock
    - cache line bouncing among readers

Starvation

- writer waits for existing readers to finish
- writer disallows new readers

Big-reader lock

lock0 lock1 lock2 lock3

reader1 : preempt_disable();
acquire(&lock0);

relese(&lock0);
preempt_enable()

- Each CPU has a private lock
- locks on different CPUs belong to different cache lines
- reader acquires the lock corresponding the current CPU
- writer acquires locks of all CPUs

Alignment

0x1001

```
struct spinlock {
    unsigned long val;
} __attribute__ ((aligned(4096)));

struct spinlock lock;
```

&lock is guaranteed to be divisible by 4096.

sizeof (struct spinlock) would be the nearest multiple of 4096.

Big-reader lock

```
struct spinlock brlocks[NUM_CPUS]; /* struct spinlock is cache line size aligned */

read_acquire:
preempt_disable();
spin_lock (&brlocks[cur_cpuid]);

read_release:
spin_unlock (&brlocks[cur_cpuid]);
preempt_enable();
```

Big-reader lock

write_acquire:

```
for (i = 0; i < num_cpus; i++) {
    spin_lock (&brlocks[i]);
}
```

write_release:

```
for (i = 0; i < num_cpus; i++) {
    spin_unlock (&brlocks[i]);
}
```

Big-reader lock

- Pros
  - cache friendly reader lock

- Cons
  - acquisition of writer lock is slow
  - space for the lock variable is proportional to the number of CPUs
  - requires information regarding cache line size

- Can we use big-reader lock in user-mode?

  No

read-copy update

- no reader lock
- can be only used in OS kernel
- only works when the writer is compatible with the lock-free reader

lock free reads

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    spin_lock (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    spin_unlock (&lock);
rcu_ free (node);
}
```

```
int search (struct node *list, int key) {
    int ret = -1;
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        list = list->next;
    }
    return ret;
}
```

lock free reads

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    spin_lock (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    spin_unlock (&lock);
rcu-free (node);
}
```

```
int search (struct node *list, int key) {
    int ret = -1;
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        assert (!list->prev || list->prev->next == list);
        list = list->next;
    }
    return ret;
}
```

RCU lock

- preemption is disabled in the read critical section
- frees are delayed in the write critical section

lock free reads

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    spin_lock (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    spin_unlock (&lock);
    rcu_free (node);
}
```

```
int search (struct node *list, int key) {
    int ret = -1;
    preempt_disable ();
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        list = list->next;
    }
    preempt_enable ();
    return ret;
}
```

How to implement rcu_free?

- rcu_free waits until every CPU has seen a quiescent state
- quiescent state is a point in code which is guaranteed to be outside of read critical section
  - e.g., schedule

wait_for_rcu

- wait_for_rcu waits for quiescent state on all CPUs
- rcu_free calls wait_for_rcu before freeing the memory

1 2 3 4

wait_for_rcu

```
wait_for_rcu () {
    /* cpus_allowed filed controls the scheduling of a thread on specific sets of CPUs */
    cpus_allowed = current->cpus_allowed;
    current->cpus_allowed = all_cpus;
    while (remove_cur_cpu(&current->cpu_allowed) && current->cpu_allowed) {
        schedule();
    }
    current->cpu_allowed = cpus_allowed;
}
```

How to use rcu_free in an interrupt handler?

- Can’t call schedule if the interrupt handler is holding a spin lock
- Instead of calling wait_for_rcu immediately, add the objects to a queue
- Free all objects from queue when it is safe to call wait_for_rcu

Read-copy update for tree

- Can we protect tree_search and tree_update routines using RCU locks?

Read-copy update for tree

- On every update
  - Create a copy of the entire tree
  - update the new copy of the tree
  - update the root of the tree with the new copy of tree in single atomic update
  - call wait_for_rcu before freeing the old tree

Locking

- disable interrupts
- semaphores
- spin lock
- ticket spin lock
- readers-writer lock
- big-reader lock
- rcu lock

TLB invalidation

- How to invalidate TLBs on other cores, when page table entries are modified

- Set an in-memory global flag
  - all CPUs periodically poll the global flag
  - TLB invalidation may have arbitrary latency depending on the polling frequency

Inter processor interrupt (IPI)

- On multiple processor system, a processor can send IPI to another processor
- The corresponding interrupt handler is called when an IPI is received
- A CPU can send IPIs to other cores and wait until all of them invalidate their TLBs