Synchronization II

COMS W4118

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
Semaphore motivation

• **Problem with lock**: ensures mutual exclusion, but no execution order

• **Producer-consumer problem**: need to enforce execution order
  – **Producer**: create resources
  – **Consumer**: use resources
  – **bounded buffer** between them
  – Execution order: producer waits if buffer full, consumer waits if buffer empty
  – E.g., $ cat 1.txt | sort | uniq | wc
Semaphore definition

• A synchronization variable that contains an integer value
  – Can’t access this integer value directly
  – **Must** initialize to some value
    • `sem_init (sem_t *s, int pshared, unsigned int value)`
  – Has two operations to manipulate this integer
    • `sem_wait (or down(), P())`
    • `sem_post (or up(), V())`

```c
int sem_wait(sem_t *s) {
    wait until value of semaphore s
    is greater than 0
    decrement the value of
    semaphore s by 1
}
```

```c
int sem_post(sem_t *s) {
    increment the value of
    semaphore s by 1
    if there are threads waiting, wake
    up one
}
```
Semaphore uses: mutual exclusion

• Mutual exclusion
  – Semaphore as mutex
  – Binary semaphore: X=1

  // initialize to X
  sem_init(&s, 0, X)

  sem_wait(&s);
  // critical section
  sem_post(&s);

• Mutual exclusion with more than one resources
  – Counting semaphore: X>1
  – Initialize to be the number of available resources
Semaphore uses: execution order

- Execution order
  - One thread waits for another
  - What should initial value be?

```c
// thread 0

... // 1st half of computation

sem_post(&s);

// thread 1

sem_wait(&s);

... // 2nd half of computation
```
How to implement semaphores?

Pretty much the same as the mutex implementation we saw last time (note the direct transfer of semaphore):

```
Semaphore { int value = 0; int guard = 0; }
```

```c
P() {
    while (test_and_set(guard))
        ;
    if (value == 0) {
        Add to wait queue;
        Sleep and set guard to 0;
    } else {
        value--;
        guard = 0;
    }
}
```

```c
V() {
    while (test_and_set(guard))
        ;
    if (wait queue not empty) {
        Remove from wait queue;
        Add to ready queue;
    } else {
        value++;
    }
    guard = 0;
}
```
Producer-Consumer (Bounded-Buffer) Problem

- **Bounded buffer**: size N, Access entry 0... N-1, then “wrap around” to 0 again
- **Producer** process writes data to buffer
- **Consumer** process reads data from buffer
- Execution order constraints
  - Producer shouldn’t try to produce if buffer is full
  - Consumer shouldn’t try to consume if buffer is empty
Solving Producer-Consumer problem

• Two semaphores
  – sem_t full; // # of filled slots
  – sem_t empty; // # of empty slots

• What should initial values be?

• Problem: mutual exclusion?

```c
sem_init(&full, 0, X);
sem_init(&empty, 0, Y);

producer() {
    sem_wait(&empty);
    ... // fill a slot
    sem_post(&full);
}

consumer() {
    sem_wait(&full);
    ... // empty a slot
    sem_post(&empty);
}
```
Solving Producer-Consumer problem: final

- Three semaphores
  - `sem_t full;` // # of filled slots
  - `sem_t empty;` // # of empty slots
  - `sem_t mutex;` // mutual exclusion

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```c
producer() {
    sem_wait(&empty);
    sem_wait(&mutex);
    ... // fill a slot
    sem_post(&mutex);
    sem_post(&full);
}
```

```c
consumer() {
    sem_wait(&full);
    sem_wait(&mutex);
    ... // empty a slot
    sem_post(&mutex);
    sem_post(&empty);
}
```
Monitors

• Background: concurrent programming meets object-oriented programming
  – When concurrent programming became a big deal, object-oriented programming too
  – People started to think about ways to make concurrent programming more structured

• Monitor: object with a set of monitor procedures and only one thread may be active (i.e. running one of the monitor procedures) at a time
How to implement monitor?

Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures.

```java
class account {
    int balance;
    public synchronized void deposit() {
        ++balance;
    }
    public synchronized void withdraw() {
        --balance;
    }
};
```

```java
lock(this.m);
++balance;
unlock(this.m);
lock(this.m);
--balance;
unlock(this.m);
```
Condition Variables

- **Condition variable operations**
  - `wait()`: suspends the calling thread and releases the lock. When it resumes, reacquire the lock.
  - `signal()`: resumes one thread waiting in `wait()` if any.
  - `broadcast()`: resumes all threads waiting in `wait()`.

- **A monitor is 1 mutex + N cond var in a class object**
  - In Java, it’s 1 mutex + 1 condition variable
Condition variables vs. semaphores

• Semaphores are **sticky**: they have memory, `sem_post()` will increment the semaphore counter, even if no one has called `sem_wait()`

• Condition variables are not: if no one is waiting for a `signal()`, this `signal()` is not saved

• Despite the difference, *they are as powerful*
  – Easy to implement semaphore with cond var
  – Can implement cond var with semaphore, but tricky
• Reader-writer lock still too slow, even for reading
  – Counter variable access needs expensive atomic instructions and memory barriers
  – Does not scale with large number of CPUs
• Can we just get rid of locks?
  – Sometimes we get lucky when we forget to lock
  – Can we just replicate the luck all the time?
• Read-Copy-Update (RCU):
  – Many readers + one writer can run simultaneously
  – Readers may read old, but consistent data
  – No lock!
struct foo {
    int a;
    int b;
} *global_foo;

// global_foo initialized elsewhere
DEFINE_SPINLOCK(foo_lock);

void set(int x, int y) {
    struct foo *new_foo = kmalloc(...);
    struct foo *old_foo = global_foo;

    // 1) synchronize multiple writers
    spin_lock(&foo_lock);

    // 2) copy old pointer once
    old_foo = global_foo;

    // 3) update data
    new_foo->a = old_foo->a + x;
    new_foo->b = old_foo->b + y;

    // 4) switch pointer
    global_foo = new_foo;

    spin_unlock(&foo_lock);

    // 5) wait a bit for old readers

    // 6) free old struct
    kfree(old_foo);
}

void get(int *p, int *q) {
    struct foo *copy_foo = global_foo;

    // 1) begin reading (no lock)

    // 2) copy pointer once
    copy_foo = global_foo;

    // 3) access data using copy_foo
    *p = copy_foo->a;
    *q = copy_foo->b;

    // 4) end reading (no unlock)
}
struct foo {
    int a;
    int b;
} *global_foo;

// global_foo initialized elsewhere

DEFINE_SPINLOCK(foo_lock);

void set(int x, int y){
    struct foo *new_foo = kmalloc(...);
    struct foo *old_foo;

    // 1) synchronize multiple writers
    spin_lock(&foo_lock);

    // 2) copy old pointer once
    old_foo = rcu_dereference_protected(
        global_foo, lockdep_is_held(&foo_lock));

    // 3) update data
    new_foo->a = old_foo->a + x;
    new_foo->b = old_foo->b + y;

    // 4) switch pointer
    rcu_assign_pointer(global_foo, new_foo);

    spin_unlock(&foo_lock);

    // 5) wait a bit for old readers
    synchronize_rcu();

    // 6) free old struct
    kfree(old_foo);
}

void get(int *p, int *q) {
    struct foo *copy_foo;

    // 1) begin reading (no lock)
    rcu_read_lock();

    // 2) copy pointer once
    copy_foo = rcu_dereference(global_foo);

    // 3) access data using copy_foo
    *p = copy_foo->a;
    *q = copy_foo->b;

    // 4) end reading (no unlock)
    rcu_read_unlock();
}
#1: Using RW Lock

```c
DEFINE_RWLOCK(global_rwlock);

void rcu_read_lock(void) {
    read_lock(&global_rwlock);
}

void rcu_read_unlock(void) {
    read_unlock(&global_rwlock);
}

void synchronize_rcu(void) {
    write_lock(&global_rwlock);
    write_unlock(&global_rwlock);
}
```

#2: “Classic” RCU

```c
void rcu_read_lock(void) {
    prermpt_disable[cpu_id()]++;
}

void rcu_read_unlock(void) {
    prermpt_disable[cpu_id()]--;
}

void synchronize_rcu(void) {
    int cpu;
    for_each_possible_cpu(cpu)
        run_on(cpu);
```
RCU Today

- Linux kernel
  - TREE_RCU: high perf, super complex implementation of grace period handling
    - https://twitter.com/joel_linux/status/1175700053056512000/photo/1
  - Rich set of API

- User-space implementations
  - Ex) C++ standard library

- Use cases beyond reader-writer paradigm

- References:
  - Paul McKenney's RCU home page: http://www2.rdrop.com/users/paulmck/RCU/
  - Linux RCU API as of 2019: https://lwn.net/Articles/777036/