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

// 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; }

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

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);
}
```
Outline

• Semaphores

• Monitors and condition variables
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
- Can think of a monitor as **one big lock** for a set of operations/methods

- In other words, a language implementation of mutexes
How to implement monitor?

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

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

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

• Need wait and wakeup as in semaphores

• Monitor uses **Condition Variables**
  – Conceptually associated with some conditions

• Operations on condition variables:
  – `wait()`: suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true
  – `signal()`: resumes one thread waiting in `wait()` if any. Called when condition becomes true and wants to wake up one waiting thread
  – `broadcast()`: resumes all threads waiting in `wait()`. Called when condition becomes true and wants to wake up all waiting threads
Monitor with condition variables

So, a good way to think about a monitor: 1 mutex + N condition variables 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
  – Exercise: implement one using the other
Producer-consumer with monitors

```c
monitor ProducerConsumer {
    int nfull = 0;
    cond has_empty, has_full;

    producer() {
        if (nfull == N)
            wait (has_empty);
        ... // fill a slot
        ++ nfull;
        signal (has_full);
    }

    consumer() {
        if (nfull == 0)
            wait (has_full);
        ... // empty a slot
        -- nfull;
        signal (has_empty);
    }
}
```

- **Two condition variables**
  - `has_empty`: buffer has at least one empty slot
  - `has_full`: buffer has at least one full slot

- **nfull**: number of filled slots
  - Need to do our own counting for condition variables
Condition variable semantics

• Design question: when `signal()` wakes up a waiting thread, which thread to run inside the monitor, the signaling thread, or the waiting thread?

• **Hoare semantics**: suspends the signaling thread, and immediately transfers control to the woken thread
  – Difficult to implement in practice

• **Mesa semantics**: `signal()` moves a single waiting thread from the blocked state to a runnable state, then the signaling thread continues until it exits the monitor
  – Easy to implement
  – **Problem: race!** Before a woken consumer continues, another consumer comes in and grabs the buffer
Fixing the race in mesa monitors

monitor ProducerConsumer {
    int nfull = 0;
    cond has_empty, has_full;
    producer() {
        while (nfull == N)
            wait (has_empty);
        ... // fill slot
        ++ nfull;
        signal (has_full);
    }
    consumer() {
        while (nfull == 0)
            wait (has_full);
        ... // empty slot
        -- nfull
        signal (has_empty);
    }
};

• The fix: when woken up, a thread must **recheck the condition** it was waiting on

• Most systems use mesa semantics
  – E.g., pthread

• You should use **while**!
Monitor and condition variable in pthread

```cpp
class ProducerConsumer {
  int nfull = 0;
  pthread_mutex_t m;
  pthread_cond_t has_empty, has_full;

public:
  producer() {
    pthread_mutex_lock(&m);
    while (nfull == N)
      pthread_cond_wait(&has_empty, &m);
    // fill slot
    ++nfull;
    pthread_cond_signal(has_full);
    pthread_mutex_unlock(&m);
  }
  ...
};
```

- C/C++ don’t provide monitors; but we can implement monitors using pthread mutex and condition variable.
- For producer-consumer problem, need 1 pthread mutex and 2 pthread condition variables (`pthread_cond_t`).
- Manually lock and unlock mutex for monitor procedures.
- `pthread_cond_wait(cv, m)`: atomically waits on `cv` and releases `m`.