CSCE 311 - Operating Systems
Synchronization w/o Busy Waiting

Qiang Zeng, Ph.D.
Fall 2018
Previous Class...

• Very important concepts
  – Race condition
  – Critical section
  – Mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• This lecture: synchronization w/o busy waiting
Can you give an example of race condition? How to resolve such race condition bugs?

Example 1: counter++;
Example 2: if (account >= 100) account -=100;
Solution: identify *critical sections* in the program, and enforce proper synchronization, e.g., *mutual exclusion*
Implement a busy-waiting lock based on the test-and-set instruction

```c
enter_region() { // lock()
   while(test_and_set(&lock) == 1) ;
}

leave_region() { // unlock()
   lock = 0;
}
```
Outline

• Synchronization w/o busy waiting
  – Kernel-assisted solutions: semaphore
  – Language-assisted solutions: monitor
Semaphore


• Previous solutions rely on busy waiting
• Semaphore internals:
  – A counter to indicate the number of resources
  – A queue for processes waiting for the semaphore
  – A busy-waiting lock (usually based on test-and-set)
• Semaphore needs operating system support
Semaphore – Two main APIs

// Atomic
down(S) { // also called P operation
    if(S.num>0)
        S.num--;
    else
        put the current process in queue;
        the current process blocks;
}

// Atomic
up(S) { // also called V operation
    if(any process is in S’s wait queue)
        pop a process from the queue;
        resume this process;
    else
        S.num++;
}
Implementation of Semaphore in Linux

- https://www.quora.com/How-are-semaphores-implemented

**Down()**
- list_add_tail(&waiter.list, &sem->wait_list); // add the process to the queue
- __set_task_state(task, state); // set TASK_UNINTERRUPTIBLE state
- raw_spin_unlock_irq(&sem->lock);
- timeout = schedule_timeout(timeout); // context switch
- raw_spin_lock_irq(&sem->lock);

**Up()**
- raw_spin_lock_irqsave(&sem->lock, flags);
- list_del(&waiter->list);
- waiter->up = 1;
- wake_up_process(waiter->task);
- raw_spin_unlock_irqrestore(&sem->lock, flags);
Enforcing Mutual exclusion using a binary semaphore (aka. Mutex)

- If a semaphore’s counter value is restricted to 0 and 1, this semaphore is a binary semaphore.
- A binary semaphore can be used to enforce mutual exclusion:
  - The semaphore’s counter is initialize to 1
  - After a process X calls P(), all other process that call P() sleep in the wait queue of the semaphore
  - After X calls V(), one of the processes in the wait queue is waken up
Cons of Semaphore?

- When the competition for the resources is intense, suspending and resuming processes, (i.e., process switching) frequently is expensive.
- Assume a critical section contains $N$ instructions and a context switch takes $M$ instructions.
- If $N < M$, it actually saves CPU cycles to use busy waiting, compared to using Semaphore.
- That is why some systems use a hybrid method:
  - The process first spins for a while, and then
  - Blocks.
Semaphore-based solution to the multi-producer multi-consumer problem

- Two semaphore (\textit{fillCount} and \textit{emptyCount}) are used to sleep/awaken waiting producers and consumers
- A binary semaphore (\textit{S.num} can only be 0 or 1) can be used as a mutex (mutual exclusion); recall \texttt{enter\_region()} and \texttt{exit\_region()}

```plaintext
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        down(mutex);
        putItemIntoBuffer(item);
        up(mutex);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
        down(mutex);
        item = removeItemFromBuffer();
        up(mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

Is mutex required, if there is only one producer and one consumer?
Mutex vs. Semaphore – my email

• Mutex is to indicate whether a privilege is being occupied or not. For example, the privilege of incrementing a counter, and the privilege to accessing "account"

• Semaphore is to indicate the number of resources. For example, in the producer-consumer problem, in the view of a producer, an empty slot is a piece of resource, which is waited for by a producer but generated by a consumer.

• A binary semaphore (whose internal counter only be either 0 or 1) can be used as a mutex, but that is just a special use of semaphore. You regard the privilege as one piece of resource

• All the mutexs we covered use busy-waiting (so they are called spinlocks), while semaphore turns a process to sleep (or, be blocked), so it saves CPU cycles in some sense (why semaphore doesn't always save CPU time?)
# Semaphore and pthread_mutex in Linux

<table>
<thead>
<tr>
<th>function</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_open()</code></td>
<td>Create or connect to a named semaphore (and increment the reference count), which can be conveniently shared inter-process</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Remove the named semaphore once its reference count = 0</td>
</tr>
<tr>
<td><code>sem_close()</code></td>
<td>Decrement the reference count (<code>exit()</code> does this automatically)</td>
</tr>
<tr>
<td><code>sem_init()</code></td>
<td>Create an unnamed semaphore (usually a global variable)</td>
</tr>
<tr>
<td><code>sem_destroy()</code></td>
<td>Destroy an unnamed semaphore</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>P operation</td>
</tr>
<tr>
<td><code>sem_post()</code></td>
<td>V operation</td>
</tr>
<tr>
<td><code>pthread_mutex_init()</code></td>
<td>PTHREAD_PROCESS_SHARED: process-shared mutex  Other APIs: *_destroy(), *_lock(), *_trylock(), *_unlock()</td>
</tr>
</tbody>
</table>
Monitor

- A programming language construct that supports controlled access to shared data; it encapsulates
  - Shared data
  - Procedures that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures (only one thread can execute any of the procedures at a time)

- A Monitor has a mutex lock and a queue
  - Processes has to acquire the lock before invoking a procedure in the monitor
  - If the lock has been acquired by a process, other requesting processes are put in the queue
Condition Variables/Monitors

Michael Swift

Two Classes of Synchronization Problems

• Uniform resource usage with simple scheduling constraints
  – No other variables needed to express relationships
  – Use one semaphore for every constraint
  – Examples: thread join and producer/consumer

• Complex patterns of resource usage
  – Cannot capture relationships with only semaphores
  – Need extra state variables to record information
  – Use semaphores such that
    • One is for mutual exclusion around state variables
    • One for each class of waiting

Always try to cast problems into first, easier type

Monitors

• A programming language construct that supports controlled access to shared data
  – Synchronization code added by compiler, enforced at runtime
  – why does this help?

• Monitor is a software module that encapsulates:
  – shared data structures
  – procedures that operate on the shared data
  – synchronization between concurrent processes that invoke those procedures

Monitor protects the data from unstructured access
  – guarantees only access data through procedures, hence in legitimate ways

(1) Shared data can only be accessed through procedures
(2) Only one thread can enter the monitor to invoke procedures at a time
(1) + (2) => It is guaranteed that all threads access the shared data in a mutual exclusive way
Monitor is easy to use and fool-proof, but less flexible than semaphores

Graph credit: Michael Swift @ wisc univ
Monitor in Java

• “Synchronized” non-static methods are to implement a Monitor object
  – A thread acquires the lock of an object to execute a synchronized method
  – If another thread tries to execute any of the synchronized methods, the thread blocks

• “synchronized” static method is to implement a Monitor class

• “synchronized block” is more flexible
Conditional variable (CV)

- If we revisit the previous faulty solution to the producer-consumer problem, we can notice it is natural and easy to code, while the solution based on semaphore is subtle.
- Is there an easy and correct solution?

<table>
<thead>
<tr>
<th>Function</th>
<th>Semantics</th>
</tr>
</thead>
</table>
| pthread_cond_wait (cond, mutex) | **Precondition: the calling process must own the mutex before calling this function.**  
It atomically releases the mutex and waits on a condition variable.  
When it is unblocked by `pthread_cond_signal(cond)`,  
the calling process contends for the mutex as if it had called `pthread_mutex_lock()` |
A solution based on mutex and CV

typedef struct {
    char buf[BSIZE];
    int occupied, in, out;
    pthread_mutex_t mutex;
    pthread_cond_t slot, item;
} buffer_t;
buffer_t b;

void producer(char c)
{
    pthread_mutex_lock(&b->mutex);
    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->slot, &b->mutex);
    b->buf[b->in] = c;
    b->in = (b->in+1) % BSIZE;
    b->occupied++;
    pthread_cond_signal(&b->item);
    pthread_mutex_unlock(&b->mutex);
}

char consumer()
{
    char c;
    pthread_mutex_lock(&b->mutex);
    while (b->occupied <= 0)
        pthread_cond_wait(&b->item, &b->mutex);
    i = b->buf[b->out];
    b->out %=(b->out + 1) % BSIZE;
    b->occupied--;
    pthread_cond_signal(&b->slot);
    pthread_mutex_unlock(&b->mutex);
    return(c);
}

This is much easier to code and understand!
Can we do even better? We’ll see.
What if a process has acquired a lock but cannot progress because some condition is not met?

• Consider the following problem:
  – A process acquired the lock and did a lot of computation, but then it finds that it cannot make progress as some condition (e.g., account >= 100) is not met

• Does the thread have to give up all the computation it has done? (In some cases, it may even have to rollback)

• Any better solution?
Condition Variable (CV)

- Note that a monitor contains a wait queue (waiting for the lock), while a CV has its own wait queue

- Three operations for CVs
  - Wait(c): does two things simultaneously
    - The calling process is put into the wait queue of the CV
    - Release the monitor lock (so somebody else can get in)
  - Signal(c):
    - Move a process $p$, if any, from the CV’s queue to the monitor’s queue
    - Once $p$ gets the lock, it resumes execution from where it was suspended (usually the line after wait())
  - Broadcast(c)
    - Wake up all processes from the CV’s wait queue to the monitor’s queue
## Condition Variable

<table>
<thead>
<tr>
<th>Operations</th>
<th>Posix</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(c)</td>
<td><code>pthread_cond_wait (cond, mutex)</code></td>
<td><code>wait()</code></td>
</tr>
<tr>
<td>Signal(c)</td>
<td><code>pthread_cond_signal(cond)</code></td>
<td><code>notify()</code></td>
</tr>
<tr>
<td>Broadcast(c)</td>
<td><code>pthread_cond_broadcast(cond)</code></td>
<td><code>notifyAll()</code></td>
</tr>
</tbody>
</table>
How to use a Condition Variable

1. Always hold the lock while performing CV operations
2. Always put the wait operation in a loop
   - In order to check whether the condition is true
   - Note the the signal(c) only hints the conditions changes; it does not imply that the condition is definitely true. Moreover, after signaling, some other processes may affect the condition
“Always hold the lock while performing CV operations”, why?

• The correct use of “Wait(c)” requires that the caller holds the lock, since the semantics of this call implies the release of the lock.

• If the caller of “signal(c)” does not hold the lock, there may be race conditions:
  – Consider w.1 -> s.1 -> s.2 -> w.2 => signal lost

Strange_withdraw(x) {
   ...
   pthread_mutex_lock(&m);
   while(account < x) // w.1
      pthread_cond_wait(&c, &m); // w.2
   account -= x;
   pthread_mutex_unlock(&m);
}

Deposit(y){
   // pthread_mutex_lock(&m);
   account += y; // s.1
   pthread_cond_signal(&c); // s.2
   // pthread_mutex_unlock(&m);
}
Relations between Condition Variable & Monitor

• A Monitor may contain zero or more CVs
  – Very often, procedures in Monitor rely on CVs to implement complex synchronization
  – Recall that a CV has to be used with a lock; a Monitor can provide the lock, so you do not have to explicitly use a lock for employing a CV in a Monitor

• The use of CVs is not limited to Monitors
  – E.g., Pthread library provides CVs but not Monitors
Critical points

- A common misunderstanding: a condition variable is intrinsically associated with some condition, which is wrong.
- The signal(c) only indicates an event occurs, while wait(c) is to wait for the signaling of the occurrence of an event.
- But the wait(c) operation is indeed associated with some loop which checks whether the expected condition is true.
Examples of Monitor and Condition variable

- As a library (pthreads)
  ```c
  pthread_mutex_t mu;
  pthread_cond_t co;
  boolean ready;
  void foo() {
    pthread_mutex_lock(&mu);
    while(!ready)
      pthread_cond_wait(&co, &mu);
    ... 
    ready = TRUE;
    pthread_cond_signal(&co);  // unlock and signal
    pthread_mutex_unlock(&mu);
  }
  ```

- As a language (Java)
  ```java
  synchronized withdraw(int amount) {
    while (balance < amount) {
      wait();
      balance -= amount;
      if (balance == 0) {
        notify();
      }
  }
  ```
Condition variable VS Semaphore

• A Semaphore has a counter and a wait queue, while a Condition Variable only has a wait queue
  – Plus, you need to initialize the counter when using a Semaphore, and the initialization value means the number of resources. A Condition Variable has no notion of “the number of resources”

• Condition Variables allow broadcast() operation, while Semaphores do not
Readers and writers problem

Reads are writes are mutual exclusive
Concurrent reads are allowed; concurrent writes are not.

```c
// mutex for rc
semaphore rc_mutex = 1;
// mutex for db
semaphore db = 1;

void writer(void) {
    while(true) {
        d = get_data();
        down(&db);
        write_data(d);
        up(&db)
    }
}

void reader(void) {
    while(true) {
        down(&rc_mutex);
        ++rc;
        if(rc == 1) down(&db);
        up(&rc_mutex);
        d = read_data();
        down(&rc_mutex);
        --rc;
        if(rc == 0) up(&db);
        up(&rc_mutex);
    }
}
```
A lock-free solution to single-producer single-consumer problem

```c
volatile unsigned int produceCount, consumeCount;
char buffer[BUFFER_SIZE];

void producer(void) {
    while (1) {
        while (produceCount - consumeCount == BUFFER_SIZE)
            sched_yield(); // buffer is full

        buffer[produceCount % BUFFER_SIZE] = produceChar();
        ++produceCount;
    }
}

void consumer(void) {
    while (1) {
        while (produceCount - consumeCount == 0)
            sched_yield(); // buffer is empty

        consumeChar(buffer[consumeCount % BUFFER_SIZE]);
        ++consumeCount;
    }
}
```

The lock still has its advantage: it can be used to address general concurrent computing problems, while there are only limited types of lock-free data structures.
Big picture of synchronization primitives

Sync. Primitives

Busy-waiting
- Software solutions (Dekker’, Bakery, etc.)
- Hardware-assisted solutions (based on atomic read-modify-write instructions)

Semaphore: it contains an internal counter indicating the number of resources.
- Binary Semaphore is a special semaphore, whose counter value can only be 0 or 1; it can be used as a mutex

Blocking
- Monitor: a high-level synchronization construct; shared data + procedures + lock

Condition variable: guarded by a lock to wait on some event
Take away…

• Semaphores and binary semaphores
• Monitors and Conditional variables