CS 318 Principles of Operating Systems

Fall 2017

Lecture 7: Semaphores and Monitors

Ryan Huang
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have limited semantics
  - Just provide mutual exclusion
- Instead, we want synchronization mechanisms that
  - Block waiters, leave interrupts enabled in critical sections
  - Provide semantics beyond mutual exclusion
- Look at two common high-level mechanisms
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables
- Use them to solve common synchronization problems
Semaphores

• **An abstract data type** to provide mutual exclusion to critical sections
  - Described by Dijkstra in the “THE” system in 1968
• **Semaphores can also be used as atomic counters**
  - More later
• **Semaphores** are “integers” that support two operations:
  - ```Semaphore::P()``` : decrement, block until semaphore is open
    - after the Dutch word “Proberen” (to try), also `Wait()`
  - ```Semaphore::V()``` : increment, allow another thread to enter
    - after the Dutch word “Verhogen” (increment), also `Signal()`
  - That's it! No other operations – not even just reading its value
• **Semaphore safety property**: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes

- **When** $P()$ **is called by a thread:**
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue

- **Then** $V()$ **opens the semaphore:**
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, $V()$ has “history” (c.f., condition vars later)
    - This “history” is a counter
Semaphore Types

• Semaphores come in two types

  • **Mutex semaphore (or binary semaphore)**
    - Represents single access to a resource
    - Guarantees mutual exclusion to a critical section

  • **Counting semaphore (or general semaphore)**
    - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
    - Multiple threads can pass the semaphore
    - Number of threads determined by the semaphore “count”
      • mutex has count = 1, counting has count = N
Using Semaphores

• Use is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

Threads block critical section

It is undefined which thread runs after a signal

```c
wait(S);
balance = get_balance(account);
balance = balance - amount;
wait(S);
put_balance(account, balance);
signal(S);
```

```c
wait(S);
wait(S);
put_balance(account, balance);
signal(S);
...
signal(S);
...
signal(S);
```
Semaphores in Pintos

```
void sema_down(struct semaphore *sema) {
    enum intr_level old_level;
    old_level = intr_disable();
    while (sema->value == 0) {
        list_push_back(&sema->waiters, &thread_current()->elem);
        thread_block();
    }
    sema->value--;
    intr_set_level(old_level);
}

void sema_up(struct semaphore *sema) {
    enum intr_level old_level;
    old_level = intr_disable();
    if (!list_empty(&sema->waiters))
        thread_unblock(list_entry(list_pop_front(&sema->waiters), struct thread, elem));
    sema->value++;
    intr_set_level(old_level);
}
```

- **To reference current thread:** thread_current()

- **thread_block() assumes interrupts are disabled**
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?
Interrupts Disabled During Context Switch

```
thread_yield() {
    Disable interrupts;
    add current thread to ready_list;
    schedule(); // context switch
    Enable interrupts;
}
```

```
sema_down()
    Disable interrupts;
    while(value == 0) {
        add current thread to waiters;
        thread_block();
    }
    value--;
    Enable interrupts;
}
```

```
[thread_yield]
    Disable interrupts;
    add current thread to ready_list;
    schedule();

[sema_down]
    Disable interrupts;
    while(value == 0) {
        add current thread to waiters;
        thread_block();
    }
    value--;
    Enable interrupts;
```

```
[thread_yield]
    (Returns from schedule())
    Enable interrupts;

[thread_yield]
    (Returns from schedule())
    Enable interrupts;
```
Using Semaphores

• We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account

• Now we’re going to use semaphores to look at more interesting examples
  - Readers/Writers
  - Bounded Buffers
Readers/Writers Problem

• Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer
    • Let \( r \) be the number of readers, \( w \) be the number of writers
    • Safety: \((r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \Rightarrow (w = 0))\)

• How can we use semaphores to control access to the object to implement this protocol?

• Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex);    // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);    // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount
}
• \texttt{w\_or\_r} provides mutex between readers and writers
  - writer wait/signal, reader wait/signal when \texttt{readcount} goes from 0 to 1 or from 1 to 0.

• If a writer is writing, where will readers be waiting?

• Once a writer exits, all readers can fall through
  - Which reader gets to go first?
  - Is it guaranteed that all readers will fall through?

• If readers and writers are waiting, and a writer exits, \textbf{who goes first}?

• Why do readers use \texttt{mutex}?

• Why don't writers use \texttt{mutex}?

• What if the signal is above “if (readcount == 1)”?
Bounded Buffer

- **Problem:** There is a set of resource buffers shared by producer and consumer threads
  - **Producer** inserts resources into the buffer set
    - Output, disk blocks, memory pages, processes, etc.
  - **Consumer** removes resources from the buffer set
    - Whatever is generated by the producer

- **Producer and consumer execute at different rates**
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff

- **Safety:**
  - Sequence of consumed values is prefix of sequence of produced values
  - If $nc$ is number consumed, $np$ number produced, and $N$ the size of the buffer, then $0 \leq np - nc \leq N$
Bounded Buffer (2)

• $0 \leq np - nc \leq N$ and $0 \leq (nc - np) + N \leq N$

• Use three semaphores:
  - **empty** – count of empty buffers
    • Counting semaphore
    • empty = $(nc - np) + N$
  - **full** – count of full buffers
    • Counting semaphore
    • $np - nc = full$
  - **mutex** – mutual exclusion to shared set of buffers
    • Binary semaphore
Bounded Buffer (3)

producer {
    while (1) {
        Produce new resource;
        wait(empty); // wait for empty buffer
        wait(mutex); // lock buffer list
        Add resource to an empty buffer;
        signal(mutex); // unlock buffer list
        signal(full); // note a full buffer
    }
}

c consumer {
    while (1) {
        wait(full); // wait for a full buffer
        wait(mutex); // lock buffer list
        Remove resource from a full buffer;
        signal(mutex); // unlock buffer list
        signal(empty); // note an empty buffer
        Consume resource;
    }
}

Semaphore mutex = 1; // mutual exclusion to shared set of buffers
Semaphore empty = N; // count of empty buffers (all empty to start)
Semaphore full = 0; // count of full buffers (none full to start)
Bounded Buffer (4)

• Why need the mutex at all?
• Where are the critical sections?
• What has to hold for deadlock to occur?
  - empty = 0 and full = 0
  - \((nc - np) + N = 0\) and \(np - nc = 0\)
  - \(N = 0\)
• What happens if operations on mutex and full/empty are switched around?
  - The pattern of signal/wait on full/empty is a common construct often called an interlock
• Producer-Consumer and Bounded Buffer are classic synchronization problems
Semaphore Questions

- Are there any problems that **can be solved** with counting semaphores that **cannot be solved** with mutex semaphores?

- Does it matter **which thread is unblocked** by a signal operation?
  - Hint: consider the following three threads sharing a semaphore **mutex** that is initially 1:

```
while (1) {
    wait(mutex);
    // in critical
    // section
    signal(mutex);
}
```

```
while (1) {
    wait(mutex);
    // in critical
    // section
    signal(mutex);
}
```

```
while (1) {
    wait(mutex);
    // in critical
    // section
    signal(mutex);
}
```
Semaphore Summary

• Semaphores can be used to solve any of the traditional synchronization problems

• However, they have some drawbacks
  - They are essentially shared global variables
    • Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
    • Note that I had to use comments in the code to distinguish
  - No control or guarantee of proper usage

• Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support
Monitors

• A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?

• A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures

• A monitor protects its data from unstructured access

• It guarantees that threads accessing its data through its procedures interact only in legitimate ways
A monitor guarantees mutual exclusion

- Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
- If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
  - So the monitor has to have a wait queue…
- If a thread within a monitor blocks, another one can enter

What are the implications in terms of parallelism in a monitor?
Hey, that was easy!

But what if a thread wants to wait inside the monitor?
- Such as “mutex(empty)” by reader in bounded buffer?

Account Example

Monitor account {
    double balance;

    double withdraw(amount) {
        balance = balance - amount;
        return balance;
    }
}

withdraw(amount) balance = balance - amount;
withdraw(amount) balance = balance - amount
withdraw(amount) return balance (and exit)
withdraw(amount)
balance = balance - amount return balance;
balance = balance - amount; return balance;

Threads block waiting to get into monitor

When first thread exits, another can enter. Which one is undefined.
• **A monitor invariant** is a **safety property** associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.

• **A condition variable** is associated with a **condition** needed for a thread to make progress once it is in the monitor.
  - alternative: busy waiting, bad

Monitor M {
  ...
  monitored variables
  Condition c;

  void enterMonitor (...) {
    if (extra property not true) wait(c); waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c); brings in one thread waiting on condition
  }
}
Condition Variables

• Condition variables support three operations:
  - **Wait** – release monitor lock, wait for C/V to be signaled
    • So condition variables have wait queues, too
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads

• Condition variables are not boolean objects
  - if (condition_variable) then ... does not make sense
  - if (num_resources == 0) then wait(resources_available) does
  - An example will make this more clear
Monitor bounded_buffer {
    Resource buffer[N];
    // Variables for indexing buffer
    // monitor invariant involves these vars
    Condition not_full; // space in buffer
    Condition not_empty; // value in buffer

    void put_resource (Resource R) {
        while (buffer array is full)
            wait(not_full);
        Add R to buffer array;
        signal(not_empty);
    }

    Resource get_resource() {
        while (buffer array is empty)
            wait(not_empty);
        Get resource R from buffer array;
        signal(not_full);
        return R;
    }
} // end monitor

- What happens if no threads are waiting when signal is called?
Monitor `bounded_buffer` {

    Condition not_full;
    ...other variables...
    Condition not_empty;

    void put_resource() {
        ...wait(not_full)...
        ...signal(not_empty)...
    }

    Resource get_resource() {
        ...
    }
}
Condition Vars != Semaphores

- **Condition variables != semaphores**
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement the other

- **Access to the monitor is controlled by a lock**
  - wait() blocks the calling thread, and gives up the lock
    - To call wait, the thread has to be in the monitor (hence has lock)
    - Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history
There are two flavors of monitors that differ in the scheduling semantics of `signal()`

- **Hoare** monitors (original)
  - `signal()` immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore *monitor invariants* before signaling

- **Mesa** monitors (Mesa, Java)
  - `signal()` places a waiter on the ready queue, *but signaler continues inside monitor*
  - Condition is not necessarily true when waiter runs again
    - Returning from `wait()` is only a hint that something changed
    - Must recheck conditional case
Hoare vs. Mesa Monitors

• Hoare
  
  ```
  if (empty)
    wait(condition);
  ```

• Mesa
  
  ```
  while (empty)
    wait(condition);
  ```

• Tradeoffs
  - Mesa monitors easier to use, more efficient
    • Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    • Easier to reason about the program
Monitor Readers and Writers

Using Mesa monitor semantics.

- Will have four methods: **StartRead**, **StartWrite**, **EndRead** and **EndWrite**

- Monitored data: \( nr \) (number of readers) and \( nw \) (number of writers) with the monitor invariant

\[
(nr \geq 0) \land (0 \leq nw \leq 1) \land ((nr > 0) \Rightarrow (nw = 0))
\]

- Two conditions:
  - **canRead**: \( nw = 0 \)
  - **canWrite**: \( (nr = 0) \land (nw = 0) \)
• Write with just wait()
  - Will be safe, maybe not live – why?

```c
Monitor RW {
    int nr = 0, nw = 0;
    Condition canRead, canWrite;

    void StartRead () {
        while (nw != 0) do wait(canRead);
        nr++;
    }

    void EndRead () {
        nr--;
    }

    void StartWrite () {
        while (nr != 0 || nw != 0) do wait(canWrite);
        nw++;
    }

    void EndWrite () {
        nw--;
    }
} // end monitor
```
Monitor Readers and Writers

- **add** `signal()` and `broadcast()`

```c
Monitor RW {
    int nr = 0, nw = 0;
    Condition canRead, canWrite;

    void StartRead () {
        while (nw != 0) do wait(canRead);
        nr++;
    }

    void EndRead () {
        nr--;
        if (nr == 0) signal(canWrite);
    }

    void StartWrite () {
        while (nr != 0 || nw != 0) do wait(canWrite);
        nw++;
    }

    void EndWrite () {
        nw--;
        broadcast(canRead);
        signal(canWrite);
    }

} // end monitor
```

Can we put a signal here?
Monitor Readers and Writers

• Is there any priority between readers and writers?

• What if you wanted to ensure that a waiting writer would have priority over new readers?
Condition Vars & Locks

• C/Vs are also used without monitors in conjunction with locks
  
  - void cond_init (cond_t *, ...);
  - void cond_wait (cond_t *c, mutex_t *m);
    • Atomically unlock m and sleep until c signaled
    • Then re-acquire m and resume executing
  - void cond_signal (cond_t *c);
  - void cond_broadcast (cond_t *c);
    • Wake one/all threads waiting on c
Condition Vars & Locks

- C/Vs are also used without monitors in conjunction with locks.

- A monitor \( \approx \) a module whose state includes a C/V and a lock
  - Difference is syntactic; with monitors, compiler adds the code.

- It is “just as if” each procedure in the module calls acquire() on entry and release() on exit
  - But can be done anywhere in procedure, at finer granularity.

- With condition variables, the module methods may wait and signal on independent conditions.
Condition Vars & Locks

• Why must `cond_wait` both `release` `mutex_t` & `sleep`?
  - `void cond_wait(cond_t *c, mutex_t *m);`

• Why not separate mutexes and condition variables?

```c
while (count == BUFFER_SIZE) {
    mutex_unlock(&mutex);
    cond_wait(&not_full);
    mutex_lock(&mutex);
}
```
Condition Vars & Locks

• **Why must** `cond_wait` **both release** `mutex_t` & **sleep**?
  
  ```c
  void cond_wait(cond_t *c, mutex_t *m);
  ```

• **Why not separate** mutexes and condition variables?

  **Producer**
  ```c
  while (count == BUFFER_SIZE) {
      mutex_unlock(&mutex);
      cond_wait(&not_full);
      mutex_lock(&mutex);
  }
  ```

  **Consumer**
  ```c
  mutex_lock(&mutex);
  ... count--;
  cond_signal(&not_full);
  ```
Using Cond Vars & Locks

- Alternation of two threads (ping-pong)

- Each executes the following:

```c
Lock lock;
Condition cond;
void ping_pong () {
    acquire(lock);
    while (1) {
        printf("ping or pong\n");
        signal(cond, lock);
        wait(cond, lock);
    }
    release(lock);
}
```

Must acquire lock before you can wait (similar to needing interrupts disabled to call `thread_block` in Pintos)

Wait atomically releases lock and blocks until `signal()`

Wait atomically acquires lock before it returns
Monitors and Java

• A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables

• Every object is/has a monitor
  - At most one thread can be inside an object’s monitor
  - A thread enters an object’s monitor by
    • Executing a method declared “synchronized”
      • Can mix synchronized/unsynchronized methods in same class
    • Executing the body of a “synchronized” statement
      • Supports finer-grained locking than an entire procedure
      • Identical to the Modula-2 “LOCK (m) DO” construct
  - The compiler generates code to acquire the object’s lock at the start of the method and release it just before returning
    • The lock itself is implicit, programmers do not worry about it
Monitors and Java

• Every object can be treated as a condition variable
  - Half of Object’s methods are for synchronization!

• Take a look at the Java Object class:
  - Object.wait(*) is Condition::wait()
  - Object.notify() is Condition::signal()
  - Object.notifyAll() is Condition::broadcast()
Summary

• **Semaphores**
  - `wait()`/`signal()` implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use

• **Monitors**
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - Only one thread can execute within a monitor at a time
  - Relies upon high-level language support

• **Condition variables**
  - Used by threads as a synchronization point to wait for events
  - Inside monitors, or outside with locks
Next Time…

• Read Chapters 7