Lecture 6: Synchronization (II) – Semaphores and Monitors

Ding Yuan
Higher-Level Synchronization

• We looked at using locks to provide mutual exclusion
• Locks work, but they have some drawbacks when critical regions are long
  – Spinlocks – inefficient
  – Disabling interrupts – can miss or delay important events
• Instead, we want synchronization mechanisms that
  – Block waiters
  – Leave interrupts enabled inside the critical section
• 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

• **Semaphores are an abstract data type that provide mutual exclusion to critical region**
• Semaphores can also be used as atomic counters
  – More later
• Semaphores are integers that support two operations:
  – wait(semaphore): decrement, block until semaphore is open
    • Also P(), after the Dutch word for test, or down()
  – signal(semaphore): increment, allow another thread to enter
    • Also V() after the Dutch word for increment, or up()
  – That's it! No other operations – not even just reading its value – exist
• P and V are probably the most unintuitive names you encounter in this course
  – and you have Edsger W. Dijkstra to thank to
• 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/threads
- When \( P() \) is called by a thread:
  - If semaphore is open (> 0), 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
    - \( \text{What if multiple threads are waiting on the queue?} \)
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - \( \text{In other words, } V() \text{ has “history” (c.f., condition vars later)} \)
    - 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 (P)**
  – 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) {
    P(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    V(S);
    return balance;
}
```

It is undefined which thread runs after a signal
Semaphores in OS161

```
P(sem) {
    Disable interrupts;
    while (sem->count == 0) {
        thread_sleep(sem); /* current thread
                            will sleep on this sem */
    }
    sem->count--;
    Enable interrupts;
}

V(sem) {
    Disable interrupts;
    sem->count++;
    thread_wakeup(sem); /* this will wake
                  up all the threads waiting on this
                  sem. Why wake up all threads? */
    Enable interrupts;
}
```

- `thread_sleep()` assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?

- **What happens if “while (sem->count ==0)” is an “if (sem->count != 0)”?**
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: \( (r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \implies (w = 0)) \)

- How can we use semaphores to control access to the object to implement this protocol?
First attempt: one mutex semaphore

```c
// exclusive writer or reader
Semaphore w_or_r = 1;

reader {
    P(w_or_r); // lock out writers
    read;
    V(w_or_r); // up for grabs
}

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}
```

- Does it work?
- Why?
- Which condition is satisfied and which is not?
  - (#r ≥ 0)
  - (0 ≤ #w ≤ 1)
  - ((#r > 0) ⇒ (#w = 0))
Second attempt: add a counter

```c
int readcount = 0; // record #readers
Semaphore w_or_r = 1; // mutex semaphore

reader {
    readcount++;
    if (readcount == 1){
        P(w_or_r); // lock out writers
    }
    read;
    readcount--;
    if (readcount == 0){
        V(w_or_r); // up for grabs
    }
}

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}
```

• Does it work?

• readcount is a shared variable, who protects it?

Thread 1:
```
reader {
    readcount++;
    if (readcount == 1){
        P(w_or_r);
    }
    read;
    readcount--;
    if (readcount == 0){
        V(w_or_r);
    }
}
```

Thread 2:
```
reader {
    readcount++;
    if (readcount == 1){
        P(w_or_r);
    }
    read;
    readcount--;
    if (readcount == 0){
        V(w_or_r);
    }
}
```

A context switch can happen, a writer can come in since no reader locked the semaphore!
Readers/Writers Real Solution

- Use three variables
  - int `readcount` – number of threads reading object
  - Semaphore `mutex` – control access to `readcount`
  - Semaphore `w_or_r` – exclusive writing or reading
Readers/Writers

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}

reader {
    P(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        P(w_or_r); // synch w/ writers
    V(mutex); // unlock readcount
    Read;
    P(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        V(w_or_r); // up for grabs
    V(mutex); // unlock readcount
}
Readers/Writers Notes

- `w_or_r` provides mutex between readers and writers, and also multiple writers
- Why do readers use mutex?
- What if the V(mutex) is above “if (readcount == 1)”?
- Why do we need “if (readcount == 1)”?
- Why do we need “if (readcount == 0)”?
But it still has a problem...

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}

reader {
    P(mutex);    // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        P(w_or_r); // synch w/ writers
    V(mutex);    // unlock readcount
    Read;
    P(mutex);    // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        V(w_or_r); // up for grabs
    V(mutex);    // unlock readcount
}
Problem: Starvation

• What if a writer is waiting, but readers keep coming, the writer is starved
  – If you are interested, think how to solve this problem

Ding Yuan, ECE344 Operating System
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)

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

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)

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

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

Ding Yuan, ECE344 Operating System
Bounded Buffer (4)

Producer

EMPTY = 3

EMPTY = 2

EMPTY = 1

EMPTY = 0

Consumer decrements EMPTY and blocks when buffer is full since the semaphore is at 0

Consumer

FULL = 3

FULL = 2

FULL = 1

FULL = 0

Consumer decrements FULL and blocks when buffer has no item since the semaphore FULL is at 0
Bounded Buffer (5)

Why we need both “empty” and “full” semaphores?

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)

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

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

More consumers “remove resource” than actually produced!
Bounded Buffer (6)

- Why need the mutex at all?
- Reader-Writer and Bounded Buffer are classic examples of synchronization problems
Semaphore Questions

• Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?
• If a system provides only mutex semaphores, can you use it to implement a counting semaphores?
• When to use counting semaphore?
  – Problem needs a counter
  – The maximum value is known (bounded)
Possible Deadlocks with Semaphores

Example:

Thread 1:

share two mutex semaphores S and Q
S:= 1; Q:=1;

P(S);
P(Q);
...
V(Q);
V(S);

Thread 2:

P(Q);
P(S);
...
V(S);
V(Q);

Deadlock?
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
  – 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
Monitor Semantics

• 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
    • Condition Variable

• What are the implications in terms of parallelism in monitor?
Account Example

Monitor `account` {
  double balance;

  double `withdraw`(amount) {
    balance = balance – amount;
    return balance;
  }
}

Threads block waiting to get into monitor

Withdraw(amount)
  balance = balance – amount;

When first thread exits, another can enter. Which one is undefined.

Hey, that was easy
But what if a thread wants to wait inside the monitor?
Condition Variables

- A **condition variable** is associated with a condition needed for a thread to make progress once it is in the monitor.

Monitor M {
    ... *monitored variables*
    Condition c;

    void enter_mon (...) {
        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

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) {
        if (buffer array is full)
            wait(not_full);
        Add R to buffer array;
        signal(not_empty);
    }
}

Resource `get_resource()` {
    if (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?
  - Signal is lost
Monitor bounded_buffer {

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

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

Resource get_resource () {
  ...
}

}

Waiting to enter
Waiting on condition variables
Executing inside the monitor
Condition Vars != Semaphores

• Condition variables != semaphores
  – 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::P 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::V increases the semaphore count, allowing future entry even if no thread is waiting
    • Condition variables have no history
Locks and Condition Vars

• In OS161, we don’t have monitors
• But we want to be able to use condition variables
• So we isolate condition variables and make them independent (not associated with a monitor)
• Instead, we have to associate them with a lock (mutex)
• Now, to use a condition variable...
  – Threads must first acquire the lock (mutex)
  – CV::Wait releases the lock before blocking, acquires it after waking up
Signal Semantics

- 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
  - **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)
Monitor Readers and Writers

• Write with just wait() (will be safe, maybe not “live” - why?)
  – Starvation

```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
```
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?
Summary

• Semaphores
  – P()/V() 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