Lecture 6: Semaphores and Monitors

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 2 Due 4/24
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections 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 are another data structure that provides mutual exclusion to critical sections
- Block waiters, interrupts enabled within CS
- Described by Dijkstra in THE system in 1968

Semaphores can also be used as atomic counters
- More later

Semaphores 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()
Associated with each semaphore is a queue of waiting processes.

When `wait()` is called by a thread:
- If semaphore is open, thread continues
- If semaphore is closed, thread blocks on queue

Then `signal()` 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, `signal()` has “history” (c.f. condition vars later)
  - This “history” is a counter
Semaphore Types

- Semaphores come in two types
  - **Mutex** semaphore
    - Represents single access to a resource
    - Guarantees mutual exclusion to a critical section
  - **Counting** 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;
}
```

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

\[
\begin{align*}
\text{wait (S) \{} \\
\quad & \text{Disable interrupts;} \\
\quad & \text{while (S->value == 0)} \{ \\
\quad & \quad \text{enqueue(S->q, current_thread);} \\
\quad & \quad \text{thread_sleep(current_thread);} \\
\quad & \} \\
& \quad S->value = S->value - 1; \\
& \quad \text{Enable interrupts;} \\
\} \\
\text{signal (S) \{} \\
& \quad \text{Disable interrupts;} \\
& \quad \text{thread = dequeue(S->q);} \\
& \quad \text{thread_start(thread);} \\
& \quad S->value = S->value + 1; \\
& \quad \text{Enable interrupts;} \\
\} 
\end{align*}
\]

- thread_sleep() assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?
- Need to be able to reference current thread
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

- 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
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 {
  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(w_or_r);  // synch w/ writers
  signal(mutex);  // unlock readcount
}
Readers/Writers Notes

- If there is a writer
  - First reader blocks on \texttt{w_or_r}
  - All other readers block on \texttt{mutex}
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
  - If no writer, then readers can continue
- If readers and writers are waiting on \texttt{w_or_r}, and a writer exits, who goes first?
- Why doesn’t a writer need to use \texttt{mutex}?
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
Bounded Buffer (2)

- Use three semaphores:
  - **mutex** – mutual exclusion to shared set of buffers
    » Binary semaphore
  - **empty** – count of empty buffers
    » Counting semaphore
  - **full** – count of full buffers
    » Counting 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;
        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;
    }
}

Why need the mutex at all?
Where are the critical sections?
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 examples of synchronization problems
- The Hawaiian Boat problem in Project 1 is another
- You can use semaphores to solve the problem
- Use readers/writers and bounded buffer as examples for hw
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
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 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?

Monitor account {
    double balance;

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

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

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.

- 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

- Note: 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
    Condition not_full, not_empty;

    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
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
    - 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
Next time…

- Read Chapters 5 and 7