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

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968

- 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

- 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 threads/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
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)
    - Multiple threads pass the semaphore determined by count
      - mutex has count = 1, counting has count = N
    - Represents a resource with many units available
    - or a resource allowing some unsynchronized concurrent access (e.g., reading)
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.
Using Semaphores

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
  - Homework 2 problems

- Now we’re going to use semaphores to look at more interesting examples
  - Readers/Writers
  - Producer consumer with 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: \( r \geq 0 \) \& \( 0 \leq w \leq 1 \) \& \( (r > 0) \Rightarrow (w = 0) \)

- 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(mutex);    // unlock readcount
}
Readers/Writers Notes

- `w_or_r` provides mutex between readers and writers
  - Readers wait/signal when `readcount` goes from 0 to 1 or 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, who goes first?
- Why do readers use `mutex`?
- What if the signal is above “if (readcount == 1)”?
- If read in progress when writer arrives, when can writer get access?
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

- What is desired safety property?
  - 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)

- $0 \leq np - nc \leq N$

- Use three semaphores:
  - **full** – count of full buffers
    - Counting semaphore
    - $\text{full} = ?$
      - $(np - nc)$
  - **empty** – count of empty buffers
    - Counting semaphore
    - $\text{empty} = ?$
      - $N - (np - nc)$
  - **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;
    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
  }
}

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;
  }
}
Bounded Buffer (4)

- Why need the mutex at all?

- What has to hold for deadlock to occur?
  - empty = 0 and full = 0
  - \((nc - np) + N = 0\) and \(np - nc = 0\)
  - \(N = 0\)

- 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
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 processes sharing a semaphore `mutex` that is initially 1

```c
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 the 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
Account Example

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

Hey, that was easy!

But what if a thread wants to wait inside the monitor?

- Such as “mutex(empty)” by reader in bounded buffer?
Monitors, Monitor Invariants and Condition Variables

- 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.

```c
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;
}

What happens if no threads are waiting when signal is called?
Monitor Queues

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
  - Although their operations have the same names, they have entirely different semantics
  - 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 $\text{signal}()$
  - **Hoare** monitors (original)
    - $\text{signal}()$ 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)
    - $\text{signal}()$ places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
      - Returning from $\text{wait}()$ 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?)

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()

Monitor `RW`

```c
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?
Condition Vars & Locks

- Condition variables are also used without monitors in conjunction with blocking locks
- A monitor is “just like” a module whose state includes a condition variable 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
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 Sleep in Nachos)
- 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
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()
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
Looking ahead

- Scheduling – how to decide which thread/process to run next?
  - Does it matter? Yes!

- Other concurrency issues
  - Deadlock – the deadly embrace!

- Midterm!
More synchronization practice

- Cars cross a narrow bridge. Bridge be open in one direction at a time. It can hold no more than three cars. Build threads to simulate this situation.

- Change the problem above to avoid starvation of one direction. Let’s say we don’t let more than 10 cars cross in any direction if there are cars waiting on the other direction.
More synchronization practice

- You take a break from studying cs153 to play frisbee with your friends. We have one or more frisbees, and there are two or more of you. Each student waits until they have a frisbee and their neighbor doesn’t have one and then throws the frisbee.

- What happens if the number of frisbees is equal to the number of players?
More synchronization practice

- CS153 are studying for the midterm over the national CS dish (Pizza). Each pizza pie has 8 slices. Each student eyes the pie, then grabs the next slice.
  - What race conditions can happen?
  - How can you resolve them?

- A student that grabs the last slice should order the next pie. Extend your implementation to do that