Last time

- Introduced hardware support for synchronization
  - Two flavors:
    - Atomic instructions that read and update a variable
      - E.g., test-and-set, xchange, …
    - Disable interrupts

- Blocking locks
  - Spin lock only around acquire of lock

- Introduced Semaphores
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
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;
}
```

Threads

- block

- critical section

- It is undefined which thread runs after a signal
Beyond Mutual Exclusion

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

- We’re going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples
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: \( (r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \Rightarrow (w = 0)) \)

- 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

```c
// 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: Set of buffers shared by producer and consumer threads
  - **Producer** inserts jobs into the buffer set
  - **Consumer** removes jobs from the buffer set

- 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

- Data structure should not be corrupted
  - Due to race conditions
  - Or producer writing when full
  - Or consumer deleting when empty
Bounded Buffer (2)

- $0 \leq np - nc \leq N$
- Use three semaphores:
  - **full** – count of full buffers
    - Counting semaphore
    - $full = ?$
      - $(np - nc)$
  - **empty** – count of empty buffers
    - Counting semaphore
    - $empty = ?$
      - $N - (np - nc)$
  - **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?

- 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
  - We will see and practice others
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