Lecture 9: Semaphores
Last time

- Worked through software implementation of locks
  - Good concurrency practice
  - Ended up with Dekker and Peterson’s algorithms
    - Work under assumptions of atomic and in order memory system
      - So, they do not work in practice
        - Compiler reorders
        - And memory system is not ordered

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

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it satisfy critical region requirements? (mutex, progress, bounded wait, performance?)
Another solution: Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```c
struct lock {
    disable interrupts;
}:

void acquire (lock) {
    disable interrupts;
}

void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?
On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- In a “real” system, this is only available to the kernel
  - Why?

- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don’t want interrupts disabled between acquire and release
Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

Disabling Interrupts:
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Implementing Locks (4)

- Block waiters, interrupts enabled in critical sections

```c
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    Disable interrupts;
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    Enable interrupts;
}

void release (lock) {
    Disable interrupts;
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    Enable interrupts;
}
```

acquire(lock)  
...  
**Critical section**  
...  
release(lock)
Higher-Level Synchronization

- Locks so far inefficient 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

- Plan:
  - 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

1. Semaphores come in two types
2. **Mutex** semaphore (or **binary** semaphore)
   - Represents single access to a resource
   - Guarantees mutual exclusion to a critical section

3. **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
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
Example Problem(s)

- Create a critical region where up to three threads (but no more) may enter at a time
  - Exploits the counting feature of semaphores

- Order operations across two threads; thread A executes first, then thread B executes
  - Exploits the ability to initialize semaphores to different values
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: \((r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \implies (w = 0))\)

- Use three variables
  - int \texttt{readcount} – number of threads reading object
  - Semaphore \texttt{mutex} – control access to \texttt{readcount}
  - Semaphore \texttt{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
}
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
    - $\text{full} = \text {?}$
      - $(np - nc)$
  - **empty** – count of empty buffers
    - Counting semaphore
    - $\text{empty} = \text {?}$
      - $N - (np - nc)$
  - **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.