CSE 153
Design of Operating Systems

Fall 2018

Lecture 6: 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:

```
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:

```
struct lock {
}
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)

acquire(lock)
…
Critical section
…
release(lock)

Disabling Interrupts:

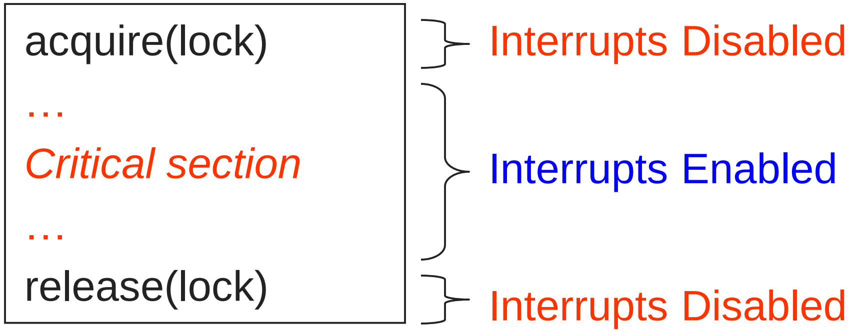
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

Implementing Locks (4)

- Block waiters, interrupts enabled in critical sections

```
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() (or P()) is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() (or V()) 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)

Protecting a critical region

```
sem mutex =1;
process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    critical region;
    V(mutex);
    noncritical region;
   }
}
```

Implementing a 2-process barrier

- Neither process can pass the barrier until both have arrived
- Must be able continuously reuse the barrier; therefore it must reinitialize after letting processes pass the barrier.
  - This will require two semaphores: a signaling semaphore for each of arrival and departure.
  - Each process x signals its arrival using a V(arrivex) and then waits on the other process' (y) semaphore with a P(arrivey);

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
  ...
  V(arrive1);  /*signal arrival */
  P(arrive2);  /*await arrival of other process */
  ...
}
process Worker2 {
  ...
  V(arrive2);
  P(arrive1);
  ..
}
```

A simple producer/consumer problem

- Use a single shared buffer
- Only one process can read or write the buffer at a time
- Producers put things in the buffer
- Consumers take things out of the buffer
- Need two semaphores
  - Empty will keep track of whether the buffer is empty
  - Full will keep track of whether the buffer is full

```
typeT buf;  /* a buffer of some type */
sem empty =1; /*initially buffer is empty */
sem full = 0;  /*initially buffer is not full */
process Producer [i = 1 to m] {
  while (true) {
    ...
    /*produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer [j=1 to n] {
  while (true) {
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

Bounded Buffers: Resource Counting

- Several messages can be queued between a producer and a consumer
- Use counting semaphores to keep track of how many buffers are full and how many are empty

```
typeT buf[n];  /*an array to hold the queue of messages*/
int front =0, rear =0-;
sem empty =n, full = 0; /*n -2 <= empty+full <= n*/
process Producer {
  while (true) {
    ...
    /*produce message data and deposit it in the buffer;*/
    P(empty);
    buf[rear] = data; rear = (rear+1)%n;
    V(full);
  }
}
process Consumer {
  while (true) {
    /*fetch message result and consume it */
    P(full);
    result = buf[front]; front = (front + 1)%n;
   V(empty);
    ...
  }
}
```

Multiple Producers and Consumers

- Since multiple producers can access deposit at the same time and multiple consumers can access fetch at the same time, we need critical regions

```
typeT buf[n]  /* an array of data*/
int front =0, rear =0;
sem empty = n, full =0;
sem mutexD =1, mutexF =1; /* semaphores for mutual exclusion*/
process Producer [i= 1 to M] {
  while (true) {
    ...
    /* produce message and deposit it in the buffer */
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear +1) %n;
    V(mutexD);
    V(full);
  }
}
process Consumer [i = 1 to N] {
  while (true) {
    ...
    /*fetch message and consumer it */
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

More complex situations…

- How to get it right?
- What if it is not done right?
  - Race condition
  - Erroneous program behavior
  - 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
  - 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