CSE 120
Principles of Operating Systems

Spring 2020

Lecture 6: Semaphores

Yiying Zhang
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization.
- We need mechanisms to control access to these shared resources in the face of concurrency.
  - So we can reason about how the program will operate.
- Our example was updating a shared bank account.
- Also necessary for synchronizing access to any shared data structure.
  - Buffers, queues, lists, hash tables, etc.
Critical Section Requirements

1) Mutual exclusion
   ♦ If one thread is in the critical section, then no other is

2) Progress
   ♦ If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
   ♦ A thread in the critical section will eventually leave it

3) Bounded waiting (no starvation)
   ♦ If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) Performance
   ♦ The overhead of entering and exiting the critical section is small with respect to the work being done within it
acquire(lock);
if (no Milk)
    buy milk;
release(lock);
} Critical Section
• Two independent threads may both notice that a lock has been released and thereby acquire it.

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (lock.held) ;
    lock.held = 1;
}

d void release (lock) {
    lock.held = 0;
}
```

A context switch can occur here, causing a race condition.
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
  - Interrupts are only disabled on a per-core basis
Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to true
  - Return the old value
- Hardware executes it atomically!

    ```c
    bool test_and_set (bool *flag) {
        bool old = *flag;
        *flag = True;
        return old;
    }
    ```

- When executing test-and-set on “flag”
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?
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?
• What about multiprocessors?
The problem with spinlocks is that they are wasteful (busy wait!)

- If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor)

How did the lock holder give up the CPU in the first place?

- Lock holder calls yield or sleep (voluntary), or
- Involuntary context switch

Only want to use spinlocks as primitives to build higher-level synchronization constructs
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 (CS) 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 chance for lock holder to be interrupted

**Disabling Interrupts:**
- Doesn’t work on multiprocessor
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are “primitive” – can’t do anything other than mutual exclusion
- Need higher-level synchronization primitives that:
  - Move waiters to the blocked queue (block waiters)
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement them
CSE 120 – Lecture 5 – Synchronization

[lec 3] Process State Transition

- **Running**
  - Scheduler dispatch
  - Wait for resource
  - Resource becomes available

- **Ready**
  - Create a process
  - Resource becomes available

- **Blocked**
  - Terminate
Implementing Locks

- If cannot hold lock, give up CPU (move to block queue)
- Use a `guard` on the lock itself

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}

void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock->guard)) ;
    if (lock->held == 0) {
        lock->held = 1;
        lock->guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock->Q;
    lock->guard = 0;
    go to sleep;
    enable interrupts;
}

void release (lock) {
    disable interrupts;
    while (test-and-set(lock->guard)) ;
    lock->held = 0;
    if (lock->Q is not empty)
        move a waiting thread to the ready queue;
    lock->guard = 0;
    enable interrupts;
}
```
Deep Thinking

• Why is this busy waiting (the while loop) not a concern?
  ♦ What’s our critical section here?

• Can we remove the disable/enable interrupts?
  ♦ With interrupts, when a process that gets the guard (pass the while loop) get context switched out, all other wait processes on other cores will be busy waiting
Today
Often times, we have to wait for shared resources
Producer & Consumer Problem

- **Producer**: creates copies of a resource
- **Consumer**: uses up (destroys) copies of a resource.
- **Buffers**: fixed size, used to hold resource produced by producer before consumed by consumer.

![Diagram of producer and consumer with buffers]

- N = 4
- 2 empty slots
- 2 occupied slots
Producer & Consumer Problem

• 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

• **Synchronization**: ensuring concurrent producers & consumers access the buffer in a correct way
  ◦ What’s a “correct way”?

• Happens inside OS all the time (e.g., I/Os)
Producer & Consumer

Producer

while (1) {

    produce an item;

    while (pool is full) ;

    insert(item to pool);

}

Consumer

While (1) {

    while (pool is empty) ;

    remove(item from pool);

    consume the item;

}
Producer

while (1) {
    produce an item;
    while (pool is full) ;
    acq(lock);
    insert(item to pool);
    rel(lock);
}

Consumer

While (1) {
    while (pool is empty) ;
    acq(lock);
    remove(item from pool);
    rel(lock);
    consume the item;
}

Producer & Consumer – Locks?
Producer

while (1) {

produce an item;

acq(lock);

while (pool is full) ;

insert(item to pool);
rel(lock);

}

Consumer

While (1) {

acq(lock);
while (pool is empty) ;
remove(item from pool);
rel(lock);

consume the item;

}

Producer & Consumer – Locks?
Often times, we have to wait for shared resources

• Busy waiting is a bad idea

• Checking resources itself needs to be in critical section!

• Busying waiting inside CS even worse!
  ◆ No one else can check!

→ Need a more powerful sync. primitive!
→ Want the simplest primitive that can check & wait
Higher-Level Synchronization

• We looked at using locks to provide mutual exclusion
• Locks work, but they have limited semantics
  ♦ Just provide mutual exclusion
• Instead, we want synchronization mechanisms that
  ♦ Provide semantics beyond mutual exclusion
• We now look at two high-level mechanisms
  ♦ **Semaphores**: binary (mutex) and counting
  ♦ **Conditional variables**: next lecture
• Use them to solve common synchronization problems
Semaphore

• A synchronization variable that takes on non-negative integer values
  ♦ Invented by Edsger Dijkstra in the mid 60’s

• Two primitive operations
  ♦ `wait(semaphore)`: an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
  ♦ `signal(semaphore)`: an atomic operation that increments semaphore by 1
Semaphore

```c
wait(S) {
    while (S<=0) ;
    S--;  
}
```

```c
signal(S) {
    S++;  
}
```

- Historically, `wait()` is known as `P()`, `signal` is known as `V()``;
- In reality, `wait/signal` are not implemented as above
Blocking in Semaphores

- 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
Binary Semaphore

Init: $S = 1$;

wait($S$) {
    while ($S == 0$)
        ;
    $S --$;
}

signal($S$) {
    if ($S == 0$)
        $S ++$;
}

• **Binary semaphores**: only take 0 or 1
• **Sounds familiar?**
  ♦ $S = 0 \rightarrow$ someone is holding the lock!
Semaphore

P(S)

wait(S) {
    while (S <= 0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 1

- P1: P(S), …, V(S)
- P2: P(S), …, V(S)
- P3: P(S), …, V(S)
Semaphore

P(S)  V(S)

\[
\text{wait}(S) \{
\text{while } (S<=0); \\
S--; \\
\}
\]

\[
\text{signal}(S) \{
S++; \\
\}
\]

What happens if initially \( S = 1 \)

- P1: P(S), …., V(S)
- P2: P(S) \rightarrow, …, V(S)
- P3: P(S) \rightarrow, …, V(S)
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 1

- P1: P(S), ..., V(S)
- P2: V(S), ..., P(S)
- P3: V(S), ..., P(S)
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--; 
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 2

• P1: P(S), …, V(S)
• P2: P(S), …, V(S)
• P3: P(S), …, V(S)
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 2

• P1: P(S), …, V(S)
• P2: P(S), …, V(S)
• P3: P(S)---------→, …, V(S)
semaphore has built-in counting!

- signal(S) simply increments S
  - “just produced an item”
  - S value = how many items have been produced

- wait(S) will return without waiting only if S > 0;
  - Wait(S) is saying “waited until there is at least one item, and just consumed an item”
Two usages of semaphores

- For mutual exclusion:
  - to ensure that only one process is accessing shared info at a time.
  - Semaphores or binary semaphores?

- For condition synchronization:
  - to permit processes to wait for certain things to happen
  - Semaphores or binary semaphores?
Semaphore benefits over locks

- Has a value => more semantics
  - When greater than 1, can allow multiple threads to access critical resource
  - When equal to 1, can be used for mutual exclusion (only one thread in critical section)
Producer & Consumer Problem

- **Producer**: creates copies of a resource
- **Consumer**: uses up (destroys) copies of a resource.
- **Buffers**: fixed size, used to hold resource produced by producer before consumed by consumer.

![Diagram of Producer and Consumer Problem](image_url)

- N = 4
- 2 empty slots
- 2 occupied slots
• Define constraints (what is “correct”)
  - Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffers, if all buffer space is in use (mutual excl. or condition sync?)

• Use a separate semaphore for each constraint
  - Full = 0
  - Empty = N
Producer & Consumer – semaphore attempt, what’s wrong?

**Producer**
```
while (1) {
    produce an item;
    wait(EMPTY);
    insert(item to pool);
    signal(FULL)
}
```

**Consumer**
```
While (1) {
    wait(FULL);
    remove(item from pool);
    signal(EMPTY);
    consume the item;
}
```

Init: FULL = 0; EMPTY = N;
int buffer[MAX];
int fill = 0;
int use = 0;

insert (int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX
}

int get() {
    int tmp = buffer[use]
    use = (use + 1) % MAX
    return tmp;
}

Need to protect shared resource (critical section)!
• Define constraints (what is “correct”)
  ♦ Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  ♦ Producer must wait for consumer to empty buffers, if all buffer space is in use (mutual excl. or condition sync?)
  ♦ Only one process must manipulate buffer pool at once (mutual excl. or condition sync?)

• Use a separate semaphore for each constraint
  ♦ Full = 0
  ♦ Empty = N
  ♦ Mutex = 1
Producer & Consumer – semaphore attempt 2, what’s wrong?

**Producer**

```c
while (1) {
    produce an item;
    acq(lock);
    wait(EMPTY);
    insert(item to pool);
    signal(FULL)
    rel(lock);
}
```

**Consumer**

```c
While (1) {
    acq(lock);
    wait(FULL);
    remove(item from pool);
    signal(EMPTY);
    rel(lock);

    consume the item;
}
```

Init: FULL = 0; EMPTY = N; Mutex = 1;

Deadlock!
Producer & Consumer – semaphore working

Producer

while (1) {
    produce an item;
    wait(EMPTY);
    acq(lock);
    insert(item to pool);
    rel(lock);
    signal(FULL)
}

Consumer

While (1) {
    wait(FULL);
    acq(lock);
    remove(item from pool);
    rel(lock)
    signal(EMPTY);
    consume the item;
}

Init: FULL = 0; EMPTY = N; Mutex = 1;
Readers-Writers problem

- A data object is shared among multiple processes
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)
Readers-Writers problem (Solution 1)

- Constraints:
  - Writers can only proceed if there are no readers/writers
  - Readers can proceed only if there are no writers
  - use a single semaphore BlockWrite
    - To keep track of how many are reading
  - use a shared variable
    - To count the current number of readers
  - use semaphore Mutex
    - Only one process manipulates the shared variable at once

- Initialization:
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - int Readers = 0; // count of readers reading in critical section
  - semaphore Mutex = 1; // binary semaphore (basic lock)
Reader

P(Mutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
   P(BlockWrite);
V(Mutex);

< Do the Reading >

P(Mutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
   V(BlockWrite);
   V(Mutex);
Writer

P(BlockWrite); // wait to lock the shared resource for a writer

< Do the Writing >

V(BlockWrite);
What will happen in different scenarios?

1. The first reader blocks if there is a writer; any other readers who try to enter block on mutex.
2. The last reader to exit signals a waiting writer.
3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
4. If a writer exits and a reader goes next, then all readers that are waiting will fall through.
5. Does this solution guarantee all threads will make progress?

Writes can starve
=> Read preference
What is a good solution?

- Only one process inside a critical section
- Processes outside of critical section should not block other processes
- No one waits forever
- No assumption about CPU speeds
- Works for multiprocessors
Readers-Writers problem (Solution 2)

- How do we let reads yield to writes?
  - semaphore BlockRead = 1; // used to block readers
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - int Readers = 0, Writers = 0; // count of readers and writers in critical section
  - semaphore RMutex = 1; // binary semaphore for Readers
  - Semaphore WMutex = 1; // binary semaphore for Writers
Reader

P(BlockRead); // at most one reader can go before a pending write
P(RMutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
    P(BlockWrite);
V(RMutex);
V(BlockRead);

< Do the Reading >

P(RMutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
    V(BlockWrite);
V(RMuxet);
Write

P(WMutex);
Writers++;
if (Writers == 1) // block readers
    P(BlockRead);
V(Wmutex);

P(BlockWrite); // ensures only one writer
< Do the Writing >
V(BlockWrite);

P(WMutex);
Writers--;
if (Writers == 0) // enable readers
    V(BlockRead);
V(WMutex);
Problem of solution 2

- Reader starvation

- Is there a solution that’s fair to both reads and writes?
  - An idea: use a FIFO queue for all readers and writers
Semaphore Summary

• Semaphores can be used to solve many 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
  ♦ No control or guarantee of proper usage
  ♦ Sometimes hard to use and prone to bugs
    » Difficult to get the counting right (e.g., initial value)
Next time...

• Read Chapter 30, 32
Backup Slides
Semaphore implementation

```c
wait(S) {
    while (S<=0);
    S--;
}

signal(S) {
    S++;
}
```

- Can they be implemented in the user space?
  - An intuitive argument?
- No existing hardware implements them directly
  - Scheduling/queuing cannot be easily done in HW

⇒ Semaphore must be done in OS, typically with low-level synchronization support from hardware
Use TAS to implement semaphores on multiprocessor

```c
void wait(semaphore s) {
    disable interrupts;
    while (1 == tas(&lock, 1));
    if (s->count > 0) {
        s->count --;
        lock = 0;
        enable interrupts;
        return;
    }
    add(s->q, current_process);
    lock = 0;
    enable interrupts;
    sleep(); /* re-dispatch */
    enable interrupts;
}

void signal(semaphore s) {
    disable interrupts;
    while (1 == tas(&lock, 1));
    s->count ++;
    if (!isEmpty(s->q)) {
        thread = removeFirst(s->q);
        wakeup(process);
        /* put process on Ready Q */
    }
    lock = 0;
    enable interrupts;
}
```
To reference current thread: KThread.currentThread()

KThread.sleep() assumes interrupts are disabled
- Note that interrupts are disabled only to enter/leave critical section
- How can it sleep with interrupts disabled?