CSE 120
Principles of Operating Systems

Fall 2021

Lecture 6: Semaphores

Yiying Zhang
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);
How can we separate “checking” from “buying milk” and only lock “checking”? 

```c
local_flag = FALSE;

Acquire(lock);
if (no note && noMilk){
    leave note;
    local_flag = true;
}
Release(lock);

If (local_flag) buy milk;
Acquire(lock)
If (local_flag){
    local_flag = FALSE;
    remove note;
}
Release(lock);
```
Implementing Locks

- 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)

```plaintext
acquire(lock)  
...  
Critical section  
...  
release(lock)
```
Implementing Locks with a queue

- 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\rightarrow guard)) ;
    if (lock\rightarrow held == 0) {
        lock\rightarrow held = 1;
        lock\rightarrow guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock\rightarrow Q;
    lock\rightarrow guard = 0;
    go to sleep;
    enable interrupts;
}

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

• Lock is useful when implementing critical sections

• Critical sections’ semantic is to mutually exclude

• Mutual exclusion does not solve all the synchronization problems

• Sometimes, we need other semantics
  ♦ E.g., wait for shared resources to become available
  ♦ E.g., allowing multiple threads to generate/get (different) shared resources
  ♦ E.g., use certain conditions to decide when to enter CS
Producer & Consumer Problem

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

![Diagram](image_url)

- 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
  ♦ There can be multiple producers and multiple consumers
  ♦ 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

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 & Consumer – Locks?

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;
}
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 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
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

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

signal(S) {
    S++;
}
\end{verbatim}

- 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 threads
- 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: \( S = 1; \)

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

\[
\text{signal}(S) \{
    \text{if (} S == 0\text{)}
    S++; \\
\}
\]

- **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 \( S = 1 \)

- T1: P(S), ..., V(S)
- T2: P(S), ..., V(S)
- T3: P(S), ..., V(S)
Semaphore

\[
P(S) \\
\text{wait}(S) \{ \\
\text{while } (S \leq 0); \\
S--; \\
\}
\]

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

What happens if initially \( S = 1 \)
- T1: \( P(S), \ldots, V(S) \)
- T2: \( P(S) \quad \rightarrow \quad, \ldots, V(S) \)
- T3: \( P(S) \quad \rightarrow \quad, \ldots, V(S) \)
Semaphore

\[
\begin{align*}
\text{P}(S) & \quad \text{V}(S) \\
\text{wait}(S) & \quad \text{signal}(S) \\
\text{wait}(S) & \quad \text{signal}(S) \\
\text{while } (S \leq 0); & \quad S++; \\
\text{S--}; & \quad \} \\
\} & \\
\end{align*}
\]

What happens if initially \( S = 1 \)

- T1: P(S), …, V(S)
- T2: V(S), …, P(S)
- T3: V(S), …, P(S)
Semaphore

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

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

What happens if initially S = 2
• T1: P(S), …, V(S)
• T2: P(S), …, V(S)
• T3: P(S), …, V(S)
Semaphore

P(S)

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

V(S)

signal(S) {
    S++;
}

What happens if initially S = 2

- T1: P(S), …, V(S)
- T2: P(S), …, V(S)
- T3: 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 thread is accessing shared info at a time.
  ♦ Non-binary semaphores or binary semaphores?

• For condition synchronization:
  ♦ to permit threads to wait for certain things to happen
  ♦ Non-binary 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 with buffers]

- N = 4
- 2 empty slots
- 2 occupied slots
Producer & Consumer (cont)

• 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
  ♦ FILLED = 0
  ♦ EMPTY = N
Producer & Consumer – semaphore attempt, what’s wrong?

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

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

Init: FILLED = 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)!
Producer & Consumer (cont)

• 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 thread must manipulate buffer pool at once (mutual excl. or condition sync?)

• Use a separate semaphore for each constraint
  ♦ FILLED = 0
  ♦ EMPTY = N
  ♦ MUTEX = 1 (binary semaphore), or use a lock
Producer & Consumer – semaphore attempt 2, what’s wrong?

Init: FILLED = 0; EMPTY = N;

Producer
while (1) {
    produce an item;
    acquire(lock);
    wait(EMPTY);
    insert(item to pool);
    signal(FILLED)
    release(lock);}

Consumer
While (1) {
    acquire (lock);
    wait(FILLED);
    remove(item from pool);
    signal(EMPTY);
    release (lock);
    consume the item;
}

Deadlock!
Producer & Consumer – semaphore working

Producer
while (1) {
    produce an item;
    wait(EMPTY);
    acquire(lock);
    insert(item to pool);
    release(lock);
    signal(FILLED)
}

Consumer
While (1) {
    wait(FILLED);
    acquire(lock);
    remove(item from pool);
    release(lock);
    signal(EMPTY);
    consume the item;
}

Init: FILLED = 0; EMPTY = N;
Readers-Writers problem

- A data object is shared among multiple threads
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)

- Used a lot in many data accessing systems
  - E.g., our old bank account balance example
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 semaphore BlockWrite
to block writers when there are readers or a writer

→ 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)
P(BlockWrite); // wait until there is no readers

< Do the Writing >

V(BlockWrite);
Reader

Readers++;

< Do the Reading >

Readers--;
Reader

Readers++;
if (Readers == 1) // first reader needs to wait until no writer
    P(BlockWrite);

< Do the Reading >

Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
Reader

P(Mutex); // protect the manipulation of "Readers"
Readers++;
if (Readers == 1) // first reader needs to wait until no writer
    P(BlockWrite);
V(Mutex);

< Do the Reading >

P(Mutex);
Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
V(Mutex);
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 the mutex.
2. The last reader exiting 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? Is it fair?

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

• Only one thread inside a critical section
• Threads outside of critical section should not block other processes
• No one waits forever (no starvation)
• No assumption about CPU speeds or scheduling decisions
• Works for multiprocessors
Readers-Writers problem (Solution 2)

• How do we let reads yield to writes?
  ♦ semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  ♦ int Readers = 0; // count of readers in critical section
  ♦ semaphore RMutex = 1; // binary semaphore for Readers

  ♦ semaphore BlockRead = 1; // used to block readers
  ♦ int Writers = 0; // count of writers in critical section
  ♦ 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 needs to wait until no writer
    P(BlockWrite);
V(RMutex);
V(BlockRead);

< Do the Reading >

P(RMutex);
Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
V(RMutex);
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
  - *Work this out on your own*
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
    » Relies on programmers to properly guard and use it
    » No control or guarantee of proper usage
  ♦ No connection between the semaphore and the data being controlled by the semaphore
  ♦ Sometimes hard to use and prone to bugs
    » Difficult to get the counting right (e.g., initial value)
Semaphore implementation

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

```c
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
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_thread);
    lock=0;
    sleep(); /* re-dispatch */
    enable interrupts;
}

void signal(semaphore s) {
    disable interrupts;
    while (1 == tas(&lock, 1));
    if (isEmpty(s->q) {
        s->count ++;
    } else {
        thread = removeFirst(s->q);
        wakeup(thread);
        /* put thread on Ready Q */
    }
    lock = 0;
    enable interrupts;
}
Next time...

• Read Chapter 30, 32
Semaphores in Nachos

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?