CS-446/646

Semaphores & Monitors

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Semaphore Motivation

Problem with *Lock:*

- Ensures *Mutual Exclusion*, but not execution order

Producer-Consumer problem: Ensuring execution order makes sense

- *Producer*: Creates resources
- *Consumer*: Uses resources
- *Bounded Buffer*: Shared between them
- Execution order: *Producer* should just wait if *Bounded Buffer* is full, *Consumer* should just wait if *Bounded Buffer* is empty
  - e.g. **`$ cat entries.txt | sort | uniq | wc`**

Semaphore Definition

Abstract data type (i.e. a high-level mechanism) to provide *Synchronization*

- Described by Dijkstra in the "THE (Technische Hogeschool Eindhoven) Operating System" in 1968

A *Synchronization* object that **contains an integer counter** variable

- No operation to access integer counter variable directly
- *Semaphore* safety property: Integer counter value never allowed to go below 0
- Integer counter variable must be initialized to some value:
  - **`sem_init (sem_t *s, int pshared, unsigned int value)`**
- Operations to manipulate integer counter variable:
- **`sem_wait`** (or **`down()`**, **`P()`** -robieren): Decrements, *Blocks* until semaphore is *Open*
- **`sem_post`** (or **`up()`**, **`V()`** -erhogen): Increments, allows another *Thread* to enter

```
int sem_wait(sem_t *s) {
  // 1. wait until value of
  // semaphore s becomes > 0
  // 2. decrement value by 1
}
```

```
int sem_post(sem_t *s) {
  // 1. increment value of s by 1
  // 2. if there are 1 or more
  // threads waiting, wake 1
}
```

Blocking* in *Semaphores

Associated with each *Semaphore* is a *Queue* of waiting *Threads*

When **`P()`** / **`sem_wait()`** is called by a *Thread*:

- If *Semaphore* is *Open*, *Thread* continues
- If *Semaphore* is *Closed*, *Thread* will *Block* on *Queue*

When **`V()`** / **`sem_post()`** *Opens* the *Semaphore*:

- If a *Thread* is waiting on the *Queue*, it is *Unblocked*
- If no *Threads* are waiting on the *Queue*, the **signal is remembered** for the next *Thread*
  - In other words, **`V()`** has "memory"
    - In contrast to *Condition Vars* (will see these later)
  - This "memory" property is derived from the integer counter value

Semaphore Types

Mutex Semaphore (or *Binary Semaphore*)

- Represents single access to a resource; **`X=1`**
- Guarantees *Mutual Exclusion* to a *Critical Section*

Counting Semaphore (or *General Semaphore*)

- Represents a resource with many units available, or a resource to which we want to limit concurrent access (e.g. reading); **`X>1`**
  - Is initialized to number of resources available
- Multiple *Threads* can pass the *Semaphore* "wait" test
- Number of *Threads* determined by *Semaphore* "counter"

Note:
No direct access to counter

```
sem_init(s, 0, X)
…
sem_wait(s);
// critical section
sem_post(s);
```

```
int sem_init(sem_t *sem,
             int pshared,
             unsigned int value);
```

Initializes the *Semaphore* at **`sem`**.
`value` specifies the initial value for it.
`pshared` indicates whether this *Semaphore* is to be shared between the *Threads* of a *Process*, or between *Processes* (**`sem`** can be part of a region of *Shared Memory*).

```
int sem_post(sem_t *sem);
```

Increments (*Unlocks*) the *Semaphore* at **`sem`**.

```
int sem_wait(sem_t *sem);
```

Decrements (*Locks*) the *Semaphore* at **`sem`**.
If the *Semaphore* currently has the value zero, then the call *Blocks* until either it becomes possible to perform the decrement or a *Signal* handler *Interrupts* the call.

Semaphore Uses

Mutual Exclusion

- Case of *Binary Semaphore*

`sem_init(s, 0` or `1, X=1)`

```
sem_wait(s);
// critical section
sem_post(s);
```

```
sem_wait(s);
// critical section
sem_post(s);
```

Execution Ordering

- Case of Limiting Concurrent Access → *Counting Semaphore*

`sem_init(s, 0` or `1, X=0)`

```
// 1st half
// of computation
sem_post(s);
```

```
sem_wait(s);
// 2nd half
// of computation
```

Producer-Consumer (***Bounded-Buffer***) **Problem**

Bounded Buffer

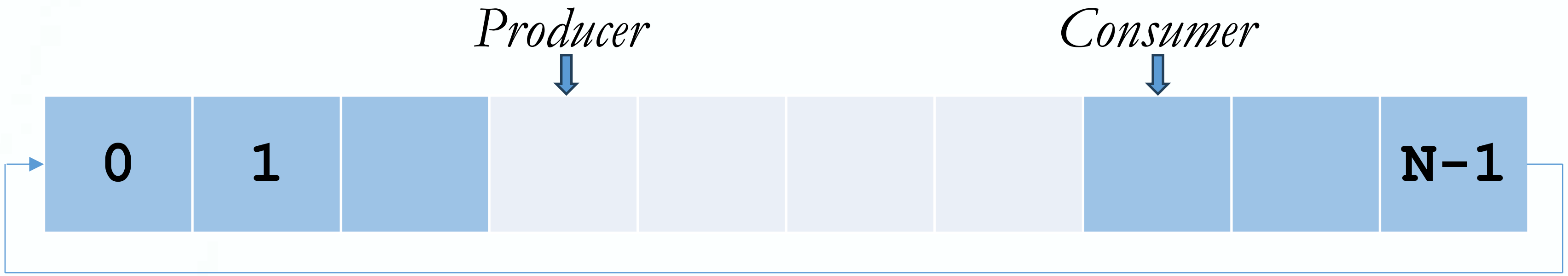
- size N, Access entry 0… N-1, then "wraps around" to 0 again

Producer Thread : Writes data to *Bounded Buffer*

Consumer Thread : Reads data from *Bounded Buffer*

Execution ordering constraints:

- *Producer* shouldn't try to produce if *Bounded Buffer* is full
- *Consumer* shouldn't try to consume if *Bounded Buffer* is empty



Producer-Consumer (*Bounded-Buffer*) Problem

Solution – 1st version

Two *Semaphores*

- `sem_t filled; // # of filled slots`
- `sem_t empty; // # of empty slots`
- *Problem:* Does this also achieve *Mutual Exclusion* ?

```
sem_init(&filled, 0, 0);
sem_init(&empty, 0, N);
```

```
void* producer(void* _arg) {
  sem_wait(&empty);
  … // fill a slot
  sem_post(&filled);
}
```

```
void* consumer(void* _arg) {
  sem_wait(&filled);
  … // empty a slot
  sem_post(&empty);
}
```

Note: Sequencing operations

Producer-Consumer (*Bounded-Buffer*) Problem

Solution – Final version

Three *Semaphores*

- `sem_t filled; // # of filled slots`
- `sem_t empty; // # of empty slots`
- `sem_t mutex; // # mutual exclusion`

Note: Can also use a **`pthread_mutex_t`**

```
sem_init(&filled, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```
void* producer(void* _arg) {
  sem_wait(&empty);
  sem_wait(&mutex);
  … // fill a slot
  sem_post(&mutex);
  sem_post(&filled);
}
```

```
void* consumer(void* _arg) {
  sem_wait(&filled);
  sem_wait(&mutex);
  … // empty a slot
  sem_post(&mutex);
  sem_post(&empty);
}
```

Data Structure "internal access" *Critical Section*

Note: Fill / Empty operations correspond to manipulating the *Circular Buffer*'s **`head`** & **`tail`**

Semaphore Summary

- *Semaphores* can be used to solve any of the traditional *Synchronization* problems

- Drawbacks:
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in Program
  - No direct connection between the *Semaphore* and the data being controlled by it
  - Used for both *Critical Sections* (*Mutual Exclusion*) and *Execution Ordering* (*Scheduling*)
  - No control or guarantees for their proper usage

- When used in complex code can lead to bugginess
  - Solution: Leverage *Object-Oriented Programming* to support controlled behaviors

Monitors

Monitors

An *Object-Oriented Language* construct that controls access to shared data

- *Synchronization* code added by compiler, enforced at runtime

A module that encapsulates

- ***Shared Data Structures***
- ***Procedures*** that operate on the shared data structures
- ***Synchronization*** between concurrent *Threads* that invoke these procedures

- Guarantees that access of its data through *Threads* is done in legitimate ways only

Monitors

A *Monitor* guarantees *Mutual Exclusion*

- Only one *Thread* can execute **any** *Monitor* Procedure **at a time**
  - The *Thread* is "inside the *Monitor*"
- If a second *Thread* invokes a *Monitor* procedure when a first *Thread* is already executing one, the second *Thread* shall *Block*
  - i.e. the *Monitor* has to have a *Wait Queue*
- If a *Thread* that is "inside a *Monitor*" *Blocks*, then another *Thread* can enter the *Monitor*

Note: A *Monitor Invariant* is a safety property associated with the *Monitor*

- It's an assertion regarding the *Monitored Variables*
- It holds whenever a *Thread* enters or exits the *Monitor*
  - i.e. the assertion holds whenever there is no *Thread* executing "inside the *Monitor*"

Monitors

A *Monitor* is like one big *Super-Lock* for a set of operations/methods

- It is however a *Language*-level implementation
  - Compiler automatically inserts the necessary *Synchronization* operations upon entry and exit of *Monitor* Procedures

Monitor Procedures

```
monitor account {
  int balance;
  public void deposit() {
    ++balance;
  }
  public void withdraw() {
    --balance;
  }
};
```

Example of (part of) the operations inserted at *Compile-Time*.

```
lock(this.m);
…
++balance;
…
unlock(this.m);
```

```
lock(this.m);
…
--balance;
…
unlock(this.m);
```

  - C++ does not have *Monitors*

Note: But check out **synchronized**, C++20 *Synchronized Blocks (experimental)*:
https://en.cppreference.com/w/cpp/language/transactional_memory

Monitors

Condition Variables

Remember: A *Monitor* also needs to take care of Wait, Wakeup, Queueing functionalities

- Not just *Locking*
- What if a *Thread* has to wait for something to happen/change, but is already "inside the *Monitor*"?
    - Bad if left to just *Busy-Wait*
    - Worse: No one can now get "inside the *Monitor*" (e.g. not even to take corrective actions)
  - Have to be able to let a different *Thread* enter "inside the *Monitor*"

In order to achieve the above, a *Monitor* can use a different *Synchronization* mechanism:

Condition Variables

- A *Condition Variable* is associated with a condition needed for a *Thread* to make progress once it is "inside the *Monitor*"

Condition Variables (with respect to *Monitors*)

Operations on *Condition Variables*

`wait()`

Suspends the calling *Thread* and releases the *Monitor Lock* (when it resumes, it will reacquire the *Lock*)

For **`wait()`** to be called, the *Thread* has to already be "inside the *Monitor*"

➢ (Should be) called when the *Condition Predicate* is **`false`**

`signal()`

Resumes one *Thread* waiting in **`wait()`**, if any

➢ (Should be) called once *Condition Predicate* becomes **`true`**, and wants to **Wakeup one** waiting *Thread*

`broadcast()`: Resumes all *Threads* waiting in **`wait()`**

➢ (Should be) called once *Condition Predicate* becomes **`true`**, and wants to **Wakeup all** waiting *Threads*

Note: Condition Variables are not **`bool`**ean objects; they are *associated* with a **`bool`**ean *Condition Predicate*

o **`if (cv) then`** … does not make sense

✓ **`if (num_resources == 0) then wait(cv)`** does

Monitors

Condition Variables (with respect to *Monitors*)

Although operations have similar names with *Semaphores*, they are different

- But one can be used to implement the other

Access to the *Monitor* is controlled by a *Lock*

`wait()`: *Blocks* the calling *Thread*, and gives up the *Lock*

- To call **`wait()`**, the *Thread* has to be "inside the Monitor" (hence holds the *Lock*)
  - *Semaphore*'s **`sem_wait()`** just blocks the *Thread* on the *Queue*

`signal()`: Causes a waiting *Thread* to Wakeup

- If there is no **`wait()`**ing *Thread*, the **`signal()`** is lost
  - *Remember: Semaphore*'s **`sem_post()`** increases its count, allowing future entry even if no *Thread* is Waiting right now

- I.e. *Semaphores* are "sticky", *Condition Variables* have no "memory"
  - If no one is Waiting for a **`signal()`**, it is lost

Condition Variables (with respect to *Monitors*)

Producer-Consumer with *Monitors*

```
monitor ProducerConsumer {
  int nfilled = 0;
  cond has_empty, has_filled;
  void produce() {
    if (nfilled == N)
      wait (has_empty);
    … // fill a slot
    ++ nfilled;
    signal (has_filled);
  }
  void consume() {
    if (nfilled == 0)
      wait (has_filled);
    … // empty a slot
    -- nfilled;
    signal (has_empty);
  }
};
```

A (one) *Monitor* with two *Condition Variables*:

- **`has_empty`**: Buffer has at least one empty slot
- **`has_filled`**: Buffer has at least one filled slot

`nfilled`: Number of filled slots

E.g.:

- If a *Thread* tries to **`consume()`** and the Buffer is empty, it will be blocked at the *CV*. If another *Thread* tries to **`consume()`** again, it will also be blocked at the *CV*, etc.
- If a third Thread tries to **`produce()`**, it will pass the other *CV*'s wait, and **`signal()`** (one of) the first 2 *Threads*

I.e. (each) *Condition Variable* also has to have a *Queue*

Condition Variable Signal Semantics

When `signal()` wakes up a waiting *Thread*, which *Thread* to run "inside the *Monitor*"?

- The *Signaling Thread (/Signaler)*, or the *Waiting Thread (/Waiter)* ?

Hoare Semantics:

Suspends *Signaler,* and immediately transfers control to a *Waiter*

- The *Condition* that the *Waiter* was anticipating is guaranteed to hold when waiter executes
- Difficult to implement in practice, *Signaler* must restore *Monitor Invariants* before signaling

Remember: Assertions that hold whenever no *Thread* is "inside the *Monitor*"; i.e. implementation needs to remember state because *Thread* hasn't completed yet

Mesa Semantics

Signal moves a single *Waiter* from the blocked state to a runnable state, then the *Signaler* continues until it "exits the *Monitor*"

- Problem: *Condition Variable's Predicate* is not necessarily true when *Waiter* gets to run again
  - Return from `wait()` is only a hint that something changed, **always have to recheck** *Predicate*
- E.g. *Spurious Wakeup* – Fill one single slot and `signal()`, but before a scheduled woken consumer grabs the *Queue Lock* to continue, a different (e.g. fourth) *Thread* enters the *Queue,* grabs the *Lock*, consumes the one filled slot. The woken *Thread* will find the *Predicate* changed once it runs.

Condition Variables

Producer-Consumer with *Monitors*

```
monitor ProducerConsumer {
  int nfilled = 0;
  cond has_empty, has_filled;

  void produce() {
    while (nfilled == N)
      wait (has_empty);
    … // fill a slot
    ++ nfilled;
    signal (has_filled);
  }

  void consume() {
    while (nfilled == 0)
      wait (has_filled);
    … // empty a slot
    -- nfilled;
    signal (has_empty);
  }
};
```

Spurious Wakeup – **`pthread`**

- **`pthread_cond_signal()`** is only guaranteed to unblock **at least one** *Thread*
- Even worse, a *Thread* blocked in **`pthread_cond_wait`** can return with no **`pthread_cond_signal/broadcast()`** call

Spurious Wakeup Fix:

- When woken up, a *Thread* **must** recheck the *Predicate* associated to the *Condition Variable* it was waiting on
- Most systems use *Mesa* Semantics
  - e.g. **`pthread`**

Monitor & Condition Variables with `pthread`

Producer-Consumer with *Monitors*

```
class ProducerConsumer {
  int nfull = 0;
  pthread_mutex_t m;
  pthread_cond_t has_empty,
                 has_full;
  public:
  void produce() {
    pthread_mutex_lock(&m);
    while (nfull == N)
      pthread_cond_wait(&has_empty,
                        &m);
    … // fill slot
    ++ nfull;
    pthread_cond_signal(has_full);
    pthread_mutex_unlock(&m);
  }
  …
};
```

C/C++ don't provide *Monitors*, but we can implement such functionality using **`pthread_mutex_t`** and **`pthread_cond_t`**

For the *Producer-Consumer* problem, we need 1 *Mutex* and 2 *Condition Variables*

- Manually lock and unlock *Mutex* for *Monitor* procedures
- ```
  int pthread_cond_wait(
       pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex );
  ```

  Atomically waits on **`cond`** and releases **`mutex`**

The function shall *Block* on a *Condition Variable*. It shall be called with *mutex* locked by the calling *Thread* or Undefined Behavior results. The function **atomically** releases **`mutex`** and causes the calling *Thread* to *Block* on **`cond`**… Upon successful return, the **`mutex`** shall have been locked and shall be owned by the calling *Thread*.
  ```

Monitor & Condition Variables with **pthread**

Producer-Consumer with *Monitors*

```
class ProducerConsumer {
  int nfull = 0;
  pthread_mutex_t m;
  pthread_cond_t has_empty,
                 has_full;
  public:
  void produce() {
    pthread_mutex_lock(&m);
    while (nfull == N)
      pthread_cond_wait(&has_empty,
                        &m);
    … // fill slot
    ++ nfull;
    pthread_cond_signal(has_full);
    pthread_mutex_unlock(&m);
  }
  …
};
```

Note: Unlock the *Mutex* **after** calling **pthread_cond_signal()**

C/C++ don't provide *Monitors*, but we can implement such functionality using **pthread_mutex_t** and **pthread_cond_t**

For the *Producer-Consumer* problem, we need 1 *Mutex* and 2 *Condition Variables*

- Manually lock and unlock *Mutex* for *Monitor* procedures
- **int pthread_cond_signal(pthread_cond_t * *cond*);**

  Atomically waits on **cond** and releases **mutex**

The function shall *Unblock* **at least one** of the *Threads* that are *Blocked* on the specified *Condition Variable* **cond**… may be called by a *Thread* whether or not it currently owns the *Mutex* that *Threads* calling **pthread_cond_wait()** … have associated with the *Condition Variable*… however, if predictable *Scheduling* behaviour is required, then that *Mutex* is *Locked* by the **pthread_cond_signal()**-calling Thread

CS-446/646

Time for Questions !