Semaphores and Monitors: High-level Synchronization Constructs

Key idea of Semaphores vs. Locks

- Locks: Mutual exclusion only (1-exclusion)
- Semaphores: k-exclusion
  - $k = 1$, equivalent to a lock
  - Sometimes called a mutex, or binary semaphore
  - $k = 2^+$, up to $k$ threads at a time
- Many semaphore implementations use “up” and “down”, rather than Dutch names (P and V, respectively)
- ‘cause how many programmers speak Dutch?
- Semaphore starts at $k$
  - Acquire with down(), which decrements the count
  - Blocks if count is 0
  - Release with up(), which increments the count and never blocks

Important properties of Semaphores

- Semaphores are non-negative integers
- The only operations you can use to change the value of a semaphore are P(down) and V(up) (except for the initial setup)
  - P(down) can block, but V(up) never blocks
- Semaphores are used both for
  - Mutual exclusion, and
  - Conditional synchronization
- Two types of semaphores
  - Binary semaphores: Can either be 0 or 1
  - General/Counting semaphores: Can take any non-negative value
- Binary semaphores are as expressive as general semaphores (given one can implement the other)

How many possible values can a binary semaphore take?

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4
Using Semaphores for Mutual Exclusion

- Use a binary semaphore for mutual exclusion
  
  Semaphore = new Semaphore(1);

- Using Semaphores for producer-consumer with bounded buffer
  
  int count;
  Semaphore mutex;
  Semaphore fullBuffers;
  Semaphore emptyBuffers;

Coke Machine Example

- Coke machine as a shared buffer
- Two types of users
  - Producer: Restocks the coke machine
  - Consumer: Removes coke from the machine
- Requirements
  - Only a single person can access the machine at any time
  - If the machine is out of coke, wait until coke is restocked
  - If machine is full, wait for consumers to drink coke prior to restocking
- How will we implement this?
  - How many lock and condition variables do we need?
    - A. 1 B. 2 C. 3 D. 4 E. 5

Revisiting Coke Machine Example

Class CokeMachine{
  int count;
  Semaphore new mutex(1);
  Semaphore new fullBuffers(0);
  Semaphore new emptyBuffers(numBuffers);
}

CokeMachine::Deposit() {
  emptyBuffers->P();
  mutex->P();
  Add coke to the machine;
  count++;
  mutex->V();
  fullBuffers->V();
}

CokeMachine::Remove() {
  fullBuffers->P();
  mutex->P();
  Remove coke from to the machine;
  count--;
  mutex->V();
  emptyBuffers->V();
}

Does the order of P matter? Order of V matter?

Implementing Semaphores

Semaphore::P() {
  while (0 > atomic_dec(&value)) {
    Put TCB on wait queue for semaphore;
    Switch(); // dispatch a ready thread
    atomic_inc(&value);
  }
}

Semaphore::V() {
  int notify = atomic_inc(&value);
  // atomic_inc returns new value
  if (notify <= 0) {
    Move a waiting thread to ready queue;
  }
}

value:
  1..k = Resource available
  0 = All resources used, no waiters
  <0 = -1 * number of waiters

The Problem with Semaphores

- Semaphores are used for dual purpose
  - Mutual exclusion
  - Conditional synchronization
- Difficult to read/develop code
- Waiting for condition is independent of mutual exclusion
  - Programmer needs to be clever about using semaphores

Semaphore::P() {
  if (0 > atomic_dec(&value)) {
    Put TCB on wait queue for semaphore;
    Switch(); // dispatch a ready thread
    atomic_inc(&value);
  }
}

Semaphore::V() {
  int notify = atomic_inc(&value);
  // atomic_inc returns new value
  if (notify <= 0) {
    Move a waiting thread to ready queue;
  }
}
Introducing Monitors

- Separate the concerns of mutual exclusion and conditional synchronization.
  - What is a monitor?
    - One lock, and zero or more condition variables for managing concurrent access to shared data.
  - General approach:
    - Collect related shared data into an object/module.
    - Define methods for accessing the shared data.
  - Monitors first introduced as a programming language construct.
    - Calling a method defined in the monitor automatically acquires the lock.
    - Examples: Mesa, Java (synchronized methods).
  - Monitors also define a programming convention.
    - Can be used in any language (C, C++, ...).

Monitors: Recap

- Lock acquire and release: often incorporated into method definitions on object.
  - E.g., Java’s synchronized methods.
  - Programmer may not have to explicitly acquire/release.
- But, methods on a monitor object do execute under mutual exclusion.
- Introduce idea of condition variable.

Critical Section: Monitors

- Basic idea:
  - Restrict programming model.
  - Permit access to shared variables only within a critical section.
- General program structure:
  - Entry section:
    - “Lock” before entering critical section.
    - Wait if already locked, or invariant doesn’t hold.
    - Key point: synchronization may involve wait.
  - Critical section code.
  - Exit section:
    - “Unlock” when leaving the critical section.
- Object-oriented programming style:
  - Associate a lock with each shared object.
  - Methods that access shared object are critical sections.
  - Acquire/release locks when entering/exiting a method that defines a critical section.

Remember Condition Variables

- Locks:
  - Provide mutual exclusion.
  - Support two methods:
    - Lock::Acquire() – wait until lock is free, then grab it.
    - Lock::Release() – release the lock, waking up a waiter, if any.
- Condition variables:
  - Support conditional synchronization.
  - Three operations:
    - Wait(): Release lock; wait for the condition to become true; reacquire lock upon return (Java wait()).
    - Signal(): Wake up a waiter, if any (Java notify()).
    - Broadcast(): Wake up all the waiters (Java notifyAll()).
  - Two semantics for implementation of wait() and signal():
    - Hoare monitor semantics.
    - Hansen (Mesa) monitor semantics.

So what is the big idea?

- (Editorial) Integrate idea of condition variable with language.
  - Facilitate proof.
  - Avoid error-prone boiler-plate code.

Coke Machine – Example Monitor

```c
int count = 0;
Condition notFull, notEmpty;

CokeMachine::Deposit()
{
  lock.acquire();
  while (count == n) {
    notFull.wait(&lock);
  }
  Add coke to the machine;
  count++;
  notEmpty.signal();
  lock.release();
}
```

```c
CokeMachine::Remove()
{
  lock.acquire();
  while (count < 0) {
    notEmpty.wait(&lock);
  }
  Remove coke from the machine;
  count--;
  notFull.signal();
  lock.release();
}
```
Every monitor function should start with what?
- A. wait
- B. signal
- C. lock acquire
- D. lock release
- E. signalAll

Hoare Monitors: Semantics
- Assume thread T1 is waiting on condition x
- Assume thread T2 is in the monitor
- Assume thread T2 calls x.signal
- T2 gives up monitor, T2 blocked
- T1 takes over monitor, runs
- T1 gives up monitor
- T2 takes over monitor, resumes

Example
T1
- fd(...)
- x.wait // T1 blocked
- x.signal // T2 continues
- T1 resumes
- T2 blocked!
- T2 resumes

Hansen [Mesa] Monitors: Semantics
- Assume thread T1 is waiting on condition x
- Assume thread T2 is in the monitor
- Assume thread T2 calls x.signal; wake up T1
- When T1 get a chance to run, T1 takes over monitor, runs
- T1 finishes, gives up monitor

Example:
fn1(…)
- x.wait // T1 blocked
- x.signal // T2 continues
- T1 resumes
- T1 finishes

Tradeoff

Hoare
- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- but
  - Inefficient implementation
  - Not modular – correctness depends on correct use and implementation of signal

Hansen
- Signal is only a hint that the condition may be true
- Need to check condition again before proceeding
- Can lead to synchronization bugs
- Used by most systems (e.g., Java)

Benefits:
- Efficient implementation
- Condition guaranteed to be true once you are out of while!

CokeMachine::Deposit(){
  lock.acquire();
  if (count == n) {
    notFull.wait(&lock); }
  Add coke to the machine;
  count++;
  notEmpty.signal();
  lock.release();
}

CokeTruck::Unload(){
  lock.acquire();
  while (soda.atDoor() != coke) {
    cokeAvailable.wait(&lock);
  }
  Unload soda closest to door;
  soda.pop();
  Signal availability for soda.atDoor();
  lock.release();
}

Problems with Monitors

Nested Monitor Calls
- What happens when one monitor calls into another?
  - What happens to CokeMachine:lock if thread sleeps in CokeTruck:Unlock?
  - What happens if truck unloader wants a coke?
Comparing Semaphores and Monitors

CokeMachine::Deposit()
{
  lock.acquire();
  while (count == n) {
    notFull.wait(&lock);
  }
  Add coke to the machine;
  count++;
  notEmpty.notify();
  lock.release();
}

CokeMachine::Remove()
{
  fullBuffers.P();
  mutex.P();
  Remove coke from the machine;
  count--;
  mutex.V();
  emptyBuffers.V();
}

Which is better?
A.Semaphore
B. Monitors

CokeMachine::Deposit()
{
  lock.acquire();
  while (count == n) {
    notFull.wait(&lock);
  }
  Add coke to the machine;
  count++;
  notEmpty.notify();
  lock.release();
}

CokeMachine::Remove()
{
  lock.acquire();
  while (count == 0) {
    notEmpty.wait(&lock);
  }
  Remove coke from the machine;
  count--;
  notFull.notify();
  lock.release();
}

Other Interesting Topics

- Exception handling
  - What if a process waiting in a monitor needs to time out?

- Naked notify
  - How do we synchronize with I/O devices that do not grab monitor locks, but can notify condition variables.

- Butler Lampson and David Redell, “Experience with Processes and Monitors in Mesa.”

Summary

- Synchronization
  - Coordinating execution of multiple threads that share data structures

- Past lectures:
  - Locks → provide mutual exclusion
  - Condition variables → provide conditional synchronization

- Today:
  - Semaphores
    - Introduced by Dijkstra in 1960s
    - Two types: binary semaphores and counting semaphores
    - Supports both mutual exclusion and conditional synchronization
  - Monitors
    - Separate mutual exclusion and conditional synchronization