Concurrency and Multi-threading

May 2, 2015

Overview

Implement Event Manager

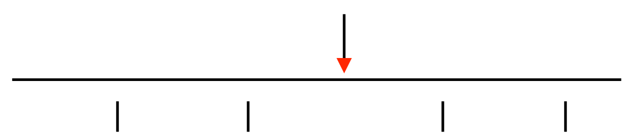
Basics

Producer-Consumer Problem

Implement Event Manger

Implement the following class.

```
// An event manager that can register callbacks,
// but only process callbacks after the event is triggered.
// Callbacks registered after the event are directly processed.
class EventManager {
 public:
  void RegisterForEvent(void (*callback)(void));
  void TriggerEvent();
}
```



```
// Example Usage
void foo() {
  EventManager em;
  em.RegisterForEvent(callback1); // callback1 not called.
  em.RegisterForEvent(callback2); // callback2 not called.
  em.TriggerEvent(); // callback1 and callback2 called.
  em.RegisterForEvent(callback3); // callback3 called.
  em.RegisterForEvent(callback4); // callback4 called.
}
```

Level 1

```
class EventManager {
 public:
  RegisterForEvent(void (*callback)(void));
  TriggerEvent();
 private:
  bool triggered_ = false;
  std::vector<void (*)(void)> registered_callbacks;
}

void EventManager::RegisterForEvent(void (*callback)(void)) {
  if (triggered_) {
    (*callback)();
  } else {
    registered_callbacks.push_back(callback);
  }
}
void EventManager::TriggerEvent() {
  triggered_ = true;
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

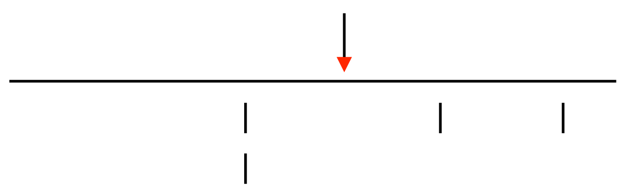
Support Multithreading

```
// Example Usage

// Event manager shared by all threads.
EventManager em;

void thread1() {
  em.RegisterForEvent(callback1);
}
void thread2() {
  em.RegisterForEvent(callback2);
}
void thread3() {
  em.TriggerEvent();
}
void thread4() {
  em.RegisterForEvent(callback3);
}
void thread5() {
  em.RegisterForEvent(callback4);
}
```

What’s the problem?



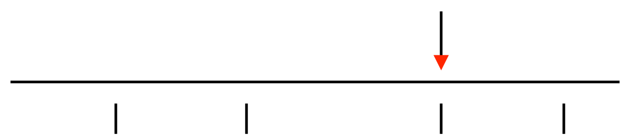
```
void EventManager::RegisterForEvent(void (*callback)(void)) {
  if (triggered_) {
    (*callback)();
  } else {
    registered_callbacks.push_back(callback);
  }
}
void EventManager::TriggerEvent() {
  triggered_ = true;
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

Level 2

```
class EventManager {
 public:
  RegisterForEvent(void (*callback)(void));
  TriggerEvent();
 private:
  bool triggered_ = false;
  mutex mu_;
  std::vector<void (*)(void)> registered_callbacks;
}


void EventManager::RegisterForEvent(void (*callback)(void)) {
  if (triggered_) {
    (*callback)();
  } else {
    mu_.lock();
    registered_callbacks.push_back(callback);
    mu_.unlock();
  }
}
void EventManager::TriggerEvent() {
  triggered_ = true;
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

Any other problems?



```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  if (triggered_) {
    (*callback)();
  } else {






    mu_.lock();
    registered_callbacks.push_back(callback);
    mu_.unlock();
  }
}
```

```
void EventManager::TriggerEvent() {




  triggered_ = true;
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

Level 3

```
class EventManager {
 private:
  mutex mu_trigger_;
  bool triggered_ = false;
  mutex mu_callbacks_;
  std::vector<void (*)(void)> registered_callbacks;
}
```

```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  mu_trigger_.lock();
  if (triggered_) {
    (*callback)();
  } else {
    mu_callbacks_.lock();
    registered_callbacks.push_back(callback);
    mu_callbacks_.unlock();
  }
  mu_trigger_.unlock();
}
```

```
void EventManager::TriggerEvent() {
  mu_trigger_.lock();
  triggered_ = true;
  mu_trigger_.unlock();
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

Any Problems?

```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  mu_trigger_.lock();
  if (triggered_) {
    (*callback)();
  } else {
    mu_callbacks_.lock();
    registered_callbacks.push_back(callback);
    mu_callbacks_.unlock();
  }
  mu_trigger_.unlock();
}
```

```
void EventManager::TriggerEvent() {
  mu_trigger_.lock();
  triggered_ = true;
  mu_trigger_.unlock();
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

1. Concurrent callbacks after the event will be serialized.
2. More seriously, a callback can register another event, causing deadlock on mu_trigger_.

Level 4

```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  mu_trigger_.reader_lock();
  if (triggered_) {
    (*callback)();
  } else {
    mu_callbacks_.lock();
    registered_callbacks.push_back(callback);
    mu_callbacks_.unlock();
  }
  mu_trigger_.reader_unlock();
}
```

```
void EventManager::TriggerEvent() {
  mu_trigger_.lock();
  triggered_ = true;
  mu_trigger_.unlock();
  int size = registered_callbacks.size();
  for (int i = 0; i < size; i++) {
    (*registered_callbacks[i])();
  }
}
```

Summary

General Steps

- Write single-thread version.
- Identify race conditions. (thread-safety)
- Add mutexes to ensure mutual exclusion.
- Identify deadlocks.
- Reduce unnecessary serializations to maximize concurrency.

*Another Solution

```
class EventManager {
 private:
  mutex mu_;
  condition_variable cv_;
}
```

```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  mu_.lock();
  cv_.wait(mu_);
  mu_.unlock();
  (*callback)();
}
```

```
void EventManager::TriggerEvent() {
  cv_.notify_all();
}
```

condition variable: wait until being notified.

- cv_.wait(mu_): **atomically** release mu_, blocks the current executing thread, and adds it to the **list** of threads waiting on cv_.
- cv_.notify() / cv_.notify_all(): unblock one/all waiting threads on cv_.

This solution is **WRONG**: RegisterForEvent() after TriggerEvent() will block forever.

* Another Solution (Correct)

```
class EventManager {
 private:
  mutex mu_;
  condition_variable cv_;
  bool wait_ = true;
}
```

```
void EventManager::RegisterForEvent(
    void (*callback)(void)) {
  mu_.lock();
  cv_.wait(mu_, []{return wait_});
  mu_.unlock();
  (*callback)();
}
```

```
void EventManager::TriggerEvent() {
  mu_.lock();
  wait_ = false;
  mu_.unlock();
  cv_.notify_all();
}
```

cv_.wait(mu_, pred):

Semantics: while (!pred()) {cv_.wait();}

Effects:

If pred() is false, no waiting. (useful for our case)

If notified spuriously (pred() is still true), continue to wait.

mu_:

Compare those two solutions: Asynchronous vs. Synchronous.

Concurrency

Concurrency: multiple sequences of execution execute at the same time or by time-sharing.

Communication between cooperating executions:

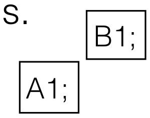
- shared-memory: light weight (like “broadcast”)
- message-passing: more scalable (allows P2P)
  - Sort of builds on shared-memory: think about the send/receive buffer.

Synchronization problem:

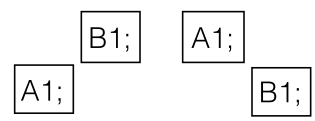
- Shared access with at least one write -> race condition

Synchronization

Synchronization: enforce constraints between cooperating executions.



- Basic synchronization patterns:
  - Signaling: A can execute A1 only after B executes B1.
  - Mutual exclusion: at most one of A and B can execute.



- Software synchronization primitives:
  - semaphore
  - mutex
    - busy-waiting mutex (i.e. spin-lock) (any advantage?)
    - non-busy-waiting mutex (i.e. mutex-lock)
  - **condition variable
  - **monitor
  - **reader-writer mutex (i.e. shared-exclusive mutex)
  - *atomic operations (bonus)

**Hardware support: atomic operations (test-and-set, compare-and-swap, fetch-and-add, etc).

**OS support for non-busy-waiting mutex; OS support for timeout.

Semaphore

- Extensively used for education purpose
  - Simple
  - Versatile
    - can be used to implement other primitives
    - **In practice, usually constructed from mutex.
- Semaphore is an integer with three methods:
  - Semaphore(int n): initialize
  - wait(): decrement n and block if n is negative
  - signal(): increment n and wake a blocked thread.

Semaphore Examples

Signaling

Rendezvous

- "barrier problem"

Mutual exclusion

- "critical section problem"

**Harder problems:

- N-thread barrier
  - N-semaphore solution
  - 2-semaphore solution
- Reader-writer problem
- Bounded-buffer problem
- Condition variable

More:"The little book of semaphores" by Allen B. Downey

```
Semaphore bArrived(0);
```

```
bArrived.wait();
A1
```

```
B1;
bArrived.signal();
```

```
Semaphore aArrived(0);
Semaphore bArrived(0);
```

```
A1;
aArrived.signal();
bArrived.wait()
A2
```

```
B1;
bArrived.signal();
aArrived.wait();
B2
```

```
Semaphore mutex(1);
```

```
mutex.wait();
A1
mutex.signal();
```

```
mutex.wait();
B1;
mutex.signal();
```

Deadlock

Live-lock

Producer-Consumer Problem / Bounded-Buffer Problem

```
class Buffer{
 public:
  void produce(Item item);
  Item consume();
 private:
  Item buffer[BUFFER_SIZE];
}
```

```
class Buffer{
 public:
  void produce(Item item);
  Item consume();
 private:
  Item buffer[BUFFER_SIZE];
  int produce_pos = 0;
  int consume_pos = 0;
}
```

```
void Buffer::produce(Item item) {
  while ()
}
```