Synchronization

Condition Variables

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Introduction

- Condition variables
  - Allow threads to **efficiently wait** for a condition to become true **without** constantly polling (i.e.., performing busy-waiting)
    - Allow threads to wait in **a race-free way** for an **arbitrary condition to occur**
    - Provide a place for threads to **Rendez-vous**
    - They facilitate communication and coordination between threads based on the state of shared data

Introduction

- POSIX, C11, and C++11 have similar primitives
  - They are based on the following key elements
    - There's some **data shared** between threads
      - There's a specific **condition** related to this shared data that one or more threads might need to wait for
    - A mutex is used to **protect** access to the shared data, and the condition check
    - The **condition variable** manages the waiting and notifying of threads
      - The wait operation needs the ability to **atomically**
        - **Unlock** the mutex **while waiting**
        - **Re**-lock it upon waking up if the condition is still false

CVs in C++

- The C++ standard library defines
  - The class std::condition_variable
  - In the header <condition_variable>
- The library has the following member functions

Type\	Meaning
wait()	Takes a reference to a std::unique_lock that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable.
notify_one()	Notify a single waiting thread, mutex does not need to be held by the caller.
notify_all()	Notify all waiting threads, mutex does not need to be held by the caller.

CV Usage

- Define a CV

```
#include <thread>
#include <mutex>
#include <condition_variable>

...

std::mutex mtx;
std::condition_variable cv;
bool done = false;
```

CV must be used with a mutex and a condition related to a shared data

CV Usage

❖ Notify threads

Lock the mutex to protect the CV

Modify shared data

```
1. mtx.lock();
2. done = true;
3. cv.notify_one();
4. mtx.unlock();
```

Notify one or all threads waiting on the CV

Release the mutex

- Function "notify" is used to notify threads that a condition (**done**, in this case) has been satisfied
  - **notify_one** will wake up at least one thread waiting on the condition
  - **notify_all** will wake up all threads waiting on the condition

CV Usage

- Waiting threads

Lock the mutex to protect the CV

```
1. mtx.lock();
   while (!done)
     cv.wait(mtx);
   mtx.unlock();
```

1. The thread obtains the mutex
   - The mutex must be locked when we run cv.wait
   - The mutex will be released in the epilogue

CV Usage

- Waiting threads

Check the condition

```
1. mtx.lock();
2. while (!done)
      cv.wait(mtx);
   mtx.unlock();
```

2. The thread tests the predicate; if the predicate is

CV Usage

- Waiting threads

Check the condition

```
1. mtx.lock();
2. while (!done)
3.   cv.wait(mtx);
   mtx.unlock();
```

The wait on the CV atomically releases the lock ahd puts the thread to sleep

2. The thread tests the predicate; if the predicate is
3. **Satisfied**, the thread executes the **wait** on the CV which **releases** the mutex and it awaits on the condition variable
   - The mutex must be released to allow other threads to check the condition
   - When the condition variable is signaled, the thread wakes up and the predicate is checked again

CV Usage

- Waiting threads

```
1. mtx.lock();
2. while (!done)
3.   cv.wait(mtx);
4. mtx.unlock();
```

Release the mutex

2. The thread tests the predicate; if the predicate is
   4. **Not satistied**, the thread goes on and it unlock the mutex

Why do we need this level of complexity?

The problem is that **there is no memory** on cv

Example

- Suppose **join** does not exist and we want to wait the termination of a thread

```
#include <thread>
#include <iostream>

void child() {
  // ...
  done = 1;
  return;
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  t.join();
  return 0;
}
```

No join !

Example

Solution with spin-lock
(never use it)

- We can use polling
  - This is grossly inefficient as it wastes CPU cycles

```
#include <thread>
#include <iostream>

void child() {
  // ...
  done = 1;
  return;
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  while (done == 0); // spin
  return 0;
}
```

No join !

Example

We can use a CV

```
std::mutex mtx;
std::condition_variable cv;
bool done = false;
```

Initialization

```
void child() {
  mtx.lock();
  done = true;
  cv.notify_one();
  mtx.unlock();
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  mtx.lock();
  while (!done)
    cv.wait(mtx);
  mtx.unlock();
  return 0;
}
```

Thread join

Does it work?

Example

```
void child() {
  mtx.lock();
  done = true;
  cv.notify_one();
  mtx.unlock();
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  mtx.lock();
  while (!done)
    cv.wait(mtx);
  mtx.unlock();
  return 0;
}
```

The parent runs first:

1. It will acquire mtx, check "done", and as done=0, it will go to sleep releasing mtx
2. The child will run, set done to true, signal the cv, release mtx, and quit
3. The parent will be woken-up by the signal with the mutex locked, unlock the mutex, check cv, proceed, check "done", proceed, return

In this case the "job" is done by the **cv** on which the parent waits

Example

```
void child() {
  mtx.lock();
  done = true;
  cv.notify_one();
  mtx.unlock();
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  mtx.lock();
  while (!done)
    cv.wait(mtx);
  mtx.unlock();
  return 0;
}
```

The child runs first:

1. It will set done to true, signal the cv, unlock the mutex, and terminate. As there is no one waiting, the signal on cv **has no effect**
2. The parent will get to the critical section, lock the mutex, as done==true it will go on, unlock the mutex, and terminate

In this case the "job" is done by the variable **done** as the parent never does a wait

Example

Is the variable **done** required?

```
void child() {
  mtx.lock();

  cv.notify_one();
  mtx.unlock();
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  mtx.lock();

    cv.wait(mtx);
  mtx.unlock();
  return 0;
}
```

The code is broken.
In fact, **iff** the child runs first:

1. It will signal the cv but as there is no one waiting, the signal has no effect
2. The parent will get to the critical section, lock the mutex, and wait on cv forever

Then, variable **done** records the status the threads are interested in knowing

Example

Is the **mutex** m required?

```
void child() {

  done = true;
  cv.notify_one();

}

int main(int argc, char *argv[]) {
  std::thread t(child);

  while (!done)
    cv.wait(    );

  return 0;
}
```

The code is broken.

There is a subtle **race condition**:

1. The parent runs first, and it checks done. As done==false it is ready to go to sleep on the cv.wait, but before going on the wait the child runs
2. The child set done to true and signal cv. But the parent is not waiting, thus **the signal is lost**
3. The parent will go on the wait and wait forever

This is not a correct implementation, because there is no mutex. **Let us suppose it is correct just for the sake of the example**

Using the mutex may not be always required around the signal but it is **always** required around the wait

Example

Is the **while** required or we can use an if?

```
void child() {
  mtx.lock();
  done = true;
  cv.notify_one();
  mtx.unlock();
}

int main(int argc, char *argv[]) {
  std::thread t(child);
  mtx.lock();
  if (!done)
    cv.wait(mtx);
  mtx.unlock();
  return 0;
}
```

The code is broken.
More than one thread may be awoken, because **notify_all** has been called or a race between two processors simultaneously woke two threads. The first thread locking the mutex will block all other threads. Thus, the predicate may have changed when the second thread gets the mutex. In general, whenever a CV returns, the thread should reevaluate the predicate

In other words, the C and C++ standard allows CVs to wake-up spuriously; thus, the condition must be re-checked

Signaling a thread wakes-it up but there is **no** guarantee that when it runs the **state** will still **be the same. The while is required.**

Summary I

- When using a condition variable
  - The mutex
    - Is used to protect the condition variable
    - Must be locked before waiting
  - The wait will "atomically" unlock the mutex, allowing others access to the condition variable
  - When the condition variable is signalled (or broadcast to) one or more of the threads on the waiting list will be woken up and the mutex will be magically locked again for that thread

Summary II

- Condition variables allow a thread to notify other threads that something happened
  - A condition variable relieves the user of the burden of polling some condition and waiting for the condition without wasting resources
  - They avoid busy waiting

```
while (done == 0);
```

```
1. mtx.lock();
2. while (!done)
3.    cv.wait(mtx);
4. mtx.unlock();
```

- Used when one or more threads are waiting for a specific condition to come true

Summary III

- Condition variables versus semaphores
  - Semaphores are very general and sophisticated
    - They are expensive
    - It is essentially a counter with a mutex (and a waiting queue)
    - Used for general synch schemes and to control the access to a finite number of resources
  - A condition variable represents a condition related to a shared data
    - It needs a mutex
    - Used when to synchronize threads based on the condition of the shared data

Exercise 01

- Only C++20 supports semaphores
  - In contrast to a mutex a semaphore is **not** bound to a thread
  - This means that the acquire and release call of a semaphore can happen on different threads
- Suppose C++20 does not exist yet
- Implement a C++ semaphore using a mutex and a CV

Solution 01

Solution with polling
Never use it

```
struct Semaphore {
  int count;
  mutex m;
  ...
}
```

Constructor

```
Semaphore (int n) {
  count = n;
  return;
}
```

At most **n** workers in the critical section

```
void sem_wait() {
  while (1) {
    while (count <= 0) {}
    m.lock();
    if (count <= 0) {
      m.unlock();
      continue;
    }
    count--;
    m.unlock();
    break;
  }
}
```

Polling

Polling wait

Re-check after aquiring the lock

If the sem cannot be acquired, cycle (wait) again

```
void sem_signal () {
  m.lock();
  count++;
  m.unlock();
}
```

Solution 02

Solution with 2 mutexes
BUGGY

```
struct Semaphore {
  int count;
  mutex m, wait;
  ...
}
```

Constructor

```
Semaphore (int n) {
  count = n;
  return;
}
```

At most **n** workers in the critical section

The first mutex is to protect the CS, the second one to make threads wait

```
void sem_wait() {
  m.lock();
  cout--;
  if (count < 0) {
    m.unlock();
    wait.lock();
  } else {
    m.unlock();
  }
}
```

Buggy **because** locks have a unique owner

```
void sem_signal () {
  m.lock();
  count++;
  if (count <= 0) {
    wait.unlock();
  }
  m.unlock();
}
```

Solution 03

Solution with a mutex and a condition variable

```
#include <mutex>
#include <condition_variable>
using ...
struct Semaphore {
  int count;
  mutex m;
  condition_variable cv;
  ...
}
```

Constructor

```
Semaphore (int n) {
  count = n;
  return;
}
```

At most **n** workers in the critical section

```
void sem_wait() {
  m.lock();
  count--;
  while (count <= 0) {
    cv.wait(m);
  }
  m.unlock();
}
```

Predicate

CV

Mutex

```
void notify( int tid ) {
  m.lock();
  count++;
  cv.notify_one();
  m.unlock();
}
```

Predicate

CV

Mutex

Exercise 02

Exam of January 19, 2021

- Write a C++ program in which
  - A thread **admin** initializes an integer variable **var** to 10 and then waits for 5 **adder** threads
  - Each **adder** thread adds a random number between 1 and 2 to **var**
  - The program terminates when
    - All threads finish or
    - When **var** becomes equal or greater than 15
  - When the program ends the **admin** thread is awakened and prints the final value

Solution

Premises

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <fstream>

std::mutex m;
std::condition_variable adminCV;
std::condition_variable adderCV;
int var = 0;

void admin_f();
void adder_f();
```

Solution

Main

```
int main() {
  std::vector<std::thread> adders;

  // Run admin thread
  std::thread admin_t(admin_f);
  for(int i=0; i<5; i++){
    // Makes the seed different for each thread
    srand ((unsigned)time(NULL));
    // Run adder threads
    adders.emplace_back(std::thread (adder_f));
  }
  for(auto &i: adders) {
    i.join();
  }
  adminCV.notify_one();
  admin_t.join();
  return 0;
}
```

Run thread admin

Run three threads adder_f

Wait for them

Wake-up admin_f when all adders have finished

Solution

Thread admin

Mutex

```
void admin_f () {
  std::unique_lock<std::mutex> admin_lock{m};
  var = 10;
  cout << "Variable initialized to 10" << endl;

  // Notify adders
  adderCV.notify_all();

  // Wait adders
  while (var < 15)
    adminCV.wait(admin_lock);

  cout << "Variable value = " << var << endl;
}
```

Predicate

CV

Mutex

Set the condition for the adder (var=10) and wakes them (adderCV.notify_all).
Then, is waits on its predicated and CV (adminCV)

Solution

```
void adder_f () {
  std::unique_lock<std::mutex> adder_lock{m};
  // Wait for initialization
  while (var == 0) {
    // Unlock the mutex
    adderCV.wait(adder_lock);
  }
  // If var is over the threshold, notify admin and exit
  if (var < 15) {
    int n = 1 + rand() % 2;
    var +=  n;
    cout << "Added = " << n << " Sum = " << var << endl;
    if (var >= 15) {
      adminCV.notify_one();
    }
  }
  return;
}
```

Mutex

Adder threads

Predicate 1

CV 1

Predicate 2

CV 2

Exercise 03

Exam of January 16, 2023

- Write a C++ program that operates on a vector of integers v managing the synchronization of the following threads
  - A thread **writer** adds a random number in the range [1,10] to the vector every 3 seconds
  - A thread **ui** constantly checks for user input from the console and update the global variable **command** every 1 second
  - A thread **worker** executes the commands specified in the variable **command** when thread **ui** wakes it

Exercise 03

- The valid commands are the following
  - 0 terminates the program
  - 1 displays all elements in v
  - 2 displays the last element of v
  - 3 deletes all elements in v

Solution

```
#include <iostream>
#include <thread>
#include <vector>
#include <condition_variable>

using namespace std;

bool running = true;
int command = -1;
condition_variable cv;
mutex mx;
vector<int> vt;

void writer();
void ui();
void worker()
```

Runs and waits threads

```
int main(){
  cout << "START" << endl;
  thread t_wr(writer);
  thread t_u(ui);
  thread t_w(worker);
  t_w.join();
  t_u.join();
  cv.notify_one();
  t_wr.join();
  cout << "END" << endl;
}
```

Solution

The **writer** adds a random number in the range [1,10] to the vector every 5 seconds

Insert a new value in the array every 3 seconds

```
void writer(){
  while(running) {
    this_thread::sleep_for(chrono::milliseconds(3000));
    unique_lock<mutex> l_w(mx);
    vt.emplace_back(rand()%10+1);
    l_w.unlock();
  }
  return;
}
```

Add a value in the range [1,10]

Solution

The ui checks for user input from the console and update the global variable command

Let other threads running too

```
void ui(){
  while(running) {
    this_thread::sleep_for(chrono::milliseconds(1000));
    cout << "Command (0,1,2,3): ";
    unique_lock<mutex> l_ui(mx);
    cin >> command;
    if (command==0){
      running = false;
    }
    cv.notify_one();
    l_ui.unlock();
  }
}
```

Read user commands

The variable "running" should be protected here, in the writer, and in the worker

Solution

```
void worker(){
  while(running) {
    unique_lock<mutex> l_r(mx);
    while(vt.empty() || command==-1)
      cv.wait(l_r);
    switch (command){
      case 1: cout << " ### Current elements: ";
        for(auto &e: vt)
          cout << e << " ";
        cout << endl;
        break;
      case 2: cout << " ### Last element: " << vt.back() << endl;
        break;
      case 3: cout << " ### All elements removed" << endl;
        vt.clear();
        break;
    }
    command = -1;
    l_r.unlock();
  }
}
```

The worker executes the commands specified in the variable command when thread ui wakes it

Execute commands in **command** (terminates, display, display, delete)

Reset command to run cv.wait again

Exercise 04

- Implement a Producer-Consumer scheme
  - The main thread runs NP producers and NC consumers
    - Producers and consumers communicate using **a single variable**
  - Each producer stores a predefined number of (random) integers in **buffer**
  - Each consumer displays (on standard output) a predefined number of integers, reading them from **buffer**
  - Use condition variables to perform synchronization

Solution

Premises

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

// Shared data structure
std::queue<int> dataQueue;
std::mutex mtx;
std::condition_variable cv;
bool stop = false;
```

Buffer

Initially empty

Solution

```
void producer() {
  for (int i = 0; i < 10; ++i) {
    {
    std::lock_guard<std::mutex> lock(mtx);
    dataQueue.push(i);
    }
    std::cout << "Produced: " << i << std::endl;
    cv.notify_one(); // Notify consumer
    // Simulate production time
    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
  {
    std::lock_guard<std::mutex> lock(mtx);
    stop = true; // Signal end of production
  }
  cv.notify_one(); // Notify consumer about end
}
```

Solution

```
void consumer() {
  while (true) {
    std::unique_lock<std::mutex> lock(mtx);
    // Wait for data or stop signal
    cv.wait(lock, []{ return !dataQueue.empty() || stop; });

    if (stop && dataQueue.empty()) {
      break;
      // Exit loop if no more data and stop signal received
    }

    int data = dataQueue.front();
    dataQueue.pop();
    // Unlock before processing to allow producer to continue
    lock.unlock();
    std::cout << "Consumed: " << data << std::endl;
    }
}
```

Solution

```
int main() {
    std::thread producerThread(producer);
    std::thread consumerThread(consumer);

    producerThread.join();
    consumerThread.join();

    return 0;
}
```

Exercise 05

- Implement the First Reader-Writer scheme using
  - Condition Variables

Readers

```
wait (meR);
  nR++;
  if (nR==1)
    wait (w);
signal (meR);
...
wait (meR);
  nR--;
  if (nR==0)
    signal (w);
signal (meR);
```

Initialization

```
nR = 0;
init (meR, 1);
init (meW, 1);
init (w, 1);
```

Writers

```
wait (meW);
wait (w);
...
signal (w);
signal (meW);
```

Logic behavior

Solution

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
int readersCount = 0;
bool writerActive = false;

// Function Prototypes ...
void reader();
void writer();
```

Solution

```
int main() {
    std::thread readers[5];
    std::thread writers[5];

    for (int i = 0; i < 5; ++i) {
        readers[i] = std::thread(reader);
        writers[i] = std::thread(writer);
    }

    for (int i = 0; i < 5; ++i) {
        readers[i].join();
        writers[i].join();
    }

    return 0;
}
```

Solution

```
void reader() {
  {
    std::lock_guard<std::mutex> lock(mtx);
    while (writerActive) {
      cv.wait(mtx);            // Wait for writer to finish
    }
    readersCount++;
  }
  std::cout << "Reader is reading..." << std::endl;
  {
    std::lock_guard<std::mutex> lock(mtx);
    readersCount--;
    if (readersCount == 0) {
      // Notify writer that readers have finished
      cv.notify_all();
    }
  }
}
```

Solution

```
void writer() {
  {
     std::unique_lock<std::mutex> lock(mtx);
     while (readersCount > 0 || writerActive) {
       // Wait for readers to finish and no other writer
       cv.wait(lock);
     }
     writerActive = true;
  }
  // Write operation here
  std::cout << "Writer is writing..." << std::endl;
  {
    std::lock_guard<std::mutex> lock(mtx);
    writerActive = false;
    // Notify readers and other writers
    cv.notify_all();
  }
}
```