# 进程和多线程

## 线程

多线程编程是为了提高程序的执行效率和响应能力。C++ 标准库和其他第三方库提供了多种多线程技术和工具来实现并发编程。下面是一些常用的多线程技术及其用法示例：

## （1）多线程的通信

### ****1.互斥量（Mutexes）****

* **定义**：互斥量用于防止多个线程同时访问共享资源。

#include <iostream>#include <thread>#include <mutex>

std::mutex mtx;

void printThread(int id) {

std::lock\_guard<std::mutex> lock(mtx);

std::cout << "Thread " << id << " is running\n";

}

int main() {

std::thread t1(printThread, 1);

std::thread t2(printThread, 2);

t1.join();

t2.join();

return 0;

}

### 2 .****条件变量（Condition Variables）****

* · **定义**：条件变量用于在线程之间同步某个条件的发生。

#include <iostream>#include <thread>#include <condition\_variable>

std::mutex mtx;

std::condition\_variable cv;bool ready = false;

void printThread(int id) {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, []{ return ready; });

std::cout << "Thread " << id << " is running\n";

}

int main() {

std::thread t1(printThread, 1);

std::thread t2(printThread, 2);

{

std::lock\_guard<std::mutex> lock(mtx);

ready = true;

}

cv.notify\_all();

t1.join();

t2.join();

return 0;

}

### ****3.信号量（Semaphores）****

* **定义**：信号量是一个计数器，用于控制多个线程对共享资源的访问。

#include <iostream>#include <thread>#include <semaphore.h>

std::binary\_semaphore semaphore(1);

void printThread(int id) {

semaphore.acquire();

std::cout << "Thread " << id << " is running\n";

semaphore.release();

}

int main() {

std::thread t1(printThread, 1);

std::thread t2(printThread, 2);

t1.join();

t2.join();

return 0;

}

### ****4.事件（Events）****

* **定义**：事件用于线程间的同步，通常是通过信号来通知事件的发生。

#include <iostream>#include <thread>#include <mutex>#include <condition\_variable>

std::mutex mtx;

std::condition\_variable cv;bool eventOccurred = false;

void waitForEvent(int id) {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, []{ return eventOccurred; });

std::cout << "Thread " << id << " received event\n";

}

void triggerEvent() {

std::lock\_guard<std::mutex> lock(mtx);

eventOccurred = true;

cv.notify\_all();

}

int main() {

std::thread t1(waitForEvent, 1);

std::thread t2(waitForEvent, 2);

std::this\_thread::sleep\_for(std::chrono::seconds(1));

triggerEvent();

t1.join();

t2.join();

return 0;

}

### ****5.消息队列（Message Queues）****

* **定义**：消息队列允许线程通过消息传递进行通信，每个消息带有一个类型标识。

**实现**（可通过C++标准库或第三方库实现，例如Boost）：

#include <iostream>#include <thread>#include <queue>#include <mutex>#include <condition\_variable>

std::queue<int> messageQueue;

std::mutex mtx;

std::condition\_variable cv;

void producer() {

for (int i = 0; i < 10; ++i) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

{

std::lock\_guard<std::mutex> lock(mtx);

messageQueue.push(i);

}

cv.notify\_one();

}

}

void consumer() {

while (true) {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, []{ return !messageQueue.empty(); });

int msg = messageQueue.front();

messageQueue.pop();

lock.unlock();

std::cout << "Consumed: " << msg << std::endl;

if (msg == 9) break; // Exit condition

}

}

int main() {

std::thread prod(producer);

std::thread cons(consumer);

prod.join();

cons.join();

return 0;

}

## （2）多线程技术

### 1.互斥量（Mutexes）

std::mutex 用于保护共享资源，防止多个线程同时访问。

**互斥量（Mutex, Mutual Exclusion）** 是多线程编程中用于防止多个线程同时访问共享资源的一种同步原语。它确保在任一时刻，只有一个线程能够进入临界区，从而避免竞态条件和数据不一致的问题。

##### 1. ****互斥量的基本概念****

**互斥**：互斥量用于保护临界区，即对共享资源的访问被互斥量保护起来，确保同一时刻只有一个线程能够访问。

**锁定（Lock）**：线程在进入临界区前需要获得互斥量的锁，如果锁已经被其他线程持有，当前线程将被阻塞，直到锁被释放。

**解锁（Unlock）**：当线程完成了对共享资源的操作后，需要释放互斥量，以便其他等待线程可以获得锁，继续执行。

##### 2. ****互斥量的使用技术****

互斥量的主要作用是解决多线程环境中的同步问题。它通过确保同一时刻只有一个线程进入临界区，避免了多个线程同时修改共享资源而导致的不确定行为。

##### ****1. 使用互斥量的步骤****

在使用互斥量时，通常涉及以下几个步骤：

1. **创建互斥量**：在程序开始时，通常在全局或类中定义一个互斥量。
2. **锁定互斥量**：线程在进入临界区前，首先要获取互斥量的锁。
3. **访问共享资源**：在成功获取锁后，线程可以安全地访问或修改共享资源。
4. **解锁互斥量**：完成对共享资源的操作后，线程必须释放互斥量，允许其他等待的线程获取锁。

示例：

#include <iostream>#include <thread>#include <mutex>

std::mutex mtx;

void printThread(int id) {

std::lock\_guard<std::mutex> lock(mtx); // 加锁

std::cout << "Thread " << id << " is running\n";

}

int main() {

std::thread t1(printThread, 1);

std::thread t2(printThread, 2);

t1.join();

t2.join();

return 0;

}

##### 3. ****互斥量使用中的常见问题****

###### ****1. 死锁（Deadlock）****

* **问题**：如果一个线程在持有互斥量的同时尝试获取另一个已经被锁定的互斥量，就可能导致死锁，两个或多个线程互相等待对方释放锁，导致程序无法继续执行。
* **解决方案**：使用正确的锁顺序，避免交叉锁定；使用 std::timed\_mutex 或 std::try\_lock 允许线程尝试获取锁并设置超时时间。

###### ****2. 优先级反转（Priority Inversion）****

* **问题**：在具有优先级调度的系统中，低优先级线程持有锁，高优先级线程需要等待，导致系统性能下降。
* **解决方案**：使用优先级继承协议（Priority Inheritance Protocol），在这种协议下，当高优先级线程等待低优先级线程释放锁时，低优先级线程会临时继承高优先级，以加快锁的释放。

###### ****3. 竞争条件（Race Condition）****

* **问题**：如果未正确使用互斥量，多个线程同时访问共享资源，可能导致竞态条件，产生不确定的行为。
* **解决方案**：确保所有对共享资源的访问都被正确的互斥量保护。

#### .锁机制

多线程的锁机制属于多线程编程中的**同步技术**（Synchronization）。锁机制是用来控制多个线程对共享资源的访问，确保同一时间只有一个线程能够访问或修改该资源，从而避免**竞态条件**（Race Condition）和**数据竞争**（Data Race）。

##### 1. 锁机制的主要作用

锁机制的主要作用是保护临界区（Critical Section），即那些被多个线程同时访问时可能导致数据不一致或不稳定的代码区域。通过使用锁，可以确保在任何时刻，临界区内的代码只会被一个线程

### 2. 条件变量（Condition Variables）

std::condition\_variable 用于线程之间的同步，等待某个条件满足。

#### 示例：

#include <iostream>#include <thread>#include <mutex>#include <condition\_variable>

std::mutex mtx;

std::condition\_variable cv;bool ready = false;

void printThread(int id) {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, []{ return ready; }); // 等待条件满足

std::cout << "Thread " << id << " is running\n";

}

int main() {

std::thread t1(printThread, 1);

std::thread t2(printThread, 2);

std::this\_thread::sleep\_for(std::chrono::seconds(1)); // 模拟延迟

{

std::lock\_guard<std::mutex> lock(mtx);

ready = true;

}

cv.notify\_all(); // 通知所有等待的线程

t1.join();

t2.join();

return 0;

}

### 3.原子操作（Atomic Operations）

std::atomic 提供了对基本数据类型的原子操作，确保线程安全。

#### 示例：

#include <iostream>#include <thread>#include <atomic>

std::atomic<int> counter(0);

void incrementCounter() {

for (int i = 0; i < 1000; ++i) {

++counter; // 原子操作

}

}

int main() {

std::thread t1(incrementCounter);

std::thread t2(incrementCounter);

t1.join();

t2.join();

std::cout << "Final counter value: " << counter << std::endl;

return 0;

}

### 4期望（Promise）与未来（Future）

std::promise 和 std::future 用于线程间值传递和同步。

多线程编程中的期望（Promise）和未来（Future）是C++标准库中提供的机制，用于线程间的同步和通信，特别适合于异步操作。std::promise允许一个线程设置一个值，而另一个线程可以通过std::future来获取这个值。这个机制保证了线程间的值传递和同步。

使用场景

* **异步计算**：一个线程启动一个长时间运行的计算任务，并在另一个线程中获取计算结果。
* **任务提交与结果获取**：主线程提交多个任务给子线程，并在需要时获取子线程的执行结果。

#### 示例：

#include <iostream>#include <thread>#include <future>

void setPromiseValue(std::promise<int>& prom) {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // 模拟耗时操作

prom.set\_value(42); // 设置Promise的值

}

int main() {

std::promise<int> prom; // 创建Promise对象

std::future<int> fut = prom.get\_future(); // 获取与Promise关联的Future对象

std::thread t(setPromiseValue, std::ref(prom));

std::cout << "Waiting for value from child thread..." << std::endl;

int result = fut.get(); // 阻塞直到Promise设置了值

std::cout << "Value from child thread: " << result << std::endl;

t.join(); // 等待子线程完成

return 0;

}

使用例子

#### 基本用法

#### 示例 1：基本的Promise和Future

一个简单的例子展示如何使用std::promise和std::future进行线程间的值传递。

cpp

复制代码

#include <iostream>#include <thread>#include <future>

// 函数运行在子线程，设置Promise值void setPromiseValue(std::promise<int>& prom) {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // 模拟耗时操作

prom.set\_value(42); // 设置Promise的值

}

int main() {

std::promise<int> prom; // 创建Promise对象

std::future<int> fut = prom.get\_future(); // 获取与Promise关联的Future对象

// 创建并启动子线程

std::thread t(setPromiseValue, std::ref(prom));

// 在主线程等待并获取Future的值

std::cout << "Waiting for value from child thread..." << std::endl;

int result = fut.get(); // 阻塞直到Promise设置了值

std::cout << "Value from child thread: " << result << std::endl;

t.join(); // 等待子线程完成

return 0;

}

#### 示例 2：使用std::async

std::async可以简化异步操作的代码，它启动一个异步任务并返回一个std::future对象。

cpp

复制代码

#include <iostream>#include <future>

// 异步函数int asyncFunction() {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // 模拟耗时操作

return 42;

}

int main() {

// 启动异步任务

std::future<int> fut = std::async(std::launch::async, asyncFunction);

// 在主线程等待并获取Future的值

std::cout << "Waiting for value from async task..." << std::endl;

int result = fut.get(); // 阻塞直到异步任务完成并返回值

std::cout << "Value from async task: " << result << std::endl;

return 0;

}

#### 示例 3：Promise与Exception

使用std::promise可以传递异常，这在处理异步错误时非常有用。

cpp

复制代码

#include <iostream>#include <thread>#include <future>#include <stdexcept>

// 异常处理函数void setPromiseException(std::promise<int>& prom) {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // 模拟耗时操作

try {

throw std::runtime\_error("Something went wrong!");

} catch (...) {

prom.set\_exception(std::current\_exception()); // 设置Promise的异常

}

}

int main() {

std::promise<int> prom; // 创建Promise对象

std::future<int> fut = prom.get\_future(); // 获取与Promise关联的Future对象

// 创建并启动子线程

std::thread t(setPromiseException, std::ref(prom));

// 在主线程等待并获取Future的值

try {

std::cout << "Waiting for value or exception from child thread..." << std::endl;

int result = fut.get(); // 阻塞直到Promise设置了值或异常

std::cout << "Value from child thread: " << result << std::endl;

} catch (const std::exception& e) {

std::cout << "Exception from child thread: " << e.what() << std::endl;

}

t.join(); // 等待子线程完成

return 0;

}

### 5异步（Async）

std::async 启动异步任务，并返回 std::future 对象来获取任务结果。

#### 示例：

#include <iostream>#include <future>

int asyncFunction() {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // 模拟耗时操作

return 42;

}

int main() {

std::future<int> fut = std::async(std::launch::async, asyncFunction);

std::cout << "Waiting for value from async task..." << std::endl;

int result = fut.get(); // 阻塞直到异步任务完成并返回值

std::cout << "Value from async task: " << result << std::endl;

return 0;

}

### 7，线程池（Thread Pool）

虽然C++标准库没有提供线程池的直接实现，但可以使用第三方库（如Boost）或者自己实现。

#### 示例：

#include <iostream>#include <vector>#include <thread>#include <queue>#include <functional>#include <mutex>#include <condition\_variable>

class ThreadPool {public:

ThreadPool(size\_t threads);

~ThreadPool();

template<class F, class... Args>

auto enqueue(F&& f, Args&&... args) -> std::future<typename std::result\_of<F(Args...)>::type>;

private:

std::vector<std::thread> workers;

std::queue<std::function<void()>> tasks;

std::mutex queue\_mutex;

std::condition\_variable condition;

bool stop;

};

inline ThreadPool::ThreadPool(size\_t threads) : stop(false) {

for(size\_t i = 0; i < threads; ++i)

workers.emplace\_back(

[this] {

for(;;) {

std::function<void()> task;

{

std::unique\_lock<std::mutex> lock(this->queue\_mutex);

this->condition.wait(lock,

[this]{ return this->stop || !this->tasks.empty(); });

if(this->stop && this->tasks.empty())

return;

task = std::move(this->tasks.front());

this->tasks.pop();

}

task();

}

}

);

}

template<class F, class... Args>auto ThreadPool::enqueue(F&& f, Args&&... args)

-> std::future<typename std::result\_of<F(Args...)>::type>{

using return\_type = typename std::result\_of<F(Args...)>::type;

auto task = std::make\_shared<std::packaged\_task<return\_type()>>(

std::bind(std::forward<F>(f), std::forward<Args>(args)...)

);

std::future<return\_type> res = task->get\_future();

{

std::unique\_lock<std::mutex> lock(queue\_mutex);

if(stop)

throw std::runtime\_error("enqueue on stopped ThreadPool");

tasks.emplace([task](){ (\*task)(); });

}

condition.notify\_one();

return res;

}

inline ThreadPool::~ThreadPool() {

{

std::unique\_lock<std::mutex> lock(queue\_mutex);

stop = true;

}

condition.notify\_all();

for(std::thread &worker: workers)

worker.join();

}

// 使用线程池int main() {

ThreadPool pool(4);

auto result1 = pool.enqueue([](int answer) { return answer; }, 42);

auto result2 = pool.enqueue([](int a, int b) { return a + b; }, 5, 7);

std::cout << "Result 1: " << result1.get() << std::endl;

std::cout << "Result 2: " << result2.get() << std::endl;

return 0;

}

## 二，进程

进程间通信（Inter-Process Communication，IPC）是指在操作系统中，多个进程之间通过某种机制进行数据交换或信息传递的技术。由于进程是相互独立的，在不同的地址空间中运行，因此需要专门的机制来实现它们之间的通信。下面是几种常见的进程间通信方法及其使用场景：

### 进程间通信（IPC）方法

### ****1.管道（Pipes）****

* 1. **定义**：管道是一种半双工的通信方式，数据只能单向流动，且只能在父子进程之间使用。

#### ****使用场景：****

* **简单数据传输**：适用于父子进程之间的简单数据传输，例如在Unix/Linux系统中的命令行管道（|），将一个程序的输出作为另一个程序的输入。
* **线性数据流**：在数据流方向明确，通常是单向通信的场景中使用，例如：ls | grep txt，ls的输出通过管道传递给grep进行过滤。

ls | grep txt

#include <unistd.h>

#include <iostream>

int main() {

int fd[2];

pid\_t pid;

char buffer[20];

pipe(fd);

pid = fork();

if (pid > 0) { // Parent process

close(fd[0]); // Close unused read end

write(fd[1], "Hello, child", 13);

close(fd[1]); // Close write end

} else { // Child process

close(fd[1]); // Close unused write end

read(fd[0], buffer, 20);

std::cout << "Child received: " << buffer << std::endl;

close(fd[0]); // Close read end

}

return 0;

}

### 2. ****命名管道（Named Pipes, FIFO）****

* · **定义**：命名管道是一种特殊类型的文件，可以在无亲缘关系的进程间进行通信。

#### ****使用场景：****

* **不同进程间的通信**：适用于无亲缘关系的进程之间的数据传输，可以在进程间共享数据，尤其是当通信进程不存在父子关系时。
* **跨程序的通信**：常用于不同程序之间的通信，如客户端和服务器端程序之间的数据传递。

#### ****示例：****

* 一个进程可以使用命名管道将数据写入管道，另一个进程从管道读取数据。

#include <iostream>#include <fcntl.h>#include <sys/stat.h>#include <unistd.h>

int main() {

const char\* fifo = "/tmp/myfifo";

mkfifo(fifo, 0666);

pid\_t pid = fork();

if (pid > 0) { // Parent process

int fd = open(fifo, O\_WRONLY);

write(fd, "Hello, FIFO", 12);

close(fd);

} else { // Child process

char buffer[20];

int fd = open(fifo, O\_RDONLY);

read(fd, buffer, 20);

std::cout << "Child received: " << buffer << std::endl;

close(fd);

}

unlink(fifo);

return 0;

}

### 3 ****消息队列（Message Queues）****

* · **定义**：消息队列允许进程通过消息传递进行通信，每个消息带有一个类型标识。

#### ****使用场景：****

* **异步通信**：适用于需要异步消息传递的场景，允许进程以队列的形式发送和接收消息。
* **可靠的消息传递**：在要求消息传递顺序和可靠性的重要应用中使用，例如任务调度系统、分布式系统中的消息传递。

#### ****示例：****

* 操作系统内核中任务调度的通知机制

#include <sys/ipc.h>#include <sys/msg.h>#include <iostream>#include <cstring>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key = ftok("progfile", 65);

int msgid = msgget(key, 0666 | IPC\_CREAT);

msg\_buffer message;

pid\_t pid = fork();

if (pid > 0) { // Parent process

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello, Message Queue");

msgsnd(msgid, &message, sizeof(message), 0);

} else { // Child process

msgrcv(msgid, &message, sizeof(message), 1, 0);

std::cout << "Child received: " << message.msg\_text << std::endl;

}

msgctl(msgid, IPC\_RMID, nullptr); // Destroy the message queue

return 0;

}

### 4 ****共享内存（Shared Memory）****

* · **定义**：共享内存是一种高效的进程间通信机制，允许多个进程访问同一块内存。

#### ****使用场景：****

* **大数据量传输**：适用于需要在进程间高效共享大量数据的场景。由于共享内存是所有IPC方式中速度最快的，所以在需要高性能的数据交换时优先使用。
* **低延迟通信**：适合需要低延迟的数据传输的场景，如实时数据处理、图像处理或多媒体应用。

#### ****示例：****

* 进程 A 将一个大数据块放入共享内存区，进程 B 从该共享内存区读取数据。

#include <sys/ipc.h>#include <sys/shm.h>#include <cstring>#include <iostream>

int main() {

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

char\* str = (char\*) shmat(shmid, nullptr, 0);

pid\_t pid = fork();

if (pid > 0) { // Parent process

strcpy(str, "Hello, Shared Memory");

shmdt(str);

} else { // Child process

sleep(1); // Ensure parent writes first

std::cout << "Child received: " << str << std::endl;

shmdt(str);

}

shmctl(shmid, IPC\_RMID, nullptr); // Destroy the shared memory

return 0;

}

### 5 ****信号（Signals）****

* · **定义**：信号是一种异步通信机制，允许一个进程通知另一个进程发生了某个事件。

#### ****使用场景：****

* **事件通知**：适用于进程之间的事件通知和控制，例如进程 A 向进程 B 发送信号，通知其需要执行某种操作（如终止、暂停、继续等）。
* **异常处理**：处理异常情况时使用，如进程接收到 SIGINT 信号时，执行特定的清理操作后退出。

#### ****示例：****

* 一个进程在执行长时间任务时，可以设置处理 SIGINT 信号的处理函数，以便在用户按下 Ctrl+C 时进行清理操作。

#include <csignal>#include <iostream>#include <unistd.h>

void signalHandler(int signum) {

std::cout << "Interrupt signal (" << signum << ") received.\n";

}

int main() {

signal(SIGUSR1, signalHandler);

pid\_t pid = fork();

if (pid > 0) { // Parent process

sleep(1);

kill(pid, SIGUSR1);

} else { // Child process

pause(); // Wait for signal

}

return 0;

}

### 进程间通信的其他场景

### 6. ****信号量（Semaphores）****

#### ****使用场景：****

* **进程同步**：适用于进程间的同步，控制多个进程对共享资源的访问，防止竞争条件（Race Condition）。
* **互斥锁**：常用于实现进程间的互斥，如在共享资源（如文件、内存）上进行读写操作时，防止数据竞争。

#### ****示例：****

* 多个进程需要控制对共享资源的访问，比如多个进程同时写入同一个文件时，需要通过信号量控制。

### 7. ****套接字（Sockets）****

#### ****使用场景：****

* **网络通信**：适用于不同主机之间的进程通信，特别是分布式系统、客户端-服务器架构中，使用 TCP 或 UDP 协议进行跨网络的进程通信。
* **跨平台通信**：可以用于不同操作系统之间的进程通信，如一个进程运行在Linux，另一个进程运行在Windows，通过网络套接字进行通信。

#### ****示例：****

* 一个Web服务器接收来自浏览器的请求，然后通过套接字将数据发送回浏览器。