Internal-used only

Threads synchronization

Contents

- Thread basic
- Thread mutex locking
- Thread advance locking and condition variables
- Atomic types

VC IVI Development Center Vietnam

Hanoi, November 2018

Threads

- Threads allow parallel execution of processes or distinct parts of a single process.
- All threads within a process share:
  - The same address space
  - Process instructions
  - Most data
  - Open files (descriptors)
  - Signals and signal handlers
  - Current working directory
  - User and group id

Threads

- Each thread has a unique:
  - Thread ID (tid)
  - Set of registers, stack pointer
  - Stack for local variables, return addresses
  - Signal mask
  - Priority
  - Return value: errno

- Basic thread operations:
  - Creation
  - Termination
  - Joining
  - Synchronization

Problems

In the general case, you often use shared objects between the threads. And when you do it, you will face another problem: synchronization.

Solution: provide functions that will block one thread if another thread is trying to access data that it is currently using.

- There are several ways to fix this problem:
- Semaphores
- Atomic references
- Monitors
- Condition codes
- Compare and swap
- etc.

Use a mutex to make a thread-safe

- There are two important methods on a mutex: lock() and unlock(). As their names indicate, the first one enable a thread to obtain the lock and the second releases the lock. The lock() method is blocking. The thread will only return from the lock() method when the lock has been obtained.

Automatic management of locks

It exists a good solution to avoid forgetting to release the lock: **std::lock_guard**.

This class is a simple smart manager for a lock. When the std::lock_guard is created, it automatically calls lock() on the mutex. When the guard gets destructed, it also releases the lock.

Recursive locking

The threads tries to acquire the lock again, but the lock is already locked. This is a case of deadlock. By default, a thread cannot acquire the same mutex twice.

Timed locking

Sometimes, you doesn't want a thread to wait ad infinitum for a mutex. For example, if your thread can do something else when waiting for the thread.

the standard library has a solution: **std::timed_mutex** and **std::recursive_timed_mutex**

You have access to the same functions as a **std::mutex**: *lock()* and *unlock()*, but you have also two new functions: *try_lock_for()* and *try_lock_until()*.

Condition variables

A condition variable manages a list of threads waiting until another thread notify them.

Each thread that wants to wait on the condition variable has to acquire a lock first. The lock is then released when the thread starts to wait on the condition and the lock is acquired again when the thread is awakened.

Atomic Types

The C++11 Concurrency Library introduces Atomic Types as a template class: std::atomic.

The main advantage of this technique is its performance.

std::atomic is specialized for all integral types to provide member functions specific to integral.

Thank you!