How would you implement a basic event-loop?

Asked 15 years, 9 months ago Modified 1 year, 8 months ago Viewed 54k times

76

If you have worked with gui toolkits, you know that there is a event-loop/main-loop that should be executed after everything is done, and that will keep the application alive and responsive to different events. For example, for Qt, you would do this in main():

```
int main() {
    QApplication app(argc, argv);
    // init code
    return app.exec();
}
```

Which in this case, app.exec() is the application's main-loop.

The obvious way to implement this kind of loop would be:

```
void exec() {
    while (1) {
        process_events(); // create a thread for each new event (possibly?)
    }
}
```

But this caps the CPU to 100% and is practicaly useless. Now, how can I implement such an event loop that is responsive without eating the CPU altogether?

Answers are appreciated in Python and/or C++. Thanks.

Footnote: For the sake of learning, I will implement my own signals/slots, and I would use those to generate custom events (e.g. `go_forward_event(steps)`). But if you know how I can use system events manually, I would like to know about that too.

c++ python blocking event-loop

Share
Improve this question
Follow

edited Mar 18, 2009 at 14:37
Iraimbilanja

asked Mar 18, 2009 at 14:12
fengshaun
2,170 ● 1 ● 16 ● 26

4 I believe you can delve into Qt's source code and see exactly what they are doing in exec(). That would probably give you some good pointers. – JimDaniel Mar 18, 2009 at 14:49

7 Answers

Sorted by: Highest score (default)

87

I used to wonder a lot about the same!

A GUI main loop looks like this, in pseudo-code:

```
void App::exec() {
    for(;;) {
        vector<Waitable> waitables;
        waitables.push_back(m_networkSocket);
        waitables.push_back(m_xConnection);
        waitables.push_back(m_globalTimer);
        Waitable* whatHappened = System::waitOnAll(waitables);
        switch(whatHappened) {
            case &m_networkSocket: readAndDispatchNetworkEvent(); break;
            case &m_xConnection: readAndDispatchGuiEvent(); break;
            case &m_globalTimer: readAndDispatchTimerEvent(); break;
        }
    }
}
```

What is a "Waitable"? Well, it's system dependant. On UNIX it's called a "file descriptor" and "waitOnAll" is the ::select system call. The so-called `vector<Waitable>` is a `::fd_set` on UNIX, and "whatHappened" is actually queried via `FD_ISSET`. The actual waitable-handles are acquired in various ways, for example `m_xConnection` can be taken from ::XConnectionNumber(). X11 also provides a high-level, portable API for this -- ::XNextEvent() -- but if you were to use that, you wouldn't be able to wait on several event sources *simultaneously*.

How does the blocking work? "waitOnAll" is a syscall that tells the OS to put your process on a "sleep list". This means you are not given any CPU time until an event occurs on one of the waitables. This, then, means your process is idle, consuming 0% CPU. When an event occurs, your process will briefly react to it and then return to idle state. GUI apps spend almost *all* their time idling.

What happens to all the CPU cycles while you're sleeping? Depends. Sometimes another process will have a use for them. If not, your OS will busy-loop the CPU, or put it into temporary low-power mode, etc.

Please ask for further details!

Share
Improve this answer
Follow

edited Mar 18, 2009 at 15:00

answered Mar 18, 2009 at 14:29
Iraimbilanja

How would I implement such a waiting system to wait not for system signals, but for my own signals? – fengshaun Mar 18, 2009 at 17:17

As I said, your code only runs in reaction to events. Therefore, if you fire your own event, you're gonna do that as a reaction to some system event. And then it becomes clear that you don't actually need an event system for your custom events. Simply call the handlers directly! – Iraimbilanja Mar 18, 2009 at 17:24

For example, consider a signal "Button::clicked". It is only going to fire in response to a system event (left mouse button release). So your code becomes "virtual void Button::handleLeftRelease(Point) { clicked.invoke(); }" with no need for threads or an event queue or anything. – Iraimbilanja Mar 18, 2009 at 17:27

1 I've been wondering how evented processes were implemented on a low level for years now. Thanks so much! – Jonathan Dumaine Feb 13, 2012 at 18:55

25

Python:

You can look at the implementation of the [Twisted reactor] which is probably the best implementation for an event loop in python. Reactors in Twisted are implementations of an interface and you can specify a type reactor to run: select, epoll, kqueue (all based on a c api using those system calls), there are also reactors based on the QT and GTK toolkits.

A simple implementation would be to use select:

```
#echo server that accepts multiple client connections without forking threads

import select
import socket
import sys

host = ''
port = 50000
backlog = 5
size = 1024
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host,port))
server.listen(backlog)
input = [server,sys.stdin]
running = 1

#the eventloop running
while running:
    inputready,outputready,exceptready = select.select(input,[],[])

    for s in inputready:

        if s == server:
            # handle the server socket
            client, address = server.accept()
            input.append(client)
        elif s == sys.stdin:
            # handle standard input
            junk = sys.stdin.readline()
            running = 0

        else:
            # handle all other sockets
            data = s.recv(size)
            if data:
                s.send(data)
            else:
                s.close()
                input.remove(s)
server.close()
```

Share Improve this answer Follow

answered Mar 19, 2009 at 1:45

Vasil
38k ● 27 ● 93 ● 116

14

Generally I would do this with some sort of counting semaphore:

1. Semaphore starts at zero.
2. Event loop waits on semaphore.
3. Event(s) come in, semaphore is incremented.
4. Event handler unblocks and decrements the semaphore and processes the event.
5. When all events are processed, semaphore is zero and event loop blocks again.

If you don't want to get that complicated, you could just add a sleep() call in your while loop with a trivially small sleep time. That will cause your message processing thread to yield it's CPU time to other threads. The CPU won't be pegged at 100% any more, but it's still pretty wasteful.

Share Improve this answer Follow

answered Mar 18, 2009 at 14:23

Eric Petroelje
60.4k ● 9 ● 132 ● 178

This sounds tempting, I will have to learn more about threading. Thanks. – fengshaun Mar 18, 2009 at 17:18

1 @FallingFromBed - not a busy wait, but a blocking wait on a sepmaphore. The difference is important because a blocking wait will not idly consume CPU time. – Eric Petroelje Apr 14, 2017 at 16:15

13

I would use a simple, light-weight messaging library called ZeroMQ (http://www.zeromq.org/). It is an open source library (LGPL). This is a very small library; on my server, the whole project compiles in about 60 seconds.

ZeroMQ will hugely simplify your event-driven code, AND it is also THE most efficient solution in terms of performance. Communicating between threads using ZeroMQ is much faster (in terms of speed) than using semaphores or local UNIX sockets. ZeroMQ also be a 100% portable solution, whereas all the other solutions would tie your code down to a specific operating system.

Share Improve this answer Follow

answered Mar 18, 2009 at 16:37

Anonymous

7

Here is a C++ event loop. At the creation of the object `EventLoop`, it creates a thread which continually runs any task given to it. If there are no tasks available, the main thread goes to sleep until some task is added.

First we need a thread safe queue which allow multiple producers and at least a single consumer (the `EventLoop` thread). The `EventLoop` object controls the consumers and producers. With a little change, it can be added multiple consumers (runners threads), instead of only one thread.

```
#include <stdio.h>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <iostream>
#include <set>
#include <functional>

#if defined( WIN32 )
    #include <windows.h>
#endif

class EventLoopNoElements : public std::runtime_error
{
public:
    EventLoopNoElements(const char* error)
        : std::runtime_error(error)
    {
    }
};

template <typename Type>
struct EventLoopCompare {
    typedef std::tuple<std::chrono::time_point<std::chrono::system_clock>,
Type> TimePoint;

    bool operator()(const typename EventLoopCompare<Type>::TimePoint left,
const typename EventLoopCompare<Type>::TimePoint right) {
        return std::get<0>(left) < std::get<0>(right);
    }
};

/**
 * You can enqueue any thing with this event loop. Just use lambda functions,
future and promises!
 * With lambda `event.enqueue( 1000, [myvar, myfoo](){ myvar.something(myfoo);
} )`
 * With futures we can get values from the event loop:
 * ```
 * std::promise<int> accumulate_promise;
 * event.enqueue( 2000, [&accumulate_promise](){
accumulate_promise.set_value(10); } );
 * std::future<int> accumulate_future = accumulate_promise.get_future();
 * accumulate_future.wait(); // It is not necessary to call wait, except for
syncing the output.
 * std::cout << "result=" << std::flush << accumulate_future.get() <<
std::endl;
 * ```
 * It is just not a nice ideia to add something which hang the whole event loop
queue.
 */
template <class Type>
struct EventLoop {
    typedef std::multiset<
        typename EventLoopCompare<Type>::TimePoint,
        EventLoopCompare<Type>
    > EventLoopQueue;

    bool _shutdown;
    bool _free_shutdown;

    std::mutex _mutex;
    std::condition_variable _condition_variable;
    EventLoopQueue _queue;
    std::thread _runner;

    // free_shutdown - if true, run all events on the queue before exiting
    EventLoop(bool free_shutdown)
        : _shutdown(false),
        _free_shutdown(free_shutdown),
        _runner( &EventLoop<Type>::_event_loop, this )
    {
    }

    virtual ~EventLoop() {
        std::unique_lock<std::mutex> dequeuelock(_mutex);
        _shutdown = true;
        _condition_variable.notify_all();
        dequeuelock.unlock();

        if (_runner.joinable()) {
            _runner.join();
        }
    }

    // Mutex and condition variables are not movable and there is no need for
smart pointers yet
    EventLoop(const EventLoop&) = delete;
    EventLoop& operator =(const EventLoop&) = delete;
```

```
    EventLoop(const EventLoop&&) = delete;
    EventLoop& operator =(const EventLoop&&) = delete;

    // To allow multiple threads to consume data, just add a mutex here and
create multiple threads on the constructor
    void _event_loop() {
        while ( true ) {
            try {
                Type call = dequeue();
                call();
            }
            catch (EventLoopNoElements&) {
                return;
            }
            catch (std::exception& error) {
                std::cerr << "Unexpected exception on EventLoop dequeue
running: '" << error.what() << "'" << std::endl;
            }
            catch (...) {
                std::cerr << "Unexpected exception on EventLoop dequeue
running." << std::endl;
            }
        }
        std::cerr << "The main EventLoop dequeue stopped running unexpectedly!"
<< std::endl;
    }

    // Add an element to the queue
    void enqueue(int timeout, Type element) {
        std::chrono::time_point<std::chrono::system_clock> timenow =
std::chrono::system_clock::now();
        std::chrono::time_point<std::chrono::system_clock> newtime = timenow +
std::chrono::milliseconds(timeout);

        std::unique_lock<std::mutex> dequeuelock(_mutex);
        _queue.insert(std::make_tuple(newtime, element));
        _condition_variable.notify_one();
    }

    // Blocks until getting the first-element or throw EventLoopNoElements if
it is shutting down
    // Throws EventLoopNoElements when it is shutting down and there are not
more elements
    Type dequeue() {
        typename EventLoopQueue::iterator queuebegin;
        typename EventLoopQueue::iterator queueend;
        std::chrono::time_point<std::chrono::system_clock> sleeptime;

        // _mutex prevents multiple consumers from getting the same item or
from missing the wake up
        std::unique_lock<std::mutex> dequeuelock(_mutex);
        do {
            queuebegin = _queue.begin();
            queueend = _queue.end();

            if ( queuebegin == queueend ) {
                if ( _shutdown ) {
                    throw EventLoopNoElements( "There are no more elements on
the queue because it already shutdown." );
                }
                _condition_variable.wait( dequeuelock );
            }
            else {
                if ( _shutdown ) {
                    if (_free_shutdown) {
                        break;
                    }
                    else {
                        throw EventLoopNoElements( "The queue is shutting
down." );
                    }
                }
                std::chrono::time_point<std::chrono::system_clock> timenow =
std::chrono::system_clock::now();
                sleeptime = std::get<0>( *queuebegin );
                if ( sleeptime <= timenow ) {
                    break;
                }
                _condition_variable.wait_until( dequeuelock, sleeptime );
            }
        } while ( true );

        Type firstelement = std::get<1>( *queuebegin );
        _queue.erase( queuebegin );
        dequeuelock.unlock();
        return firstelement;
    }
};
```

Utility to print the current timestamp:

```
std::string getTime() {
    char buffer[20];
#if defined( WIN32 )
    SYSTEMTIME wlocaltime;
    GetLocalTime(&wlocaltime);
    ::snprintf(buffer, sizeof buffer, "%02d:%02d:%02d.%03d ", wlocaltime.wHour,
wlocaltime.wMinute, wlocaltime.wSecond, wlocaltime.wMilliseconds);
#else
    std::chrono::time_point< std::chrono::system_clock > now =
std::chrono::system_clock::now();
    auto duration = now.time_since_epoch();
    auto hours = std::chrono::duration_cast< std::chrono::hours >( duration );
    duration -= hours;
    auto minutes = std::chrono::duration_cast< std::chrono::minutes >( duration
);
    duration -= minutes;
    auto seconds = std::chrono::duration_cast< std::chrono::seconds >( duration
);
    duration -= seconds;
    auto milliseconds = std::chrono::duration_cast< std::chrono::milliseconds >
( duration );
    duration -= milliseconds;
    time_t theTime = time( NULL );
    struct tm* aTime = localtime( &theTime );
    ::snprintf(buffer, sizeof buffer, "%02d:%02d:%02d.%03ld ", aTime->tm_hour,
aTime->tm_min, aTime->tm_sec, milliseconds.count());
#endif
    return buffer;
}
```

Example program using these:

```
// g++ -o test -Wall -Wextra -ggdb -g3 -pthread test.cpp && gdb --args ./test
// valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --
verbose ./test
// procdump -accepteula -ma -e -f "" -x c:\ myexe.exe
int main(int argc, char* argv[]) {
    std::cerr << getTime().c_str() << "Creating EventLoop" << std::endl;
    EventLoop<std::function<void()>>* eventloop = new
EventLoop<std::function<void()>>(true);

    std::cerr << getTime().c_str() << "Adding event element" << std::endl;
    eventloop->enqueue( 3000, []{ std::cerr << getTime().c_str() << "Running
task 3" << std::endl; } );
    eventloop->enqueue( 1000, []{ std::cerr << getTime().c_str() << "Running
task 1" << std::endl; } );
    eventloop->enqueue( 2000, []{ std::cerr << getTime().c_str() << "Running
task 2" << std::endl; } );

    std::this_thread::sleep_for( std::chrono::milliseconds(5000) );
    delete eventloop;
    std::cerr << getTime().c_str() << "Exiting after 10 seconds..." <<
std::endl;
    return 0;
}
```

Output test example:

```
02:08:28.960 Creating EventLoop
02:08:28.960 Adding event element
02:08:29.960 Running task 1
02:08:30.961 Running task 2
02:08:31.961 Running task 3
02:08:33.961 Exiting after 10 seconds...
```

Update

In the end, the Event Loop presented is a like a time manager. A better interface for a time manager would be not to force the user to use threads. This would be an example:

```
class TimerManager
{
public:
    std::chrono::steady_clock clock_type;
    // setup given function to be executed at given timeout
    // @return unique identifier
    uint64_t start( std::chrono::milliseconds timeout, const std::function<
void( void ) >& func );
    // cancel given unique identifier
    void cancel( uint64_t id );
    // handle all expired entries
    // @return next expiration or zero when queue is empty
    std::chrono::milliseconds run( );
}
```

Share
Improve this answer
Follow

edited Sep 30, 2022 at 18:24

answered Apr 25, 2021 at 5:18
Evandro Coan
9,388 ● 14 ● 97 ● 158

-1: Haters will hate; +1 lovers will love. – Evandro Coan Apr 27, 2021 at 23:55

2 Nice implementation and +1 from me – asitdhal Aug 1, 2021 at 17:28

1 This is a beautiful implementation. Thank you for sharing. – sleepystar96 Nov 26, 2021 at 20:43

1

This answer is for unix-like system such as Linux or Mac OS X. I do not know how this is done in Windows.

select() or pselect(). Linux also has poll().

Check the man pages for a in depth details. This syscalls want a lists of file desciptors, a timeout and/or a signal mask. This syscalls let the program wait till an event. If one of the file desciptors in the list is ready to read or write (depends on the settings, see manpages), the timeout expires or a signal arrived, this syscalls will return. The program can then read/write to the file descriptors, processes the signals or does other stuff. After that it calls (p)select/poll again and wait till the next event.

The sockets should be opened as non-blocking so that the read/write function returns when there is no data/buffer full. With the common display server X11, the GUI is handled via a socket and has a file descriptor. So it can be handled the same way.

Share Improve this answer Follow

answered Aug 20, 2020 at 17:44
12431234123412341234123
4123
1

-1

Before creating Basic application of Event-loop in python. Let's understand

`what is Event Loop ?`

> An event loop is a central component of any asynchronous I/O framework that allows you to perform I/O operations concurrently without blocking the execution of your program. An event loop runs in a single thread and is

> responsible for receiving and dispatching I/O events [like reading/writing to a file -or- keyboard interrupt] as they occur.

```
import asyncio

async def coroutine():
    print('Start')
    await asyncio.sleep(1)
    print('End')

loop = asyncio.get_event_loop()
loop.run_until_complete(coroutine())
```

Share Improve this answer Follow

answered Apr 4, 2023 at 9:09

Deepanshu Mehta

1,732 ● 16 ● 14