A look at event loops and multithreading

Stephan Soller, Computer Science and Media
Stuttgart Media University
ss312@hdm-stuttgart.de

Abstract

Recent scalability efforts like node.js made event loops popular. But event loops are often also useful (and used) for other kinds of software.

This paper introduces event loops and takes a look at their advantages and disadvantages. What they can provide and what it costs to use them. It also explores multithreaded event loops, especially multithreading one event loop with epoll.

In the end however multithreading one event loop nullifies many advantages of event loops. So they are best used in a single thread with explicit background threads.

1. Introduction

Simple programs only have to react to input from one source. A terminal for example reads a command, executes it and then waits for the next command. For this kind of question and answer interaction a simple `read()` [1] or `scanf()` [2] will suffice. If no data is available the operating system will automatically block the program. And as soon as new data arrives the program will be woken up and can resume its work.

More complex programs however need to react to input from multiple sources. Will the user click the "accept" button or close the window? Is new data available to be read from a network connection? Did a background task just complete?

Most applications usually have to take care of many such things. But they don't know in which order these events will happen. They don't know from which source (window, timer, network connection, etc.) the next event will come. So they can't wait for a single "next thing" like `scanf()` does by waiting only for new data from the standard input. Instead applications have to be ready for events from many different sources and process them as they are coming in.

Operating systems have long since provided functions for that task: For example `select()` [3] or `poll()` [4] on Linux and `WaitForMultipleObjects()` [5] on Windows. With them an application can wait for the next event from any of the supplied sources. By calling these functions in a simple loop an application then can react to all events:

```
while( wait_for_event(sources, &event) )
  process_event(event);
```

The loop in this pseudocode will wait for the next event and process it when it arrives. Then it starts over until there is no next event (e.g. the window was destroyed or a shutdown signal received). Quite literally a loop that processes events, hence it's called an "event loop".

2. Advantages of event loops

Event loops have several advantages which made them popular for many applications.

Do many things "at once"

Event loops allow an single threaded application to appear like it's reacting to many different things at the same time.

A GUI application can redraw it's window when needed, update a progress bar and can react when it's window is closed. A server can accept new connections and coordinate the data flow from and to many already established connections. This is achieved by reacting to events so quickly that the user doesn't notice that the application does only one thing at a time. This is also called "event multiplexing".

Very efficient

Event loops handle large number of events very efficiently. APIs like epoll [6] and I/O Completion Ports [7] have little overhead and can be used to monitor thousands of network connections for events.

Network servers are usually I/O heavy applications. Often there is little CPU intensive work to be done but there are very many network connections that have to be monitored and processed. Servers like nginx [8] and Apache 2 (the event MPM [9]) use event loops to optimize their performance and scalability [10]. The throughput oriented node.js [11] even uses the event loop as its primary programing paradigm.

No deadlocks or race conditions

Event loops don't need threads. Thus a single threaded event loop avoids the overhead, difficulties and errors of locking and thread synchronization.

This allows to write code that coordinates a program in a straight forward way. Event handlers can modify arbitrary complex structures like elements in the DOM tree of web browsers. No need to think about locks, deadlocks or lost updates. This enormously simplifies any coordination code.

Also programmers usually don't notice it all these advantages have put event loops into much of todays software. Most graphical user interfaces (GUIs) are event driven. You write event handlers and don't care how it gets called. GUIs then use event loops to call the event handlers at the right time. The Win32 GUI, GTK+ and many others work this way. Even websites are no different. Many server applications use event loops. Either for performance and scalability reasons or because of the simplicity event loops bring with them when many clients need to interact with one another.

3. Disadvantages

As with everything in live nothing is perfect. And neither are event loops. Using event loops comes with some major drawbacks.

No long running event handlers

Event loops only work well when all event handlers are quick enough.

While an application is busy processing an event new events are queued up. The application doesn't respond to them immediately. In other words: During event processing the application is unresponsive.

When an event handler runs for one second the user will usually notice that other stuff isn't taken care of (e.g. redrawing of the GUI). In the worst case an event handler runs for several seconds and the user or system might think that the application just crashed. And some users will simply kill the application.

In practice there are two sources for long running event handlers:

- Functions that block and wait for data.
- Expensive calculations that actually take a long time to complete.

Avoid blocking calls in event handlers

Blocking functions will also block the entire event loop. Those are functions that wait for data or events by blocking the current thread until the data arrives (like `scanf()`).

Waiting for data (or events) is what the event loop is for. When a function *within an event handler* waits until something is supposed to happen our entire event loop doesn't continue. While that function keeps our thread blocked the event loop can't process new events. Our application is unresponsive.

So everything that might block the current thread to wait for data should instead use the event loop. Unfortunately blocking function calls can hide in many places. Especially if libraries are used.

The most obvious candidates are network connections. Establishing a network connection and reading or writing from it can take time. In case of internet connections or slow servers quite a long time. For example if you have a function that connects to a server and checks for updates you'll have to rewrite it to use the event loop. Otherwise it will block the entire application until it got its answer from the server.

Even working with files can take time and block a thread: Maybe the notebooks hard drive was in energy saving mode and needs to spin up again. Even if the drive is ready it can take the operating system quite some time to fetch or write data. All time during which our event loop could process new events.

Actually for local file systems this is often good enough and users learned to live with small application "hiccups" or "hangs". But network file systems which are common in enterprise environments can drastically increase the time of file I/O. This can make an application quite unresponsive or simply unproductive to work with.

Integration of APIs into an event loop

It can be rather difficult up to impossible to properly integrate all blocking function calls into an event loop.

On Linux non-blocking I/O and event monitoring works well for almost all operating system "objects": sockets, pipes, timers, signals, Video4Linux 2 devices (webcams), etc. Even an epoll event loop itself can be monitored from another event loop. But plain disk files don't support non-blocking operations. And monitoring them with `poll()` or `epoll()` doesn't work because disk files are always "ready" for `read()` or `write()` but can still force the thread to sleep [12].

On windows many `HANDLE`s can be waited upon. With `MsgWaitForMultipleObjects()` an application can wait for `HANDLE`s and window messages at the same time. But again for file I/O there are many quirks and exceptions to this. Sometimes asynchronous calls can be silently converted to synchronous ones and block the thread [13] [14].

Event loop libraries like libeio [15] and libuv [16] (used by node.js) internally use thread pools to emulate a non-blocking interface to these functions.

Unfortunately some APIs simply don't play well with event loops. For example OpenGL provides it's own synchronization primitives (GL_ARB_sync [17]). But at the time of writing (November 2014) these sync primitives can't be converted to `HANDLE`s or file descriptors which could be used in an event loop. This functionality was proposed in the spec around 2006 [18] but never followed up upon or implemented. So again an extra thread has to be used to isolate the event loop from the blocking function calls of OpenGL. But this can have other implications since especially on Windows a lot of implicit state is tied to the thread instead of the process (e.g. input queues of windows).

While event loops suffer from this drawback it's not strictly related only to them. Rather many APIs and libraries don't expose proper event handling primitives. A problem also encountered when coordinating multiple threads. Making it hard to focus on the causality of the programs execution and requiring additional layers of coordination. Even outside of event loops this leads to unnecessary spawning of new threads, not to utilize multiple cores for number crunching but to work around blocking function calls.

Avoid CPU intensive work in the event loop

CPU heavy event handlers can also keep the event loop busy for a long time. They execute expensive calculations that simply take up a lot of CPU time to finish. Such code can also hide in a lot of places. Especially since most libraries don't mention how expensive a function can be.

For example loading and processing images is fast for small images but can take several seconds for larger images or more complex formats and operations. Video and audio processing like decoding, encoding or filtering of multimedia content can take a lot more CPU time. In the worst case keeping the event loop busy almost all the time with just an occasional event slipping through. Often resulting in a "not yet dead but quite laggy" application.

To reliably identify those situations the developer has to understand what actually happens. For example he has to understand that JPEG decoding is CPU intensive and takes longer with larger images. Libraries can be used to avoid writing a decoder but developers have to understand how long a function can take. Either by understanding the task itself or by profiling.

Again, to keep these CPU intensive functions from "blocking" the event loop they have to be moved to their own threads. Preferably these threads should have a lower priority than the event loop thread. Event handlers can then spawn these threads and use event objects [19] or eventfds [20] in the event loop to wait for their completion. Those techniques are nothing new and were already used with Windows 98 [21].

Programming in continuation passing style

Event loops heavily impact the style the code has to be written in. Usually code is written in a straightforward way, step for step. Disregarding what exactly the code does or how long it takes to execute (the main advantage of encapsulation).

```
void on_click() {
  void* data = read(filename);
  void* result = do_calculation(data);
  write(filename, result);
  show("Done!");
}
```

With event loops it becomes important if code can block and how long a CPU intensive calculation can run.

If the `read()` and `write()` functions in the above example can block we need to restructure the code. Each time we can possibly block we have to return to the event loop and give it some object it can monitor. The waiting has to done by the event loop. As soon as the event loop sees that this object is readable or writable we have to continue with our code in another event handler.

```
cont_t on_click() {
  event_t read_done = read_async(filename);
  return (cont_t){
    .when = read_done,
    .call = on_data_ready
  };
}

cont_t on_data_ready(void* data) {
  void* result = do_calculation(data);
  return (cont_t){
    .when = write_async(filename, result),
    .call = on_write_done
  };
}

cont_t on_write_done() {
  show("Done!");
}
```

In this C style pseudocode each event handler returns a `cont_t` struct that tells the event loop which event handler (`.call` member) should be called on which event (`.when` member). Instead of the blocking `read()` and `write()` functions we also need asynchronous versions of these functions that return an event (an file descriptor or `HANDLE`). The event loop then monitors these events and calls the event handlers when triggered.

While this now works nice with event loops every blocking function call splits the code into one more event handler. And for complex processes such an event handler chain can get quite long. Making simple code harder to read.

node.js uses this approach. There inline function literals and closures mitigate the problem a bit:

```
function on_click() {
  fs.readFile(filename, function(err, data){
    var result = do_calculation(data);
    fs.writeFile(filename, result,
      function(err){
      show("Done!");
    });
  });
}
```

All in all event loops force the programmer to explicitly handle when code can continue. Also all the data an event handler needs has to be stored and passed explicitly. Formerly this data would just be kept on the stack.

In the context of programming languages and interpreters this programming style is known as "continuation passing style". Each function returns what needs to be done next (a "continuation"). In that context the event loop corresponds to the trampoline that executes the continuations as soon as possible.

This is quite a different style of thinking and can get complicated. To simplify code some system employ light-weight user level threads (e.g. POSIX `ucontext.h`). The system then provides functions that instead of blocking hand control back to the event loop. And the event loop resumes the user level threads when new data is available for them. Then the programmer can write code more or less in the same style as before. But how much this then differs from actually using threads directly depends on the platform, used libraries and the time spend to optimize the system (e.g. user level threads stack size).

Continuation passing style is usually perceived as a disadvantage of event loops. But depending on the project and problem the explicit state and causality handling can offer advantages.

The explicit state management makes it rather easy to optimize servers for minimal client state. All state a client needs can be put into one data structure (e.g. a simple C struct) which can be optimized to only contain necessary data. This way a server can scale to very high client counts.

The explicit handling of events from different sources also makes prioritization simple. Each time the event loop is notified from the system that something has happened it simply calls the important event handlers first. It's also possible to defers low priority events until no high priority events are reported.

For example a server can accept new connections only when all established connections are already served. The event loop first calls all event handlers that receive, send or process data from established connections that are ready for I/O. It then would usually call the event handlers that accept new connections. But instead these events are left unhandled and the event loop again polls for new events. If new data for established connections is ready we do the same as before (handle and poll again). But if the event loop reports no data for established connections we then call the event handlers to accept new connections. This scheme makes sure that a server only establishes new connections when it still has capacity to do so. If not it will serve already connected clients at its maximum capacity and new connections time out.

4. Multithreaded event loops

A single threaded event loop can only use one CPU. Even when all event handlers are pretty fast the sheer number of events on a high-load server can make the event loop CPU bound instead of I/O bound. So high performance servers try to use multiple CPU cores for their event loops.

A common approach is to create one event loop per core and have some kind of load balancer distribute work to the individual event loops. This can use all available cores but allows to keep each event loop single threaded and avoids thread synchronization issues. But the approach requires some kind of request level parallelism the load balancer can use to easily distribute the workload. That usually is the case in server applications where many (mostly unrelated) requests need to be processed.

GUI applications don't offer easy request level parallelism since most event handlers need to modify complex global state (e.g. the DOM tree of a website). On the other hand GUI applications seldomly receive so many events that they become CPU bound. Instead they mostly suffer from long running event handlers that need to be manually handled in extra threads anyway. Either because an API can't be integrated into the event loop (e.g. OpenGL) or because of actual CPU intensive work.

5. Multithreading one event loop

Details of epoll [6] open up ways to distribute event handlers of even a single event loop over multiple threads. We explore that alley a bit in order to see what a single multithreaded event loop might look like. This chapter focuses only on epoll and the Linux kernel. Platform independence would easily exceed the scope of this paper.

epoll basically is a central list of file descriptors (sockets, timer fds, event fds, etc.) that are monitored for changes (new data ready to be read, etc.). Multiple threads can wait on a single epoll instance. When an event occurs one of the waiting threads is notified. This effectively distributes events over all waiting threads. Given at least one thread per core and enough CPU heavy events the event handlers can actually use all available CPU cores to do their work. Utilizing the full CPU.

This distribution however results in two problems:

- Since we're working with threads we have to avoid race conditions, lost updates and deadlocks.
- The second problem is a bit more subtle: We have to make sure that what the programmer thinks about as "one event" will actually only be processed by one and only one event handler.

Keeping "one event" one event

The second problem is best clarified with an example:

We have an HTTP server with 4 threads that uses one central epoll instance to handle many TCP network connections to clients. epoll notifies one thread as soon as new data on a connection is ready. Say we received 150 bytes on that connection. Now the notified thread executes the event handler that reads the pending data into a small buffer reserved for that connection and does some processing.

While that event handler is running another data packet arrives on that connection. Say an additional 300 bytes. New data is available and since the original thread is still busy it's possible that another thread will be notified. So the second thread executes the same event handler on the same connection. This is a typical race condition. We don't know how much data the first thread already read and if it already updated the number of bytes in the client buffer. So the second thread might overwrite the buffer data of the first thread with newer received data.

To avoid that we have to make sure that an event on a connection is fired only once and then disabled until the event handler is done. Fortunately epoll offers this mechanism: The `EPOLLONESHOT` flag (described in the man page). When telling epoll to monitor a file descriptor for changes this flag can be set. It then makes sure that only one event is triggered on that file descriptor. After the event is triggered the user has to rearm it (e.g. at the end of the event handler) to receive another event. With `EPOLLONESHOT` we can make sure that the first thread can read all data and do it's processing without a second thread working on the same connection.

Avoid race conditions in general

One big advantage of a single threaded event loop is that state can be manipulated without thinking about any synchronization. We lost that advantage so care has to be taken when working with state.

For global state (e.g. number of connected clients, total bytes served, etc.) atomic instructions and structures provide a way to avoid race conditions and deadlocks. Normal locking and synchronization primitives like mutexes or semaphors won't work because they would block the event handlers execution (and the entire thread). Instead Linux offers "eventfds" [20]. These are file descriptors that can be used as very lightweight semaphors. These eventfds can be integrated into the event loop and can be used for locking and signaling.

But since we don't know in which order epoll will trigger event handlers we can't avoid deadlocks by sorting the locks. This means that such locking can not be nested without risking deadlocks. So a locked list must not contain objects that require locking. Otherwise it's just a mater of time until the entire server deadlocks. So atomic operations should be preferred over locking whenever possible.

The same limitations also apply to state that is used to coordinate multiple clients, e.g. a queue where one client pushes data and many other clients read that data. But for some situations there are simple (but slower) alternatives to complex atomic structures. For example UNIX domain sockets of type SOCK_DGRAM or SOCK_SEQPACKET can be used as simple message queues.

In case of server applications most state is usually per request or connection. Global state is often handled by databases or key-value stores. Since event loops force the programmer to write code in continuation passing style even single threaded servers explicitly handle per connection state. When a new connection is established they allocate a structure containing all relevant data for the connection (buffer space, temporary variables, request state, what to do next, etc.).

A multithreaded event loop profits from this explicit "infrastructure". If we can ensure that this state data structure is only accessed by one event handler at a time most synchronization can be avoided. The easiest way is to ensure that for each state data structure only one event handler at a time can fire. So a request can either wait for new data from the network connection or from the database. But it can't wait for both. Otherwise the event handlers for these events might be called in different threads at the same time.

Worth the effort?

Handling of global state is complex. Atomic data structures require careful thinking in more complex scenarios. And this takes away a lot of the simplicity of event loops.

For server applications there usually is easy request level parallelism to exploit. Distributing load over multiple servers would require a load balancer anyway. And that load balancer can also be used to distribute work over multiple CPU cores. Threaded event loops would only add complexity to the entire system with no apparent advantage. So in these situations it's probably not worth the effort.

For GUI applications it depends heavily on the used environment and libraries. On windows many data structures of the UI are implicitly thread local. Distribution of the event handlers is very problematic in this case.

Even when designing a new GUI system multithreading would probably not be worth the effort. GUIs usually require a tree or graph based content model which would be difficult to handle in a multithreaded environment. It's probably easier and a better investment of time and complexity to use a single threaded event loop and move CPU heavy tasks into a worker thread pool. These worker threads can then communicate with the event loop via eventfds or other means.

6. Conclusion

Event loops offer a simple way to implement coordination of complex applications. That's their main advantage. Coordination usually isn't a CPU heavy task so it doesn't profit from parallelization. By putting event loops in a multithreaded context we're complicating coordination of events enormously but gain little by it.

7. Further ideas

Event loops (try to) centralize the high level coordination of a program (the programs "causality" so to speak). This opens up an interesting question: Can every application be divided into "coordinator" and "worker" parts? Or is there some CPU intensive high-level coordination that has to be done while crunching numbers? It might be that in general multithreading (parallelism) is a pattern to handle "worker" parts (number crunching) while event loops (or event multiplexing) is a pattern to handle "coordinator" parts.

It would also be interesting if such a division is hindered by technical problems (e.g. APIs) or if its just a matter of programming and thinking in a different way. Would that make development of complex applications easier? Could it even be a natural way to avoid many high-level synchronization bugs?

8. References

[1] read(2) man page - Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/read.2.html
Retrieved 2014-11-06

[2] scanf(3) man page - Linux Programmer's Manual
http://man7.org/linux/man-pages/man3/scanf.3.html
Retrieved 2014-11-06

[3] select(2) man page - Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/select.2.html
Retrieved 2014-11-06

[4] poll(2) man page - Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/poll.2.html
Retrieved 2014-11-06

[5] WaitForMultipleObjects function - Windows Dev Center - Desktop
http://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx
Retrieved 2014-10-30

[6] epoll - I/O event notification facility - Linux Programmer's Manual
http://man7.org/linux/man-pages/man7/epoll.7.html
Retrieved 2014-10-29

[7] I/O Completion Ports - Windows Dev Center - Desktop
http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx
Retrieved 2014-11-07

[8] nginx webserver
http://nginx.org/
Retrieved 2014-11-07

[9] Apache MPM event
http://httpd.apache.org/docs/2.4/mod/event.html
Retrieved 2014-11-07

[10] The C10K problem
http://www.kegel.com/c10k.html
Retrieved 2014-10-30

[11] node.js
http://nodejs.org/
Retrieved 2014-10-30

[12] Non-blocking I/O with regular files
http://www.remlab.net/op/nonblock.shtml
Retrieved 2014-10-29

[13] Nonblocking disk IO
http://neugierig.org/software/blog/2011/12/nonblocking-disk-io.html
Retrieved 2014-10-29

[14] Asynchronous Disk I/O Appears as Synchronous on Windows NT, Windows 2000, and Windows XP - Microsoft support
http://support.microsoft.com/kb/156932
Retrieved 2014-10-29

[15] libeio - Event-based fully asynchronous I/O library for C
http://software.schmorp.de/pkg/libeio.html
Retrieved 2014-10-30

[16] libuv - Cross-platform asynchronous I/O
https://github.com/joyent/libuv
Retrieved 2014-10-30

[17] OpenGL GL_ARB_sync extension
https://www.opengl.org/registry/specs/ARB/sync.txt
Retrieved 2014-10-31

[18] Issue 18. B6 of GL_ARB_sync extension
https://www.opengl.org/registry/specs/ARB/sync.txt
Retrieved 2014-10-31

[19] Event Objects - Windows Dev Center - Desktop
http://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx
Retrieved 2014-11-07

[20] eventfd - create a file descriptor for event notification - Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/eventfd.2.html
Retrieved 2014-10-30

[21] Programming Windows, 5th Edition - Chapter 20 Multitasking and Multithreading - Event Signaling - The Event Object, page 1235
ISBN: 1-57231-995-X, Microsoft Press
November 11, 1998