# libasync-mp: A Muliprocessor Aware C++ Asynchronous Programming Library

William Friedman, Ben Pick, Mark Rosen, Joe Hastings
[williamf, bpick, mrosen, jrh]@mit.edu

## Abstract

We have created a C++ library based upon *libasync* that allows for asynchronous programs to be distributed across multiple processors in FreeBSD4.3. We achieved nearly 2x performance increases on heavily CPU bound applications, and meet or exceed the performance of *libasync* in all other circumstances. We discuss which sets of applications are suitable for linking against our library and the effort necessary to perform such tasks. Finally, we discuss the difficulties in writing such a library as well as its performance and limitations.

## 1. Introduction

The asynchronous programming model is a powerful paradigm for writing efficient programs that perform concurrent operations. It eliminates the need for error-prone context synchronization code and time-consuming switches, while preserving one of the primary benefits of threads: the ability for the application to perform other useful work while waiting for a blocking call to complete. However, the libasync implementation of the asynchronous programming model eliminates one of the primary benefits of threads: automatic scheduling across multiple CPUs. This project's goal is to extend libasync to efficiently distribute events across multiple processors.

A key strength of the asynchronous model is its ease of use; there is no need to worry about critical sections, deadlocks, or any number of complex and difficult to debug eventualities. Threads will necessarily need to be introduced for events to be distributed across multiple processors, but the addition of threads will break a lot of asynchronous code, because the asynchronous programming style specifically assumes that there is no concurrent execution. Our asynchronous library provides no support for synchronization of events across multiple processors, but enforces a contract that allows all of the events that compose a specific connection to share data.

## 2. Related Work

### 2.1 Libasync

Our asynchronous threads package is inspired by libasync. We chose to write our own library to avoid the often confusing SFS internals. However, there is no thing to prevent our code from integration into libasync in the future. We provide nearly identical functionality to libasync, with the addition of multiplexing events across multiple processors. We do not implement the wrap function or reference counting, but nonetheless, programs require very little re-coding before they work with our library.[1]

### 2.2 SEDA

SEDA is significantly different from our system but is worth mentioning. The key insight of SEDA is that applications exhibit different levels of parallelism at different stages of their execution. SEDA decomposes an application into discrete stages that are separated by queues. The amount of parallelism at a specific stage is depending on both the load on that specific stage and the amount of hardware parallelism available to execute that stage. For example, it makes little sense for a CPU bound stage to have more threads allotted it than there are CPUs in the system.

The SEDA project also includes the NBIO non-blocking IO package for Java. NBIO uses three threads: one for accepting connections, one for reading, and one for writing.[5]

### 2.3 kqueue

The *kqueue* FreeBSD API is designed as an alternative to the *select/poll* kernel calls used by *libasync*. The kernel implementation of *kqueue* is designed to be kernel-thread friendly in anticipation of native FreeBSD kernel threads in FreeBSD5. In addition, *kqueue* is designed to support a broader range of kernel-notifiable events than *select* although most of this functionality is unsupported in FreeBSD4.x.[4] Finally, *kqueue*'s re-entrant nature makes programming with multiple kernel threads easier because the callee does not need to worry about the current status of *kqueue* in other threads.

## 2.4 Next Generation POSIX Threads

The NGPT library is a library supporting $M : N$ mappings of user-threads to kernel-threads. This POSIX standard is based partly upon the *Linux threads* library which was ported to FreeBSD to support kernel threads through the *rfork* system call. NGPT allows multiple-processor oriented applications to avoid needing to call *fork/rfork* directly and provides the convenience of address-space sharing from multiple processors.[2]

# 3. libasync-mp

## 3.1 Overview of libasync-mp

Our multiprocessor aware library functions at a high-level by creating $N$ completely independent event queues, generally one for each processor. While the library itself does not force any specific mode of use, when a connection is accepted, it is generally assigned a processor. It will remain with this processor for its entire lifetime. This design decision is crucial to our concurrency model, as it allows all of the events that compose a connection to share data without worrying about race conditions, synchronization, etc.

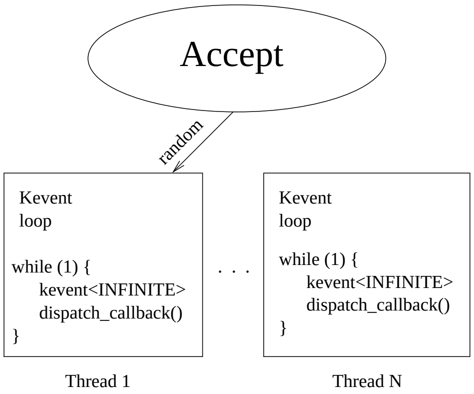


**Figure 1: Schematic of our system.**

Figure 1 shows an overview of our system. As connections are accepted, they are assigned one of N threads. The threads use *kevent* to wait for events to become ready to be executed, and then immediately execute the events as they leave *kevent*. Each of the $N$ event loops run as a separate kernel thread, using Next Generation POSIX Threads to handle complex $rfork$ intricacies.

## 3.2 Typical Server Implementation

A typical server application requires a single callback function for accepting connections. In asynchronous programming, this callback is implemented by requesting to be notified of a socket read. When this callback is invoked, it, in turn, sets up a new socket and creates a new object for the connection. Because a $fdcb$ call can only be associated with a single processor, all accept functions will, by default, be handled on the same processor for the duration of the program. The accept callback calls *accept* to accept the connection, and then spawns a new connection. It randomly assigns the new connection to a processor – all future read and write callbacks will be handled on that specific processor. We assume that the typical server application will receive a large number of connections that require approximately the same amount of work – if this assumption is true, then the read and write events will naturally be evenly distributed across all available CPUs.

This design also assumes that the accept function takes a negligible amount of time. This topic will be discussed further in the Performance section, but we found this assumption to be incorrect. As a fix, we periodically reassign the callback that handles the accept function to another processor. When this fix was incorporated, we observed even utilization of multiple CPUs.

## 3.3 libasync-mp Design Details

The event loop executed in each thread is conceptually similar to the *amain* function of *libasync*. When the kernel determines that an event can be executed without blocking, it returns from *kevent* and provides an array of events that are ready to be executed. The callback functions associated with these events are then executed. We instruct *kevent* to return only one item in order to avoid the following difficult case:

- An event closes its connection and deletes the object associated with it.
- There exists an event later in the array that deals with the same connection. $libasync{-}mp$ will crash when it attempts to dispatch that event's callback.
- One can refuse to execute any callback that has since been removed using $fdcb(callback\_fd, NULL)$, but under high load the file descriptor for the recently deleted connection may be reassigned immediately and a bad callback mistakenly executed. We believe that *libasync* should also experience the same problem, as *select/pool* return multiple file descriptors, but we cannot find where *libasync* deals with this problem. *kevent* removes events from its queue when their associated file descriptor is closed, which provides an easy solution to this problem.

Calls to *fdcb* are simplified by the fact that *kevent* is re-entrant with respect to kernel threads. *fdcb* need only call *kevent* with a parameter that instructs it to return immediately. Our performance measurements shows that calls to *kevent* are actually quite expensive – queuing adds and removes would be desirable. However, queuing in a multiprocessor environment introduces complications – callbacks such as accept may need to register callbacks in another processor. Queues work well for events that run within the same processor – because of the structure of our code, kqueue will be called when the callback executes, and any queued items added to the *kevent* list. Trying to queue an event in another processor will not work – that processor may be blocking inside *kevent*. *kevent* knows nothing about items in our queue. A solution to this problem is to only queue items that are intended for the current processor. Events destined for other processors are added using an expensive *kevent* call. We did not discover how expensive *kevent* calls were until late in the performance testing of our project. We were able to modify $libasync-mp$ to queue items in this manner, and we believe that this would result in substantially less library overhead, but the code tended to hang after several hundred connections, so our results for this paper cover a version of $libasync-mp$ without these optimizations.[3]

### 3.4 Semantic Differences with libasync

We have imposed several minor limitations not present in libasync in the interests of simplicity:

- We do not support the *libasync* wrap function. Instead, we take a function pointer whose arguments include a void pointer. Callbacks either use the void pointer to pass the *this* pointer of their class, or allocate a struct which wraps any necessary arguments. In addition to the void pointer, we also pass a number of other arguments to our callback functions, including the number of bytes available for read or write, and the EOF status of the socket. *kevent* provides us with this data, which is another advantage of *kqueue* over *select/poll*. We found the number of bytes available for reading to be extremely helpful in designing efficient buffer management code and for avoiding EAGAIN errors.[1]

- In $libasync-mp$, we require that the user pass a $processor$ argument to $fdcb$. This was done for simplicity, and there is no reason that the library could not maintain a file descriptor > processor mapping internally.

- We also do not implement reference counting. This can create headaches for the programmer, and there may even be memory leaks if the user allocates memory to pass to another callback, but the callback is never called or never de-registered. We have not found this to be a problem in practice, though it requires more careful coding on the user's part. We also do not implement *libasync*'s str and strbuf class, which can make porting *libasync* code difficult.

- *libasync* contained a very complex asynchronous DNS lookup module. We did not port this module. We use the blocking *gethostbyname* function to reverse DNS addresses to IPs.

- Finally, users must use our special *thread_malloc* and *thread_free* functions. The default malloc, even if it is compiled with *dmallocth* or any number of thread safe $\#defines$, is not thread safe. We wrapped malloc and free with our own mutexes. This error plagued us for a very long time.

### 3.5 Concurrency

The move from a single-CPU asynchronous library to a multiprocessor-aware asynchronous library has the potential to introduce and almost infinite variety of concurrency problems. Threaded programming is notoriously difficult, and mutexes are simply not appropriate for asynchronous programming. We provide a limited means for handling programmer-directed concurrency. Each connection is typically assigned to run on a single CPU. Because of this, users of $libasync-mp$ may share data between the events that compose a connection with the need for mutexes or any other form of synchronization. We do not support access to shared global objects – this is a complicated topic in and of itself. To give practical examples, $libasync-mp$ supports a sufficient concurrency model to run the tcpproxy lab, but not the webproxy lab.

## 4. Performance

$libasync-mp$ meets or exceeds the performance of *libasync* in all measures, and, most critically, provides a nearly 2x performance improvement when running CPU intensive code.

### 4.1 Test Applications

Our testing plan was initially fairly ambitious: we planned on building a "gzipserver" to test our client. The gzipserver stripped of all compression code (an echo server, essentially) would serve as the IO benchmark, the gzipserver with compression enabled would test a real-world mix of CPU and IO, and an overzealous gzipserver that compressed data several dozen times would serve as the CPU intensive benchmark. We also intended to port webproxy to provide an even more real-world test than the gzipserver. However, a host of problems unrelated with our library prevented us from writing such a comprehensive test suite. The gzip code was affected by the previously discussed malloc problem – the gzip code we found called malloc, which in turn generated a variety of odd errors. We were unable to develop webproxy because the *connect* function would consistently hang for no apparent reason. We believe that it has to do – again – with not linking with a thread safe version of the library. In both cases, the errors could haven been solved, only we lacked the time to do so.
We developed two test applications to benchmark our library. They are:

- Echo server: the IO-bound application

- Spin server: a CPU bound application that spins for a length of time proportional to the number of bytes being sent (similar to the CPU characteristics of a gzipserver, if only a lot less useful).

Tests were performed with the server running on an almost unloaded, two-processor class machine. The client code was invoked from several other class machines to provide the requisite load. The servers included throttling code to prevent themselves from accepting too many concurrent connections. Measurements were taken once the clients were observed to reach full request speed.

## 4.2 CPU-Bound Applications

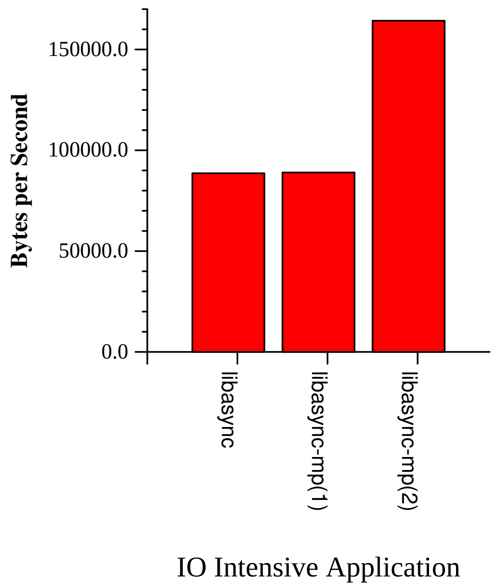


**Figure 2: Graph of performance of a CPU-bound application.**

The primary goal of $libasync-mp$ is to increase the performance of CPU bound applications, and in that respect, $libasync-mp$ is an success. As is apparent from Figure 2, a version of $libasync-mp$ that utilizes two processors is 1.85 times faster than $libasync$ and 1.84 times faster $libasync-mp$ using one CPU. We provide an explanation for this acceptable but less than optimal result in Section 2.4.

The CPU bound application was tested using the spin server. Measurements were taken by measuring the time to transmit and receive back 250 10,000 byte files. File size is not important because CPU time heavily dominates IO time in this application.

## 4.3 IO-Bound Applications

The results of our IO bound application proved to be interesting. $libasync$, $libasync-mp(1)$, and $libasync-mp(2)$ perform nearly identically with small file sizes,

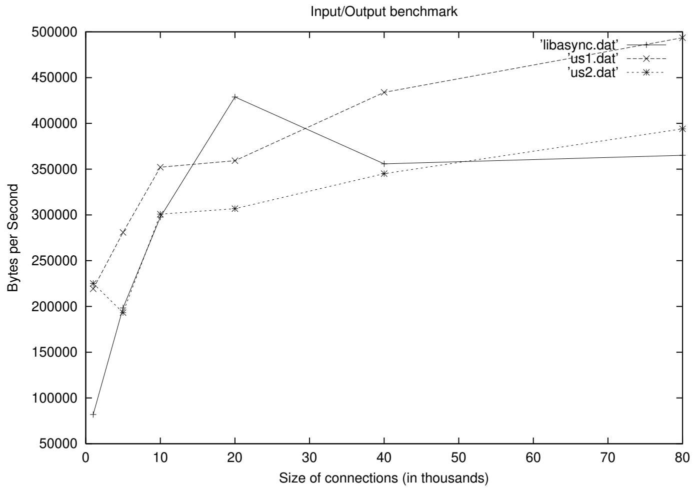


**Figure 3: Graph of performance of an IO bound application.**

perhaps due to the small number and short duration of the requests – there is little opportunity for the libraries to distinguish themselves at small file sizes. $libasync-mp(1)$ is the most efficient of the three, with $libasync$ shortly ahead of $libasync-mp(2)$. It should be emphasized that the performance gaps here are only moderate. The largest performance gap – $libasync$ and $libasync-mp(1)$ at 80k file sizes – differs by only 35%. We suspect that even our inefficient use of $kevent$ is faster than select/poll – larger files require more system calls, and the increasing difference between $libasync-mp(1)$ and $libasync$ approximately matches the relationships between $kevent$-based code and $select/poll$ based code references in the $kevent$ paper.[3] The IO bound application uses very little CPU – the multiprocessor application is most likely slower because it requires more overhead to process connections. The IO bound application was tested using the echo server application. Measurements were taken by measuring the time to transmit and receive back 250 files of varying sizes.

## 4.4 Accepts

An assumption of the initial design of $libasync-mp$ was that the accept callbacks take virtually no time to execute, and can therefore always be executed on the same CPU without hurting the performance of the system. When we instrumented our code and ran benchmarks, we found that this was not true – accepts take a significant amount of time and, for connections that transmit less than several hundred kilobytes of data, comprise a significant percentage of the total number of events executed.

Our solution to this problem was to periodically "migrate" the accept callback to be executed on another CPU. This is accomplished by deregistering the accept

callback on the old CPU, and then re-registering it on the new CPU. Once we made this simple fix, load was distributed evenly across all available processors – the difference in the number of events handled by each processor is less than 1%.
Later in the performance testing, we discovered the reason why accepts take so long. The actual call to the *accept* function takes, on average, 48 microseconds – small enough to be negligible. Our accept callbacks take approximately 1.3 milliseconds to execute. Accept functions typically register several callbacks – two to read and write on the newly created socket, and two more to "migrate" the accept function to a new processor. It was at this time we realized that there is a significant amount of overhead associated with *kevent*. When running the CPU-bound test application, accepts consume 8% of all CPU time – a figure that correlates nicely with the less-than-optimal 1.84X performance increase from $libasync - mp(2)$ over $libasync - mp(1)$.

## 5. Alternative Design

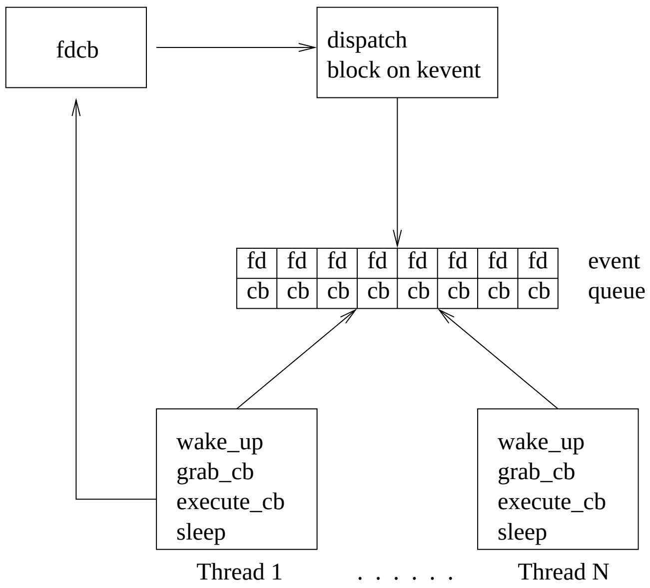


**Figure 4: Schematic of our alternative design.**

Our first attempt at a multiprocessor aware *libasync* replacement was designed much more poorly. We originally envisioned a single dispatch thread that would block on *kevent* and queue callbacks to a thread-pool. This design would be much less restrictive in terms of which callbacks any given processor could run because it would make no association between processors and descriptors. It intrinsically load balanced all events across N processors – processors that are free grab an item from the dispatch queue. However, we quickly realized that this approach has enormous synchronization problems, is relatively inefficient, and provides a concurrency model that is virtually useless. Several of the difficulties we faced and our attempted solution are listed below:

- Adding and especially removing of callbacks from within the thread-pool is extremely complicated because the thread-pool member may need to know the status of the dispatch thread.

- Callbacks are essentially queued in 2 places: inside of the kernel and in the thread-pool queue. Unfortunately, a thread-pool worker that wants to remove a callback registration cannot directly remove the item from the kernel's queue. In addition, although it can request that the item be removed from the kernel's queue, the kernel may, just at that moment, un-block on that very event and queue it to the thread-pool. The only solution to this problem is to mutex nearly every step of callback registration which produces extremely poor results.

- A callback that deletes memory or closes a descriptor must use a two-step process rather than doing the action directly because another thread may be using that resource. We devised a two-step method for closing descriptors (and deleting memory associated with them), but this violated the semantics of *libasync*. Essentially any call to "delete this;" in *libasync* code would need adjustment.

- The latency between *kevent* dispatching an event and a thread actually running the callback meant that *kevent* might fire again for the same event (especially with accepting connections). This means that code designed to accept connections would need to be careful about handling errors because many of them would be caused by calling the callback twice for the same actual connection. In addition, code such as tcpproxy that aborted on a write error would behave improperly if that error were caused by the same write event being called twice. We devised a fix for this problem.

- Two callbacks accessing a resource such as a buffer must deal with synchronization themselves (e.g. a read and a write callback for the same descriptor). This not only violates the semantics of *libasync* but also means that the programmer must deal with mutexes / critical sections which (somewhat) defeats the purpose of using this style of programming.

Our simplified design, as presented in this paper, greatly simplified the synchronization problems by tying all of the callbacks associated with a file descriptor to the same processor. Although there are no guarantees on synchronization between callbacks registered for different descriptors, this level of serialization is nonetheless useful for the class of applications that share memory only on a connection-basis. Thus, our current library is applicable only to the set of applications that can establish memory-access-closures around descriptors.

## 6. Conclusion

We have presented a C++ asynchronous programming library similar to *libasync* that can multiplex events across multiple processors. Our approach is not nearly as general as *libasync*, but we achieve nearly double the performance on CPU bound applications, and our library can serve hundreds of thousands of hits without crashing. Additionally, there is plenty of room for both algorithmic and implementation improvements.

## 7. Lessons Learned

We have learned a lot of practical UNIX development techniques, many of which we did not mention in the paper. We wrote several versions of the code, experimenting at various points with signal handlers, thread libraries, asynchronous file-io (aio), etc. We spent a lot of time debugging "impossible" errors such as the memory allocation problem (it"s very frustrating when the thread safe version of a library is not thread safe).

## References


[1] David Maziéres, w. F. D., and Peterson, E. Using tcp through sockets. http://www.pdos.lcs.mit.edu/6.824/papers/sockets.ps.

[2] IBM. Open source: Next generation posix threading project web site. http://oss.software.ibm.com/pthreads.

[3] Lennon, J. "kqueue: A generic and scalable event notification facility for freebsd". http://people.freebsd.org/~jlemon/kqueue.pdf, October 2000.

[4] McKusick, K., Design, T., and the, I. Bsd operation system, 1996.

[5] Welsh, M. "seda: An architecture for highly concurrent server applications". http://www.cs.berkeley.edu/~mdw/proj/sandstorm/, July 2001. Berkeley, California.