

An Introduction to Libaio

William K. Josephson

wkj@CS.Princeton.EDU

Princeton University
Princeton, NJ 08544

ABSTRACT

This document is a short introduction to the `libaio` event-driven programming libraries, the `9unix` libraries upon which they depend, and `config`, the basis of the `libaio` build system. The reader is expected to be reasonably well versed with the UNIX® operating system, `make`, C and C++, and concurrent programming.

February 8, 2007

An Introduction to Libaio

William K. Josephson

wkj@CS.Princeton.EDU

Princeton University
Princeton, NJ 08544

1. Introduction

This document is a short introduction to the `libaio` event-driven programming libraries, the `9unix` libraries upon which they depend, and `config`, the basis of the `libaio` build system. The `9unix` libraries provide a consistent, portable, Unix-like programming interface inspired by Plan 9, while `config` provides a flexible, configurable build system, and `libaio` provides a C++-based event-driven infrastructure intended for modern CMPs. The interface to `libaio`, although similar to some existing event-driven libraries, is new, relatively untried, and still a work in progress. The author is less than fond of continuation-passing style (aka: event-driven style) as a programming model for end users and hopes eventually to develop a source-to-source compiler in the spirit of Tame [10] to ease the task of programming with `libaio`. Unlike the SMP version of `libasync` [6], `libaio` makes concurrency explicit through barriers and locks and sports a variety of more sophisticated schedulers.

Documentation and the source for the most recently released versions of `9unix`, `libaio`, and `config` may be obtained from the author's web page:

<http://www.morphisms.net/~wkj/software>

2. The 9unix libraries

The `9unix` libraries provide a consistent, system independent, Unix-like interface across multiple platforms. The basic interface was inspired by Plan 9 and much of the code has either come directly from Plan 9 or come via Plan 9 Ports [5], although early work on `9unix` significantly predated Plan 9 Ports. `9unix` also provides a set of standard make files for program maintenance and scripts to support automated linking on a variety of platforms.

2.1. Building Programs with 9unix

`9unix` uses BSD for program maintenance. The Makefiles distributed in the `mk` subdirectory are responsible for computing dependencies and providing the rules for transforming program and documentation source into executable code and formatted text.

An internal `9unix` make file must, at a minimum, specify the root of the build directory tree as a relative path in the `root` variable and include `Makefile.inc` from the root. A Makefile that is not part of `9unix` itself sets `p9root` to point to the root of the `9unix` installation tree and include the copy of `Makefile.inc` residing in `$(p9root)/mk` instead. By convention, the path to the root of a `9unix` installation is exported to the environment via the `P9UNIX` environment variable. Most users will want to set this variable in their shell's initialization scripts (`~/.profile` for Bourne shell users).

Most Makefiles will need to set the `OFILES`, `TARG`, and `LIB` libraries. The Makefiles in `src/cmd` and `src/libbio` provide good working examples. The intrepid may venture into `mk/Makefile.inc` for further details.

The `9unix` makefiles set the variables `ARCH`, `OS`, `PICEXT`, and `SOEXT` for the user. Operating

system specific configuration resides in `mk/Makefile.${OS}`, architecture specific configuration in `mk/Makefile.${ARCH}`, and general configuration, particularly for the compiler, resides in `mk/Makefile.cfg`. Note that what some systems call x86-64 (*i.e.* IA-32 with 64-bit extensions), 9unix consistently calls `amd64` on all platforms; similarly, what some systems call `i386` or `i686`, 9unix consistently calls `386`. Furthermore, the MacOS X port on 386 and `amd64` defaults to `amd64` mode even though current versions of the system do not support 64-bit mode as fully as 32-bit mode.

The default user-callable make targets are:

- **all**: Build all programs, libraries, shared libraries, and mandatory documentation. Optional documentation may require additional programs such as `groff` or Plan 9 Ports and is not built by default.
- **install**: Execute the **all** target if necessary, construct the necessary installation directory tree, and copy programs, scripts, libraries, shared libraries, headers, macros, and installed documentation into the installation tree.
- **clean**: Remove most intermediate files.
- **nuke**: Remove all generated files.

2.2. Linking

9unix exports a C preprocessor macro `AUTOLIB` which inserts a weak symbol into any source file containing this macro. This weak symbol has a special form that can be extracted with the `nm(1)` program at link time. The 9l shell script is responsible for using these extracted symbols to build a list of library dependencies and topologically sort it for linking. For the most part, 9l knows enough about various systems to build both static and dynamic shared libraries and to insert library search paths into binaries on each of the supported systems. It is also responsible for enabling any other platform-specific linker flags and linking against auxiliary libraries as necessary (some platforms require programs using floating point mathematics routines link against the `libm.a` math library while others do not, for instance). For the gory details, read the 9l source. Originally 9l was developed simultaneously for 9unix and Plan 9 Ports; with the advent of `AUTOLIB` and dynamic linking support, 9unix switched to a lightly modified version of the Plan 9 Ports implementation.

2.3. lib9c

`Lib9c` provides the core portability layer. It is similar in spirit to Plan 9 Port's `lib9` but eschews emulation of many Plan 9 specific features in favor of a more Unix-like interface. Whereas Plan 9 Port is primarily for porting Plan 9 software to Unix, 9unix has been used primarily for new software.

The following is a list of note-worthy functions and macros available in 9unix:

queue(3): `queue(3)` contains a recent version of the BSD list macros from FreeBSD. These macros implement various singly- and double-linked lists.

tree(3): `tree(3)` contains a recent version of the BSD tree macros from FreeBSD. These macros implement red-black and splay trees.

ARGBEGIN, ARGEND: Macros for argument parsing; see `arg(3)` for details.

dial, announce, listen: `dial(3)` provides a sane interface to the Berkeley socket API; should be used instead of the raw socket API in new programs. `libaio` provides an asynchronous implementation¹.

open and **opentemp**: 9unix overrides the `open(2)` function with `open(3)` and implements the `opentemp(3)` function for safely creating temporary files.

errstr and **werrstr**: Retrieve and set, respectively, the current error string. This is the textual equivalent of Unix's `errno` variable, but is human readable and may be set to an arbitrary string by applications. The `r` format verb takes no arguments and returns the current value of the error string.

sysfatal: signal abnormal program termination, printing an error message.

¹As of this writing, the asynchronous DNS interface is not properly locked.

wait, waitnohang, waitfor, etc.: `wait(3)` is the Plan 9 equivalent of the Unix `wait(2)` family of functions, but with a more pleasant interface.

quote*, unquote*, etc.: `quote(3)` provides routines for producing quoted strings which are often convenient in configuration files, environment variables, and simple text-based network protocols. The `unquote*` routines evaluate the quotes.

tokenize and getfields: `getfields(3)` parses simply delimited strings, including `quote(3)` delimited strings.

gmtime, localtime, etc.: 9unix overrides the standard Unix time related functions. See `ctime(3)` and `time(3)` for details.

nsec: return the number of nanoseconds since the Unix epoch.

dirread, dirreadall: `dirread(3)` provides a simplified, portable interface for reading directory entries.

sendfd, recvfd: `sendfd(3)` provides a sane interface to file descriptor passing over Unix domain sockets.

USED: A macro that when applied to a variable indicates to the compiler that it should be treated as if it were used.

2.4. `libutf, libfmt, and libbio`

9unix uses the Plan 9 formatted printing and Unicode libraries. The interfaces to these libraries are documented in `print(3)`, `fmtinstall(3)`, `utf(7)`, and `rune(3)`. These interfaces support Unicode (UTF-8) natively and provide for user-specified printing verbs, unlike `stdio(3)`; new software using 9unix **must not** use `stdio`. For buffered I/O, including formatted, buffered printing, use `bio(3)`.

2.5. A Potpourri of Further Resources

- **libbin:** A binned allocator; useful for small objects. See `bin(3)` for details.
- **libxbio:** An extensible version of `bio(3)` that allows the programmer to install function pointers for read, write, and seek operations.
- **libencode:** simple-minded endian-aware serialization primitives; often used for wire protocols or stable storage (e.g. Berkeley DB key/value pairs).
- **libip:** Various useful network routines, particularly a portable `readipfc`, `udpread/udpwrite`, and formatting verbs for IP and Ethernet addresses and functions to parse them. See `ip(3)` for details.
- **libflate:** The deflate algorithm used by `gzip`, among others; see `flate(3)` for details.
- **libString:** Reference counted strings for C; probably not so useful to `libaio` users (see below).
- **libmangle:** A library interface to the standard C++ demangling algorithms; see also the 9unix `demangle` command.
- **libmempool:** Arena and pool allocators that uses `sbrk` and `mmap`; useful if you want a file-backed heap.
- **libmux:** A protocol message multiplexor; see `mux(3)` for details.
- **libobjload:** Some basic portable run-time linker routines; incomplete, but still useful if you need to run on a number of Unix-like systems.
- **libregexp9:** `regexp(3)` is an efficient implementation of `regexp(7)` regular expressions. The Plan 9 implementation is typically far more robust than Perl, Python, Ruby, and PCRE implementations. See Russ Cox's recent essay on the subject [4].

There are a few libraries in 9unix that remain mostly for backwards compatibility and which should not be used in new software. These include:

- **tls:** A simple wrapper around OpenSSL; not as portable as one might like as recent MacOS X versions don't ship with 64-bit OpenSSL. It also includes a `select(2)` state machine and therefore is not suitable for use with `libaio`.
- **misc:** A collection of useful, but now-obsolete routines; fair game for poaching.

Some useful programs:

- **lex**: A port of Plan 9 lex; useful for legacy applications, but it is often better to write your own lexical analyser.
- **yacc**: A port of Plan 9 yacc; used by rpcc, among others.
- **demangle**: A shell utility that identifies mangled C++ symbols in its input and demangles them.
- **fn.awk** and **cfn.pl**: Two utilities for extracting prototypes from properly formatted C code.

3. Config

`config` is a program for software configuration and build management inspired by the BSD kernel configuration program of the same name [12]. It takes a project description, or configuration file, and a number of additional files describing individual sources, modules, libraries, and programs, and automatically constructs a build environment and Makefile tailored for the target architecture, operating system, and configuration.

To build `config`, you will need to have a copy of a BSD-compatible `make` and `9unix` installed. With these prerequisites, all that should be necessary is to unpack the source and type `make install`; don't forget to have the `P9UNIX` environment variable set appropriately.

To construct a default configuration for `libaio`, unpack the source distribution and type `make config`. The build environment will be constructed in a sub-directory of the `compile` directory named after the configuration, operating system, and target architecture. A default build may be initiated by walking into this directory and invoking `make`.

3.1. Configuration Specifications

At the top level, the following directives are permitted:

ident: A string name identifying the configuration.

arch: Explicitly set the target architecture.

includeconfig: Causes `config` to embed a copy of the root configuration (but not any portions incorporated via the include mechanism) in `config.c`, which is linked with every program. The configuration may be retrieved from the binary with the command:

```
strings -n 4 program | grep '____' | sed -e 's;^____;;'
```

makeoption: Passes a variable definition through to the generated Makefile. A typical use is to set the value of `p9root` or to turn on additional compiler or debugging flags:

```
makeoption      p9root      ${P9UNIX}
```

param: Takes one or two arguments. The first argument is the name of a macro to define in the generated `config.h`. The second, optional argument, is the value of the macro. If no value is given, it defaults to 1.

option: Takes one or two arguments. The first argument is the name of an option to enable. The second, optional argument, is the value of the option. If no value is given, it defaults to 1. When applied at the top level, the directive defines and enables an option. When attached to another directive, it enables that directive if and only if the option is enabled and its value matches the second argument.

include: The **include** directive takes a single argument for inclusion in the configuration and further processing. It is subject to variable and automatic expansion; see below for details.

The following directives are permitted in included files:

include: See above.

option: May be attached to a source, module, library, or program, which is then enabled and compile only if the option is enabled.

module: Declares the sources and headers belonging to a module.

library: Declares the sources and modules required to build a library.

program: Declares the sources, modules, and libraries required to build a program.

Other options, which are roughly analogous to those in BSD's `config(8)` are listed below. See `config`'s source and the `libaio` configuration directory for documentation and examples.

no-obj: No object files are generated for this source.

no-implicit-rule: Do not emit implicit rules for this target.

before-dep: Indicates that a target must be built before the dependency phase.

depend: Explicit list of files upon which the source depends.

compile-with: The command to use to generate the target from the source.

link-with: Use specified command for linking; applies to programs only.

clean: List of files for the `clean` target to remove.

nuke: List of files for the `nuke` target to remove.

The `ARCH` and `OS` make variables are filled in automatically from the values returned by `uname` unless overridden by the configuration itself or by the appropriate command line switches.

Some directives can occur only at the top level. These directives lack the `IncOK` flag in `parse.c`. The utility of this is unclear and the number of such directives is shrinking.

Include file names are subject to variable expansion. Variable expansion occurs in `expandvars()` in `misc.c`.

The **option** directive tags a source, module, or library. If the option is enabled, any source, module, or library it tags is built; otherwise it is not. The value of the option is made available via `config.h`.

The **module** directive applied to a library indicates that the library depends upon the specified module. It acts in this context as a sort of "include" directive that includes the sources (including headers) contained in the module. If the module is optional and not enabled, the directive is ignored.

The **module** directive applied to a source (including headers) indicates that the source is a member of the module. A source may belong to at most one module. A source is optional either if it is explicitly marked as such or if it is a member of a module marked as such. An optional source is built only if all the corresponding options (source and module) are enabled.

Include directives invoke `autoinc()`. If "file" is included, then "file.\${OS}" is attempted as well, followed by "file.\${ARCH}". If a file has already been included, either explicitly or implicitly, it is not included again. Explicit inclusion of a file multiple times is permitted, but is almost certainly an error. More confusing still, the key used to prevent multiple inclusion is the base name of the file, not any prefix introduced by the include path. The alternative seems more surprising still.

3.2. Config and Makefiles

`Config` does not use the 9unix build system's makefiles once it has been built and installed. Rather, it has a separate set in the 9unix installation tree under the `conf` subdirectory. `Makefile.${ARCH}` contains the per-architecture makefile template used by `config`. The remaining makefiles, `config.*`, are used by the makefiles generated from the templates to drive the build. You may need to edit some of these files if you want to change the compiler options. Note that exporting `DEBUG=1` into the environment is sufficient to get a debug build. Further changes will require editing, but beware that the installed makefiles are shared by all configurations and builds.

4. The libaio libraries

Unlike 9unix, `libaio` is written in C++ as opposed to (almost) pure C99. The two primary motivations for this are described further below. Briefly, heavy use of templates are made to construct reference-counted closures. The closures and the automatic reference counting both depend upon sugar provided by C++. The libraries also make some use of templates and classes to provide polymorphic implementations of a number of standard data-structures as well as an asynchronous DNS resolver.

The public `libaio` header files may be found in `include/aio`. The private `libaio` header files in `include/aioimpl` may be included indirectly via the public ones but should **not** be included by user

applications. In general, there is a separate header file for each “module” and the interface to the module is documented by a comment at the top of the file. When in doubt, look there for documentation.

4.1. A Note on C++

Why *not* C++?

- **Performance:** BWK’s cenk (or: I’m OK, STL’s not so hot), exceptions, others?
- **Code bloat:** huge binaries are the norm and there is a tendency to over-inline with `static inline` member functions, causing unnecessary L-cache pressure.
- **Language features:** In no particular order:
 1. Operator `new` is a botch -- even `malloc` and `free` are better [13].
 2. Exceptions are expensive -- really expensive.
 3. Constructors and destructors can’t return errors without exceptions.
 4. Copy and assignment constructor problems are frequent and slicing is usually detected only at run time and often manifests itself as bizarre behavior.
 5. There are often problems forcing initialization/call order for constructors and destructors of static instances.
 6. No modules whatever, which is particularly bad in the presence of inline functions and templates.
 7. Template instantiation is a mess.
 8. Inscrutable syntax, especially operator overloading and templates.
 9. The language changes far too quickly and so the half-life for new software is short.

4.2. C++ Utilities

`libaio` provides a number of additions to the standard C++ environment.

1. An array type that wraps C-style arrays is provided since C-style arrays interact poorly with C++ templates. See the comments in `<aioimpl/array.h>` for details.
2. C++ templates can be abused to implement compile-time optimizations. The `<aioimpl/type.h>` header contains a number of templates for reflection and type manipulation at compile time. Some of the `libaio` data structures use these facilities to improve performance.
3. The `Equals` and `Compare` templates in `<aioimpl/equals.h>` provide standard interfaces for comparison objects used by the hash table and tree implementations as well as implementations for primitive types. Several other classes, including the `String` class extend these interfaces.

4.2.1. Reference Counting

XXX: somewhat complicated; will document implementation later.

Given a type `T`, `ref<T>` is a non-nullable reference counted `T`. Similarly, `ptr<T>` is a nullable reference counted `T`. Reference counted objects are allocated like so:

```
ref<T> poot = New refcounted<T>();
```

Note that a `refcounted<T>` can only take a limited number of constructor arguments since it uses automatically generated template code. The template code is generated with `bin/va.awk` and by default supports up to eight arguments to the constructor.

Passing a `ref` or `ptr` as a function argument appears to be roughly comparable in cost to passing an ordinary argument (within ten to twenty percent), assuming exceptions are not enabled.

If a class has a `finalize` method and has `refcount` as a virtual base class, then each time the reference count on the object reaches zero, the finalization method will be called instead of deleting the object and calling its destructor. You can also manipulate the reference counts on an object directly if

refcount is a public virtual base class. To prevent this and the use of `mkref` to create refs or ptrs from vanilla pointers, make the virtual base class private.

4.2.2. Closures (Curry)

The `curry` function and `Curry` template classes provide syntax and semantics analogous to function currying in modern programming languages. A `Curry<R, T1, T2, ..., Tn>` object represents a closure for a function of n arguments of types $T1$ through Tn returning a value of type R . Unused right-most arguments may be omitted from the template. As with reference counted objects, the number of arguments is fixed at configuration time in `bin/curry.awk`.

Suppose we have a function $R \text{ fn}(T1, T2, T3)$. Then:

```
curry(fn)          -> Curry<R, T1, T2, T3>::ref
curry(fn, T3)      -> Curry<R, T1, T2>::ref
curry(fn, T3, T2)  -> Curry<R, T1>::ref
curry(fn, T3, T2, T1) -> Curry<R>::ref
```

`Curry` objects have `operator()` and `apply()` functions that take any remaining arguments and evaluate the closure with the given arguments.

Beware passing references to heap allocated objects to `curry` since the resulting `Curry` object may be evaluated multiple times in the future. In typical usage, heap allocated objects that are passed to `curry` are reference counted to avoid the potential for memory corruption.

Some additional note-worthy features of the currying implementation are:

1. Notice that currying is a right-to-left operation.
2. Unfortunately, in the current implementation, repeated application of `curry` may require the definition of trivial functions to forward arguments.
3. `curry_func` is similar to `curry`, but the function to apply is the last argument.
4. `curry_func_placement` is to `curry` what `placement new` is to ordinary `new`. A `Curry` object is recycled rather than freshly allocated. This may be beneficial in performance critical applications but should be avoided in general as it is not type safe: the programmer asserts that the target object is of the correct type and size.
5. `curry_func_placement_whack` takes `curry_func_placement` a step further, allowing the programmer to change only the function inside a `Curry` object. Note that the implementation requires variadic macros, which requires a C99 preprocessor. In general, whacking closures is a bad idea and should be treated as a mere novelty.

4.2.3. Memory Allocation

All C++-style memory allocation should use the macro `New` and **not** ordinary `new`. `New` expands to code that permits limited memory debugging using `libdmalloc` (the design of C++ is fundamentally crippled when it comes to memory allocation).

All C-style memory allocation should use `aio_malloc`, `aio_free`, and `aio_realloc` to facilitate debugging and statistics gathering.

For pooled allocation, consider the allocators in `<aioimpl/pool.h>`.

1. **SimplePool**: A fixed-size pool of bytes. Explicit deallocation is not supported and neither constructors nor destructors for individual objects are run, even when the entire pool is destroyed.
2. **ExtensibleSimplePool**: A set of **SimplePools** that expands on demand.
3. **ChunkPool**: A fixed-size pool of objects of a single type, T . **ChunkPools** support explicit deallocation.
4. **ImmutablePool**: A fixed-size pool of objects of a single type, T . Explicit deallocation of individual objects is not possible, but destructors are run when the entire pool is destroyed.

4.3. Algorithms and Data Structures

libaio provides a variety of standard algorithms and data structures. In general, we eschew the STL for a variety of reasons, including portability and *reproducible, consistent* performance. The standard data structures that are available are:

1. **Bit sets:** Bit sets with platform-specific optimizations. See the comments at the top of `<aioimpl/bitset.h>` for details.
2. **Heaps:** A polymorphic heap implementation is provided. See the comments at the top of `<aioimpl/heap.h>` for details.
3. **Priority Queues:** `PriorityQueue` extends `Heap` with `push`, `top`, and `pop` operations. See `<aioimpl/queue.h>` for details.
4. **Red-Black Trees:** A polymorphic red-black tree implementation very similar to that of `queue(3)`. See `<aioimpl/rbtree.h>` for details.
5. **Hash Tables and Maps:** `<aioimpl/hash.h>` contains implementations of polymorphic, transparently resizing, chaining hash tables: `HashTables`, `HashMaps`, and `(CWBoolMaps` (a specialization of `HashMap` for boolean valued entries). The comments at the top of the header file describe the interfaces (notice that all implementations inherit `HashTableCore`'s interface). The chain links are embedded in values in `HashTables` where as a separate bucket is allocated for `HashMaps`.
6. **Linked Lists and Tail-Queues:** Similar to `queue(3)` but polymorphism is achieved using C++ templates. See the comments at the top of `<aioimpl/list.h>` for details.
7. **Vectors and stacks:** More or less what one would expect; see the comments at the top of `<aioimpl/vector.h>` for details.

libaio provides several other data structures useful for systems programming in particular:

8. **Strings:** Strings are reference counted and immutable. Both `Equals` and `Compare` are defined for `Strings`. See `<aioimpl/string.h>` for details. A mutable version and an implementation of ropes are planned.
9. **Retry:** `Retry` is a class that implements a generic interface for periodic “retry” (e.g. for retransmission). The `Retry` class implements `start`, `retry`, and `timeout`, which add a new `RetryEntry`, force an entry to “keep trying”, and remove an entry, respectively. `RetryEntry` is a template for a field embedded in another class that stores the state necessary for `Retry`. Implementation details may be found in `<aio/retry.h>` and an example usage in `src/test/retry.C` and the asynchronous DNS library.
10. **IOVec:** `IOVec` is the libaio equivalent of Unix's `struct iovec` for scatter-gather I/O. Unlike the Unix variant, however, it provides some automatic memory management facilities. `IOVec` is used by the RPC implementation and some user applications. The interface is documented in the comments at the top of `<aioimpl/iovec.h>`.

We expect to continue adding to the repertoire of algorithms and data structures. Although the efficient low-level primitives already exist for various types of Bloom Filters, we have not yet implemented the higher level abstraction. We also expect to implement Radix trees (aka: Patricia trees or crit-bit trees) as replacements for hash tables in situations where randomization is unacceptable, for instance where hash table attacks are a legitimate worry. We also plan to implement ARC [14] and possibly radix sort, suffix trees, and Google-style sparse hashes.

Finally, note that `queue(3)` and `tree(3)` are still available to programs written using libaio and may be better alternatives in some cases than the equivalent C++ implementations, particularly when constructors for statically initialized objects are involved.

4.3.1. Cryptography

XXX: not documented yet.

1. Cryptographic hashes (aka: Sechash): MD5, SHA1, SHA1, SH256, SHA384, SHA512
2. Random numbers: ARC4 PRG; TODO: SHA PRGs, AES PRG, non-uniform variates.
3. Symmetric encryption: AES in CBC and CTR modes; probably also want DES/3DES.
4. Secret sharing.
5. Possibly RSA/DH, but padding/timing attacks.
6. Possibly homomorphic encryption such as Paillier; private matching (Freedman, EURO-CRYPT'04); maybe Schnorr; some of the EC variants?
7. No SSL/TLS

4.4. Threads

The following routines are used to synchronized threads of control running in shared memory. Locks are typically spin locks, although depending upon the platform they may be blocking locks. QLocks are queuing locks and RWLocks are queueing locks that support multiple readers.

Lock blocks until the lock has been obtained. Canlock is non-blocking. It tries to obtain a lock and returns a non-zero value if it was successful and zero otherwise. Unlock releases a lock.

QLocks have the same interface but are not spin locks; if the lock cannot be acquired, the caller is suspended until the lock is released.

RWLocks manage access to a data structure that has distinct readers and writers. There may be any number of simultaneous readers, but only one writer. Moreover, if write access is granted no one may have read access until write access is released.

All types of locks must be initialized with the appropriate *lockinit function and should be deallocated with the corresponding *lockfree function. Failure to free a lock results in resource leaks.

Rendezes are rendezvous points. Each Rendez, *r*, is protected by a QLock, *r->l*, which must be held by the callers of *rsleep*, *rwakeup*, and *rwakeupall*. *Rsleep* atomically releases the lock and suspends the caller. Upon resumption, the caller holds the lock. *Rwakeup* wakes up a single thread sleeping on the Rendez, if there are any. *Rwakeupall* wakes up all threads sleeping on a Rendez. Neither form of wakeup releases the lock associated with the Rendez nor do they block.

A Ref contains a long that can be incremented and decremented atomically with *incref* and *decref*, respectively. *Decref* returns zero if the resulting value is zero and non-zero otherwise.

1. **Ref**: Atomic reference counts.
 - i. **incref(Ref *r)**: increment the reference count *r*.
 - ii. **decref(Ref *r)**: decrement the reference count *r*.
2. **Lock**: Mutual exclusion.
 - i. **void lockinit(Lock*)**: Initialize a lock (on heap, stack, or BSS) and register it with the performance monitoring subsystem. Must be called before the lock can be used.
 - ii. **void lockfree(Lock*)**: Deallocate a lock and deregister it with the performance monitoring subsystem. Failing to call *lockfree* will leak resources.
 - iii. **void lock(Lock*)**: Acquire a lock atomically. The **Lock** variant may, but need not, be a spin lock; it is intended for very short critical sections.
 - iv. **void unlock(Lock*)**: Release a lock.
 - v. **int canlock(Lock*)**: Attempt to acquire a lock atomically. If the lock can be acquired, return non-zero; otherwise do not block, but return zero immediately instead.
3. **QLock**: Supports the same operations as **Locks**. **QLocks** are intended for longer critical sections; callers blocking on these locks are guaranteed to sleep rather than spin.

qlockinit, qlockfree, qlock, qunlock, canqlock

4. **RWLock**: Reader-writer locks. **r*** variants lock for reading while **w*** variants lock for writing. **RWLocks** locked for reading must be unlocked with **runlock** and those locked for writing must be unlocked with **wunlock**.
rwlockinit, rwlockfree, rlock, runlock, canrlock, wlock, unwlock, canwlock
5. **Rendez**: Rendezvous points (condition variables).
rsleep, rwakeup, rwakeupall
6. **Thread**: **Thread** objects are private to the thread library, but a number of related functions are exported to users. The thread library provides a definition of **main**: users instead implement the **threadmain** function, which has the same signature.
 - i. **int threadcreate(void (*fn)(void*), void *arg, u32int stksize)**: create a new thread that will execute the function **fn** with the argument **arg** on a stack of size at least **stksize** bytes. Returns zero on success and a negative integer on failure.
 - ii. **uint threadid(void)**: return the opaque ID for the current thread.
 - iii. **void threadyield(void)**: cause the current thread to yield the processor.
 - iv. **void threadexits(char*)**: cause the current thread to exit with the given status string.
 - v. **void threadexitsall(char*)**: cause all threads to exit with the given status string.
 - vi. **void **threaddata(void)**: return a pointer to a per-thread pointer. Note that **libaio** allocates an array of **void** pointers and installs it into the per-thread pointer. Indices in this array are reserved in **<aio.h>**.
 - vii. **void threadmain(int, char**)**: user-supplied entry point.

4.5. Events

This section describes the event-drive core of **libaio**. The basic unit of work is the Task, which can be thought of as either a “run to completion thread” or “one-shot continuation”.

4.5.1. Types

The following is a list of a number of types that appear frequently throughout the **libaio** interface. Additional types are described below when the corresponding subsystems are introduced.

1. **typedef Curry<void>::ptr Callback**: A convenient name for nullary **Curry** objects.
2. **typedef Curry<void,int>::ptr FdHandler**: Used for functions such as **open**, **close**, **fsync**, and **so-on** that take a file descriptor.
3. **typedef Curry<void,int>::ptr ErrHandler**: Error handlers are invoked asynchronously after some events complete. The integer argument is either zero or an **errno** value.
4. **typedef Curry<void,int>::ptr SigHandler**: Signal handlers are invoked in response to delivered signals. The integer argument is the signal number that caused the handler to be invoked.
5. **typedef Curry<void,Waitmsg*>::ptr ChildHandler**: Child handlers are invoked in response to changes in the status of a child process. **Waitmsgs** contain exit status information. See **<libc.h>** for the definition of the **Waitmsg** structure.
6. **struct TimeVal**: Unix **struct timeval** with comparison operators; see **<aio/time.h>**.

4.5.2. Tasks

There are two types of Tasks: those that contain an ordinary function pointer and those that contain a **Callback** instead. When a Task is scheduled, the thunk associated with a Task is invoked with a pointer to the Task structure as its sole argument. The thunk and its callees then run until the thunk returns to the scheduler. Beware that the default stack size for Tasks, **Aio::NSTACK**, is relatively small (typically two small VM pages).

For **Callback** Tasks, the thunk to run is supplied by the library and arranges to call the **Callback**.

The reference count on the Callback is incremented when the Task is initialized and the default thunk decrements it before returning to the scheduler. Callback Tasks must manipulate reference counts explicitly since the Callback is stored as a pair of pointers to a Curry object and a refcount object in order to overlay the function Task argument block. Callback Tasks also store a file descriptor and I/O operation code that the default Callback thunk uses to automatically re-enable I/O on a file descriptor at Task completion time.

Both kinds of Tasks contain a link field for chaining, a pointer to a TaskBarrier, a pointer to the thunk, and an array **hint** of **Task::NHint u32int** values that provide hints to the scheduler for cache-friendly scheduling. Except for the array of hints, these parts are not user-serviceable even though they are marked public. Ordinary function-pointer tasks also contain an array of **Task::NArg uintptr** values that a thunk may retrieve.

The implementation details for Tasks may be found in `<aio/task.h>`, `src/libaio/task.C`, and `src/libaio/taskthread.C`.

4.5.2.1. Task Synchronization Primitives

1. **class TaskBarrier**: A barrier primitive for Tasks. TaskBarriers support three “modes”:
 1. Ordinary, non-threshold mode.
 2. Threshold mode with an infinite number of signals.
 3. Threshold mode with a bounded number of signals.

The first two are special cases of the third since the bound may be infinite (infinity is written as **TaskBarrier::Infty**). A bounded, threshold mode barrier may be signaled a bounded number of times, **n**. Once the barrier has been signaled **n** times, the task associated with the barrier is scheduled and the barrier destroys itself. If the threshold, **t**, for the barrier is less than the bound, then after every **t** signal operations, the associated task is scheduled. An ordinary, non-threshold barrier is equivalent to a bounded threshold barrier with identical bound and threshold.

The mode of a barrier is fixed when the barrier is constructed. The general barrier signal method is **admit_bounded**, however more efficient signal methods are provided. There is no static or run-time check to ensure that the proper method is invoked; the programmer must assert the mode of a barrier. Non-threshold mode barriers are most efficiently signaled with the **signal** method and threshold mode barriers that can be signaled an infinite number of times are most efficiently signaled with the **admit** method. Bounded threshold mode barriers must be signaled with **admit_bounded**.

- i. **TaskBarrier(uint n, Task *t)**: Initialize an ordinary, non-threshold barrier.
 - ii. **TaskBarrier(uint n, uint tot, Task *t)**: Initialize a barrier with threshold **n** and bound **tot**.
 - iii. **signal(void)**: Signal the barrier once. The last caller to signal the barrier will submit the associated task to the Task scheduler.
 - iv. **admit(void)**: Signal an unbounded, threshold barrier.
 - v. **admit_bounded(void)**: Signal a bounded, threshold barrier. This is the most general form of the signaling primitive.
2. **class TaskLock**: A simple lock for Tasks. The current implementation serves lockers in FIFO order, but future versions may support cache-aware queuing.
 - i. **void lock(Task *t)**: Blocking acquire. The call returns immediately whether or not the lock is acquired. Once the lock is acquired, the Task **t** will be scheduled for execution. Note that the implementation may elect to execute **t** in the context of the currently running Task.
 - ii. **bool canlock(void)**: Non-blocking acquire. Return a boolean value indicating result of the attempt.
 - iii. **bool canlock(Task *t)**: Non-blocking acquire. Return a boolean value indicating result of the attempt. Upon successful acquire, schedule the Task **t**. The

implementation may elect to execute **t** in the context of the currently running Task.

- iv. **void unlock(void)**: Unlock the TaskLock. The next Task waiting for the lock, if any, is scheduled for execution. The implementation may not run any such Tasks in the current context, but rather must submit them to the Task scheduler.

4.5.2.2. Task Scheduler Entry Points

The following functions are the only user-visible entry points into the Task scheduler. As a convenience (of dubious value) **sched*** calls have variants in the top-level **Aio** namespace that forward to the Task scheduler entrypoints of the same name.

1. **Task *allocTask(void)**: Allocate a new Task, possibly from the local thread's cache of Task structures. The state of the new Task is undefined and must be initialized by the caller.
2. **Task *allocTask(int n)**: Allocate a list of **n** new Tasks. The allocator may fail to allocate as many Tasks as requested. Failure to allocate all **n** Tasks may result from a variety of temporary conditions. This entry point may be used to reduce the impact of lock overhead when the caller expects to use a large number of Task structures.
3. **void freeTask(Task*)**: Release a Task back to the free pool.
4. **void freeTask(Task *t, Task *e)**: Release Tasks on a list starting with **t** and ending with **e** back to the pool. The Task **e** should be the last Task on the list -- *i.e.* this entry point should not be used to release a contiguous slice of a larger list of Tasks.
5. **void sched(Task*)**: Explicitly schedule a previously allocated and initialized Task.
6. **void sched(CallBack cb, int fd, int op)**: Allocate and schedule a Task to run the CallBack **cb**, recording the file descriptor **fd** and I/O operation code **op** for use by the default CallBack thunk.
7. **void sched(CallBack cb)**: Allocate and schedule a Task to run the CallBack **cb**; no file descriptor or I/O operation code will be recorded.

4.5.3. I/O Events

The following functions prepare file descriptors for asynchronous I/O and manipulate I/O call backs for a given file descriptor. I/O callbacks are expected to be very short: they are run in the context of the I/O event loop and must immediately schedule a new call back to perform the actual I/O. I/O callbacks are one-shot: that is, they are automatically de-registered when they fire. The user is responsible for re-installing the call back once the event has been handled. Note, however, that the default thunk for CallBack Tasks automatically re-enables I/O for the events specified in the I/O operation mask when the Task is initialized.

Operations such as **fsync** that may block even on a file descriptor that is marked asynchronous and operations such as **open** and **close** should be run by the I/O daemon to avoid unnecessarily blocking compute tasks.

1. **void mkAsync(int)**: mark a file descriptor for asynchronous I/O.
2. **void closeOnExec(int, bool)**: mark a file descriptor "close on exec".
3. **CallBack read(CallBack cb, int fd)**: install a callback for reading on file descriptor **fd**.
4. **CallBack write(CallBack cb, int fd)**: install a callback for writing on file descriptor **fd**.
5. **CallBack removeRead(int fd)**: remove any read callbacks installed for **fd**.
6. **CallBack removeWrite(int fd)**: remove any write callbacks installed for **fd**.

4.5.4. Timers

The **libaio** runtime provides simple one-shot and interval timers. Timer callbacks are executed in a new Task submitted to the task scheduler. As a result, the timer interface is not suitable for real-time events.

1. **void addOneShot(CallBack cb, const TimeVal& tv)**: Add a one-shot timer CallBack,

cb, to be executed after the wall clock time **tv**.

2. **void addInterval(CallBack cb, const TimeVal& tv)**: Add a recurring interval timer CallBack, **cb** with wall clock interval **tv**. Beware that unlike I/O handlers, the interval timer callback is immediately rescheduled.
3. **void removeTimer(CallBack)**: Remove all timers with the given CallBack.
4. **void removeTimer(TimeVal&)**: Remove all one-shot timers with the given deadline.

4.5.5. Networking

The following interfaces are asynchronous replacements for the BSD socket interface. The interface is similar to the Plan 9 network interface `dial(3)`. The major change is **adial**.

1. **void adial(Resolver*, FdHandler, char*, char*, char*, int*)**: Performs an asynchronous DNS lookup and in the case of TCP sockets an asynchronous connection. **Adial** returns an open file descriptor to the FdHandler and arranges for it to be scheduled in a new Task. For details about the asynchronous resolver interface, see the discussion under “High Level Services” below.
2. **void announce(CallBack, char*, char*)**: Not Implemented
3. **void listen(CallBack, char*, char*)**: Not Implemented
4. **void accept(CallBack, int, char*)**: Not Implemented
5. **void sendfile(IOVec, IOVec, int, int)**: Not Implemented

4.5.6. Signals and Process Control

The following functions

1. **SigHandler sigInstall(SigHandler cb, int sig)**: Install a signal handler, **cb**, for the signal **sig**. If the signal handler is **nil**, then revert to the default behavior for the signal.
2. **void addChildHandler(ChildHandler cb, ulong pid)**: Install a child handler, **cb**, for process **pid**.
3. **void removeChildHandler(ulong pid)**: Remove the child handler installed for process **pid**, if any.
4. **int afork(void)**: Asynchronous fork.
5. **int aspawn(char *path, char *argv[], char *env[], CallBack cb)**: Asynchronous fork followed immediately by exec of the binary **path** with the arguments **argv** and environment **env**. The callback **cb** is run in the child before the exec.

4.5.7. AIO Daemons

The AIO daemons provide a limited number of blocking contexts for use by the event library. The number of copies of the AIO daemon to run is specified to the library by **ainit** (see below). The daemons process a global queue of requests in FIFO order. Results are returned via callbacks scheduled by the task scheduler.

1. **void fsync(ErrHandler cb, int fd)**: Call `fsync(2)` on **fd** and pass the result to **cb** in a new Task.
2. **void open(ErrHandler cb, FdHandler fdcb, String path, int flags, int mode)**: Call `open(2)` with **path**, **flags**, and **mode** as arguments. On error, the ErrHandler **cb** is called with the value of **errno** in a new Task; otherwise, the FdHandler **fdcb** is called with the new file descriptor, also in a new Task.
3. **void close(ErrHandler cb, int fd)**: Call `close(2)` on **fd** and pass the result of **cb** in a new Task.
4. **void access(Curry<void,int>::ptr, const char*, int)**: Not Implemented.
5. **void sendfile(ErrHandler, IOVec, IOVec, int, int)**: Not Implemented.

6. **dirstat, dirfstat, dirwstat, and dirfwstat:** Not Implemented.

4.5.8. Miscellaneous

1. **void ainit(int nthr, int naiod):** Initialize libaio. **Nthr** is the number of task dispatch threads to run. Typically this will be the same as the number of cores or possibly hardware threads. Future versions of libaio will automatically detect the processor topology. **Naiod** is the number of background AIO daemons to run.
2. **void amain(void):** Start libaio. Typically called at the end of **threadmain**; does not return.
3. **void aexit(char*):** Cause libaio to exit.
4. **void getTimeOfDay(TimeVal&):** Fill in a **TimeVal** with the current time of day as returned, for instance, by the Unix **gettimeofday** system call.
5. The following functions are simply forward their arguments to the corresponding task scheduler function.
 - i. **void sched(Task *t)**
 - ii. **void sched(CallBack cb)**
 - iii. **void sched(CallBack cb, int fd, int op)**

The following are a number of variables exposed by libaio. With the exception of **now**, these variables are implementation details that may disappear at any time.

1. **TimeVal now:** **Now** contains the approximate current time of day. It is updated asynchronously by the internal event loop. Users that do not need precise timing information may consult this variable rather than repeatedly call into the kernel.
2. **u64int ntod:** Number of times **getTimeOfDay** has been called by the library.
3. **u64int ncheckio:** Number of times I/O event check loop has run.
4. **u64int noneshot:** Number of one-shot timers.
5. **u64int ninterval:** Number of interval timers.

4.6. Performance Monitoring

Current: lock allocations/usage, rdtsc Future: processor event sampling, etc.

4.7. Processor-specific support

libaio contains some processor-specific support. Some is architecture specific, for instance **cpu_spinwait** is implemented in terms of the pause instruction on 386 and amd64 targets. Similarly, libaio exports an **rdtsc** function on all targets that have a user-accessible cycle counter. On x86 targets this is implemented with the **rdtsc** instruction; on Sparc V8+ and later it is implemented with the **tick** instruction.

libaio borrows the **atomic(9)** macros from FreeBSD to support a reasonably portable atomic instruction interface. The user is responsible for understanding the semantics of atomic instructions and memory barriers on each target architecture, but the library provides the mechanism. This interface is used by the library for signal handling and for reference counting. It should not be used to re-implement locks since doing so may interfere with the schedulers.

1. **ffs{32,64}:** Find first set bit in an integer operand of the given size (1 indexed).
2. **fls{32,64}:** Find last set bit in an integer operand of the given size (1 indexed).
3. **ffc{32,64}:** Find first clear bit in an integer operand of the given size (1 indexed).
4. **flc{32,64}:** Find last clear bit in an integer operand of the given size (1 indexed).
5. **ilog2_{32,64}:** Integral part of logarithm if argument is non-zero, -1 otherwise.
6. **bswap{16,32,64}:** Byte-order swap for 16-, 32-, or 64-bit operands.

7. **hamming_distance**: Compute hamming distance between two arrays of given length.

4.8. High Level Services

Asynchronous DNS, XDR+RPC, scripting (Python)

4.9. Some Useful Programs

The `src/tools` directory contains programs for common maintenance and measurement tasks. The most commonly used one is a port of Poul-Henning Kamp's `ministat` program for computing basic summary statistics and the Pooled Student's T-test for statistical significance. `Ministat` can be used to evaluate the effectiveness of an optimization.

A number of potentially useful scripts may also be found in the `bin` directory of the `libaio` distribution. In addition to several used by the build process, they include:

1. **run.pl, runit**: Dirty simple performance data collection harnesses.
2. **split.awk**: Splits a performance data file as produced by `run.pl` or `runit` into individual files suitable for `ministat`.
3. **syms.pl**: Processes a list of addresses, looking them up in a binary and mapping them to source lines. The initial lookup is performed by `addr2line` and the `cached`.
4. **genprog.awk**: Given a list of sources, produces a sample **program** directive entry for `config`.
5. **genhdr**: Generates a master list of `libaio` headers for `config`.

5. Getting Started With `libaio`

In this section we attempt to provide enough details and examples to get new users started with the libraries. If you run into problems or have more examples that belong here, please send the author electronic mail.

5.1. Installing `9unix` and `libaio`

Installing the various components is relatively straight-forward on supported systems (for a list of such systems, see the next section). If your system does not already have a BSD-compatible `make`, fetch a copy from the author's web site and follow the directions for installing it. The most recent released versions of `9unix`, `libaio`, and `config` are available from the same website.

<http://www.morphisms.net/~wkj/software/index.html>

Once you have a working copy of `make`, be sure to add it to your path and arrange to invoke it instead of another `make` variant such as GNU `make`.

Next, you must set the `P9UNIX` to point to the intended, writeable installation directory. Unpack `9unix` and then compile and install it with `make`; `make install`. Do the same for `config`; it is also installed into the `P9UNIX` hierarchy. Finally, unpack `libaio` and type `make config` at the root. This should generate a default configuration. Happy hacking.

6. Supported Systems and Porting

To install `9unix` and `libaio`, you must have a recent copy of `gcc`, or possibly Intel's compiler, `icc`, and BSD-compatible `make`. A number of operating systems and target architectures are explicitly supported; others either "should" work or may require modest porting effort. In general, it should be possible to port the libraries to any system that provides a basic shared-memory threading system, an I/O event notification interface, Unix-like process control, and basic atomic operations should.

Most development work is done under FreeBSD 6.x and FreeBSD-CURRENT (7.x). As a result, some of the other ports are occasionally broken on the HEAD branch, but the effort necessary to fix these ports is usually trivial. Releases are lightly tested (*i.e.* compile tested plus some additional effort to

run the basic regression tests). Supported systems currently include:

- FreeBSD 6.x, 7.x on i386 and amd64 targets
- Linux 2.6 (RedHat Linux) on i386 and amd64 targets
- NetBSD 1.6 on i386 targets
- OpenBSD 4.0 on i386 targets
- MacOS X 10.4 on i386/amd64 targets
- Solaris 10/SunOS 5.10 on UltraSPARC T1 (Niagara)

C code in all of the libraries must be C99 compliant, although some code is annotated with additional macros that expand to GCC attributes when GCC is employed and are otherwise empty. C++ portability is a more difficult issue, but we attempt to compile test with Sun's compiler. Due to inline assembly, particularly in `atomic(9)`, it is not currently possible to run many of the resulting binaries as the offending code is replaced with `no-ops`.

The build is regularly tested with GCC 3.4 and GCC 4.0 on i386 and amd64 platforms and somewhat less often on UltraSPARC, including UltraSPARC T1 (Niagara). Some features, particularly those requiring assembly, may not be available on all platforms either due to limitations in the target architecture or native tools. For instance, many recent MacOS X systems don't have an assembler that handles DWARF symbols. In theory, GCC-style inline assembly syntax support is not necessary; external implementations should replace the inline assembly automatically, but this is rarely tested and probably broken, although not difficult to fix. The atomic operations interface is the most complex to port and currently dependent upon GCC-style inline assembly. If you have ports to new systems, efficiency improvements, or implementations that don't require GCC extensions, please send a patch.

References

1. R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services *Proc. 19th ACM Symp. on Operating System Principles*, 2003.
2. R. von Behren, J. Condit, and E. Brewer. Why Events Are A Bad Idea (for High-Concurrency Servers) *Proc. HotOS IX*, May 2003.
3. R. von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A Framework for Network Services *Proc. of the 2002 USENIX Annual Technical Conf.*, Monterey, CA, 2002.
4. R. Cox. Implementing Regular Expressions <http://swtch.com/~rsc/regexp>, January 2007.
5. R. Cox. Plan 9 From User Space (Also known as Plan 9 Ports). <http://swtch.com/plan9port>.
6. F. Dabek, N. Zeldovich, F. Kaashoek, and D. Mazieres. Event-driven Programming for Robust Software *Proc. of the 10th ACM SIGOPS European Workshop*, St.-Emilion, France, 2002.
7. V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines Technical Report TR 96-1, Dept. of Computer Science, University of Arizona, January 1996.
8. M. Frigo. Portable High-Performance Programs MIT LCS PhD Thesis, 1999.
9. S. Harizopoulos and A. Ailiamaki. Affinity Scheduling in Staged Server Architectures CMU Dept. of Computer Science Technical Report CMU-CS-02-113, March 2002.
10. M. Krohn, E. Kohler, and M. F. Kaashoek Events Can Make Sense <http://www.okws.org/doku.php?id=okws:tame>, August 2006.
11. J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance *Proc. of the 2002 USENIX Annual Technical Conf.*, Monterey, CA, 2002.
12. S. J. Leffler and M. J. Karels. Building 4.4 BSD Kernels with Config Computer Systems Research Group, EECS U. C. Berkeley, July 5, 1993.
13. D. Mazieres. My Rant on C++'s Operator New <http://www.scs.cs.nyu.edu/~dm/c++-new.html>
14. N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache *Proc. 2nd USENIX Conf. on File and Storage Technologies*, San Francisco, CA, 2003.
15. G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining *Proc. of the 38th IEEE/ACM International Symp. on Microarchitecture*, November, 2005.
16. J. Philbin, J. Edler, O. Anshus, C. Douglas, and K. Li. Thread Scheduling for Cache Locality *Architectural Support for Programming Languages and Operating Systems*, pp. 60-71, 1996.

- 17.V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System *ACM Transactions on Computer Systems*, 18(1):37-66, 2000.
- 18.R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. Agust, G. Z. N. Cai. Support for High-Frequency Streaming in CMPs *Proc. of the 39th IEEE/ACM International Symp. on Microarchitecture*, December, 2006.
- 19.M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services *Symp. on Operating System Principles*, 2001.
- 20.N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor Support for Event-Driven Programs *Proc. of the 2003 USENIX Annual Technical Conf.*, 2003.