Using libasync

David Mazières (mostly),
Frank Dabek, Eric Peterson,
and Thomer M. Gil

1 Introduction

This document builds on *Using TCP Through Sockets*, and *Using select*, and introduces a library for doing event-driven I/O, libasync.

Readers of this document are assumed to be proficient in the C and C++ programming languages including template classes and be comfortable using a Unix operating system to develop programs. Detailed knowledge of network standards or Unix I/O is not required.

Noting the following typesetting conventions may assist the reader: words set in sans serif type (such as open) are Unix system calls; any text set in a mono spaced font (such as `nbytes`) is either a C code fragment or a C object or name.

2 libasync – an asynchronous socket library

libasync alllows the programmer to associate function callbacks with readability and writability conditions on sockets. It also provides a number of helpful utility functions for creating sockets and connections and buffering data under conditions where the write system call may not write the full amount requested.

2.1 Memory allocation & debugging

libasync provides C++ support for a debugging malloc. To make the most of this support, you should emply two macros `New` and `vNew` instead of the built-in C++ operator `new`. Use `New` where you would ordinarily use `new`. It is a wrapper around `new` that also records line-number information, making it easy to debug memory leaks and other problems. `vNew` is simply a call to `New` cast to void. Use it to avoid compiler warnings when you ignore the pointer returned by `New`. You should still use `new` for per-class allocators and placement `new` (don't worry if you don't know what these are).

libasync also provides three classes for reporting errors to the user, warn, fatal, and panic. warn simply prints a message to the termial, much like the C++ standard library's cerr. For example, one might say:

```
int n = write (fd, buf, nbytes);
```

```
if (n < 0 && errno != EAGAIN) {
  warn << "could not write to socket: "
       << strerror (errno) << "\n";
  // ...
}
```

Unlike cerr, however, warn is asynchronous. warn also makes efforts to accumulate and write data to the terminal in chunks with reasonable boundaries. Thus, when several processes write to the same terminal using warn, the combined output is considerably easier to read than if they had both used cerr. fatal is like warn, except it prepends the word "`fatal:` " and exits after printing. panic causes a core dump after printing the message. It should be used for assertion failures, when the occurrence of an event indicates a bug in the program.

2.2 The suio class

The suio class maintains arrays of `iovec` structures (see readv or writev man pages if you really want to) and helps deal with the annoyance of short writes. The suio class has the following core methods:

- `void print (const void *base, size_t len);`

  suio::print adds `len` bytes of data stored at `base` to the end of the array of `iovec` structures maintained by suio. suio may or may not copy the contents of the memory at base. Thus, **you must not under any circumstances modify the data** until it has been removed from the suio. Modifying data will cause a crash when the library is compiled for memory debugging.

- `void copy (const void *base, size_t len);`

  This is the same as suio::print, except it makes a copy of the data into a temporary buffer managed by suio. Thus, one can modify the memory at `base` immediately after calling suio::copy.

- `void rembytes (size_t n);`

  Removes `n` bytes from the beginning of the `iovec` array.

- `const iovec *iov () const;`

  Returns an array of `iovec` structures corresponding to all the data that has been accumulated via suio::print calls.

- `size_t iovcnt () const;`

  Returns the length of the array returned by suio::iov.

- `size_t resid () const;`

  Returns the total number of bytes in the `iovec` array.

As an example, the following shows a rather convoluted function `greet ()`, which prints the words “hello world” to standard output. It behaves correctly even when standard output is in non-blocking mode.

```
void
writewait (int fd)
{
  fd_set fds;
  assert (fd < FD_SETSIZE);
  FD_ZERO (&fds);
  FD_SET (fd, &fds);
  select (fd + 1, NULL, &fds, NULL, NULL);
}

void
greet ()
{
  char hello[] = "hello";
  char space[] = " ";
  char world[] = "world";
  char nl[] = "\n";

  suio uio;
  uio.print (hello, sizeof (hello) - 1);
  uio.print (space, sizeof (space) - 1);
  uio.print (world, sizeof (world) - 1);
  uio.print (nl, sizeof (nl) - 1);

  while (uio.resid ()) {
    writewait (1);
    int n = writev (1, uio.iov (),
                    min<int> (uio.iovcnt (), UIO_MAXIOV));
    if (n < 0 && errno != EAGAIN)
      fatal << "stdout: " << strerror (errno) << "\n";
    if (n > 0)
      uio.rembytes (n);
  }
}
```

Writing the contents of a suio structure in this way is so common that there is a method output for doing it. For symmetry, there is also an input function which reads data from a file descriptor into memory managed by the suio.

- `int output (int fd);`

suio::output writes as much of the data in a suio structure to `fd` as possible. It returns 1 after successfully writing some data to the file descriptor, 0 if it was unable to write because of `EAGAIN`, and −1 if there was some other write error.

- `int input (int fd);`

  Reads data from file descriptor `fd` and appends it to the contents of the suio. Returns the number of bytes read, 0 on end of file, or −1 on error (including `EAGAIN`).

2.3 The str and strbuf classes

One complication of programming with callbacks is dealing with freeing dynamically allocated memory. This is particularly true of strings. If a function takes an argument of type `char *` and does something asynchronously using the string, the caller may not be allowed to free the string until later. Keeping track of who is responsible for what strings and for how long is tricky and error-prone.

For that reason, libasync has a special class str for reference-counted strings.[1] Strings are references to immutable character streams. Thus, for instance, the following code prints "`one is ‘one’`":

```
  str two = "one";
  str one = two;
  two = "two";
  warn << "one is ‘" << one << "’\n";
```

The strbuf structure allows one to build up a string by appending to it. The following code illustrates the use of strbuf:

```
void
func (str arg)
{
  strbuf sb;
  sb << "arg is ‘" << arg << "’.";
  str result = sb;
  warn << result << "\n";
}
```

strbuf is a built around the suio structure. One can access the underlying suio with the strbuf::tosuio method:

- `suio *tosuio () const;`

  Returns a pointer to the suio in which the strbuf is storing accumulated data.

[1]Note that C++ has a string class, but the standard does not specify that it has to be reference counted. Thus, a C++ string implementation can incur the overhead copying a string each time it is passed as a function argument.

2.4 Reference counting

Strings are not the only data structure for which deallocation becomes tricky in asynchronous programming. libasync therefore provides a generic mechanism for reference-counted deallocation of arbitrary dynamically allocated data structures. Refcounting is a simple form of automatic garbage collection: each time a refcounted object is referenced or goes out of scope its reference count is incremented or decremented, respectively. If the reference count of an object ever reaches zero, the object's destructor is called and it is deleted. Note that unlike real garbage collection in languages like Java, you cannot use reference-counted deallocation on data structures with pointer cycles.

Reference counted objects are created by allocating an instance of the `refcounted` template:

```
class foo : public bar { ... };

  ...
  ref<foo> f = New refcounted<foo> ( ... );
  ptr<bar> b = f;
  f = New refcounted<foo> ( ... );
  b = NULL;
```

Given a class named foo, a `refcounted<foo>` takes the same constructor arguments as `foo`, except that constructors with more than 7 arguments cannot be called due to the absence of a varargs template. A `ptr<foo> p` behaves like a `foo *p`, except that it is reference counted: `*p` and `p->field` are valid operations on `p` whichever its type. However, array subscripts will not work on a `ptr<foo>`. You can only allocate one reference counted object at a time.

A `ref<foo>` is like a `ptr<foo>`, except that a `ref<foo>` can never be NULL. If you try to assign a NULL `ptr<foo>` to a `ref<foo>` you will get an immediate core dump. The statement `ref<foo> = NULL` will generate a compile time error.

A `const ref<foo>` cannot change what it is pointing to, but the `foo` pointed to can be modified. A `ref<const foo>` points to a `foo` you cannot change. A `ref<foo>` can be converted to a `ref<const foo>`. In general, you can implicitly convert a `ref<A>` to a `ref<B>` if you can implicitly convert an A to a B. You can also implicitly convert a `ref<foo>` or `ptr<foo>` to a `foo *`. Many functions can get away with taking a `foo *` instead of a `ptr<foo>` if they don't eliminate any existing references.

On both the Pentium and Pentium Pro, a function taking a `ref<foo>` argument usually seems to take 10-15 more cycles the same function with a `foo` argument. With some versions of g++, though, this number can go as high as 50 cycles unless you compile with '-fno-exceptions'.

Sometimes you want to do something other than simply free an object when its reference count goes to 0. This can usually be accomplished by the reference counted object's destructor. However, after a destructor is run, the memory associated with an object is freed. If you don't want the object to be deleted, you can define a finalize method that gets invoked once the reference count goes to 0. Any class with a finalize method must declare a virtual base class of `refcount`. For example:

```
class foo : public virtual refcount {
  ...
  void finalize () { recycle (this); }
};
```

Occasionally you may want to generate a reference counted `ref` or `ptr` from an ordinary pointer. This might, for instance, be used by the recycle function above. You can do this with the function `mkref`, but again only if the underlying type has a virtual base class of `refcount`. Given the above definition, recycle might do this:

```
void
recycle (foo *fp)
{
  ref<foo> = mkref (fp);
  ...
}
```

Note that unlike in Java, an object's finalize method will be called every time the reference count reaches 0, not just the first time. Thus, there is nothing morally wrong with "resurrecting" objects as they are being garbage collected.

Use of `mkref` is potentially dangerous, however. You can disallow its use on a per-class basis by simply not giving your object a public virtual base class of `refcount`.

```
class foo {
  // fine, no mkref or finalize allowed
};

class foo : private virtual refcount {
  void finalize () { ... }
  // finalize will work, but not mkref
};
```

If you like to live dangerously, there are a few more things you can do (but probably shouldn't). If foo has a virtual base class of `refcount`, it will also inherit the methods `refcount_inc()` and `refcount_dec()`. You can use these to create memory leaks and crash your program, respectively.

2.5 Callbacks

Using libasync to perform socket operations also entails the use of the template class `callback`—a type that approximates function currying. An object of type `callback<R, B1, B2>` contains a member `R operator() (B1, B2)`. Thus callbacks are function objects with the first template type specifying the return of the function and the remaining arguments specifying the types of the arguments to pass the function.

Callbacks are limited to 3 arguments by a compile-time default. Template arguments that aren't specified default to void and don't need to be passed in. Thus, a `callback<int>` acts like a function with signature `int fn ()`, a `callback<int, char *>` acts like a function with signature `int fn (char *)`, and so forth.

Each callback class has two type members, `ptr` and `ref` (accessed as `::ptr` and `::ref`), specifying refcounted pointers and references to the callback object respectively (see above for a description of the refcount class)

The function `wrap` is used to create references to callbacks. Given a function with signature `R fn (A1, A2, A3)`", `wrap` can generate the following references:

```
wrap (fn) → callback<R, A1, A2, A3>::ref
wrap (fn, a1) → callback<R, A2, A3>::ref
wrap (fn, a1, a2) → callback<R, A3>::ref
wrap (fn, a1, a2, a3) → callback<R>::ref
```

When the resulting callback is actually called, it invokes `fn`. The argument list it passes fn starts with whatever arguments were initially passed to wrap and then contains whatever arguments are given to the callback object. For example, given fn above, this code ends up calling `fn (a1, a2, a3)` and assigning the return value to r:

```
R r;
A1 a1;
A2 a2;
A3 a3;
callback<R, A2, A3>::ptr cb;

cb = wrap (fn, a1);
r = (*cb) (a2, a3);
```

One can create callbacks from class methods as well as from global functions. To do this, simply pass the object as the first parameter to wrap, and the method as the second. For example:

```
struct foo {
  void bar (int, char *);
  callback<void, char *>::ref baz () {
    return wrap (this, &foo::bar, 7);
  }
};
```

Note the only way to generate pointers to class members in ANSI C++ is with fully qualified member names. `&foo::bar` cannot be abbreviated to `bar` in the above example, though some C++ compilers still accept that syntax.

If wrap is called with a refcounted ref or ptr to an object, instead of a simple pointer, the resulting callback will maintain a reference to the object, ensuring it is not deleted. For example, in the following code, baz returns a callback with a reference to the current object. This ensures that a foo will not be deleted until after the callback has been deleted. Without the call to mkref, if a callback happened after the reference count on a foo object went to zero, the foo object would previously have been deleted and its vtable pointer likely clobbered, resulting in a core dump.

```
struct foo : public virtual refcount {
  virtual void bar (int, char *);
  callback<void, char *>::ref baz () {
    return wrap (mkref (this), &foo::bar, 7);
  }
};
```

2.5.1 An example

```
void
printstrings (char *a, char *b, char *c)
{
  printf ("%s %s %s\n", a, b, c);
}

int
main ()
{
  callback<void, char *>::ref cb1 = wrap (printstrings, "cb1a", "cb1b");
  callback<void, char *, char *>::ref cb2 = wrap (printstrings, "cb2a");
  callback<void, char *, char *, char *>::ref cb3 = wrap (printstrings);

  (*cb1) ("cb1c");                  // prints: cb1a cb1b cb1c
  (*cb2) ("cb2b", "cb2c");          // prints: cb2a cb2b cb2c
  (*cb3) ("cb3a", "cb3b", "cb3c");  // prints: cb3a cb3b cb3c

  return 0;
}
```

2.6 libasync routines

Libasync provides a number of useful functions for registering callbacks (fdcb, amain) as well as performing common tasks relating to sockets (tcpconnect, inetsocket). Each of these functions is prototyped in either async.h or amisc.h.

- `void fdcb (int socket, char operation, callback<void>::ptr cb)` associates a callback with the specified condition (readability or writability) on socket. The conditions may be specified by the constants `selread` or `selwrite`. Exception conditions are not currently supported. To create the refcounted callback to pass to `fdcb` use `wrap`. For instance: `wrap(&read_cb, fd)` produces a refcounted callback which matches a function of the signature `void read_cb(int fd);`

To unregister a callback, call `fdcb` with the `cb` argument set to NULL. Note that callbacks do not unregister once they are called and that no more than one callback can be associated with the same condition on any one socket.

- `timecb_t *delaycb (time_t sec, u_int32_t nsec, callback<void>::ref cb)`

  Arranges for a callback to be called a certain number of seconds and nanoseconds in the future. Returns a pointer suitable for passing to `timecb_remove`.

- `void timecb_remove (timecb_t *)`

  Removes a scheduled timer callback.

- `void amain ()`

  Repeatedly checks to see if any registered callbacks should be triggered. `amain()` does not return and must not be called recursively from a callback.

- `void tcpconnect (str hostname, int port, callback<void, int>::ref cb)`

  creates an asynchronous connection to the specified host and port. The connected socket will be returned as the argument to the callback specified by cb, or that argument will be -1 to indicate an error.

- `int inetsocket (int type, int_16 port, u_int32_t addr)`

  creates a socket, but unlike `tcpconnect` does not connect it. `inetsocket` does, however, bind the socket to the specified local port and address. `type` should be `SOCK_STREAM` for a TCP socket, and `SOCK_DGRAM` for a UDP socket. `inetsocket` does not put the socket into non-blocking mode; you must call `make_async` yourself.

- `void make_async (int fd)`

  sets the `O_NONBLOCK` flag for the socket s and places the socket in non-blocking mode.

- `void close_on_exec (int fd)`

  sets the close-on-exec flag of the socket to true. When this flag is true any process created by calling `exec` will not inherit s.

- `void tcp_nodelay (int fd)`

  sets the TCP_NODELAY socket option to true. This command allows TCP to keep more than one small packet of data outstanding at a time. See the tcp(4) manual page or RFC0896.

2.7 multifinger.C

Using libasync, the multifinger example can be implemented in approximately 100 lines of C++ code.

Multifinger begins by registering as many callbacks for finger requests as possible (that number is limited by a conservative estimate of how many file descriptors are available). As each request completes, the `done()` function registers additional callbacks. A global variable, `ncon`, tracks the number of pending transactions: when `ncon` reaches 0 the program terminates by calling `exit`.

For each transaction, the multifinger client moves through a series of states whose boundaries are defined by system calls that potentially return `EAGAIN`. The code progresses through these states by "chaining" callbacks: for instance, when the `write` of the username is complete, the `write_cb` unregisters the write conditioned callback and registers a callback for readability on the socket.

```
#include "async.h"

#define FD_MAX 64
#define NCON_MAX FD_MAX - 8
#define FINGER_PORT 79

char **nexttarget;
char **lasttarget;
void launchmore ();

struct finger {
  static int ncon;

  const str user;
  const str host;
  int fd;
  strbuf buf;

  static void launch (const char *target);
  finger (str u, str h);
  ~finger ();
  void connected (int f);
  void senduser ();
  void recvreply ();
};

int finger::ncon;

void
finger::launch (const char *target)
{
  if (const char *at = strrchr (target, '@')) {
    str user (target, at - target);
    str host (at + 1);
    vNew finger (user, host);
  }
  else
    warn << target << ": could not parse finger target\n";
}

finger::finger (str u, str h)
  : user (u), host (h), fd (-1)
{
  ncon++;
  buf << user << "\r\n";
  tcpconnect (host, FINGER_PORT, wrap (this, &finger::connected));
}

void
finger::connected (int f)
{
  fd = f;
  if (fd < 0) {
    warn << host << ": " << strerror (errno) << "\n";
    delete this;
    return;
  }
  fdcb (fd, selwrite, wrap (this, &finger::senduser));
}

void
finger::senduser ()
{
  if (buf.tosuio ()->output (fd) < 0) {
    warn << host << ": " << strerror (errno) << "\n";
    delete this;
    return;
  }
  if (!buf.tosuio ()->resid ()) {
    buf << "[" << user << "@" << host << "]\n";
    fdcb (fd, selwrite, NULL);
    fdcb (fd, selread, wrap (this, &finger::recvreply));
  }
}

void
finger::recvreply ()
{
  switch (buf.tosuio ()->input (fd)) {
  case -1:
    if (errno != EAGAIN) {
      warn << host << ": " << strerror (errno) << "\n";
      delete this;
    }
    break;
  case 0:
    buf.tosuio ()->output (1);
    delete this;
    break;
  }
}

finger::~finger ()
{
  if (fd >= 0) {
    fdcb (fd, selread, NULL);
    fdcb (fd, selwrite, NULL);
    close (fd);
  }
  ncon--;
  launchmore ();
}

void
launchmore ()
{
  while (nexttarget < lasttarget && finger::ncon < NCON_MAX)
    finger::launch (*nexttarget++);
  if (nexttarget == lasttarget && !finger::ncon)
    exit (0);
}

int
main (int argc, char **argv)
{
  make_sync (1);
  nexttarget = argv + 1;
  lasttarget = argv + argc;
  launchmore ();
  amain ();
}
```

3 Finding out more

This document outlines the system calls needed to perform network I/O on UNIX systems. You may find that you wish to know more about the workings of these calls when you are programming. Fortunately these system calls are documented in detail in the Unix manual pages, which you can access via the man command. Section 2 of the manual corresponds to system calls. To look up the manual page for a system call such as `socket`, you can simply execute the command "`man socket`." Unfortunately, some system calls such as `write` have names that conflict with Unix commands. To see the manual page for `write`, you must explicitly specify section two of the manual page, which you can do with "`man 2 write`" on BSD machines or "`man -s 2 write`" on System V. If you are unsure in which section of the manual to look for a command, you can run "`whatis write`" to see a list of sections in which it appears.