ECE 252: Systems Programming and Concurrency Fall 2019

Lecture 31 — Asynchronous I/O with AIO

Jeff Zarnett *2019-08-10*

AIO

Previously we had talked about how just opening a file for non-blocking I/O doesn't quite do what we want. There is, however, a way to do what we want using the POSIX Asynchronous I/O Interface, which we will just call AIO from here on, because it's shorter.

The general approach is that you create a control block (`struct aiocb` – AIO Control Block) that specifies what operation you want to happen. You enqueue this structure by telling the operating system that you want this to happen. And you can, if you wish, set up a callback to run if the desired event occurs, or you can check periodically if the AIO has completed instead. Your choice. Whichever one depends on your application.

Let's look at the structure of an AIO control block [SR13]:

```
struct aiocb {
  int aio_fildes;               /* File descriptor */
  off_t aio_offset;             /* Offset for I/O */
  volatile void* aio_buf;       /* Buffer */
  size_t aio_nbytes;            /* Number of bytes to transfer */
  int aio_reqprio;              /* Request priority */
  struct sigevent aio_sigevent; /* Signal Info */
  int aio_lio_opcode;           /* Operation for List I/O */
};
```

So, looking at the fields: the first is obviously the file descriptor of the file we would like to read or write; the offset is how far into the file we start the operation. The buffer is the source (for a write) or destination (for a read). Those cover the mandatory fields and the ones that most resemble our typical read or write operation. At this point we could skip right to the example, if we knew how to enqueue a request.

A couple of notes are in order. The offset does nothing if the file has been opened in "append" mode. Also, when we do a write or read the pointer to the place where we are in the file advances as per usual, but it's not the same pointer into the file as the regular (blocking) read and write operations. If you mix the two kinds of operations you may not get the behaviour that you want.

The priority field is a suggestion that we can provide as to how the AIO events should be scheduled, but the operating system is free to ignore this if it wishes. We'll just ignore the ability to set priorities for the purposes of this course. The last parameter is about list-based asynchronous I/O (a more advanced topic), but we need to look deeper at the event first. The best way to do that is to look at the structure:

```
struct sigevent {
  int sigev_notify;                          /* Notify Type */
  int sigev_signo;                           /* Signal number */
  union sigval sigev_value;                  /* Notify argument */
  void* (*sigev_notify_function) (union sigval); /* Notify Function */
  pthread_attr_t *sigev_notify_attributes;   /* Notify attributes */
};
```

(If you're confused about a `union`: it's defined like a structure, except where the struct is made of all of the fields in the definition, a union can be any single one of the fields of the definition at a time.) A `sigval` is either an integer or a `void*` pointer.

The first field is one of the following options:

- `SIGEV_NONE`: Don't do anything when the request completes.

- `SIGEV_SIGNAL`: The signal specified in the signal number field is generated when the request is complete.
- `SIGEV_THREAD`: The function specified in the notify function field is executed, in a different, detached thread. The argument to this function is the signal value. And the attributes of the thread (including whether it is detached) can be overridden using the last parameter if you wish.

We should notice that the signature of the function is not the same as when creating pthreads: the return type is just `void` and not a void pointer; the argument is `union sigval` rather than a void pointer. The definition of the union is:

```
union sigval {
  int sival_int;
  void* sival_ptr;
};
```

Using this, we can choose what we want to happen when the AIO event is complete, if anything. The thread approach is probably the one we'll use most commonly, because it is flexible and convenient. Alright, let's go!

When we're ready to enqueue a request, the functions are:

```
int aio_read( struct aiocb* aiocb );
int aio_write( struct aiocb* aiocb );
```

These are self-explanatory, I would think. The return values are not the same as the actual I/O request return value (as we'll see). It is important to know that once we enqueue the result we can't change the control block or buffer while the operation is in progress, otherwise we might get inconsistent results. And so if we want to know if we are done [SR13]:

```
int aio_error( const struct aiocb* aiocb );
ssize_t aio_return( const struct aiocb* aiocb );
```

The `aio_error` function should probably really be called `aio_status` instead, because it tells you what the status of the operation is. A return value of 0 means the operation has completed successfully! A return of -1 means that the call failed and `errno` tells you the actual reason. If the operation is still waiting to run or in progress the return value is `EINPROGRESS`; if we get any other value then an error occurred and the value is the error code.

If the operation completed successfully, `aio_return` will get the return value from the read or write operation. Calling this while the operation in still progress returns undefined results and potentially fails. We can call this only once per AIO operation; once the value has been collected then some internal structures can be deallocated, so it is polite to call it even if you do not need it.

AIO Example: Read While You Eat. Suppose we have been asked to design a program that processes a group of files. We can use asynchronous I/O to partially parallelize this: start the read for file $n+1$ and process file n in the meantime. This doesn't work for the first file, so a blocking read takes place first. The maximum size of any file we will read is `MAX_SIZE`, so always use this size as the length of a read (even though we may read less, that's okay). We need two buffers: one for the file being processed and one where the next read is taking place.

A list of files to read will be provided as arguments on the commandline to the program. To make the code a bit more compact, we'll assume that errors won't occur and therefore we do not need to check for them (i.e., memory allocation, enqueuing an asynchronous read, actually reading the data, et cetera, will always succeed).

In this example, rather than set up a callback, we'll just do the thing where we check to see if the enqueued AIO has occurred and sleep if it's not ready. We could consider changing this to create a thread instead, but using threads is something we may save for a future exercise.

```
void process( char* buffer ); /* Implementation not shown */

int main( int argc, char** argv ) {
  char* buffer1 = malloc( MAX_SIZE * sizeof( char ));
  char* buffer2 = malloc( MAX_SIZE * sizeof( char ));
```

```
  int fd = open( argv[1], O_RDONLY );
  memset( buffer1, 0, MAX_SIZE * sizeof( char ));
  read( fd, buffer1, MAX_SIZE );
  close( fd );

  for ( int i = 2; i < argc; i++ ) {
    int nextFD = open( argv[i], O_RDONLY );

    struct aiocb cb;
    memset( &cb, 0, sizeof( struct aiocb ));

    cb.aio_nbytes = MAX_SIZE;
    cb.aio_fildes = nextFD;
    cb.aio_offset = 0;
    memset( buffer2, 0, MAX_SIZE * sizeof( char ));
    cb.aio_buf = buffer2;
    aio_read( &cb );

    process( buffer1 );

    while( aio_error( &cb ) == EINPROGRESS ) {
      sleep( 1 );
    }
    aio_return( &cb ); /* This frees some internal structures */
    close( nextFD );

    char* tmp = buffer1;
    buffer1 = buffer2;
    buffer2 = tmp;
  }
  process( buffer1 );
  free( buffer1 );
  free( buffer2 );

  return 0;
}
```

It's worth noting that we need the header `aio.h` and to compile with the `-lrt` option.

Is sleep really the best way to deal with this? I would argue no: if we have nothing to do we could get blocked instead. There is a function that allows us to do that [SR13]:

```
int aio_suspend( const struct aiocb *const list[], int nent, const struct timespec* timeout );
```

The first argument is an array of the control blocks; the second is the size of this array; the last is a timeout. This function allows us to block until one of the AIO operations in the list is complete, or the timeout elapses. If the timeout does occur then the function returns -1; if any AIO request finishes then 0 is returned. If everything was finished already by the time this function was called, the function does not block.

Callback when AIO is complete. Let's also look at setting up a callback that makes a function run when the AIO is completed:

```
#include <stdlib.h>
#include <stdio.h>
#include <aio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_SIZE 512

void worker( union sigval argument) {
    char* buffer = (char*) argument.sival_ptr;
    printf("Worker thread here. Buffer contains: %s\n", buffer);
    free( buffer );
```

The function is a void type instead of a void pointer type, unlike a pthread handler

```
}

int main( int argc, char** argv ) {
    char* buffer = malloc( MAX_SIZE * sizeof( char ));

    int fd = open( "example.txt", O_RDONLY );
    memset( buffer, 0, MAX_SIZE * sizeof( char ));
    struct aiocb cb;
    memset( &cb, 0, sizeof( struct aiocb ));

    cb.aio_nbytes = MAX_SIZE;
    cb.aio_fildes = fd;
    cb.aio_offset = 0;
    cb.aio_buf = buffer;
    cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
    cb.aio_sigevent.sigev_notify_function = worker;
    cb.aio_sigevent.sigev_value.sival_ptr = buffer;

    aio_read( &cb );

    pthread_exit( NULL );
}
```

If an AIO request is no longer needed, it can be cancelled [SR13]:

```
int aio_cancel( int fd, struct aiocb* aiocb );
```

We specify the file descriptor with the AIO operations and the specific request we wish to cancel. If `NULL` is given as the control block argument, then it tries to cancel all outstanding asynchronous I/O requests for that file.

Ah, you noticed that I said “tries” to cancel. This function returns one of four values:

- `AIO_CANCELLED`: The requested operation(s) have been cancelled.
- `AIO_NOTCANCELLED`: At least one operation could not be cancelled.
- -1: Something went wrong in cancelling; `errno` set.
- `AIO_ALLDONE`: All operations finished before they could be cancelled.

Here, a silly but complete example of creating a request and then immediately cancelling it:

```
#include <stdlib.h>
#include <stdio.h>
#include <aio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#define MAX_SIZE 512

int main( int argc, char** argv ) {
    char* buffer = malloc( MAX_SIZE * sizeof( char ));

    int fd = open( "example.txt", O_RDONLY );
    memset( buffer, 0, MAX_SIZE * sizeof( char ));

    struct aiocb cb;
    memset( &cb, 0, sizeof( struct aiocb ));

    cb.aio_nbytes = MAX_SIZE;
    cb.aio_fildes = fd;
    cb.aio_offset = 0;
    cb.aio_buf = buffer;
    aio_read( &cb );
```

```
    /* Do something */

    aio_cancel( fd, & cb );

    sleep( 5 );

    close( fd );
    free( buffer );

    return 0;
}
```

One last thing: the list. In the AIO control block there was one more parameter that we did not cover but said that we would come back to: `aio_lio_opcode`. What's this for? Well, we can submit a group of AIO requests in a single operation. Rather than calling read or write individually on each control block, the following function allows us to enqueue a list of requests all in one go [SR13]:

```
int lio_listio( int mode, struct aiocb * const list[ ], int nent,
    struct sigevent* sigev );
```

The first argument `mode` is a choice of either `LIO_WAIT` or `LIO_NOWAIT`. If we choose the wait option, then the function doesn't return until all the operations are complete. If that's the case, the last argument is ignored. If we choose no-wait, then as soon as the I/O requests are queued the function returns and we can go on.

The next parameters are the list (array) of control blocks and the number of entries in that list (array); not surprising at all. If it's going into the `lio_listio` function, in the AIO block you specify the `aio_lio_opcode` (operation code) as either `LIO_READ`, `LIO_WRITE`, or `LIO_NOP` (this one means do nothing).

The last parameter is an event that will fire once all operations are complete. You can pass in `NULL` if you don't want anything. This is separate from the individual callbacks that you can register in the control blocks; those can still fire if they are set up, either signals or threads – but this is an extra one that's available if we want it.

AIO is HARD! There's a funny (and sad) note in [Ker10] and [Cor16] about what actually happens in Linux when it comes to the POSIX Asynchronous I/O approach: it's actually implemented internally using threads that do blocking reads (ouch). Work continues on the subject, but the current state of affairs is that for a long time people have been working on a properly in-the-kernel implementation of the AIO behaviour as specified above. The current implementation does not scale very well.

Does this affect your program? No – what we covered here does still work and is the POSIX-compliant portable way of writing your code. But should you wish to create a very large number of asynchronous I/O requests then you will want to consider an alternative approach for reading your data, such as libevent.

References

[Cor16] Jonathan Corbet. Fixing asynchronous i/o, again, 2016. Online; accessed 19-February-2019. URL: `https://lwn.net/Articles/671649/`.

[Ker10] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press Inc, 2010.

[SR13] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Third Edition*. Addison-Wesley, 2013.