# Lecture 31 — Asynchronous I/O with AIO

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 10, 2019

# Asynchronous I/O for Files

Previously: opening a file as Non-Blocking doesn't work.

There is a way to do what we want using the POSIX Asynchronous I/O Interface.

Create a control block, enqueue it, with optional callback.

# AIO Control Block

```
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 */
};
```

The offset does nothing if the file has been opened in “append” mode.

The AIO pointers in the file are separate from the blocking operation pointers.

Priority can be ignored by the OS (as per usual).

What about the event?

# Event in AIO

```
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 */
};
```

`sigev_notify` one of: `SIGEV_NONE`, `SIGEV_SIGNAL`,`SIGEV_THREAD`

A `union sigval` is either an `int` or a `void*`.
But not both at the same time.

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

Function definition for the thread not exactly the same as we're used to.

# What to Do?

We can send a signal when the AIO is complete.

Mostly we expect we'll use the thread approach, because it is flexible.

# Enqueue a Request

When we have the callback configured, we can enqueue the request.

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

These are self-explanatory, I would think.

We may not change the control block or buffer while in progress.

# Are we there yet?

# Are we there yet?

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

`aio_error` should really be called `aio_status`.

If it returns 0 the operation is complete.

If the operation is still waiting to run or in progress the return value is `EINPROGRESS`.

If the operation completed successfully, `aio_return` will get the return value from the read or write operation.

It does deallocate some structures, so use it…

# Read While You Eat

Time for an example: reading while you eat.

# Reading While Consuming

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.

We need two buffers: one for the file being processed and one where the next read is taking place.

# Reading While Consuming

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.

For simplicity, we'll check completion and sleep if we need to wait.

# Breakfast is Served

```
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 );
```

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

# Breakfast is Served

```
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;
}
```

# And the last bite

```
  process( buffer1 );
  free( buffer1 );
  free( buffer2 );

  return 0;
}
```

Is sleep really the best way to deal with this?

# Snooze Button...

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

`list`: array of control blocks.

`went`: number of entries in the array.

`timeout`: how long we're willing to wait.

Returns 0 if the AIO finished; -1 if timeout is reached.

Does not block if the AIO is finished when this is called.

# Callback When Complete

```
#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 );
}
```

# Callback When Complete

```
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 );
}
```

# Cancel!


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

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

If NULL is given as the control block argument, then it tries to cancel all outstanding asynchronous I/O requests for that file.

# If You Try To Fail and Succeed, Which Have You Done?

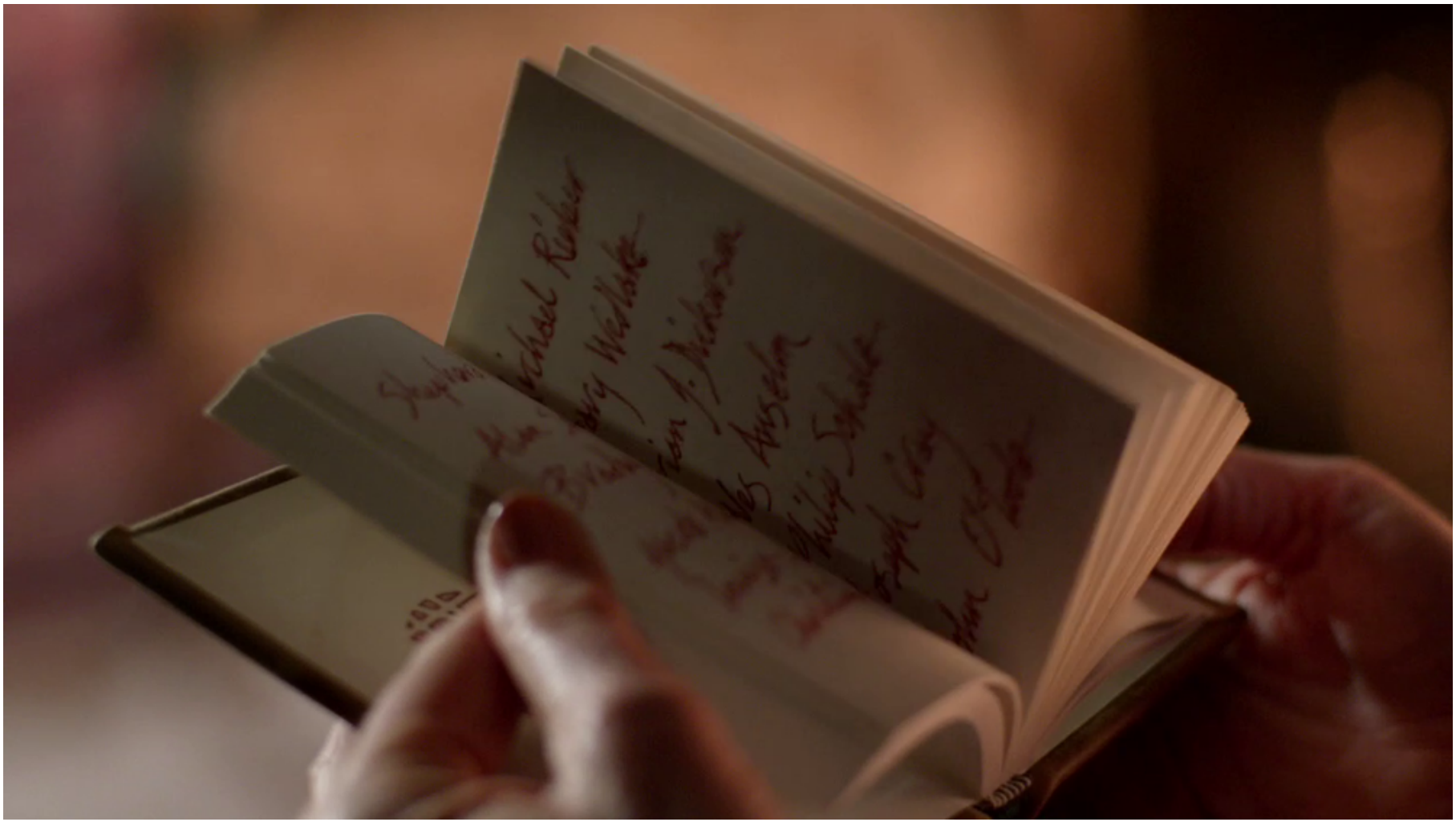
Ah, you noticed that I said “tries” to cancel. This function returns one of four values:

- `AIO_CANCELLED`
- `AIO_NOTCANCELLED`
- -1
- `AIO_ALLDONE`

# I changed my mind?

```
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;
}
```

# The List



“You can right my wrongs. You can be better than I was. You can save this city.”

# Something Left Behind

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`.

We can submit a group of AIO requests in a single operation.

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

`mode`: either `LIO_WAIT` or `LIO_NOWAIT`.

`list`: array of AIO control blocks.

`nent`: number of entries in the list.

`sigev`: event that fires when all is complete. Can be `NULL`.

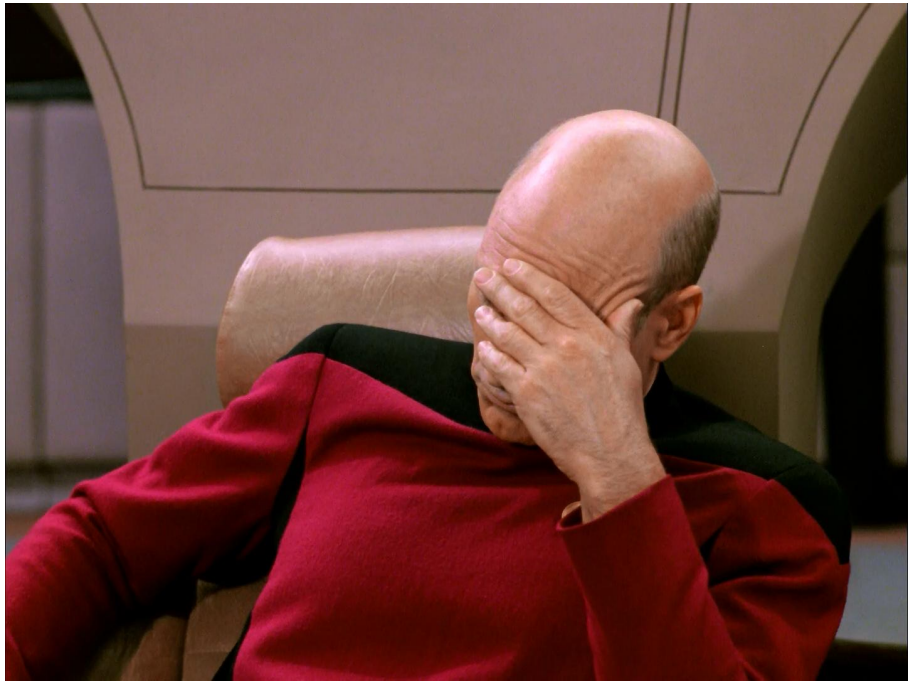
# Doing Operations as a Group

If it's going into the `lio_listio` function, in the AIO block you specify the `aio_lio_opcode` (operation code) as:

- `LIO_READ`
- `LIO_WRITE`
- `LIO_NOP`

# AIO is Hard

The Linux implementation of AIO actually uses threads that do blocking reads.



AIO is a POSIX-compliant portable way of doing asynchronous I/O.

The poor implementation doesn't affect your program, but it's not ideal…