Implementation of a Single Threaded User Level Asynchronous I/O Library using the Reactor Pattern

By

RAMAKRISHNAN KALICUT

B.Tech., Computer Science and Engineering
Jawaharlal Nehru Technological University,
Kakinada, India - 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

____

Dr. Jerry James (Chair)

____

Dr. Gary J. Minden (Committee Member)

____

Dr. Arvin Agah (Committee Member)

____

Date thesis accepted

To
My Parents & Chinna

Acknowledgements

I would like to express my sincere thanks to Dr. Jerry James, my committee chair, for his guidance throughout this thesis and for his suggestions and support during my graduate studies here at University of Kansas. Thank you! Dr. James, for all the help and advice you offered me. I would like to thank Dr. Gary Minden and Dr. Arvin Agah for giving me a chance to work in the ACE Project.

I would like to thank Raj and Sunil, with whom I worked as part of the BERT project. I thank all my friends here at KU who made the past 2 and half years of my stay in Lawrence a memorable one.

My heartfelt thanks to my parents and my brother for the love and support they offered. To them, I owe all the good things in my life.

Abstract

Asynchronous I/O is one of the methods used in systems, which are high on I/O load. Asynchronous I/O overlaps application processing with I/O operations for improved utilization of CPU and devices, and improved application performance, in a dynamic/adaptive manner. This finds its applications in database systems, web server systems etc. Many existing asynchronous I/O implementations employ multi-threading to do non-blocking I/O operations. Such implementations, due to the overhead of thread maintenance and context switching among them, do not scale well for high loads. A single-threaded asynchronous library can be used to overcome this problem. Such a library must be built on top of an event-driven framework that acts as a controlling point for I/O request processing. This thesis describes UAIO, a User-level Asynchronous I/O interface implemented in a single threaded model using a time triggered Reactor, an object-oriented event de-multiplexing framework as the underlying architecture. By eliminating threads, this implementation tries to deliver better performance of the system that is otherwise loaded with thread maintenance and thread context switches. Also, as the implementation is at the user-level, it can be ported to any operating system with minor modifications.

Table of Contents

1 INTRODUCTION 1

2 RELATED WORK 7

3 DESIGN 13

3.1 REQUEST STATES & QUEUES 14

3.2 REACTOR 15

3.3 UAIO ASYNCHRONOUS I/O MECHANISM 16

4 IMPLEMENTATION 19

4.1 DATA STRUCTURES USED IN UAIO LIBRARY 19

4.1.1 Data Structures visible to the user 19

4.1.2 Data Structures for queue/list 22

4.1.3 Data Structures for Event Handler 24

4.2 IMPLEMENTATION OF THE UAIO LIBRARY INTERFACE 25

4.3 IMPLEMENTATION OF THE REACTOR HANDLER FUNCTIONS 37

4.4 UAIO LIBRARY INITIALIZATION/FINALIZATION 41

4.5 IMPLEMENTATION LEVEL FLOW CONTROL IN THE UAIO LIBRARY 42

4.6 LIMITATIONS 45

5 TESTING 46

5.1 CORRECTNESS TESTING 46

5.2 PERFORMANCE TESTING 48

5.2.1 One single write call with varied buffer size 48

5.2.2 Varied number of I/O calls for constant write buffer size 50

5.2.3 Varied number of I/O read calls 51

5.2.4 Combination of CPU and I/O intensive tasks 53

6 CONCLUSIONS AND FUTURE WORK 54

REFERENCES 55

1 Introduction

As use of the Internet increases, high performance web servers are the need of the day. Most of the transfers involve data storage or retrieval. Due to limitations on cache size, most requests end up retrieving data from the disks. Conventional I/O is blocking in nature; the process making the I/O request waits until the request is satisfied. If the server waits for each I/O request to complete, it cannot service other clients during that time. This may lead to poor performance of the system, dissatisfied customers and thus more importantly loss of revenue for the companies. Many scientific applications involve massive amounts of I/O. Even if they employ high-performance supercomputers, the speed of the underlying devices is 2-3 orders of magnitude less. In real-time systems, the program cannot generally halt its execution while waiting for a particular I/O request to complete. Deadlines are critical in a real-time system. If it does wait for I/O completion, it may miss some deadlines and endanger the whole system. Hence disk latencies are fast becoming a performance bottleneck. This calls for somehow overlapping the I/O latencies with application processing, thereby attaining improved utilization of CPU and devices, and improved application performance.

In conventional blocked I/O, the kernel puts the application to sleep while the device is processing the I/O request. The kernel then switches to serve another process. When the kernel receives an interrupt from the I/O device signaling the completion of

the original I/O command, it wakes the sleeping process and makes it eligible to run again. This is a simplified view of how a kernel performs I/O. In reality, kernel and file systems also implement other facilities to improve performance, including read-ahead and buffered writes. This mechanism is good enough in a heavily multi-programmed environment where the overall throughput of the system is more important rather than that of individual applications. But for performance-sensitive applications, such as dedicated systems like web servers and database systems, the impact is very significant. Hence, the need for non-blocking I/O. There are many alternatives to implement non-blocking I/O so as to mask the disk latency with processing, thus ensuring better performance.

- *Using threads (multi-threading)*

  The user/programmer has to implement a threading model such that a thread is created to handle each I/O request. Whenever an I/O request is made, a thread is spawned to service the request and control returns to the main thread. The child thread does a blocking I/O on the request, but another thread can be processed at the same time. The main thread can check for I/O completion or ask to be notified of request completion. In this method, the programmer has to deal with issues related to concurrency, locks, thread maintenance and the possibility of deadlocks. The resources needed to maintain each thread and the context switching among the threads make this approach less scalable.

- *Using poll() or /dev/poll*

  Non-blocking calls (e.g. *write()* on a socket set to O_NONBLOCK) are made to start I/O, and readiness notification (e.g. poll() or /dev/poll) is used to know when it's OK to start the next I/O on that channel. This scheme has *level triggered* readiness notification, as the kernel tells whether a file descriptor is ready, whether or not anything is done with that file descriptor since the last time the kernel told about it. This standard O_NONBLOCK doesn't help, as the API doesn't take the buffer away from the user. As a result the kernel can avoid a copy only if the Memory Management Unit write protects the buffer and relies on COW (copy-on-write) to avoid overwrites while the buffer is in use. This is rather expensive due to the TLB (translation lookaside buffer) flushing requirements, especially as it involves Inter Processor Interrupts on SMP (shared memory multiprocessor) machines. This method requires a specialized device node based interface for registration and notifications of events. This *poll()* or */dev/poll* method is suitable for readiness notification on sockets (network I/O) but not for disk I/O.

- *Using Real-time signals*

  Another alternative is using real-time signals. This is just a notification mechanism and does not itself provide asynchronous I/O support. But in combination with non-blocking calls, it provides a *non-blocking I/O with edge triggered* readiness notification mechanism. In this method (edge triggered), the

kernel tells the readiness of a file descriptor only when an event occurs that might cause it to become ready. Some options need to be set on the file descriptor (like fcntl F_SETSIG) to enable *edge triggered* readiness notification. To use this, write a normal poll() outer loop, and inside it, after you've handled all the fd's noticed by poll(), you loop calling sigwaitinfo(). Signals could be lost due to signal queue overflows, especially with functions that tend to generate more signals. To overcome this problem, the signal queue can be flushed when the queue is full and control is given to the outer *poll()*. Tuning the queue length is a bit complicated (via a *sysctl* call which is used to configure kernel parameters at runtime). As the number of events increases, there is the complication of using the same signals and their signal handlers in user space. Event triggered programming (using signals) like this is difficult to implement and understand, as the flow of control is quite unpredictable.

- *Using Asynchronous I/O libraries*

  An interface to do asynchronous I/O operations is provided to the user. A user intending to do non-blocking I/O can use this interface. Control is immediately returned to the user after the I/O request is submitted. The user process can ask to be notified upon request completion or can itself check a request's status using the interface. The implementation of the interface is abstracted from the user/programmer. The implementations vary from multi-threading with little utilization of the asynchronous nature of the underlying devices to a true

asynchronous finite state machine taking full advantage of the device nature. The multi-threaded approach is similar to the *user-level threads* discussed above, except that the thread creation and management is invisible to the user (handled by the library). This and other methods of asynchronous implementation are discussed in detail in the next chapter.

This thesis aims at building an asynchronous I/O library using a single-threaded event driven approach with the view of overcoming the drawbacks of the above-mentioned alternatives. A performance comparison is also made with some of the existing implementations. The current implementation of the UAIO (User-level Asynchronous I/O) library is a single threaded model for doing non-blocking I/O. It is built on top of *Reactor*, an event driven framework [1]. The library interface is in compliance with the POSIX standard [2]. By employing only one thread of execution (the process main thread), the overhead attached with thread maintenance and context switching is eliminated. In addition UAIO does not perform any copies on user buffers.

Chapter 2 discusses various implementation methods and relevant work in the area of asynchronous I/O libraries. Chapter 3 discusses design of the user-level asynchronous I/O library. The higher-level modules of the library and the higher-level interaction of these modules with the Reactor are discussed here. Chapter 4 discusses implementation specific issues of the UAIO library. Chapter 5 discusses testing of the library, verifying its POSIX compliance and comparing its performance with some

existing asynchronous I/O implementations. Chapter 6 discusses conclusions and possible future work.

2 Related Work

With a view of overlapping I/O latencies with computation, various asynchronous I/O approaches are defined. The driving factor in choosing one implementation over another is the purpose of the system and its environment. Real-time applications, for example, must by definition deal with an unpredictable flow of external interrupt conditions that require predictable, bounded response times. In order to meet that requirement, a complete non-blocking I/O facility is needed. Violation of time constraints may be catastrophic. The overhead of context switching and thread managing in a multi-threaded asynchronous I/O implementation may deprive critical tasks of CPU cycles. In this case it is better to employ a signaling model. In a broader perspective there are two methods for asynchronous I/O implementation.

- Building the entire system in an asynchronous model. The Microsoft Windows NT I/O subsystem is built with this method. It provides support for *overlapped files* within the Win32 API [3]. The operations can be invoked in synchronous or asynchronous mode. This method is useful if the system is built from scratch. But if it is built on top of existing synchronous interfaces, changes to these interfaces need to be made.

- A separate asynchronous operations interface is created. All asynchronous operations follow a different path from the synchronous operations. This can be used along with the existing synchronous interfaces. This is the approach most

Linux implementations of asynchronous I/O take, as it can coexist with the existing synchronous interfaces.

When considered from an implementation point of view, there are three approaches for asynchronous I/O implementation.

- ❖ The first approach is to use threads to process the I/O requests without blocking the main thread (the process making the asynchronous request), thus making it look asynchronous to the application/user. In a multiprocessor machine, different processors can execute threads at the same time, thus attaining the actual asynchronous behavior. This is done in two ways.

  - ➢ Using threads

    A thread is created to process each asynchronous I/O request. The slave threads block for request completion. The *glibc* Asynchronous I/O library (part of the Real-time library) falls in this category [4]. The user is abstracted from the threading implemented in this library. This method has its drawbacks. As the number of outstanding requests increases, more threads are created and take significant process time in context switching. Therefore, this approach exhibits poor scalability and performance.

- Using a thread pool

  In this approach, a pool of threads services the requests. The request is stored into an internal queue. Each thread polls this queue for requests and services a task (request). The Solaris Asynchronous I/O library *laio* uses this approach [5]. While the creation of a pool of worker threads up front helps provide better scalability than the user-threads asynchronous I/O facility, there is a trade-off between degree of concurrency and resource consumption.

- While the threads library implementation of asynchronous I/O works well enough for many applications, it does not necessarily provide optimal performance for applications that make heavy use of the asynchronous I/O facilities. Overhead associated with the creation, management, and scheduling of user threads motivated the decision that an implementation that required less overhead and provided better performance and scalability was in order. This gave rise to a *hybrid approach* that makes use of the asynchronous behavior of the underlying operation. For this purpose there needs to be specific asynchronous I/O read and write routines at the device-driver level.

  - The SGI KAIO implementation employs this method [6].
    - If the underlying operation is asynchronous in nature (i.e., the request is *kaio* supported), *kaio* queues the request at the device and calls the

asynchronous I/O routine in the appropriate device driver. In this case the degree of asynchrony depends on the device.

- If the operation is synchronous in nature (like filesystem I/O), *kaio* uses the *thread-pool approach* or *single thread per operation* approach depending on the implementation platform. In this case the number of slave threads determines the degree of concurrency.

- The third approach is to implement a true asynchronous state machine for each type of asynchronous I/O operation. This is achieved through a sequence of non-blocking steps, with state transitions driven by IRQ techniques and event threads. This approach is very hard to implement but provides greater flexibility and greater asynchrony in comparison to the other two approaches. The asynchronous I/O interface being developed at IBM uses these techniques (with some caveats) [7].

Some other asynchronous I/O implementations are discussed below. Block Asynchronous I/O (BAIO) developed at the University of Wisconsin-Madison is a mechanism that strives to bypass the kernel abstraction of the file system [8]. In an effort to obtain reasonable performance over different file systems, the kernel provides a generic set of file-system polices. This generality leads to suboptimal performance of the system. File systems based on these general set of policies end up ignoring the application-level understanding of the data being stored, which might

facilitate highly optimized policies and decisions. The BAIO implementation transfers this file system to the user level where it can use application level information, enabling it to layout the data on the disk, caching and prefetching the data in an optimal way. Unlike SGI KAIO that uses slave threads on a per task basis, BAIO employs a service thread for each process. The service thread (BAIO service) checks for request completion in a non-blocking manner. It is notified by the device driver about the completion and then in turn notifies the user application.

The Message Passing Interface (MPI) Standard is a library specification for message passing, which is proposed as a standard in the parallel computing industry [9]. Its latest MPI 2.0 Specification has included primitives for buffered asynchronous I/O through the MPI_FILE_IREAD_* and MPI_FILE_IWRITE_* initiation functions. The non-blocking data access routines indicate that MPI can start a data access and associate a request handle with the I/O operation. Non-blocking operations are completed/synchronized via MPI_TEST, MPI_WAIT, or any of their variants. This specification allows implementations to choose the old blocking I/O primitives, if the underlying hardware is ill-fitted for asynchronous operations.

Titanium is an explicitly parallel dialect of Java developed at UC Berkeley to support high-performance scientific computing on large-scale multiprocessors [10]. Titanium is based on a parallel SPMD (Single Program, Multiple Data) model of computation in which the same program runs on different processors each having its own data.

Even so, the I/O operation at each processor is essentially blocked I/O. Due to the high I/O of intensive scientific applications, conventional blocking I/O has become a performance bottleneck. To overcome the limitations of file I/O in Java, an asynchronous I/O library is built to mask this disk latency with overlapped computation.

Whichever model they use, all the above-mentioned implementations prove the fact that asynchronous I/O is recognized as an important part of the design of a high-performance parallel computing systems. This is a sign that in the near future, more and more OS vendors will implement this feature in their system call or standard library interfaces.

3 Design

UAIO is a user-level asynchronous I/O library based on the single-threaded model. This library is built on top of the Reactor, an Object Oriented (OO) event demultiplexing framework. The UAIO API is POSIX compliant and provides support for asynchronous read and writes, batched asynchronous read and write, synchronization, asynchronous request cancellation and signaling. This section outlines the design behind development of the UAIO library. There are three major components of this library. They are:

- Asynchronous I/O interface – This interface is in compliance with the POSIX standard and can be used by the user to do asynchronous I/O.
- Library Internal Queue – This is internal to the UAIO library. The user requests are stored in this queue.
- Reactor – Processes the requests in the queue and provides notification (if desired) of the request completion.

As the interface provided by the UAIO library is well known (similar to the glibc/librt AIO interface), it is not discussed here. However, the interface is dealt with in detail in the next chapter (Implementation). This chapter first discusses the states corresponding to a request, then the internal queue properties. Then the concept of the Reactor with respect to I/O handling is explained in detail. Finally as all pieces of the puzzle are defined, the placement of these pieces in the system and the interactions among them are discussed.

3.1 Request States & Queues

In the UAIO library all request states are stored at the user-level. The possible request states are:

- *Incomplete or Pending* – In this case the request is enqueued in the queue and is waiting its turn to be processed.
- *Running* – The request is being processed (by the reactor).
- *Completed or Terminated* – The request is either terminated successfully (request is processed) or unsuccessfully (failed to process the request).

The following queues[1] are part of the UAIO library. The asynchronous I/O requests are stored in these queues depending on their state.

- *Free list* – empty list
- *Run List* – contains the requests that need to be processed
- *Done List* – contains the processed requests.

During the library initialization, a list is created (*free list*) which can hold some maximum number of elements (requests). When a request is submitted, it is stored in an element of the *free list*. This element is then placed in the *run list*, and the request state is set to *pending*. The allocation of resources (creation of the *free list*) is done with a view to increase performance. The request is stored in the queue until its turn to be processed by the Reactor. When its turn comes, the state of the request changes to *running*. Upon processing of the request, this request element is placed in the *done list*. The request is in the *completed* state. A request that is canceled by the user is also placed in the *done list*.

[1] The library internal queues are implemented as linked lists. In this document, the terms queue and linked list both mean the same unless otherwise specified

3.2 Reactor

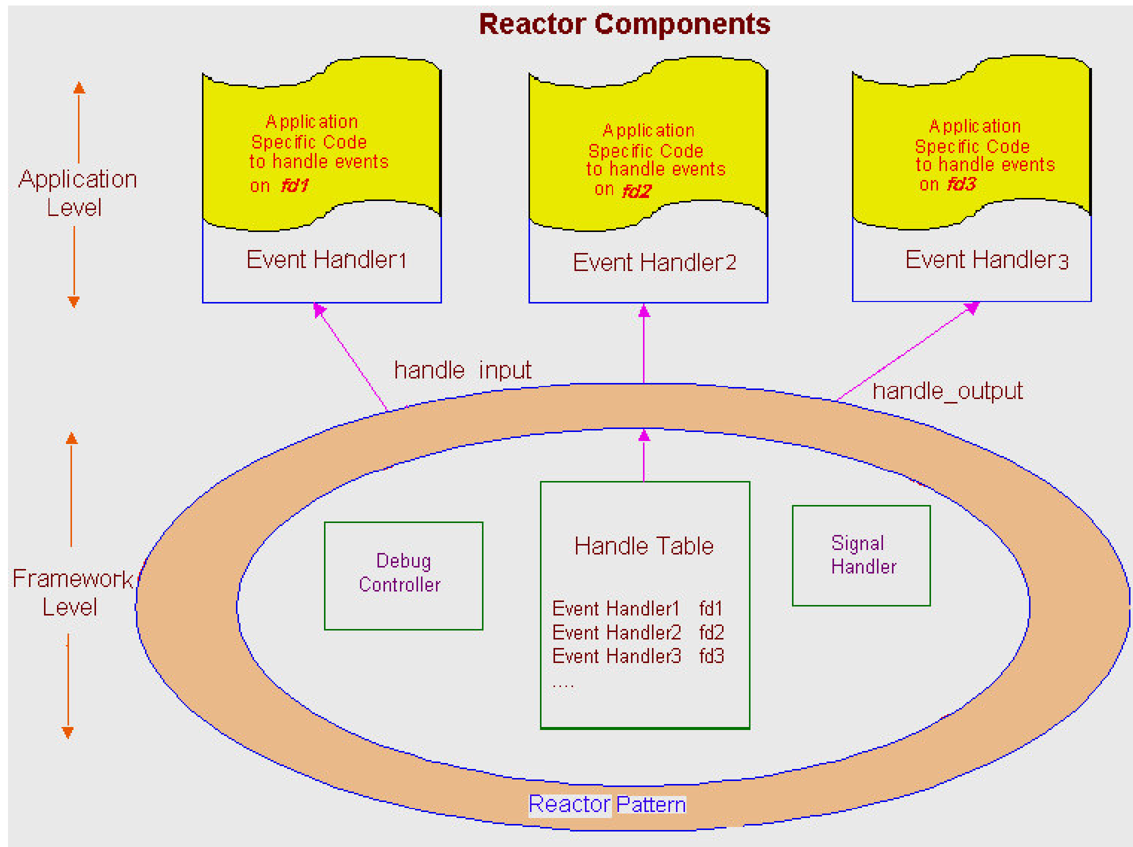


Figure 3-1 Reactor Pattern for I/O Handling [11]

The Reactor provides an object-oriented event demultiplexing and distribution framework. The reactor handles events by using event handlers. A separate event handler is generated that represents each service of the application. This event handler is responsible for dispatching requests specific to this service. Each event handler is registered with the Reactor. There are three prominent methods that each Reactor objects defines. They are:

- handle_input() – called whenever data is ready to be read
- handle_output() – called whenever the output buffer is free to send data
- handle_close() – used to deregister the handler upon completion of the request

A handler is generated for each request and is registered with the Reactor either as *input type* or *output type*. All the handlers registered with the Reactor are stored in the reactor queue. At the core of the Reactor is a *select* call which checks for I/O request that can be satisfied. The corresponding handler functions are called to process the request. Demultiplexing of the requests (events), by mapping the event detected on a file descriptor to a particular event handler, is performed by a synchronous demultiplexer. This Reactor framework, event-driven in nature, eliminates the need of threads to process the requests.

3.3 UAIO Asynchronous I/O mechanism

As discussed in the above section, the primary objective of the Reactor framework is to provide the ability to register callback functions that are called when I/O is ready. In the *glibc* version, a thread is created to satisfy each asynchronous I/O request. Each service thread blocks until the I/O request is completed. In UAIO library, which employs the single threaded model, this method causes the process as a whole to stop. But using the concept of the Reactor, we can make the UAIO library provide for asynchronous operations as we can register the asynchronous I/O events we are interested in with the reactor. When the reactor detects that the I/O can be satisfied, the request is completed and a notification (if requested) is sent to the user process.

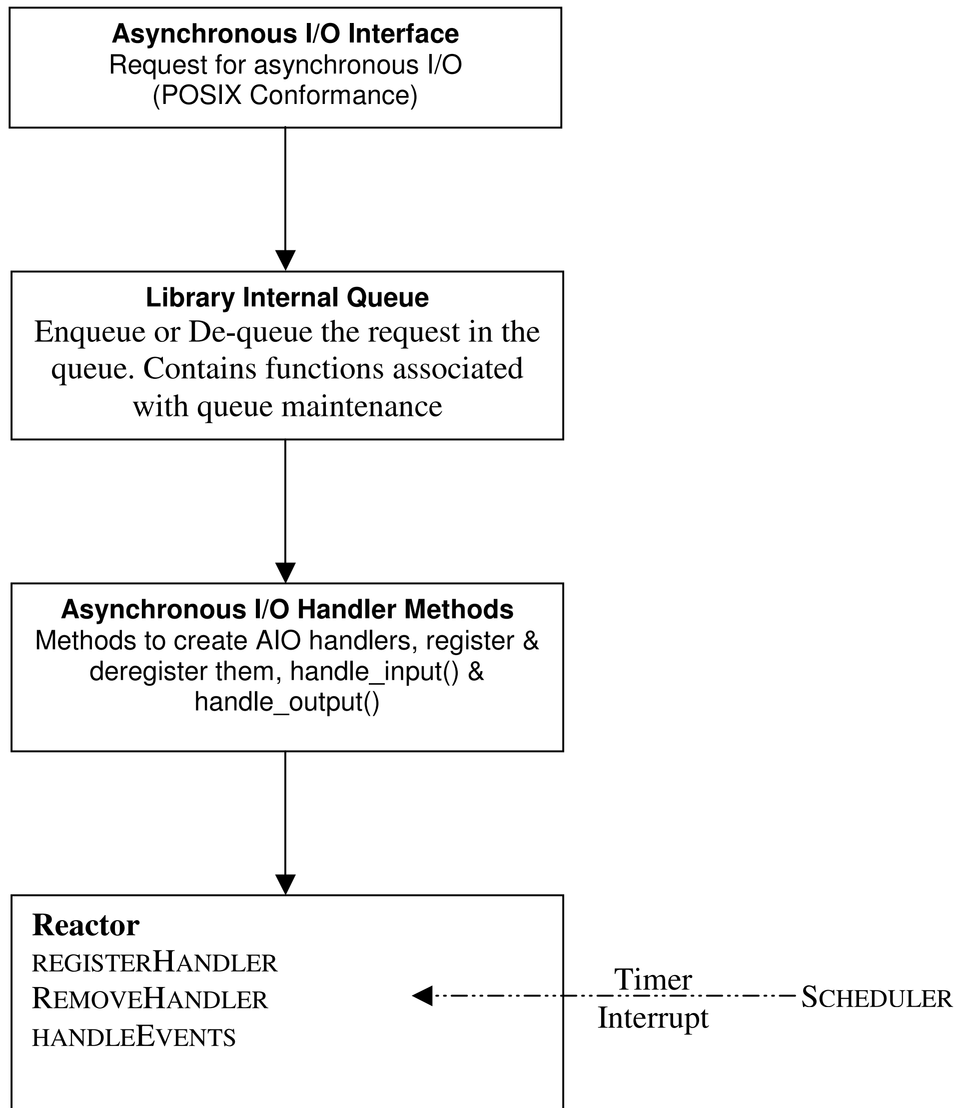


Figure 3-2 Asynchronous I/O in UAIO library

As shown in the figure 3-2, a POSIX compliant interface is provided to the user to invoke asynchronous I/O operations. The user can ask to be notified about request completion or can check for completion. When a request is made using this interface, the request is queued in the library internal queue. A new handler for the request is generated and is registered with the reactor using the registration method that extends

the *registerHandler* method of the reactor. The reactor (invoked for every 1µsec timer interrupt) checks if any of the requests can be satisfied. If the selected request is registered for input, the *handle_input()* function is called, and if it is registered for output, the *handle_output()* function is invoked. The results of the operation are stored in the corresponding queue entry and a notification (if requested) is sent to the user process indicating completion of the request. The user can retrieve the results using the interface provided.

4 Implementation

This chapter discusses the implementation details of the UAIO single-threaded library. First the data structures visible to the user and those internal to the library are discussed. The next two sections discuss the implementation of the interface and the library internal queue. Then the handle functions associated with each event handler are explained. Finally the implementation level flow control within the system is defined.

4.1 Data Structures Used in UAIO library

4.1.1 Data Structures visible to the user

These are provided to the user to submit asynchronous I/O requests and to tune the parameters in the library.

- *'struct aiocb'* – All AIO operations operate on files that were opened previously. There might be arbitrarily many operations running for one file.

```
struct aiocb{
    int aio_fildes;
    int aio_lio_opcode;
    int aio_reqprio;
    volatile void *aio_buf;
    size_t  aio_nbytes;
    struct sigevent aio_sigevent;
    off_t aio_offset;
};
```

The asynchronous I/O operations are controlled using a data structure named *'struct aiocb'* (AIO control block). Though the asynchronous I/O interface is POSIX compliant, the valid set of values for some of these elements is constrained to suit the implementation. All such changes are explicitly mentioned when discussing the corresponding structure elements.

`int aio_fildes' – File descriptor which is used for the operation. The device on which the file is opened must allow the seek operation.

`off_t aio_offset' – Specifies at which offset in the file the operation (input or output) is performed.

*'volatile void *aio_buf'* – Pointer to buffer with the data to be written or place where the read data is stored.

'size_t aio_nbytes' – Length of the buffer pointed to by *'aio_buf'*.

'int aio_reqprio' – Specifies the priority of the request.

'struct sigevent aio_sigevent' –
Specifies how the calling process is notified once the operation terminates.
If *sigev_notify* is
SIGEV_NONE - No notification is send
SIGEV_SIGNAL - signal send (*sigev_signo*)

The SIGEV_THREAD option where a thread is created to do the notification (provided by the *glibc* asynchronous I/O implementation) is not valid as this implementation is single-threaded.

'int aio_lio_opcode' – Assumes significance only when used in the *'lio_listio'* function. This parameter is used to specify the action to be performed on each request.
Possible values:
LIO_READ - Start a read operation.
LIO_WRITE - Start a write operation.
LIO_NOP - Do nothing for this control block.

- *'struct aioinit'* – This data type is used to pass the configuration or tunable parameters to the implementation. This is provided to initialize the library in a way that is optimal for the specific application.

```
struct aioinit{
    int aio_threads;
    int aio_num;
    int aio_locks;
    int aio_usedba;
    int aio_debug;
    int aio_numsers;
    int aio_reserved[2];
};
```

'int aio_threads' – Maximum number of threads which must be used at any one time

'int aio_num' – Maximum number of simultaneously enqueued requests

'int aio_locks' – Initial number of pre-allocated needed data structures. Used by real-time programs to pre-allocate needed data structures so that real time programs do not need to allocate them in critical areas.

'int aio_usedba' – Try to use Database Administration(DBA) for raw I/O in *lio_listio*

'int aio_debug' – Turn on debugging

'int aio_numusers' – Maximum number of user sprocs or pthreads making aio_* calls (as in multithreaded programs). Passing a number that is too small can result in program deadlocks and other errors.

'int aio_reserved' – Reserved for future use

The glibc version uses the first two parameters to optimize the asynchronous I/O library. The rest are left unused. As the UAIO implementation is single-threaded, *aio_threads* is irrelevant. So, in this implementation only *aio_num* of this *aioinit* structure is used.

4.1.2 Data Structures for queue/list

The library internal queue is implemented as a linked list. Elements are arranged in descending order of priority with 0 as the minimum priority. Each element in the list consists of some elements that represent the request and a pointer to the next element. The parameter *type* is used to determine the operation performed on the request (such as read, write or signaling). The event handler created for the request is stored in *handle*. *errorState* indicates the current state of the request. *errorSate* of a pending request is EINPROGRESS. If the request is completed successfully, *errorSate* is set to 0, else set to the corresponding *errno*. *returnStatus* holds the result of the request completion. For a request of type read or write, *returnSatus* stores the number of bytes read/written. The last parameter

*struct Notify *notify* is used only when the user process is explicitly waiting on this request. The results of completion of all the requests that the process is waiting on are copied to this object. The library checks this object to decide when to revoke the waiting process[2].

```
/* List element structure */
struct requestList{
      int type;                    // specifies the type of operation
      struct aiocb *aiocbp;        // aio control block
      aioHandler handle;           // handler for the request
      int errorState;              // error state of the request (error condition)
      int returnStatus;            // return status of the request (result)
      struct Notify *notify;       // if used, specifies the place to store the results
                                   // Used only when ulio_listio() is called in
                                   // LIO_WAIT mode or uaio_suspend() is called
                                   // on this  request
      struct requestList *next_prio;
};
```

The *struct Notify* data type contains the following elements.

'int count' – Indicates the total number of requests completed that the process is waiting on.

'int result' – Indicates the total result of all the completed requests. It is 0 if all the requests completed successfully, else it is set to –1.

'int signalFlag' – Turn on if any of the completed requests asked for signal notification upon completion of the request.

[2] As the library is user-level and single-threaded, it is a single process (library reactor + user program). So, waiting is accomplished using busy waiting with a condition. System commands like sleep() and wait() are not used, as they block the whole process.

```
struct Notify{
        int count;          // total # of requests(waiting for) completed
        int result;          // cumulative result
        int signalFlag; // turn on if any request completed
                              // has signal notification
};
```

4.1.3 Data Structures for Event Handler

Each request that requires an I/O operation must have a corresponding event handler registered with the reactor. The handler object extends the default handler provided by the reactor. The default handler *'EventHandler * eh'* links the handle functions like *handle_input()* and *handle_output()* with the handler. These functions are called when the specific event occurs. *aiocbp* is the aio control block that contains the asynchronous I/O request. *fd* is the file descriptor on which the request is made. This is used by the select call to find if that file descriptor is

```
typedef struct{
        /* This structure must be included in all the modules that use reactor and the
        function pointers in it must be set to appropriate values */
       EventHandler * eh;

        /* The fileds below are specific to aio implementation */
        struct aiocb *aiocbp;      // aio control block (request)
        int fd;                           // file descriptor.(=aiocbp->aio_fildes)
        int isRegisteredAsInput; // if the handler is registered as input

        int isRegisteredAsOutput; // if the handler is registered as output
}aioHandler_struct;
```

ready for I/O. A handler can be registered for input (for reading) or output (for writing). *isRegisteredAsInput* is set when the handler is registered to do an asynchronous read operation. *isRegisteredAsOutput* is set when the handler is registered to do an asynchronous write operation.

4.2 Implementation of the UAIO library interface

This section discusses the implementation of the interface provided to the user. The asynchronous I/O interface is in conformance with POSIX standards. In the *glibc* asynchronous I/O implementation, a thread is created to handle each request. But, as UAIO is based on the Reactor (an event-driven model), the implementation does not involve any multithreading.

- ***int checkRequest (int mode,struct aiocb *aiocbp)***

  This function is internal to the library. This function is used to check the validity of the asynchronous I/O control block, i.e. if the control block is checked for invalid file descriptors, file offsets etc. The first parameter *'mode'* is used to specify the type (read/write) for which the request is to be validated. 'a*iocbp'* is the aio control block.

  This function returns:

  0 - if *'aiocbp'* is valid and the library queue is not full

  -1 - otherwise, and *errno* is set accordingly

```
Input: [for checkRequest]
      mode of request (read or write)
      asynchronous I/O control block aiocb (request)

Flow:
      If queue is full (resource limitation)
            Set errno to EAGAIN;
            Return –1;
      End if

      If file descriptor in the request is not valid for the
      mode specified
            Set errno to EBADF;
            Return –1;
      End if

      If file offset in not valid (i.e., offset is –ve or exceeds the current size of file) or #
      of bytes to be read/written is -ve or invalid priority
            Set errno to EINVAL;
            Return –1;
      End if

      Return 0;
```

Possible *errno* values are:

EAGAIN – Indicating that the request cannot be queued due to exceeding resource (queue) limitations.

EBADF – File descriptor *'aio_fildes'* is not valid for reading.

EINVAL – 'aio_offset' *or* 'aio_reqprio' *value is invalid*

- ***int uaio_read (struct aiocb *)***

This function is used to submit an asynchronous read request. An aio control block that corresponds to the request is passed as an argument. A handler is

generated for the request. The request is written to the library internal queue with an element type LIO_READ and the handler is registered with the reactor. The function returns immediately after it attempts to do the said tasks. This function returns:

0 – if no error is detected before the request is enqueued and its handler is registered

-1 – otherwise, and *errno* is set accordingly

Possible *errno* values are:

EAGAIN – Indicating that the request cannot be queued due to exceeding resource (queue) limitations.

- ***int uaio_write (struct aiocb *)***

  This function is used to submit an asynchronous write request. This is similar to *uaio_read* except that the handler is register with type *LIO_WRITE*.

```
Input: [for uaio_read/uaio_write]
      Asynchronous I/O control block (request)


Flow:
      Check if the request is valid for read/write

      If checkRequest (read/write, request) is valid
            If enqueuing the request does not cause queue overflow
            (resource limitation)
                  Create a handler for the request;
                  Enqueue the request in the internal queue with type
                  LIO_READ/LIO_WRITE;
                  Register the handler with the reactor as input/output  type;
                  Return 0;
            Else
                  Set errno to EAGAIN;
                  Return –1;
            End if
      Else
            Appropriate errno is set by checkRequest( ) function;
            Return –1;
      End if
```

- ***int ulio_listio (int mode, struct aiocb *const list[], int nent, struct sigevent *sig)***

  The *ulio_listio* function allows the calling process to initiate a list of I/O requests with a single function call. The *mode* parameter defines the function behavior after having enqueued all the requests. If *mode* is *LIO_WAIT*, the calling process waits until all requests are terminated. And if it is *LIO_NOWAIT*, the calling process returns immediately after enqueuing all the requests. *list[]* contains the batch of requests that are submitted at one time. *'nent'* indicates the # of elements in *list[]*. The last parameter *'sig'*, used in LIO_NOWAIT mode, is used to notify the completion of all the requests using the *sigev_signo* signal.

Input: [for lio_listio]

```
    Mode indicates if the function waits for request completion
    List of asynchronous I/O control blocks (requests)
    # of elements in the list
    Signal used to indicate completion of the requests
```

Flow:

```
    If # of request in list > max requests allowed at one time
        Set errno to EINVAL;
        Return –1;
    End if

    If mode is neither LIO_WAIT nor LIO_NOWAIT
        Set errno to EINVAL;
        Return –1;
    End if

    For each element in the list
        If checkRequest(read/write, request) is valid
            If enqueuing the request does not cause queue overflow
            (resource limitation)
                Create a handler for the request;
                Enqueue the request in the internal queue with
                LIO_READ/LIO_WRITE type;
                Register the handler with the reactor as input/output type;
            Else
                Set errno to EAGAIN;
            End if
        End if
    End for

    If no request is enqueued and mode is LIO_NOWAIT
        Raise the signal (if specified) indicating completion;
        Return –1;
    Else
        If mode is LIO_WAIT
            Wait until all requests in the list are completed;
            Return 0, if all requests are completed successfully;
            Else errno set to EIO and return –1;
        Else  //if mode is LIO_NOWAIT and signal is specified
            Enqueue another request containing the signal as LIO_SIGNAL type;
            Return 0;
        End if
    End if

    Return –1;
```

This function returns:

0 – if all requests are enqueued/completed correctly

-1 – otherwise, and *errno* is set accordingly

Possible *errno* values are:

EAGAIN – Resources necessary to queue all the requests are not available at the moment. *errorState* of each request in the list must be checked to find which requests failed to be enqueued.

EINVAL – *'mode'* is invalid or *'nent'* is larger than the maximum allowed requests at one time.

EIO – One or more request's failed

- ***int uaio_fsync (int op, struct aiocb *aiocbp)***

This function is used to force all I/O operations queued at the time of this function call operating on the file descriptor *'aio_fildes'* in *aiocbp* into the synchronized I/O completion state; i.e., the requests for the file descriptor are honored only after all the requests prior to the *uaio_fsync* call are completed. *'op'* indicates if fsync (*O_SYNC*) or fdatasync (*O_DSYNC*) need to be performed. '*aio_fildes*' in *aiocbp* gives the file descriptor on which *fsync* or *fdatasync* is performed. This function returns immediately after enqueuing the request but the notification through the method described in '*aio_sigevent*' is performed only after all requests for this file descriptor have terminated and the file is synchronized.

This function returns:

0 - if request was successfully filed

-1 - otherwise, and *errno* is set accordingly

Possible *errno* values are:

EAGAIN - Temporary lack of resources

EBADF - '*aio_fildes'* is not valid or not open for writing

EINVAL - Implementation does not support I/O synchronization or the OP parameter is other than *O_DSYNC* and *O_SYNC*

```
Input: [for aio_fsync]
    Option indicating whether to do fsync or fdatasync
    Asynchronous I/O control block (request)

Flow:
    If enqueuing the request cause queue overflow
    (resource limitation)
        Set errno to EAGAIN;
        Return –1;
    End if

    If file descriptor is not opened for writing
        Set errno to EBADF;
        Return –1;
    End if

    Create a handler for the request
    If option is LIO_SYNC
        Enqueue the request in the internal queue as LIO_SYNC type;
    Else
        If option is LIO_DSYNC
            Enqueue the request in the internal queue as LIO_DSYNC type;
        End if
    End if

    Return 0;
```

- ***int uaio_suspend (const struct aiocb *const list[], int nent,***

  const struct timespec *timeout)

When called, this function suspends the calling thread until at least one of the requests pointed to by the *nent* elements of the array *list[]* has completed. If any of the requests in *list[]* is already completed, *uaio_suspend* returns immediately. If any element in the *list[]* is NULL, the element is ignored. If *timeout* is specified (not NULL), the process remains suspended until a request is finished or *timeout* expires, whichever occurs first. Otherwise, the process is not awakened until a request is finished.

```
Input: [for aio_suspend]
    List of asynchronous I/O control blocks (requests) that the process is
    waiting on
    # of elements in the list
    Timeout used to indicate duration to wait without any request completion


Flow:
    If list is empty // process blocking for no requests
        Return 0;
    End if

    For each element in the list
        If the request is completed
            Return 0;
        End if
    End for

    While no request in the list is completed
        Schedule the reactor to process the requests;
        If a request is completed
            Return 0 if no signaling is required;
            Else set errno EINTR and return –1;
        End if

        If timeout !=NULL and timeout is expired
            Set errno to EAGAIN;
            Return –1;
        End if
    End while
```

This function returns:

0 - if one or more requests from the list have terminated

-1 - otherwise, and *errno* is set accordingly

Possible *errno* values are:

EAGAIN – None of the requests from *list* completed in time specified by *timeout*

EINTR - Signal interrupted the *uaio_suspend.* This signal might also be sent by AIO implementation while notifying termination of one of the requests

- ***int aio_cancel (int fildes, struct aiocb *aiocbp)***

This function is used to cancel one or more outstanding requests. The first parameter *fildes* denotes the file descriptor for which all the requests will be canceled. This is used only when *aiocbp* is NULL. If *aiocbp* is specified, the request corresponding to the specific control block is canceled. When a request is canceled, notification based on its *'struct sigevent'* is performed. If a request cannot be cancelled, it terminates the usual way after performing the operation. The *errorState* of the canceled request is set to *ECANCELED* and *returnStatus* is set to –1.

This function returns:

AIO_CANCELED - Requests exist and are cancelled successfully

Input: [for aio_cancel]

File descriptor used to cancel all outstanding request on that file
Asynchronous I/O control block used to cancel a specific request

Flow:

```
If file descriptor is –ve
    Set errno to EBADF;
    Return -1;
End if

If async control block is NULL
    // delete all requests corresponding to the file descriptor
    no_of_request_not_yet_completed = 0;
    unable_to_cancel=0;

    For each element in the library internal queue
        If same request's file descriptor and request not yet completed
            Deregister the request from the reactor;
            If deregistration successful
                Update request element in the queue for cancellation;
            Else
                Update request element in the queue for non-cancellation;
                unable_to_cancel=1;
            End if
            no_of_request_not_yet_completed++;
        End if
    End for

    If no_of_request_not_yet_completed == 0
        // all requests are already terminated
        Set errno AIO_ALLDONE;
        Return –1;
    End if

    If unable_to_cancel == 1
        // at least one outstanding request could not be cancelled
        Set errno AIO_NOTCANCELED;
        Return –1;
    End if
Else
    // cancel the specific request
    Find the element corresponding to the request in the queue;
    If request not yet completed
        Deregister the element from the reactor;
        If deregistration successful
            Update request element in the queue for cancellation;
            Return AIO_CANCELED;
        Else
            Update request element in the queue for non-cancellation;
            Return AIO_NOTCANCELED;
        End if
    End if

Return AIO_CANCELED;
```

AIO_NOTCANCELED	-	one or more requests exist that couldn't be cancelled
AIO_ALLDONE	-	all requests already terminated
-1	-	error occurred during execution of *uaio_cancel* *errno* is set to *EBADF* indicating *fildes* is not valid

- ***int uaio_error (const struct aiocb *aiocbp)***

This function returns the error status *errorState* associated with the *aiocbp* passed in. The error status for an asynchronous I/O operation is the *errno* value that would be set by the corresponding *read*, *write*, *fsync* or *uaio_cancel* operation. This function returns:

EINPROGRESS	-	If request not yet terminated
errorStatus value	-	If the request is completed.
-1	-	If *aiocbp* does not refer to an asynchronous operation whose return status is not yet known. *errno* is set *EINVAL*

Input: [for aio_error]
Asynchronous I/O control block (request)

Flow:
Find the element corresponding to the request in the queue;

If element is found
Return the error state of the request;
Else
Errno is set by the sub-function to EINVAL;
Return –1;
End if

- ***int aio_return (const struct aiocb *aiocbp)***

This function returns the return status associated with the *aiocbp* passed in. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding *read*, *write* or *fsync* operation. When *uaio_error* returns *EINPROGRESS*, this function returns 0 (in the *glibc* version, the behavior is undefined). Upon completion of the request, *uaio_return* must be used exactly once to retrieve the return value and the request is erased from the queue. *uaio_error* is used to check if the request is completed and only when the request is completed, *uaio_return* must be used to get the result.

```
Input: [for aio_return]
    Asynchronous I/O control block (request)

Flow:
    If the error state of the request is not EINPROGRESS
        Find the corresponding element in done list
        If such element exists
            Delete the request from the queue;
        End if
        Return the return status of the request;
    Else
        Errno is set by the sub-function to EINVAL;
        Return –1;
    End if
```

- ***void uaio_init (const struct aioinit *init)***

The optional *uaio_init()* function allows the calling process to initialize the asynchronous I/O interface. In the *glibc* asynchronous I/O implementation, initialization of the interface may include starting slave threads that can be used to

carry out I/O requests, and allocating data structures so that *malloc* as latter stages can be avoided. But in UAIO, only optimization need is to pre-allocate the queue. As part of this function, memory for the *free_list* queue is allocated for *aio_num* of elements (in *struct aioinit*). A timer is also set to schedule the Reactor every 1μsec.

4.3 Implementation of the reactor handler functions

This section discusses the handler functions associated with the asynchronous I/O event handler objects. The reactor invokes these functions when the system is ready to handle I/O. When the file descriptor is ready for an I/O operation, the reactor triggers the handle functions *handle_input()* or *handle_output()* on the specific handler. The input handle function *handle_input()* is called if the handler associated with the file descriptor is registered for input operations (like *read*, *uaio_read*). The *handle_output* function is invoked if the handler associated with the selected file descriptor is registered for output operations (like *write*, *uaio_write*). The selected file descriptor and its associated asynchronous I/O handler are passed as arguments to these handler functions.

- ***int handle_input (void *arg, int filedescr)***

  This method is called whenever the reactor detects that data is available for reading on the file descriptor. The asynchronous I/O control block *aiocbp* is extracted from the handler and its elements are used to perform a *pread* operation. The results of the read operation are stored in the corresponding queue element

and the handler is removed from the reactor. The queue is checked to see if the next element in the queue is of type *LIO_SIGNAL*. Such an element is added to the queue when *ulio_listio* is used in *LIO_NOWAIT* mode with a valid signal request for notification. This is discussed in detail in the *ulio_listio* implementation section. If so, the specified signal is raised to notify of completion of requests submitted using *ulio_listio*. The element is removed from the *run_list*.

```
Input: [for handle_input]
        void *arg
        int filedescr

Flow:
    aioHandler aioh = (aioHandler) arg;

    // Retrieve the aio control block from the handler
    aioCB=aioh->aiocbp;

    // Perform the read operation
    bytes_read= pread(filedescr,(void *)aioCB->aio_buf,aioCB->aio_nbytes,
                        aioCB->aio_offset);

    If read operation fails (i.e., bytes_read==-1)
        // Update the corresponding element in the queue with its Error
        // state set to the  'errno' value of pread.  Return status is set to –1.
        update_request(aioCB,errno,-1);
    Else
        // Upon successful completion of the operation, error state is set
        // 0 and return status indicates the # of bytes read
        update_request(aioCB,0,bytes_read);
    End if

    Find the next element in the list;

    If the next element in the list is of type LIO_SIGNAL
            Raise the signal specified in aio_sigevent (part of the element)
            Delete the element from the queue;
    End if

    Deregister the handler 'aioh';
    Delete the handler;
    Return SUCCESS;
```

- ***int handle_output (void *arg, int filedescr)***

This method is called whenever the reactor detects that an output port is available and data can be written to the file descriptor. This function is similar to *handle_input()* except that it checks if the next element is of *LIO_SYNC* or *LIO_DSYNC* type. If so, *fsync()* or *fdatasync()* is performed on the file descriptor and the element is updated accordingly.

Input: [for handle_output]

```
void *arg
int filedescr
```

Flow:

```
 aioHandler aioh = (aioHandler) arg;

// Retrieve the aio control block from the handler
aioCB=aioh->aiocbp;

// Perform the write operation
bytes_written= pwrite(filedescr,(void *)aioCB->aio_buf,aioCB->aio_nbytes,
                          aioCB->aio_offset);

If write operation fails (i.e., bytes_written==-1)
    // Update the corresponding element in the queue with its Error
    // state set to the  'errno' value of pwrite.  Return status is set to –1.
    update_request(aioCB,errno,-1);
Else
    // Upon successful completion of the operation, error state is set
    // 0 and return status indicates the # of bytes written
    update_request(aioCB,0,bytes_read);
End if

Find the next element in the list;

While next element in the list is either LIO_SIGNAL or LIO_SYNC or LIO_DSYNC type
    If the next element in the list is of type LIO_SIGNAL
        Raise the signal specified in aio_sigevent (part of the element)
        Delete the element from the queue;
        Continue;
    End if

    If the next element in the list is of type LIO_SYNC
        Perform fsync() on the file descriptor aio_fildes.
        Update the error state & return status of element in queue.
        Raise the signal specified in aio_sigevent (part of the element)
        Continue;
    End if

    If the next element in the list is of type LIO_DSYNC
        Perform fdatasync() on the file descriptor aio_fildes.
        Update the error state & return status of element in queue.
        Raise the signal specified in aio_sigevent (part of the element)
        Continue;
    End If
End While

Deregister the handler 'aioh';
Delete the handler;
Return SUCCESS;
```

4.4 UAIO Library Initialization/Finalization

The asynchronous I/O library initialization and finalization are accomplished using the *uaio_init()* and *atexit()* functions. The *uaio_init* function takes a *struct aioinit* as its parameter. The *aio_num* parameter of this structure is used to pass the initial desired number of free list elements. This function, if called, must be the first before any other function of the UAIO interface is invoked. If not explicitly called by the user, this function is invoked when the first asynchronous I/O request is submitted. The initialization function performs the following tasks:

- Allocation of queue memory – This is done with a view for performance improvement. As the memory is allocated early on in the process, the need to do *malloc* to create queue elements is deferred. If the initially allocated resources are not sufficient, then additional resources are allocated when required.
- Setting the timer to schedule the Reactor – A timer is set to schedule the reactor every 1µsec[3]. As the reactor is part of the user process, scheduling it effectively means calling the function *handleEvents()*, which detects the events.

As part of initialization, *atexit()* is used to specify the function that is invoked during the normal termination of the program. This method frees all the resources (queues & other data structures) allocated to the library.

---

[3] Even though the timer interrupt is set for each µsec, the timer is triggered once every 10000 µsec only due to the coarse granularity of the system clock in Linux.

4.5 Implementation level flow control in the UAIO library

This section discusses the flow of control in the library. As discussed in the previous section, the library initialization is accomplished using the *aio_init()* function. As part of initialization, a timer is set to invoke the reactor every microsecond. Memory allocation for the internal queue is also done to avoid allocation at a later stage. This is done to improve the performance of the system. When an asynchronous I/O request is made, the request can be a *read*, *write*, *cancel*, *suspend* or *fsync* request. There are status requests (*uaio_error* & *uaio_return*) used to check the status of a request. Here a request is used in a general perspective, and does not refer to the asynchronous I/O control block. The reactor is used to do the asynchronous read or write operation. So, only the tasks that require some I/O operation are registered with the reactor.

When the user submits a request, the request is checked for validity. If the request is a read or write request, the request is stored in the library internal queue and is registered with the reactor. All other types of requests are not registered with the reactor. The *uaio_fsync()* and *uaio_suspend()* functions are the synchronization primitives provided by the interface. These are discussed in detail in the previous sections. The flow diagram depicted below illustrates the functional flow depending on the type of user request. Status requests not included as they are involve only the queue retrieval type functionality.

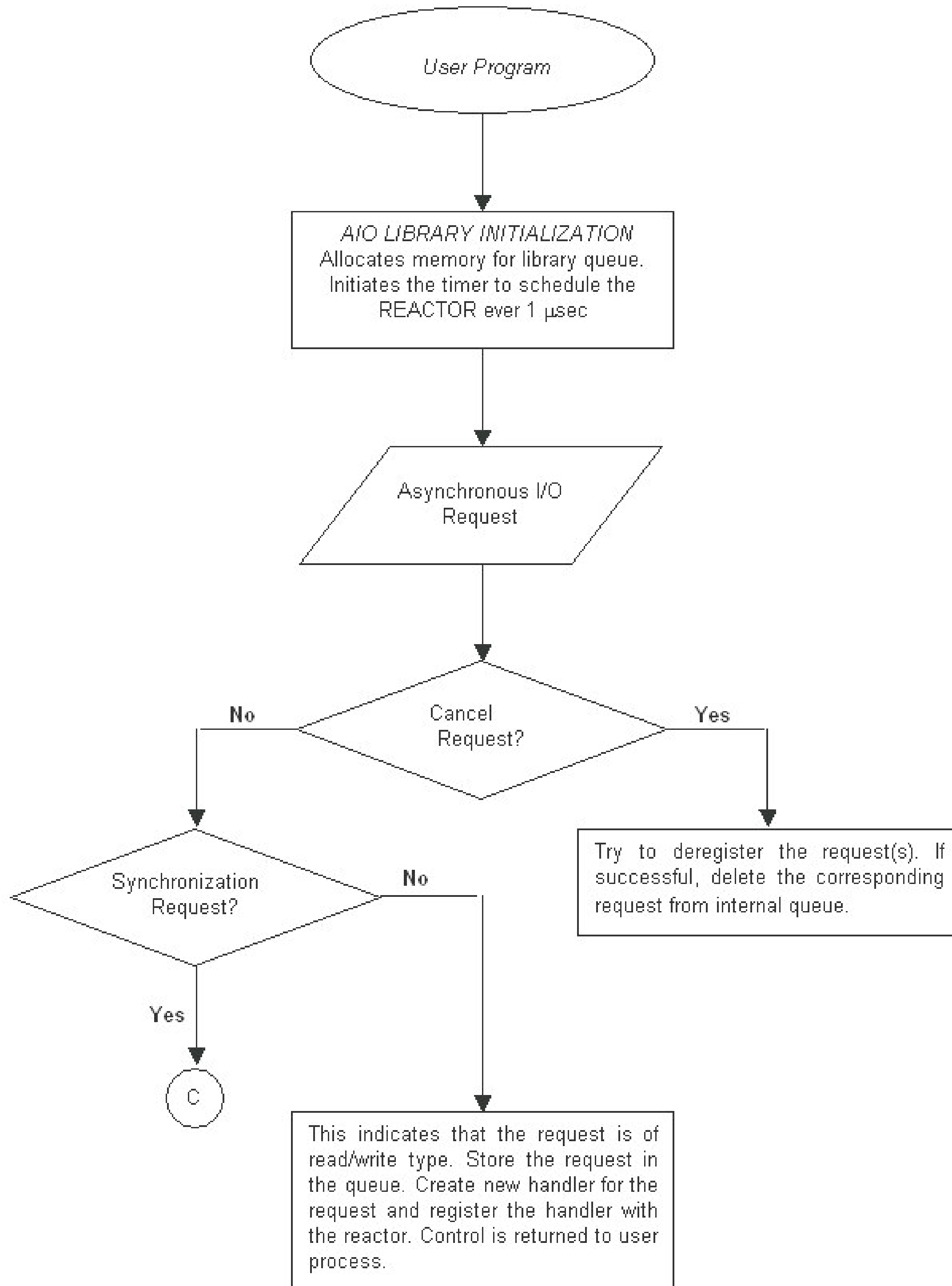


Figure 4-1 Process Flowchart

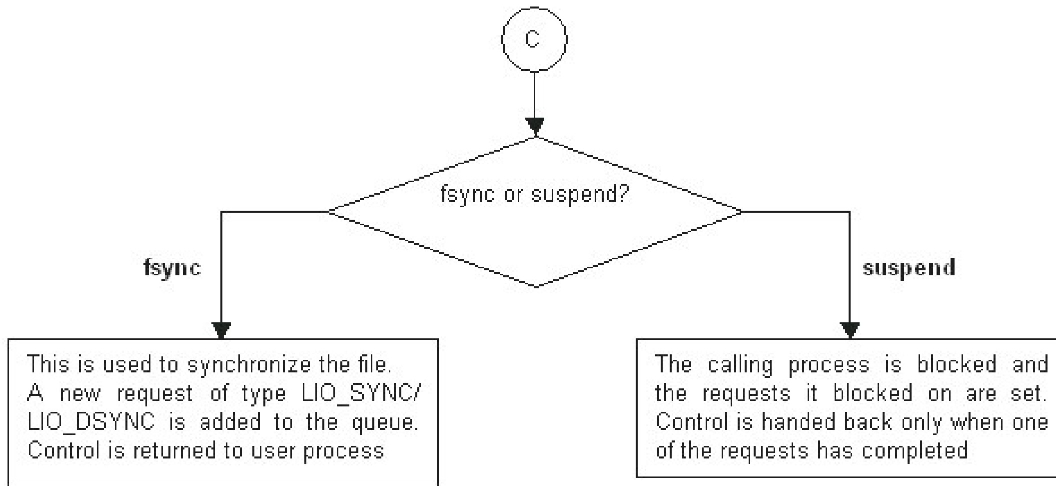


Figure 4-2 Process Flowchart Contd.

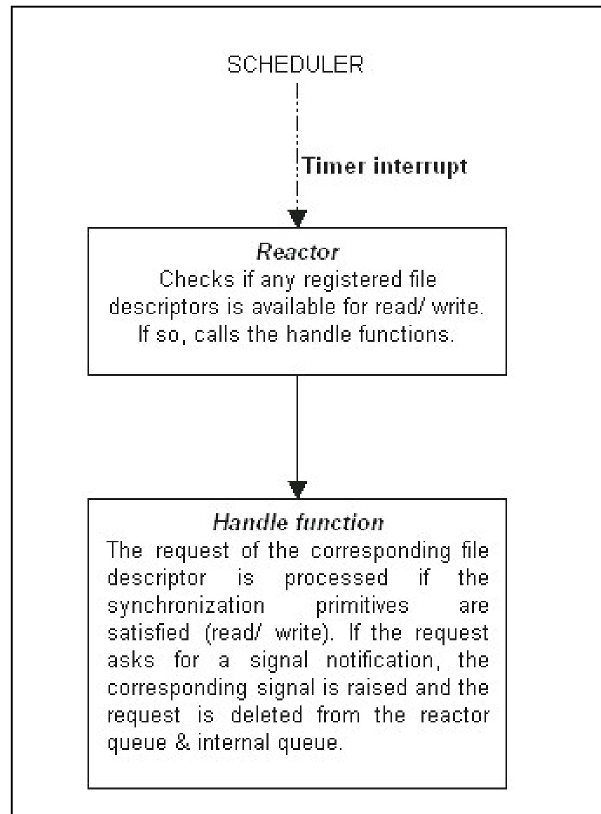


Figure 4-3 Reactor Flow

4.6 Limitations

This section discusses the limitations of the UAIO library. As the reactor and library are part of the user process, the user process cannot wait (wait for a signal or sleep). If the user process waits, the Reactor, being part of the process, is also forced to sleep. This is not an issue in the *glibc* version of asynchronous I/O, as a thread is created to handle each request. (In Linux, each thread is represented as a separate process.) Another limitation of this library is its inability to distribute its load on different processors in a multi-processor machine, because the library forms part of the user process (single process). In contrast, the *glibc* version invokes separate threads (which are processes) to process each request. Each thread/process can be given to a processor and thus the parallelism of the underlying hardware is exploited.

5 Testing

This section describes how testing of the UAIO library is done. Testing is done to verify correct operation of the library and its POSIX compliance. Performance testing is also done to compare the performance of UAIO with the *glibc* asynchronous I/O implementation.

5.1 Correctness Testing

Correctness comes first, with performance to follow. One of the goals of this implementation is its POSIX compliance. The various functions provided as part of the library interface are tested.

- Test 1 - Initially the functions are tested for their interaction with the user process. Faulty requests are submitted to the library, and the return values, error states are checked. This ensured the POSIX compliance of the library.
- Test 2 – Both explicit and implicit initialization of the library is checked. Explicit initialization is done by invoking the *uaio_init()* function in the user program. Implicit initialization is accomplished by submitting an I/O request. The library is then checked for initialization. If not initialized, the library is initialized. Finalization of the library (clean-up work on exit) is also verified.
- Test 3 – This test verifies the interaction between the library interface and the Reactor. The basic function of asynchronous read *uaio_read()* and write *uaio_write()* are tested. An I/O request is submitted to the library. Notification mechanism upon request completion is also specified as part of the request. The

Reactor, triggered for each timer interrupt, process the request and updates the corresponding fields of the request. Notification mechanism specified by the user is used to notify the request completion. Signal notification and no notification methods are also tested here.

- Test 4 – Upon validating the above said tests, batch submitted asynchronous I/O operation (*ulio_listio*) is verified. For this a list of I/O read/write requests are submitted using *ulio_listio()*. The completion of all the requests is verified. *ulio_listio()* in both *wait* mode and *no-wait* mode is verified.
- Test 5 – In this test a list of requests is submitted using the *ulio_listio()* function in *no-wait* mode. Then one of these requests is selected and cancellation function *uaio_cancel()* is invoked to cancel the request. The error state of the request, obtained using *uaio_error()*, is checked for successful request cancellation.
- Test 6 – The synchronization primitive *uaio_fsync* is checked as part of this test. *uaio_fsync()* is used to do *fsync()* or *fdatasync()* on the specified file before any further I/O requests on this file are honored. This is tested by submitting a list of I/O requests, followed by a *uaio_fsync()* request and some more list I/O requests. The *fsync* operation is found to occur after the first batch of requests is completed and before the second batch is processed.
- Test 7 – In this test the *uaio_suspend()* function is used to suspend the calling process until any one of the requests specified is complete or a timeout occurred. This is verified by submitting a batch of requests in *no-wait* mode and invoking

uaio_suspend() on some of these requests. The control return to the calling process upon timeout or request completion is verified.

During all these above tests, if there is no notification primitive indicated by the user, the state of the requests is obtained using the *uaio_error()* and *uaio_return()* functions.

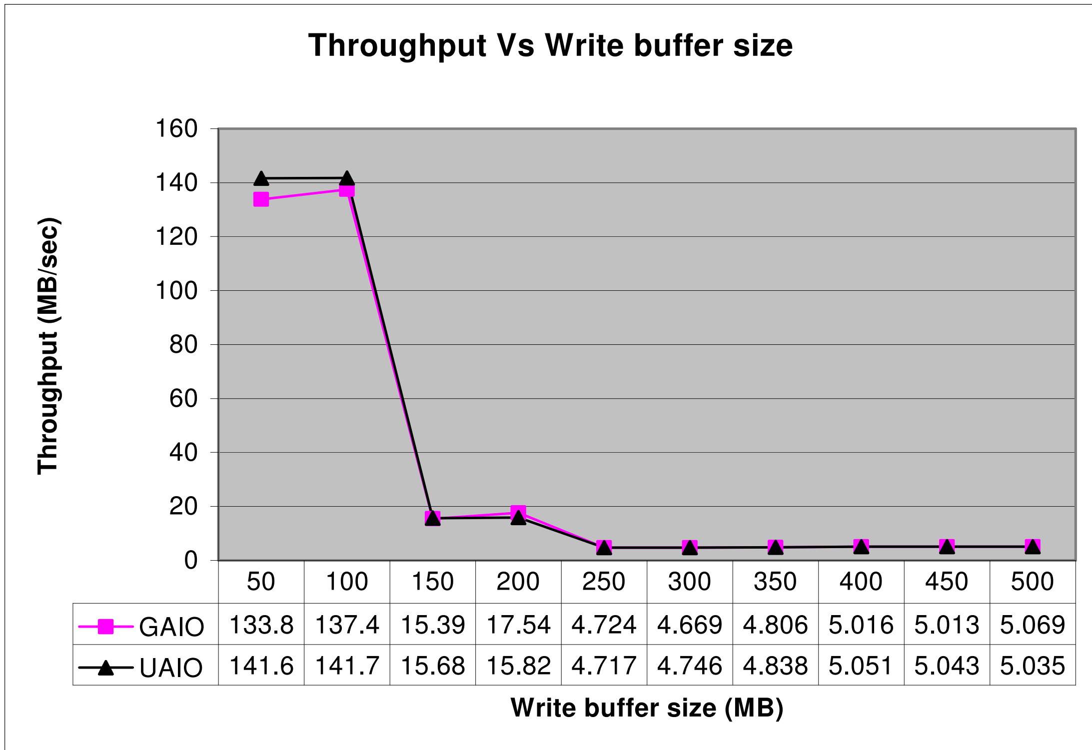
5.2 Performance Testing

As mentioned above, the UAIO library is single threaded in nature, vying for better performance by saving the CPU cycles needed for thread maintenance and context switching among various threads. Comparing the UAIO library to its *glibc* equivalent, which is a multi-threaded approach, is the best way to demonstrate the power (if not, the weakness) of this implementation[4].

5.2.1 One single write call with varied buffer size

Buffer size of a single *aio_write* call is varied and the corresponding throughput is measured for UAIO and *glibc* AIO. 10 samples of each instance are taken. The *glibc* version spawns one thread for each asynchronous I/O call. As only one AIO call (*aio_write*) is used as part of this test, the *glibc* version generates only one slave thread to process the request. So, there isn't much noticeable difference between the throughput statistics for UAIO and *glibc* AIO.

[4] All the tests are conducted on a Pentium-III 802.933MHz machine with 256MB RAM.

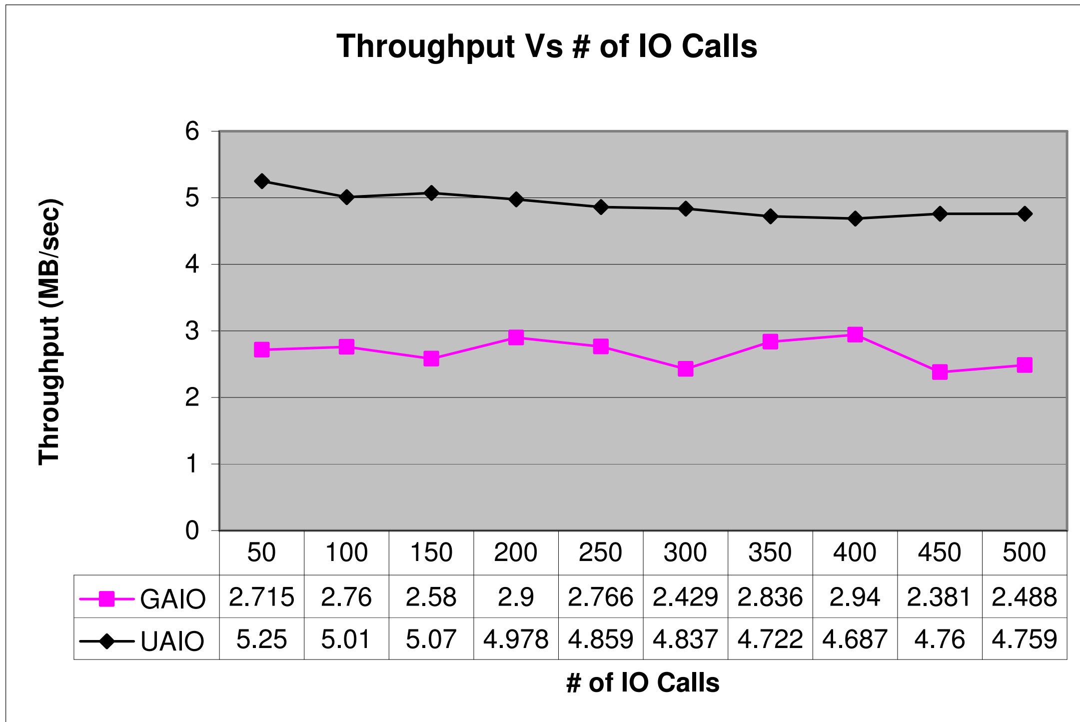


Write Buffer Size (MB)	GAIO		UAIO	
	Mean	95% Confidence Interval	Mean	95% Confidence Interval
50	133.8254	+/- 0.2719	141.6114	+/- 1.1762
100	137.4333	+/- 1.1622	141.6908	+/- 1.1590
150	15.391	+/- 0.4540	15.68189	+/- 0.4579
200	17.5362	+/- 0.6211	15.81765	+/- 0.4457
250	4.72431	+/- 0.0581	4.717336	+/- 0.0447
300	4.669149	+/- 0.1060	4.745678	+/- 0.0526
350	4.806358	+/- 0.0750	4.838455	+/- 0.0693
400	5.016221	+/- 0.0750	5.050673	+/- 0.0686
450	5.013109	+/- 0.0535	5.043232	+/- 0.0225
500	5.068782	+/- 0.0534	5.035199	+/- 0.0617

5.2.2 Varied number of I/O calls for constant write buffer size

In this test, the number of I/O calls or requests is varied and the corresponding throughput is measured for UAIO and *glibc* AIO. The size of the total data written is kept constant throughout the test. A list based I/O (*lio_listio*) is used to submit a bunch of I/O requests. Each I/O call is a write request that writes into a file. 20 samples of each instance are taken.

As the number of threads created is directly proportional to the number of I/O calls, we see a noticeable difference in the throughput statistics due to the overhead involved in creating and context switching threads for *glibc* AIO.



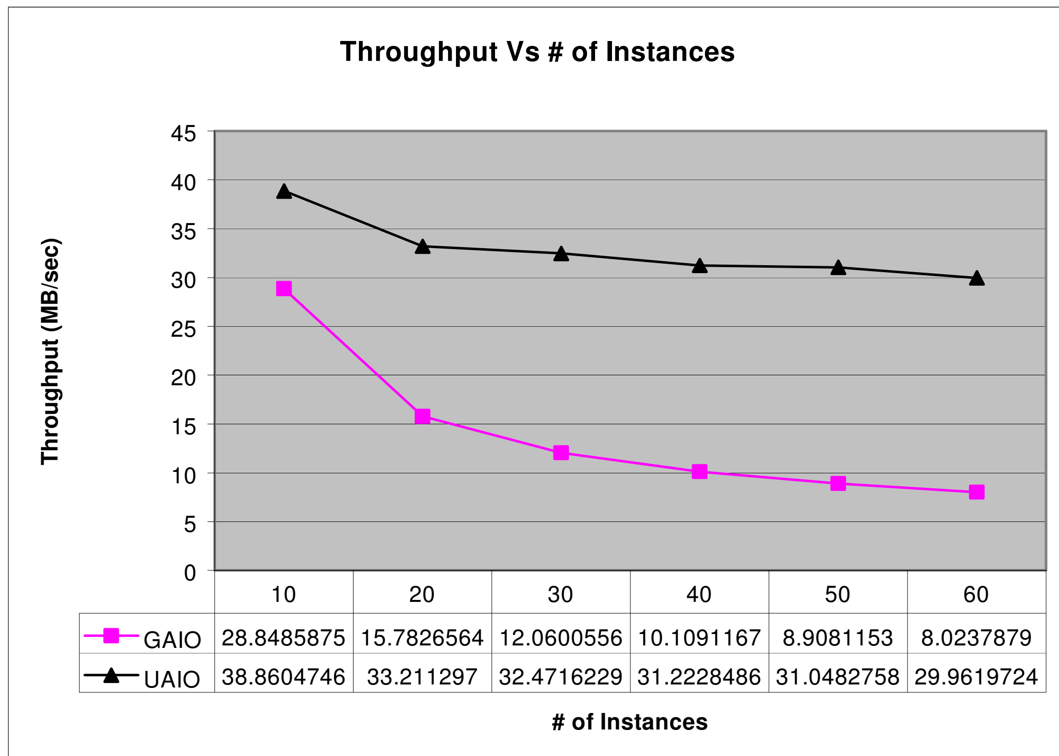
# of IO Calls	50	100	150	200	250	300	350	400	450	500
GAIO	2.715	2.76	2.58	2.9	2.766	2.429	2.836	2.94	2.381	2.488
UAIO	5.25	5.01	5.07	4.978	4.859	4.837	4.722	4.687	4.76	4.759

No. Of I/O Calls	GAIO		UAIO	
	Mean	95% Confidence Interval	Mean	95% Confidence Interval
50	2.715086	+/- 0.0589	5.24993315	+/- 0.1154
100	2.760442	+/- 0.0307	5.0101565	+/- 0.1420
150	2.57972	+/- 0.0441	5.06975625	+/- 0.0673
200	2.8996	+/- 0.0254	4.97798985	+/- 0.0832
250	2.766457	+/- 0.0453	4.85852445	+/- 0.0885
300	2.42936	+/- 0.0286	4.83662635	+/- 0.0588
350	2.835771	+/- 0.0474	4.7223428	+/- 0.0749
400	2.939973	+/- 0.0367	4.6865348	+/- 0.0605
450	2.381276	+/- 0.0210	4.7603343	+/- 0.0647
500	2.487636	+/- 0.0208	4.7586084	+/- 0.0811

5.2.3 Varied number of I/O read calls

In this test, the number of instances of a process that execute concurrently is varied and the corresponding throughput is measured. Each process does a list based I/O (*lio_listio*) operation, using which a batch of 200 I/O read requests are submitted. In the *glibc* version, as each process makes 200 I/O calls and each is assigned a thread, running 50 instances of the process amounts for ~10000 threads.

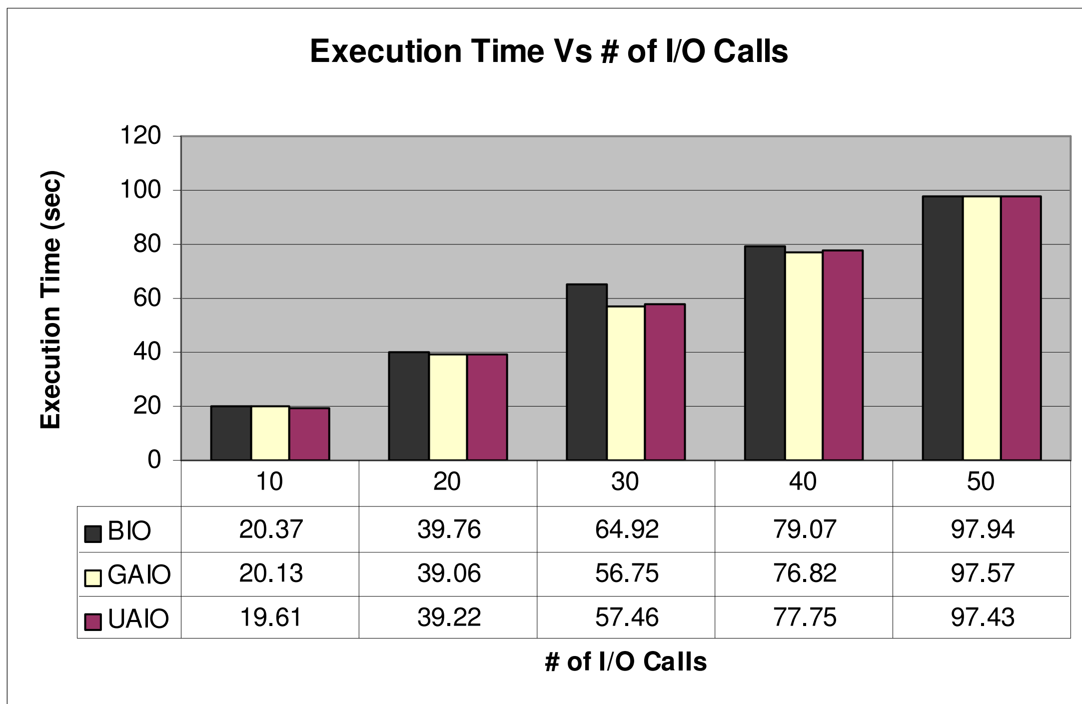
In UAIO, throughput drops slowly with the increase in the number of instances of the same process that execute concurrently. But, the throughput in the case of *glibc* AIO (GAIO) drops drastically due to the overhead associated with the threads.



Write Buffer Size (MB)	GAIO		UAIO	
	Mean	95% Confidence Interval	Mean	95% Confidence Interval
10	28.8485875	+/- 0.2489	38.8604746	+/- 0.8961
20	15.7826564	+/- 0.1049	33.211297	+/- 2.0953
30	12.0600556	+/- 0.0913	32.4716229	+/- 2.6864
40	10.1091167	+/- 0.0495	31.2228486	+/- 1.9044
50	8.9081153	+/- 0.0720	31.0482758	+/- 2.0664
60	8.0237879	+/- 0.0726	29.9619724	+/- 1.9132

5.2.4 Combination of CPU and I/O intensive tasks

While the previous tests are aimed at evaluating the library performance with respect to I/O intensive jobs, this test aims at tasks that are both I/O and CPU intensive. Audio data is read from different files of same size and Fast Fourier Transform (F.F.T) is applied on the audio samples. The number of reads performed is varied to create different I/O loads. The reads are done using standard blocking I/O, *glibc* and UAIO asynchronous I/O methods. The performance metric is chosen to be total execution time of the process. It was observed that the performance of all the three implementations was almost the same. A plausible reason for this result is that CPU processing was masking I/O processing, i.e., CPU processing dominated I/O processing. Hence performance gain due to UAIO could not be seen.



	10	20	30	40	50
BIO	20.37	39.76	64.92	79.07	97.94
GAIO	20.13	39.06	56.75	76.82	97.57
UAIO	19.61	39.22	57.46	77.75	97.43

6 Conclusions and Future Work

A user level single threaded asynchronous I/O library is implemented. The library is POSIX compliant and can be used for I/O-intensive applications. We have tested it with system data loads in excess of 300MB and it functioned correctly. It is concluded from the performance tests that by eliminating the threads and associated context-switching overhead, the UAIO implementation offers better performance in comparison with the libraries that employ multi-threading (like the *glibc* version).

Future work includes writing wrappers to blocking system calls like *wait()* and *sigsuspend()*, allowing the user to use programs written using these calls with minimum modifications. Another possible extension includes developing a conditional-threading framework within the library that uses the multi-processor nature of the machine. In multi-processor systems, separate threads can be created to handle requests corresponding to different files. Having threads to process requests for the same file may not be worth the trouble as all these threads queue for the same resource. This library can be used in the development of a Proactor, a pattern that supports the demultiplexing and dispatching of multiple event handlers, which are triggered by the completion of asynchronous events [12]. The regions of its use in database systems can also be explored.

References

[1] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, pages 529-545, Aug 1995.

[2] *Information Technology -- Portable Operating System Interface (POSIX) -- Part1: System Application Program Interface (API) [C Language]*, chapter 6.7 Asynchronous I/O, pages 165-180. The Institute of Electrical and Electronics Engineers, Inc, July 1996.

[3] Microsoft Windows 32-bit API, 1999.

[4] *glibc/librt Asynchronous I/O Interface Reference man pages*.

[5] *Sun Solaris man pages for Asynchronous I/O interface laio*.

[6] Silicon Graphics, Inc. *Kernel Asynchronous I/O (KAIO) for Linux*. http://oss.sgi.com/projects/kaio/

[7] Suparna Bhattacharya. Design notes on Asynchronous I/O for Linux. Technical report, IBM Software Labs, India, 2002.

[8] Remzi H. Arpaci-Dusseau, Muthian S., Venkateshwaran Venkataramani. Block asynchronous I/O: A flexible infrastructure for user-level filesystems. In *The International Conference on High-Performance Computing (HiPC '01)*, pages 249-261, Hyderabad, India, Dec 17-20 2001. HiPC '01.

[9] *Message Passing Interface (MPI) Standard 2.0 Specification*, chapter 9. 1997.

[10] Dan Bonachea. Bulk file I/O extensions to java. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 16-25, San Francisco, CA, June 2000. ACM.

[11] Rajukumar Girimaji. Reactor, a software pattern for building, simulating and debugging distributed and concurrent systems. Master's thesis, The University of Kansas, 2002.

[12] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt and Thomas D. Jordan. Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. In the 4th Annual Pattern Languages of Programming conference, Allerton Park, Illinois, September, 1997.

[13] Bill Gallmeister. *POSIX.4: Programming for the Real World*. Sebastopol CA: O'Reilly &Associates, 1 edition, 1995.

[14] Jim Mauro. Asynchronous I/O and large file support in Solaris. In *SunWorld*, Vol. 12 No.7, July 1998.

[15] K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, Phil C., and A. Aiken. Titanium: A high-performance java dialect. In *Workshop on Java for High-Performance Network Computing,* pages 1-13, Stanford, CA, February 1998. ACM.

[16] *glibc/librt* Source Code.