Names: Ashritha Nagavaram (nagav001) , Gaurav Khandelwal (khand052), Gopal Sarda (sarda014)

Compilation:

To compile, on Linux, run

make -f Makefilelinux

To compile, on solaris, run

make -f Makefilesolaris

To run server on both these platforms, run ./object_file and use port no. as parameter
Example, to run question1 on port no. 5000, run
./question1 5000

Note: In all above scenarios, our server never crashes.

Result Evaluation:

For Q1, we were keeping track of time taken by each thread for completion. The average time is reported by taking random sample of 50 threads from the run and calculating their mean.

For thread , Q1

#request	size of data	Linux (avg time/thread)	Solaris (avg time/thread)
100	100 KB	49350	680398
	10 MB	9411476	85473201
5000	100 KB	258357	1025810
	10 MB	89945129	45089262
10000	100 KB	746690	1159376
	1 MB	6507722	17060381
	10MB	72094851	63544752

For aio_read() , Q2

#request	size of data	Linux (avg time/thread)	Solaris (avg time/thread)
100	100 KB	58890	621654
	10 MB	10885022	131882610
5000	100 KB	773693	5998503
	10 MB	112175175	683017145
10000	100 KB	873024	5496734
	1 MB	12013034	73334539
	10MB	107206358	281476289

For read with fcntl (polling) , Q3

#request	size of data	Linux (avg time/thread)	Solaris (avg time/thread)
100	100 KB	47295	555366
	10 MB	8302003	88496487

5000	**100 KB**	**152354**	**2037867**
	10 MB	**77682289**	**358592193**
10000	**100 KB**	**134713**	**4185090**
	1 MB	**3645655**	**21330630**
	10MB	**76233937**	**343096908**

For read with select() , Q4

#request	**size of data**	**Linux (avg time/thread)**	**Solaris (avg time/thread)**
100	**100 KB**	**38370**	**595020**
	10 MB	**8357026**	**145810731**
5000	**100 KB**	**219252**	**1974581**
	10 MB	**85843699**	**232657691**
10000	**100 KB**	**424063**	**1971924**
	1 MB	**3945988**	**19043344**
	10MB	**76733875**	**229038775**

Explanation for few observed scenarios

We have computed our average per thread result. Results might vary due to various factors such as network congestion, the other background processes running etc. During our analysis, we observed better performance when both client and server were running on same machine, mainly due to zero network latency. “Too many open files” error were encountered sometimes during the testing. When we increased the number of file descriptor quota of each user, our system worked smoothly.

1. With increasing data size average time taken increases because we are reading data in chunks, to read data we are issuing multiple read system calls.
2. With increase in number of requests average time increases because server is occupied with the previous requests.
3. Linux system performed better compared to Solaris due to limited Solaris resources.
4. For a particular number of threads and data size, performance is in the following order: select() > read with fcntl (asynchronous/ non-blocking) ~ thread > aio_read with polling.
5. select() performs better because Select system call places CPU into low power waiting time and once our read is ready, select will interrupt and we will iterate to find out which selection is ready and we will read data accordingly. So, advantage here we can see is no need to poll on read system call. (reference : http://stackoverflow.com/questions/11496059/how-do-system-calls-like-select-or-poll-work-under-the-hood).
6. aio_read is least performed because it takes time to iterate over the data structure , there is also additional computational cost for creating the data structure, allocating the structure for aiocb. Multiple system calls are used such as aio_read, aio_error etc.

Flow of Control

For Thread : The server assigns one thread per client. The thread is created using pthread_create command which creates thread in user space. The user level thread calls a system call read which reads from the specified socket id, since this is a system call the control is transferred to kernel space. The system is completed once it reads 1024KB, it then returns the control to the user space. The threads which have pending data to read continues in the same process by switching control between user thread and kernel space. If the read is completed, the user thread closes the connection using close() system call, the control goes to the kernel and the socket is closed. The thread then return from the function and terminates.

For aio_read() : We continuously check for client requests using the non-blocking accept system call. Once, a valid socket id is returned from accept, we set an asynchronous I/O control block using the socket id and issue an asynchronous read system call (aio_read()). Since, the read is asynchronous, we can continue checking for more client requests while aio_read() has data ready for us to be read. A linked list is used to keep track of all the active asynchronous control blocks. Once, data has been completely read from a control block, it is removed from the linked list. aio_error() system call is used to check if data from a socket is ready to be read. If aio_error() returns value other than EINPROGRESS, that indicates the socket is ready. Once a socket is ready, you can read the data using aio_return() system call.

For read with fcntl (polling) : Here, instead of using aio_read() system call, we used normal read system call but make it work as asynchronously (non-blocking). We have used linked list for iterating our list structure, which stores socket and buffer. When a request comes to a server, our server application accepts the request via accept system call (user is communicating with kernel via system call). Since, the read is asynchronous, we can continue checking for more client requests. We are also polling to check if client has sent data. So, once read succeeds, we read it via read system call. Once read is done, we close socket file descriptor and remove it from our pending list of sockets.

For read with select() : This part is similar to above part except that instead of polling, we are using select system call. We don't have to maintain a separate structure or linked list. We uses predefined set provided by select.

Test Cases:

- ❏ Single request , multiple request from a client where both server and client have Linux environment,Solaris environment
- ❏ Multiple requests from a single client where both server and client are same
- ❏ Single request from multiple client having same data size where both server and client have Linux environment, server on Solaris environment or mixed server environment
- ❏ Multiple request from multiple client having same data size where both server and client have Linux environment
- ❏ Multiple request from multiple client having different data size where both server and client have Linux environment and server has Solaris environment
- ❏ 1000 threads with mixed data size upto 10MB
- ❏ 5000 threads with same data size of 1MB
- ❏ 10000 threads with mixed data size upto 10MB
- ❏ 1 thread with 10MB

Negative Test Cases

- ❏ Closing server abruptly without serving the clients for the request coming from multiple clients when server is running on Linux ,Solaris environment
- ❏ Opening listening socket of server which is reserved for other applications such as port 25
- ❏ Opening listening socket of server which are blocked by firewall
- ❏ Single and multiple request from a single client having data size greater than 10MB where both server and client have Linux ,Solaris environment
- ❏ Single and multiple request from a single client having data size greater than 10MB where both server and client are same
- ❏ Single and multiple request from multiple clients, where few data sizes are greater than than 10MB when both server and client have Linux ,Solaris server environment
- ❏ Single and multiple request from multiple clients, where all data sizes are greater than than 10MB when both server and client have Linux , Solaris server environment