CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

Important Assumptions:

- Server and client only uses resolved IP addresses and not the DNS names
- Both client and server should be started with same number of connections. Else results will be unexpected
- For variable size, our assumption is that always the input size should be 10MB so that we cover all the sizes in our valid range.

Test cases covered:

1. Running server and client on same host (linux) vs on different hosts(linux)
   Running on same host does a lot better than running on different hosts. As seen below in the results, the time of execution also significantly changes.
2. Running server on solaris and client on (linux), I was not able to run python on those machines. So could not run client on the solaris. Also, solaris machine was too slow and throwing out of memory error, so we could only run 2 3 and 4 for a small set of instances. Results mentioned below.
3. Start client when server is not available
   All client connections fail with Connection refused error
4. When server is already up and running, one client connects with x number of parallel requests already. Server is only expecting n connections i.e., it can only serve n-x requests more. At this moment, a new instance of client connects to the same server with another n new requests. Performed some experiments by using the standard commands to run as mentioned below.
5. When the file size is beyond 10MB
   If both client and server are on same host, then most of the runs are successful even upto 100MB for 5K connections. But when they are on different hosts, they fail with multiple errors ‘Too many open files’ ‘Connection reset’ ‘Connection timeout’
6. Server expects n connections but clients sum up to less than n requests
   In this case, server keeps waiting for the rest of the connections. So you may not see the throughput/time values which you see when server logically exits the program.
7. Without giving MB/KB in size
   It still works as expected. Ran some experiments (could not capture results in the document though). However, for variable size, our assumption is that always the input size should be 10MB so that we cover all the sizes in our valid range.

Control Flow

Pthreads

When a new thread is created using pthreads, we are using the abstraction provided by the OS to implement multi-threading using kernel threads. The control flow in brief is given below

- Pthreads use 1:1 kernel level thread model. Which means when a new pthread is created, a new thread context is created and this thread context is handed over to kernel to be scheduled using the kernel threads.

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

- When a new thread is created, user library allocates a user level stack for the thread and makes a system call. Then we switch to kernel mode from user mode.
- Kernel allocates the TCB and interruption stack and arranges the state of the kernel to start execution on the user level stack.
- After the thread creation, kernel puts new thread on ready list to be scheduled like any other thread and returns an identifier to user so that user can access it (for example join, yield, etc).
- All other actions like join, yield, etc also get executed in the same way, the user makes a system call to manage the threads and their activity.

Asynchronous I/O + Polling

Asynchronous IO is a way to allow a single threaded process to issue multiple concurrent IO requests at same time. The control flow here is

- Process makes a system call to make an IO request (like aio_read/asynchronous read) and then immediately returns the control back to the execution without waiting for result.
- At a later time, OS provides the result to the process by either
    - Polling
    - Calling signal handler
    - Placing result in a queue in the processes memory
    - Storing the result in the kernel memory until process requests for it
- In the context of the original read system call, read call blocks the execution under the data is available to read. Instead if we change the read (using fcntl to make it asynchronous or use aio_read), then this system call tells the OS to initiate the read operation and immediately return.
- The application makes a system call read/aio_read by calling the relevant library routine. This library version of the system call places the proper arguments in relevant registers (%eax) and issue some kind of trap instruction based on the value in eax register.
- Any arguments passed to the method are put on the stack. Then the actual system call takes place.
- The return path is the same, syscall -> trap -> restore all the state back for the user mode -> switch to user mode -> return to the read call.
- Once OS identifies that the requested data is available, a signal is thrown at the application or as mentioned above stored until application asks for it.
- Then the read data is actually read by the application. From the above mentioned approaches (in this problem we use custom defined polling), we later use polling to determine whether the read operation has finished. If yes, we make calls to subsequent reads until all the data is read.
- If read is used, then we check for the return value of read, if it is zero, it means data has come to an end. If < 0, there is an error and if > 0 data has been read and there could be more data, in this case we make subsequent calls.
- Similar procedure is used to make the system calls aio_error and aio_return. If aio_read is used, then we use aio_error and aio_return to poll every time, to make sure we have read all the data successfully same as above.

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

Asynchronous IO + select

Asynchronous IO is a way to allow a single threaded process to issue multiple concurrent IO requests at same time. The control flow here is

- Process makes a system call to make an IO request (like asynchronous read) and then immediately returns the control back to the execution without waiting for result.
- At a later time, OS provides the result to the process by either
    - Polling
    - Calling signal handler
    - Placing result in a queue in the processes memory
    - Storing the result in the kernel memory until process requests for it
- In the context of the original read system call, read call blocks the execution under the data is available to read. Instead if we change the read (using fcntl to make it asynchronous or use aio_read), then this system call tells the OS to initiate the read operation and immediately return.
- The application makes a system call read by calling the relevant library routine. This library version of the system call places the proper arguments in relevant registers (%eax) and issue some kind of trap instruction based on the value in eax register.
- Any arguments passed to the method are put on the stack. Then the actual system call takes place.
- The return path is the same, syscall -> trap -> restore all the state back for the user mode -> switch to user mode -> return to the read call.
- Once OS identifies that the requested data is available, a signal is thrown at the application or as mentioned above stored until application asks for it. Then the read data is actually read by the application. From the above mentioned approaches (in this problem we use select call), to determine whether the read operation has finished. If yes, we make calls to subsequent reads until all the data is read.
- Then we check for the return value of read, if it is zero, it means data has come to an end. If < 0, there is an error and if > 0 data has been read and there could be more data, in this case we make subsequent calls.
- The system call select plays a crucial role in the process and the control flow of the same is mentioned below
    - select() system call is used to wait for data in a socket. Once there is some data available on one or more of the file descriptors, there is a context switch from the user mode to the kernel mode.
    - The Network Interface Controller interrupts the processor when some data arrives. This data is received by the kernel in the interrupt routine.
    - There is a kernel thread that takes this data and wakes up the network code inside the kernel to process that packet.
    - When the packet is processed, the kernel determines the socket that was expecting for it, saves the data in the socket buffer and returns the system call back to user space.

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

Issues Known

- Even though complete data is read by server, when server closes the socket, we see an error at client 'error: [Errno 104] Connection reset by peer'.
- Sometimes (especially when client and server are on different hosts), there is a frequent error at client saying (Too many open files), because of which, server keeps waiting as its maximum number of requests have not been reached because of the missed connections from client. Try on some other host, most of the times it works!
- Sometimes even though server is up, client says connection refused. Just restart server, wait for a few seconds and start client. It should work.
- Sometimes I see this error on client, because of which some of the connections fail to reach server. Hence server keeps waiting. 'error: [Errno 110] Connection timed out'
- Sometimes it so happens that the results displayed don't get displayed until you press ENTER. This is some issue with the machines in CS labs especially when you use putty.
- Please only use the resolved IP names to test the code.
- In question1, inspite of adding a thread_join, the threads exit before completing the reading completely. Because of which, we have two version (one with join and one with infinite loop), to measure the time taken to serve all the requests.
- Question1 fails to execute when client and server are on different hosts, sorry for this, we have realized this in the last moment. So we thought it would be better to mention it here.

Comparison of 2a – 2b

Times taken 2a

Number of connections			
1	Size of Data	Same Host	Different Hosts
	10MB	92	93
	5MB	48	14
	1MB	10	3
	100KB	2	2
	10KB	1	1
	1KB	<1	<1
100	Size of Data	Same Host	Different Hosts
	10MB	9160	9096
	5MB	4552	4554
	1MB	1046	1121

Times taken 2b

Number of connections			
1	Size of Data	Same Host	Different Hosts
	10MB	5	98
	5MB	3	48
	1MB	2	10
	100KB	1	1
	10KB	<1	<1
	1KB	<1	<1
100	Size of Data	Same Host	Different Hosts
	10MB	571	9591
	5MB	296	4806
	1MB	69	1055

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

	100KB	89	300
	10KB	12	12
	1KB	8	8
1000	Size of Data	Same Host	Different Hosts
	10MB	92664	92117
	7MB	77891	69638
	5MB	45238	49755
	1MB	5186	46200
	100KB	2472	4704
	10KB	1932	2023
	1KB	1816	1884
5k	Size of Data	Same Host	Different Hosts
	10MB	54748	449461
	5MB	25913	304123
	1MB	13165	29777
	100KB	10719	25220
	10KB	10357	10310
	1KB	10277	10223
10k	Size of Data	Same Host	Different Hosts
	10MB	109621	698227
	5MB	109509	301418
	3MB	107927	250383
	1MB	105947	200531
	100KB	51386	51001

	100KB	17	1047
	10KB	11	37
	1KB	10	10
1000	Size of Data	Same Host	Different Hosts
	10MB	10503	99667
	5MB	10179	76774
	1MB	5144	46410
	100KB	1963	4041
	10KB	1908	2041
	1KB	1899	1934
5k	Size of Data	Same Host	Different Hosts
	10MB	54458	511045
	5MB	54033	297119
	1MB	52128	29112
	100KB	49896	29717
	10KB	10377	12087
	1KB	10263	10440
10k	Size of Data	Same Host	Different Hosts
	10MB	109324	506465
	5MB	109103	252987
	1MB	25953	150270
	100KB	21537	51382
	10KB	20934	51414
	1KB	20788	21150

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

	10KB	20910	20833
	1KB	20768	20657

As we see in the results below, we see that both Asynchronous IO+Polling and Asynchronous read, almost take up the same times. This can be explained basically in terms of the way both of them work. Both of them are asynchronous calls which means after we make the system call, the control returns back. Also, the way we handle poll to check for the read operations that are completed are the same. In both of them, we continuously loop (along with checking for active new connections) to see if the current requested data is received. If we see that there is more data coming in, then we make subsequent reads, else we just close that socket and mark it as done to make sure we will not go through the socket when we poll the next time.

- If read is used, then we check for the return value of read, if it is zero, it means data has come to an end. If < 0, there is an error and if > 0 data has been read and there could be more data, in this case we make subsequent calls.
- Similar procedure is used to make the system calls aio_error and aio_return. If aio_read is used, then we use aio_error and aio_return to poll every time, to make sure we have read all the data successfully same as above.

Comparing 1-2-3

As per our results mentioned below, we see that the performance of 1 (using pthreads) is the best of all, next followed by AsynchronousIO+select and the last AsynchronousIO+polling(both read/aio_read are pretty close). Both 2 and 3 are almost close though. More experiments might show that both 2 and 3 are almost the same as well.

The obvious reason for this behavior could be that, since threads make full use of the OS parallelism/multi core features, they take up lesser time when compared to 2 and 3.

Reference (Stack Overflow): On the other hand, among select and the polling methods, poll and select are basically the same speed-wise: slow because they both handle file descriptors in a linear way i.e., both of them check each of the candidate one after the other to see if there are any updates. The more descriptors you ask them to check, the slower they get.

The select() API with a "max fds" as first argument of course forces a scan over the bitmasks to find the exact file descriptors to check for, which the poll() API avoids. select() only uses (at maximum) three bits of data per file descriptor, while poll() typically uses 64 bits per file descriptor. In each syscall invoke poll() thus needs to copy a lot more over to kernel space. A small win for select().

The bitmask passed to select() has to be large enough to accomodate the highest descriptor — so whole ranges of hundreds of bits will be unset that the operating system has to loop across on every select() call just to discover that they are unset. Once select() returns, the caller has to loop over all three bitmasks to determine what events took place. In very many typical applications only one or two file descriptors will get new traffic even though we end up checking all of them.

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

Because the operating system signals you about activity by rewriting the bitmasks, they are no longer valid, so we have to rebuild the whole bitmask or keep a duplicate copy. So the poll() approach works much better in this aspect because you can keep re-using the same data structure.

So may be because of each of their advantages and disadvantages, both poll and select are doing almost the same.

1

10K, 10MB	97000
10K, 5MB	47490
10K, 1MB	11123
10K, 100KB	1300
10K, 10KB	89
10K, 1KB	15

5K, 10MB	48900
5K, 5MB	25123
5K, 1MB	6125
5K, 100KB	700
5K, 10K	42
5K, 1K	7

1K, 10MB	9400
1K, 5MB	5012
1K, 1MB	1200
1K, 100KB	213
1K, 10K	11
1K, 1K	4

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

3a

Times taken

Number of connections		
1	Size of Data	Same Hosts
	10MB	12058
	5MB	8951
	1MB	4105
	100KB	1248
	10KB	1093
	1KB	1050
100	Size of Data	Same Hosts
	10MB	26428
	5MB	14577
	1MB	1411
	100KB	777
	10KB	801
	1KB	1143
1000	Size of Data	Same Hosts
	10MB	85836
	5MB	64564
	1MB	37171
	100KB	3806
	10KB	1961
	1KB	801
5k	Size of Data	Different Hosts
	10MB (9)	470990
	5MB(5)	261608
	1MB(2)	102995
	100KB(0.5)	26746

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

	10KB(0.2)	12001
	1KB(0.2)	11396
10k	Size of Data	Different Hosts
	10MB(9)	922699
	5MB	510968
	1MB	250231
	100KB	55001
	10KB	35000
	1KB	21671

Solaris

3q	100 connections, 100KB	2062
2a	100 connections, 100KB	2238
2b	100 connections, 100KB	1853
1	We tried to run, but we got an error, no memory left to run for a long time. So could	

Variable Size

1

10K, 10MB	44000
5K 10MB	21700
1K 10MB	4200

2.a

Number of connections	Same host	Different host
10	37	429
100	382	4126
1000	3942	41027
5k	20073	205022
10k	40419	410041

CSCI 5103 – Project 1

Ravali Kandur (5084769) kandu009@umn.edu
Charandeep Parisineti(5103173) paris102@umn.edu

2.b

Number of connections	Same host	Different host
10	35	436
100	359	4456
1000	3742	50328
5k	19076	253204
10k	37860	506585

3

Number of connections	Different host
10	613
100	5178
1000	56123
5k	269670
10k	525377

How to Compile

Linux

1. gcc thread_server.c -pthread -o thread_server
2.
   a. g++ 2_a.cxx -o 2_a -lrt
   b. g++ 2_b.cxx -o 2_b -lrt
3. gcc select_server.c -o select_server

Solaris:

1. cc -g -o thread_server thread_server.c -lsocket -lrt
2.
   a. CC -g -o 2_a 2_a.cxx -lsocket -lresolv -lnsl -lrt
   b. CC -g -o 2_b 2_b.cxx -lsocket -lresolv -lnsl -lrt
3. cc -g -o select_server select_server.c –lrt -lsocket

How to Run

- For different test cases
  - Fixed/ Variable size , Same/Different host
    - Client: (each question has different ways to test using clients, it is mentioned as <2a|2b|1|3>_<fixed|variable>_size_<same|different>host.py
      - Generic python <client_script> <number_of_connections> <size> <IP> <port>
    - Server
      - 1 ./thread_server <portNo>
      - 2.a ./2_a max_clients <server_port>
      - 2.b ./2_b max_clients <server_port>
      - 3 ./select_server num_connections <port>