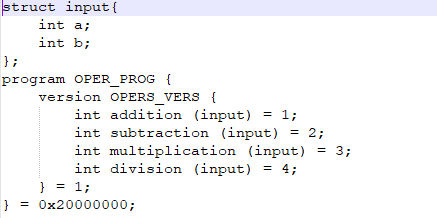
Parallel/Distributed Computing (CSEG414/CSE5414) Assignment #2

120190211 한장훈

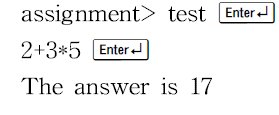
**1. RPC 프로그래밍**

SUN RPC를 통해 구현하였다. Oper.x 파일을 Rpcgen을 통해 stub파일과 service파일을 생성하였다.



**Oper.x 파일**

연산자를 2개를 넘겨줘야 하는데 rpc에서 제공하지 않으므로 struct를 만들어서 파라미터를 전달하도록 하였다. 전체적인 구조는 클라이언트에서 잡다한 연산을 하고 사칙연산때는 서버에 요청을 해서 계산하도록 하였다.



위의 계산 처럼 클라이언트 프로그램을 실행하면 assignment>가 실행되고 test를 입력하면 test파일을 읽어 거기 안에 있는 연산을 수행한다. 사칙계산 알고리즘은 infix to postfix로 식을 바꾸고 postfix 식을 계산하는 것으로 하였다. 예외 처리를 거의 안했고 따라서 숫자가 **2자리 이거나 띄어쓰기가 있을경우** 오작동 한다. 하지만 예시처럼 한자리 사칙연산에 대해서는 우선순서에 맞게 프로그램이 작성되었다.

서버측 화면.

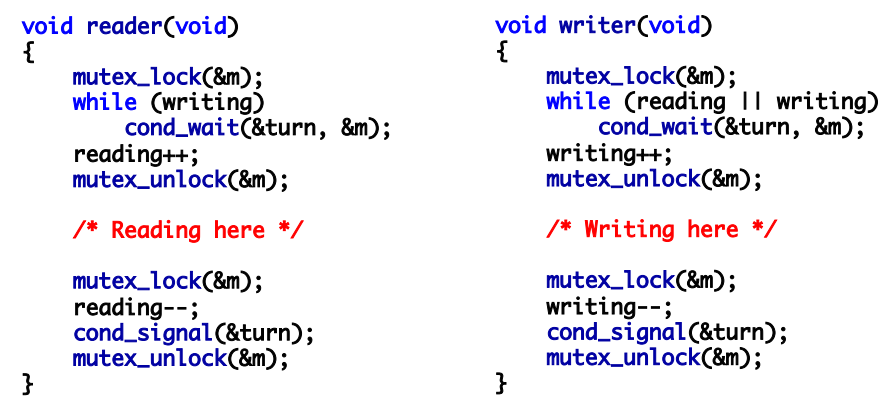


클라이언트측 화면.



**2. Multithreaded Programming**

수업때 배운 reader writer problem에 관한 프로그램이다. 주어진 pthread read-write lock 코드를 참고하여 나만의 read-write lock 을 구현하였다. 조건 대로 first reader writer lock을 mutex lock과condition variable을 상용하여 만들었다.



**First reader writer problem**

**Case 1**

|  |
| --- |
| thread\_count = 10  keys inserted = 100  # of operations = 100  percentage of searches = 0.7  percentage of inserts = 0.1 |

**using pthread read-write lock** Elapsed time = **4.300101e-04 seconds**

**using my read-write lock** Elapsed time = **4.630089e-04 seconds**

**Case 2**

|  |
| --- |
| thread\_count = 30  keys inserted = 1000  # of operations = 1000  percentage of searches = 0.7  percentage of inserts = 0.1 |

**using pthread read-write lock** Elapsed time = **5.737066e-03 seconds**

**using my read-write lock** Elapsed time = **7.205963e-03 seconds**

**Case 3**

|  |
| --- |
| thread\_count = 40  keys inserted = 10000  # of operations = 10000  percentage of searches = 0.7  percentage of inserts = 0.1 |

**using pthread read-write lock** Elapsed time = **7.526410e-01 seconds**

**using my read-write lock** Elapsed time = **8.041470e-01 seconds**

처음에 문제 크기를 100으로 하고 10배씩 늘려갔을 때 실험결과이다. 직접 구현한 lock이 pthread에서 제공하는 lock 보다 성능이 떨어지는 것을 알 수 있다. 하지만 이 경우는 search의 비율이 insert 보다 높았다.

아무래도 pthread가 제공하는 lock이 성능이 높은 것을 알 수 있다.

**Case 4**

|  |
| --- |
| thread\_count = 20  keys inserted = 100  # of operations = 100  percentage of searches = 0.6  percentage of inserts = 0.2 |

**using pthread read-write lock** Elapsed time = **5.059242e-04 seconds**

**using my read-write lock** Elapsed time = **9.851456e-04 seconds**

**Case 5**

|  |
| --- |
| thread\_count = 20  keys inserted = 100  # of operations = 100  percentage of searches = 0.2  percentage of inserts = 0.6 |

**using pthread read-write lock** Elapsed time = **7.979870e-04 seconds**

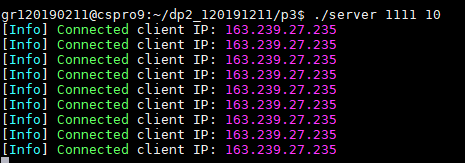
**using my read-write lock** Elapsed time = **7.059574e-04 seconds**

위에선 문제 사이즈에 대해서 성능의 변화를 봤다면 이번에는 search와 insert의 비율에 따른 비교를 해보았다. Search가 많은 경우 pthread가 제공하는 rwlock이 성능이 좋았으며 insert가 search 보다 많은 경우 직접 구현한 lock의 성능이 약간 좋았다.

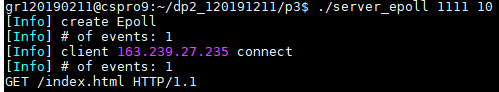
아무래도 read가 많은 경우가 ptherad에서 제공하는 lock의 성능이 좋았다. 직접 구현한 경우는 wirte 과정에서 starvation과 조건을 만족하지 못할 때 wait하는 overhead 때문에 느려졌다고 판단된다.

**3. Multithreaded Server**

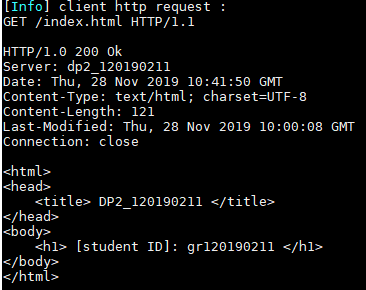
**1. Test your web server using your client program.**



**Serverside.**



**Epoll\_serverside.**



**Clientside**

첫번째 그림은 server 두번째 그림은 epollserver 세번째는 client의 실행화면이다.

클라이언트의 경우

request.txt 파일로부터 서버에 요청할 파일이름을 읽는다. 이 파일리스트를 토대로 여러 개의 쓰레드를 통해 각각 쓰레드가 connect를 요청한다. 요청은 쓰레드당 request 개수만큼 요청한다.

서버의 경우

메인쓰레드에서 통신을 받아 소켓을 연결하고 client 큐에 집어 넣게 된다. 이때 다른 쓰레드들이 큐에 있으면 쓰레드별로 큐에서 꺼내 파일이름을 받아서 파일을 읽고 클라이언트에 보내준다.

Epoll 서버의 경우

peer model with thread pool의 방식으로 실행시 포트와 쓰레드수를 명시한다. 출력시 event와 클라이언트가 요청한 query를 출력한다.

**2. Vary the number of threads in both client and server and report your observations as well as the performance of the server**

* **쓰레드당 request는 10으로 동일하다.**

**Client thread 변화에 따른 소요 시간**

|  |  |  |
| --- | --- | --- |
| **# of client threads** | **# of server threads** | **time taken for all responses** |
| **10** | **10** | **0.018s** |
| **20** | **10** | **0.032s** |
| **30** | **10** | **0.054s** |
| **40** | **10** | **0.069s** |

클라이언트 쓰레드가 증가할수록 요청이 증가하므로 총 소요 시간도 증가한다.

**server thread 변화에 따른 소요 시간**

|  |  |  |
| --- | --- | --- |
| **# of client threads** | **# of server threads** | **time taken for all responses** |
| **20** | **10** | **0.040s** |
| **20** | **20** | **0.031s** |
| **20** | **30** | **0.029s** |
| **20** | **40** | **0.031s** |

서버 쓰레드가 증가할수록 소요 시간은 감소하다가 감소하는 정도가 줄어든다. 이는 서버의 쓰레드 수가 과도하게 많아질경우 context switching이 발생하기 때문이다. 너무 많이 쓰레드수를 느리면 오히려 더 느려졌다.

**3. Vary also other parameters (e.g., size of file, interval, etc) and report your observations as well as the performance of the server**

* **쓰레드당 request는 10으로 동일하다**

**134byte 크기파일에 따른 소요 시간**

|  |  |  |
| --- | --- | --- |
| **# of client threads** | **# of server threads** | **time taken for all responses** |
| **10** | **20** | **0.013s** |
| **20** | **20** | **0.021s** |
| **30** | **20** | **0.032s** |
| **40** | **20** | **0.039s** |

**1094byte 크기파일에 따른 소요시간**

|  |  |  |
| --- | --- | --- |
| **# of client threads** | **# of server threads** | **time taken for all responses** |
| **10** | **20** | **0.019s** |
| **20** | **20** | **0.040s** |
| **30** | **20** | **0.061s** |
| **40** | **20** | **0.074s** |

파일크기가 10배정도 커져도 소요시간은 2배가 채 차이 나지 않은것으로 보아 multithread server -client 구조는 파일크기가 증가하는 대 비해 증가하는 소요시간은 상대적으로 작다. 즉 파일크기에 robust 하다고 할 수 있다.

**4. Be careful of the caching effect. Since you are making repeated requests, your files are likely to be cached. Check the performance of your server when the files are cached, thereby a little disk I/O may occur. How much is the performance benefit in this environment ? Try to make your server having substantial I/O and check the performance benefits. Report your observations.**

작은 크기의 request를 여러 번 보내는것과 큰 파일을 여러 번 실행해 보았다.

리퀘스트는 파일의 총 크기를 맞춰주기위해 첫번째의 경우를 두번째보다 쓰레드당 request를 많이 주었다

**9kbyte 크기파일에 따른 소요시간(request: 10)**

|  |  |  |
| --- | --- | --- |
| **# of client threads** | **# of server threads** | **time taken for all responses** |
| **10** | **20** | **0.086s** |
| **20** | **20** | **0.15s** |
| **30** | **20** | **0.22s** |

파일이 커져서 원래 캐시 효과를 받았던 3번의 결과가 안나오고 상당히 느려진 것을 알 수 있다. 3번의 결과는 10배가 차이나도 시간은 2배가 차이 나지 않았지만 현재는 8배정도 문제사이즈가 차이나면 속도는 4배정도 느려진는 것을 알 수있다. 이는 캐시에서 어느정도 미스가 나기 시작했기 때문이라고 할 수 있다.

**5. Try to modify the client and server codes so that they maintain the connections (persistent connections) as long as each client has something to send to the server. In this case, the server should be modified to follow peer model with thread pool and each worker uses epoll mechanism (https://en.wikipedia.org/wiki/Epoll) to monitor multiple connections. Compare the performance with original implementation by varying various parameters and report the results as well as how you have implemented client/server codes.**

1) non-epoll: Master work model with thread pool (server.cpp)

- 쓰레드 풀을 가진다. 하나의 single queue가 있어 클라이언트가 요청될 때마다 accept에서 반환되어 client의 socket을 얻고, 이를 해당 queue에 삽입한다. thread pool에서 single queue를 모니터링하고 있다가 하나의 connection에 대해 하나의 thread를 배정해주는 방식이다.

2) epoll: Peer model with thread pool using epoll (server\_epoll.cpp)

-쓰레드 풀을 가지고 있고 epoll에서 지원하는 edge 트리거 방식을 사용하기 때문에 데이터 수신과 처리되는 시점을 분리할 수 있다는 장점이 있다

클라이언트의 http\_request 단위로 큐에 들어가기 때문에 병렬성이 증가하는 경향이 있다.

또한 다수 쓰레드가 없어도 동시에 request를 처리할 수 있다. 메시지는 epollwait에서 반환되기 때문에 http\_request사용하는 것이 유리하다.

Request 10으로 동일.

**NON-epoll**

|  |  |
| --- | --- |
| **# of client threads** | **time taken for all responses** |
| **10** | **0.018s** |
| **20** | **0.032s** |
| **30** | **0.054s** |
| **40** | **0.069s** |

**epoll**

|  |  |
| --- | --- |
| **# of client threads** | **time taken for all responses** |
| **10** | **0.015s** |
| **20** | **0.029s** |
| **30** | **0.045s** |
| **40** | **0.057s** |

Epoll이 조금더 성능이 좋은 것을 볼 수 있다.