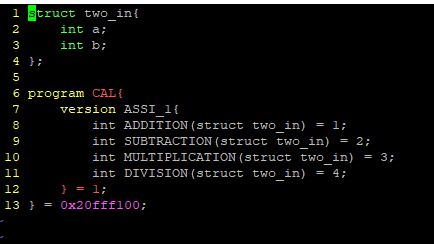
**Parallel / Distributed computing assignment #2**

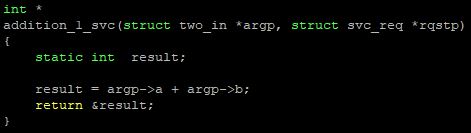
**20141602**

**황기덕**

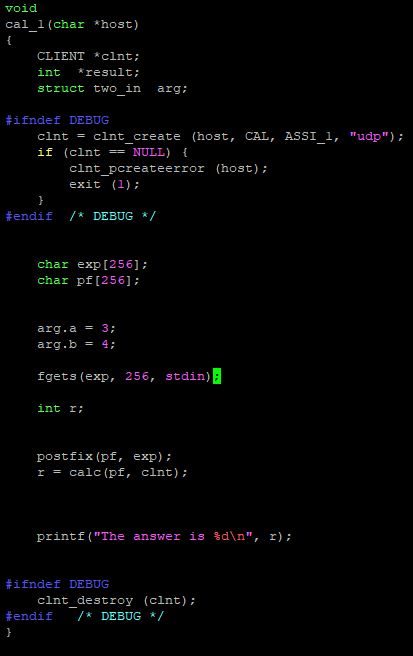
1. **RPC Programming**



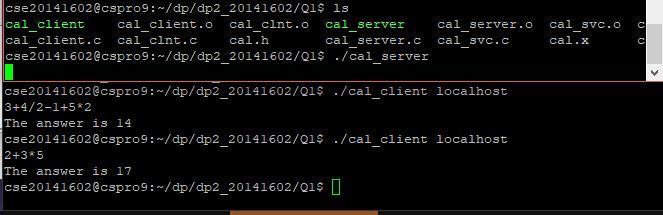
위와 같이 IDL로 contarct를 만들어 주고



server skelton은 다음과 같이 바꾸어 주었으며

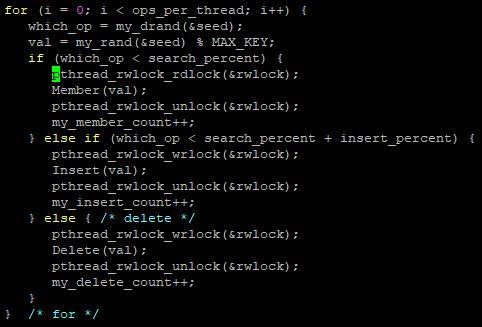


client skeleton은 다음과 같이 호출하고, 연산자식을 받으면 후위 표기식으로 바꾸어 client 내부에서 계산해 주었다.



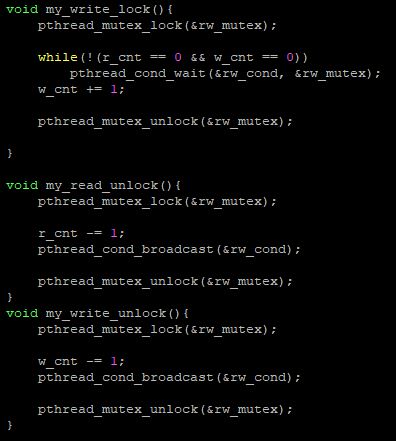
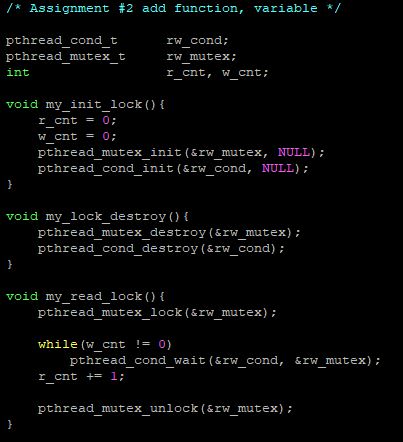
server / client 실행 모습

1. **Multithreaded Programming**



예시 코드이다. (pth\_rwlock.c) 여기에서는 rwlock을 이용하여 thread read / wirte lock을 구현하였다.

이를 아래와 같이 표현해 주었다.

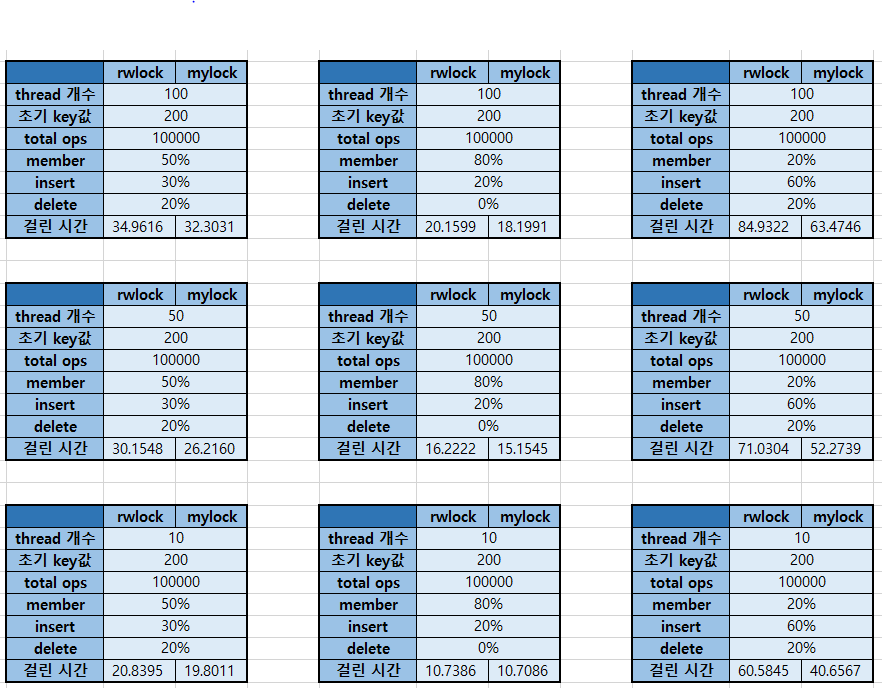


맨 처음 init 으로 초기화 해 주고, read 전용의 lock, unlock write 전용의 lock, unlock이 나뉘어져 있다. read 도중에는 read가 계속 들어올 수 있고, write일 때는 못 들어오니 read lock 부분에서 w\_cnt가 0이 아닐 경우에는 들어오게 해 주었다.

write lock에서는 write 이든, read 이든 들어온 thread가 하나라도 있다면 들어오지 못하도록 막아 주었다.

unlock에서는 각각의 cnt를 줄여 주고, broadcast로 cond var를 가지고 있는 thread를 깨워줌으로써 rwlock을 구현하였다.

다음은 thread 개수, search, insert, delete의 개수를 변환해가며 실행해 본 결과이다.



Thread 개수를 바꾸어 가며, total ops와 key 값은 고정시켜 두고 member, insert, delete의 비율을 달리해가며 비교해 보았다.

Thread의 개수가 늘어날수록 대체적으로 시간이 더 많이 걸렸다. 이는 context swtiching으로 인한 delay가 생겼기 때문에 thread 개수가 적을 때가 더 시간이 적게 걸렸음을 알 수 있다.

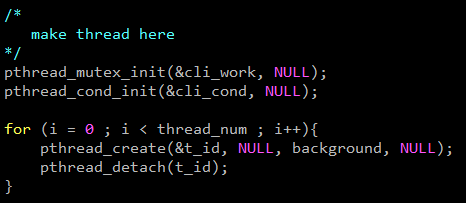
Search, 즉 member function의 비율이 높을수록 시간이 적게 걸렸다. 이는 rwlock에서는 write는 하나밖에 들어올 수 없지만 read는 바로바로 들어올 수 있기 때문에 시간 절약이 되었을 것이다. 그래서 member function이 50/ 80 일때의 시간이 훨신 작았다.

1. **Multithreaded Server**

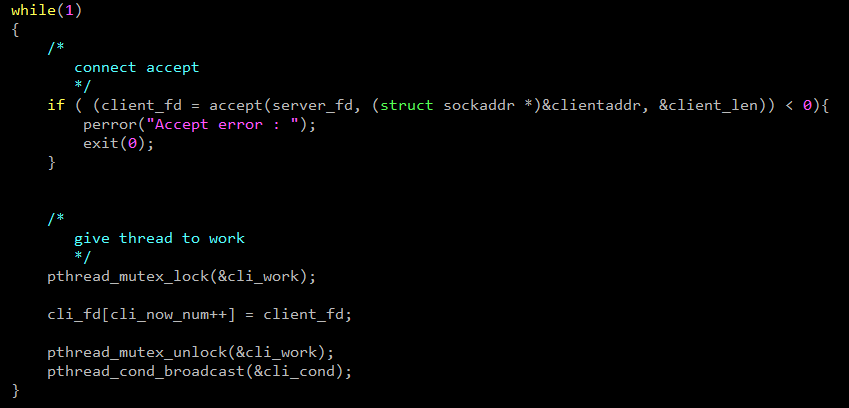
**Server.c**

실행 방법 :

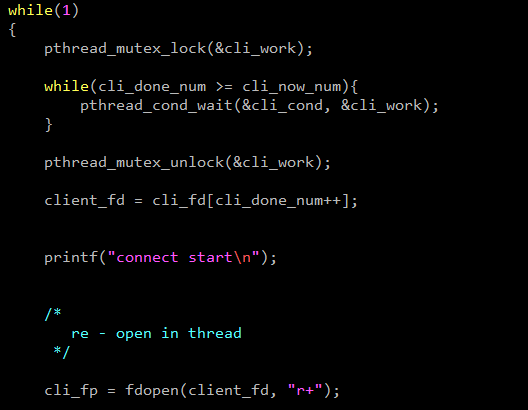
./server [portnum] [num of threads]



위와 같이 server 단에서 thread를 만들고, 각각을 detach 해 주어서 프로그램이 만약 끝난다면 알아서 자원을 반환하게끔 만들어 주었다.



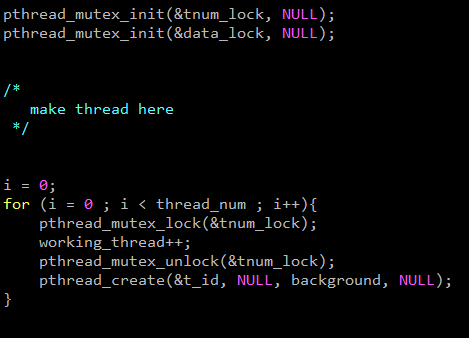
위와 같이 main thread에서만 socket 을 받고, 각 thread에게 할 일을 나누어 주었다. Cli\_fd에는 현재 들어온 client에 대한 fd를 저장해 두고, 각 thread에서 그것을 꺼내서 사용할 수 있게끔 만들어 주었다.



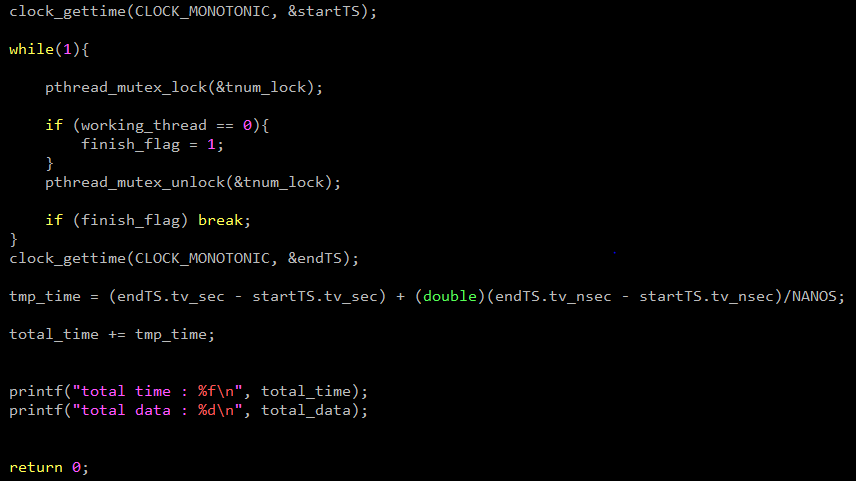
Background. Thread가 도는 함수 내부 모습이다. client에서 온 요청의 개수가 현재까지 완료한 client의 요청의 개수보다 많으면 안으로 들어가서 그 client에 대한 일을 하도록 만들어 주었다.

**Client.c**

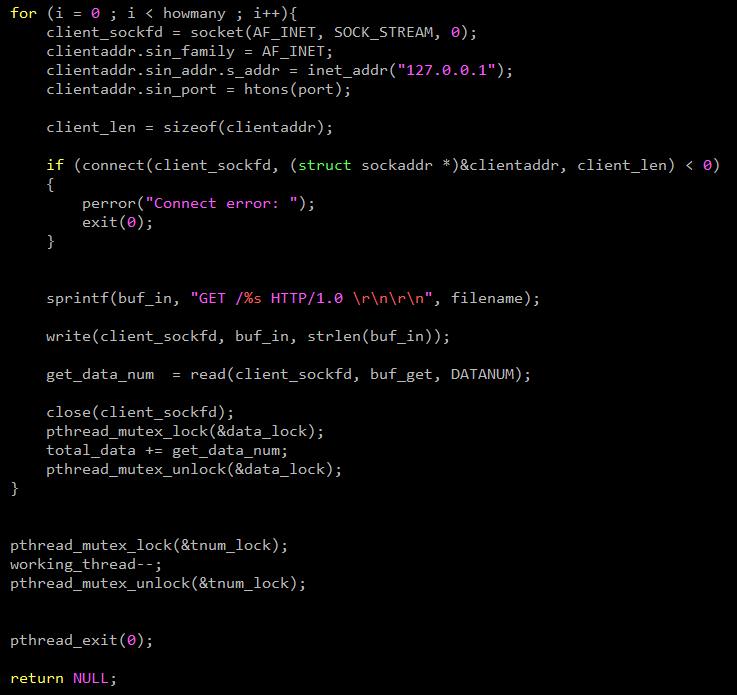
./client [portnum] [num of threads] [read filename] [loop count]



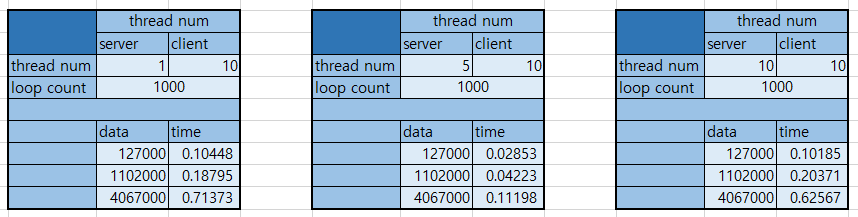
Main thread에서 thread를 만들어 주었다.



그 후, working thread가 모두 끝나면 총 시간과 받아온 데이터를 출력해 주었다.



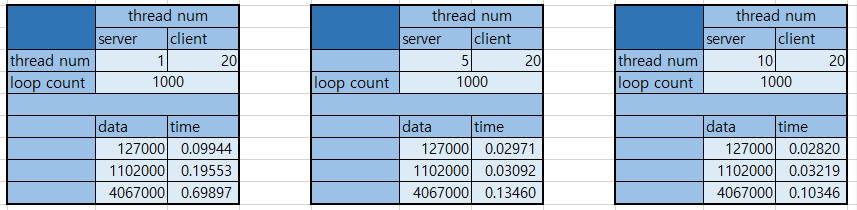
Background, thread가 각각 일하는 함수에선 connect를 각자 하고 받아온 data의 양을 추가해 주었다.



Client의 개수는 10개로 고정시켜 두고, server의 개수를 바꿔가며 한 결과이다. Client request의 값은 loop count / client 개수 이므로 각 client마다 100의 요청을 보낸다.

Data가 많을수록 time은 당연히 오래 걸렸다. 이 때, server의 thread 개수가 5일 때 가장 좋은 성능을 보였는데, 이는 thread 개수가 1에서 5로 늘어남에 따라 분산 처리가 되어서 시간이 줄어들었지만, 5에서 10으로 늘어남에 따라 context switch의 overhead가 커져서 시간이 더 이상 줄지 않고 오히려 늘어난 것으로 보인다.

하지만 server의 thread 개수가 1일 때보단 10일 때가 더 성능이 좋았다.



Client의 thread 개수를 20개로 늘인 다음 측정한 결과이다. 전체적으로 client의 thread 개수가 10일 때보다 성능이 좋아졌으며, 이는 server의 thread가 많아질수록 더 좋은 성능을 보였다. 여기에서는 server의 thread가 10일 때 가장 좋은 성능을 보였다. 이는 server thread의 context switch overhead가 io overhead에 비해 무시할만한 수준이 되었다는 것을 알 수 있다.