

POSIX Thread를 이용한 소수 판별 멀티스레드 서버 구현

소프트웨어학과 201521032 한태희
소프트웨어학과 201720733 신승현

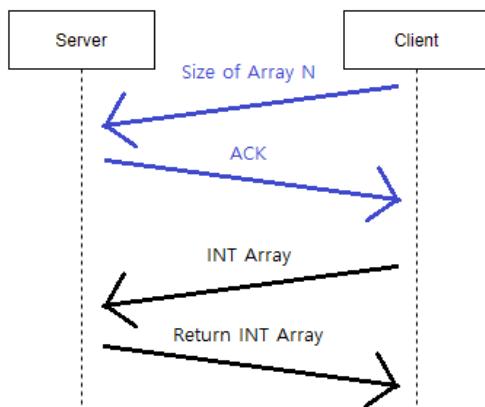
1. 개요

기존의 TCP socket을 사용한 단일 스레드 방식의 echo server를 토대로 POSIX thread를 활용하여 멀티스레드 서버를 구현한다. 서버는 client 측으로부터 랜덤으로 생성된 N개의 숫자를 수신 받으면, 각 숫자가 소수인지 판별하여 결과를 client 측으로 송신한다.

2. 구현내용

2.1. 클라이언트 측

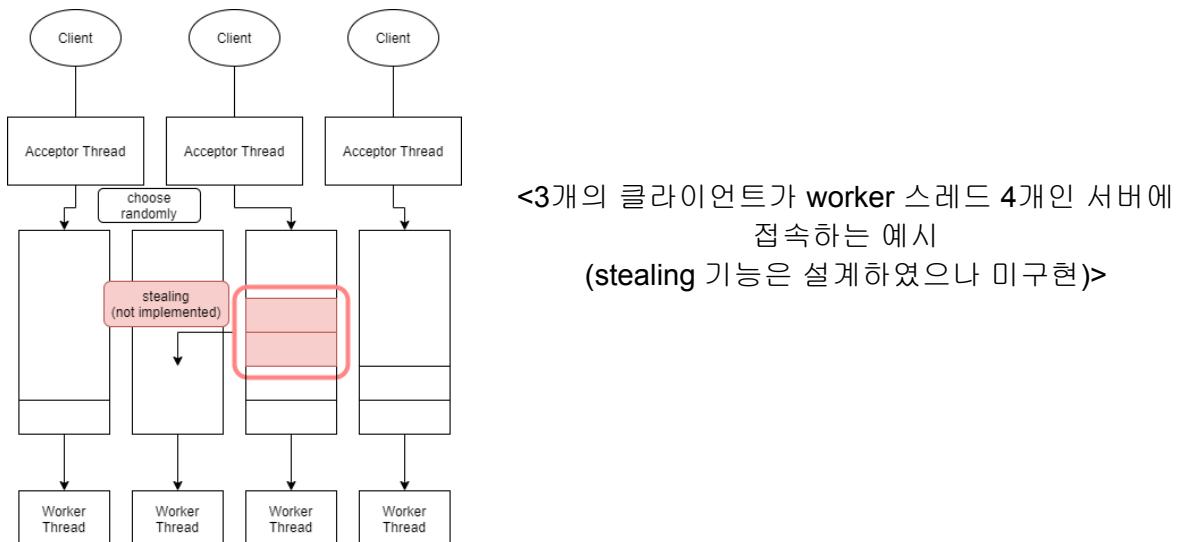
echocli.c에는 하나의 클라이언트가 N개의 정수 배열을 1번 서버로 반복 전송하는 기능이 구현되어 있다. 이때 정수 배열을 보내거나 받을 때에는 TCP 기능을 이용하여 배열 전체를 한번에 송수신한다. 클라이언트는 서버를 다음과 같은 응답을 하는 블랙박스로 생각한다.



또한 실습을 위해서 템플릿으로 제공된 driver.c의 여러개의 프로세스를 동시에 생성하는 기능을 사용했다. 이를 통해 클라이언트를 병렬로 여러개 실행해 동시에 서버에게 작업을 요청하는 환경을 구현했다. 병렬 클라이언트 요청을 실제로 발생시켜 서버가 어떤식으로 작업을 진행하는지 측정하였다.

2.2. 서버 측

서버는 동시에 많은 클라이언트의 요청을 받을 수 있도록 acceptor thread, waiting queue, worker thread의 세 가지 구조로 나뉜다.



2.2.1. Acceptor thread

Acceptor 스레드는 클라이언트가 서버에 접속하면 생성되는 스레드이다. 생성 개수 제한이 없으며, **socket accept**이 발생할 경우 생성되는 **descriptor**를 인자로 받아 독립적으로 처리하고 종료된다. 만약 **listen buffer**가 가득 찼을 경우엔 **accept** 되지 못한다.

Acceptor 스레드는 현재 서버의 **waiting queue** 중 한 곳을 랜덤으로 선택하여 클라이언트 측에서 보낸 데이터를 작업 큐에 삽입하는 역할을 한다. 클라이언트 측에서 보내는 숫자의 개수 **N**을 전달받으면 이에 대한 신호로 **ACK**를 전송하고, 이어서 배열 데이터를 전달받는다.

이 때 전달받은 배열의 크기가 **20** 이상이라면, 하나의 데이터 크기가 너무 커 특정 **worker** 스레드의 작업시간이 길어지는 상황을 방지하기 위해 **20**의 크기로 배열을 나누어 임의의 큐부터 한칸씩 인덱스를 뒤로 옮기며 (**round index**) 집어넣는다. 임의의 큐에 시작점을 정하는 이유는 큐에서 정보를 가져와 집어넣을 곳을 판단하려면 전체 큐의 정보를 한번 확인해야하는데, 그러면 전체 큐의 뮤텍스를 모두 잡아야 해서 작업 큐를 여러개로 나눈 의미가 없어지기 때문이다. 또한 스레드간 넣은 위치를 공유하면 그것도 통신으로 인한 오버헤드가 생기기 때문에, 작업 분배는 작업 스레드의 **work steal**(미구현)에만 맡기는 것이 오히려 성능 향상에 도움이 된다고 판단했기 때문이다.

임의의 큐에 배열 데이터를 삽입하고 나면 **worker** 스레드 측에서 하나의 작업을 끝내고 전달해주는 완료 신호 **flag**를 지속적으로 확인하여 나누어진(또는 하나의) 데이터에 대한 작업이 모두 끝났는지 확인한다. 모든 **flag**가 활성화되어 소수 판별 작업이 완료된 것을 확인하면 클라이언트 측에 결과 데이터를 전송하고, 접속 소켓을 종료한다.

2.2.2. Waiting queue

Waiting queue는 **acceptor thread**에서 삽입한 데이터를 저장하는 **linked list** 방식의 **queue**이다. 작업 큐는 **Worker Thread**당 하나씩 생성된다.

큐 구현에 배열이 아닌 링크드 리스트 방식을 사용했는데, **linked queue**는 배열로 구현한 형식보다 메모리 접근 효율성 측면에서 열등 할 것이다. 하지만 일반적인 배열 방식의 큐에 비해 데이터의 삽입/삭제의 시간이 짧고 이후에 **stealing** 기능이 추가될 것을 고려하여 **linked list** 방식을 사용하게 되었다.

삽입되는 하나의 데이터는 구조체를 활용하여 클라이언트 측에서 입력받은 배열 데이터와 소수 판별 결과를 담는 배열 데이터, 작업 완료 신호를 담는 **flag** 배열 데이터 등의 정보를 포함하고 있다.

큐에 데이터를 읽거나 쓰는 동안은 해당 큐에 다른 스레드가 접근 할 수 없도록 해당 큐에 대한 **mutex lock**을 활성화하고, 데이터를 삽입 한 후 **unlock** 한다.

2.2.3. Worker thread

Worker thread는 **waiting** 큐에서 원소 하나를 꺼내어 소수 판별에 대한 계산을 진행한 후, 결과를 저장하고 작업 완료 **flag**를 활성화한다. 서버 실행 시 인자를 통해 **worker** 스레드의 개수가 정해지며, 각 스레드는 **waiting** 큐 하나씩을 갖게 된다.

현재 자신의 큐가 비어있는지를 확인하기 위해 **mutex lock**을 사용하며, 만약 비어있지 않다면 원소 하나를 꺼내고 소수 판별 계산을 시작한다. 소수 판별 알고리즘은 \sqrt{N} 이하의 수들을 모두 나누어봐서 약수가 있는지 확인하는, $O(\sqrt{N})$ 의 시간 복잡도를 갖는 알고리즘을 사용하였다.

이후 서술하는 **stealing** 부분은 설계 단계까지 마쳤으나, 구현 단계에서 어려움으로 인해 실제 구현하지 못한 부분이다.

만약 자신의 큐가 비어있다면, 자신의 다음 인덱스에 해당하는 큐의 상태를 확인하고 그 크기가 **2** 이상이라면 **stealing**을 진행한다. 자신의 다음 인덱스의 큐만 확인하는 이유는

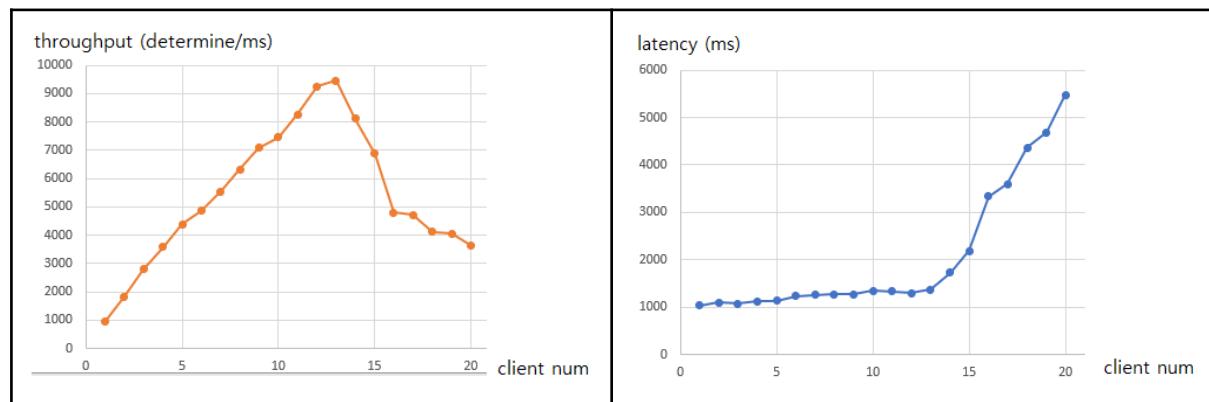
큐를 확인할 때마다 해당 큐에 대한 **mutex lock**을 걸어줘야 하고, 이 과정이 지나치게 많아진다면 여러 스레드가 진행될 경우 큰 지연시간이 생길 수 있기 때문이다. **Stealing**은 작업이 쌓인 큐의 절반만큼의 원소를 자신의 빈 큐로 가져오는 방식을 사용한다.

3. Throughput, latency 측정 및 비교

3.1. 측정의 설명

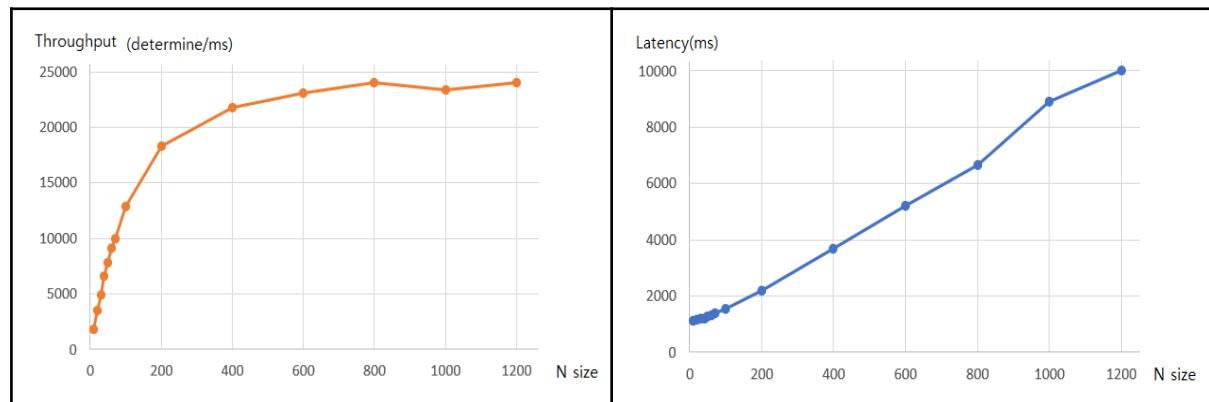
측정은 클라이언트의 수, 배열 크기, 작업 스레드 개수, 배열 분할 크기 (배열을 받았을 때 한 작업에 할당할 크기) 4가지 조건을 독립적으로 변화시켜이며 Throughput과 latency를 측정해, 가장 이상적인 입력값을 찾는 것을 목적으로 한다. Throughput의 단위는 밀리세컨드당 판별한 숫자 개수이고 (determine/ms) latency 기본값은 클라이언트 10개, 배열 크기 50, 작업 스레드 4개, 배열 분할 크기 20이다. 최종적으로 기본값으로 측정했을 때의 성능과 찾아낸 최적의 값으로 측정했을 때의 성능 차이를 비교한다.

3.2. 클라이언트 수 변화에 따른 성능 비교



클라이언트의 개수를 변화시켜 보았을 때는 Throughput이 13일 때까지 증가하다가, 그 뒤로 급격하게 떨어지는 것을 확인할 수 있었다. Latency는 13일 때까지 증가폭이 매우 낮다가 그 이후 급격하게 시간이 늘어나는 것을 볼 수 있다. 그러므로 동시에 처리하는 클라이언트를 **13개**로 제한한다면 처리량에서 가장 좋은 효율을 얻을 수 있고, 응답시간의 지연도 클라이언트의 체감을 크게 해치지 않는 수준이 될 것이다.

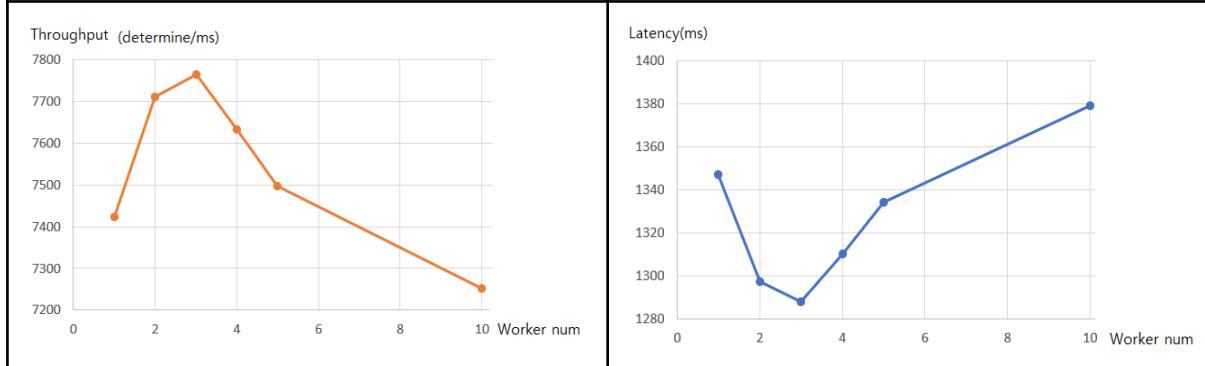
3.3. 입력 배열 크기 변화에 따른 성능 비교



입력 숫자 배열의 크기에 따라 처음부터 200까지는 급격하게 throughput이 증가했고, 약 600까지 점진적으로 증가했다. 600 이후부터는 배열 크기가 늘어났음에도 throughput의 한계점에 도달해 거의 차이를 보이지 않았다. Latency는 배열 크기에 비례하게

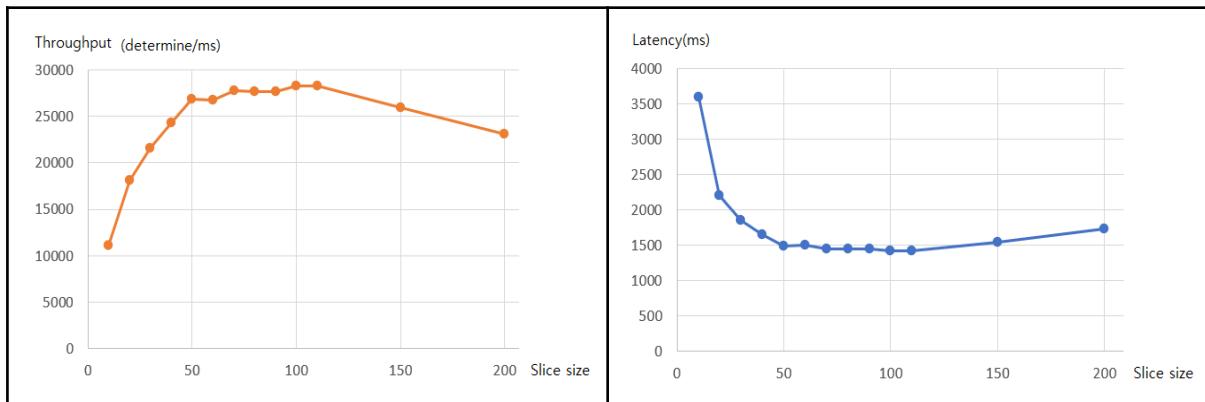
증가했으므로, 입력받는 배열 크기를 **200**으로 제한한다면 데이터 크기 증가에 따른 throughput 증가량을 높게 유지하면서 무의미한 지연시간을 갖지 않아 효율적인 서버관리가 가능할 것이다.

3.4. 작업 스레드 개수 변화에 따른 성능 비교



작업 스레드가 3개일 때 까지는 급격한 throughput 상승을 보였지만, 이후부터는 급격히 줄어들어 스레드 1개일 때 보다 급격한 성능 저하를 보인다. Latency 또한 3개의 작업 스레드를 사용할 때 최소값을 가지며 그 이후부터는 증가한다. 따라서 3개의 스레드를 사용할 때 가장 높은 효율을 보이고, 이후부터는 스레드 당 작업 효율이 떨어진다고 할 수 있다. 이론적으로는 워커 인스턴스의 개수가 늘어날수록 Throughput이 증가해야 하지만, 실제 환경에서는 Pthread로 만든 인스턴스가 CPU에서 완벽히 병렬 실행되지는 않는다고 추측해볼 수 있다.

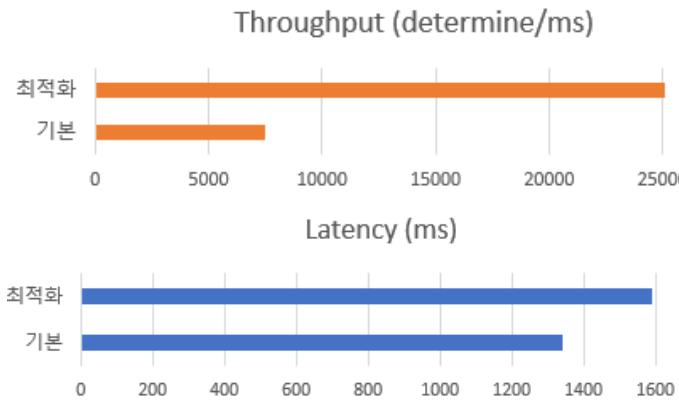
3.5. 배열 분할 크기 변화에 따른 성능 비교



앞에서 측정한 세 경우는 변경하는 변수를 제외하고는 3번 문단의 처음 설명했던 기본값으로 고정하고 측정을 진행했다. 하지만, 배열 분할은 3.2.에서 측정한 클라이언트의 입력값에 크게 영향을 받는 변수라고 볼 수 있기 때문에 3.2.에서 찾았던 최적의 값인 배열 크기가 200개인 조건에 맞추어서 테스트를 진행하였다.

측정 결과, $\frac{1}{4}$ 를 분할하여 작업에 할당하는 50개 분할 까지는 가파르게 Throughput이 증가하고 Latency가 감소하다 그 후가 대폭 둔화된다. 그리고 $\frac{1}{4}$ 분할인 100개 분할을 넘어가면, 오히려 성능이 다시 나빠지는 것을 확인할 수 있다. 분할 개수가 50에서 100 사이인 경우를 선택해야 하는데, 더 잘게 작업을 잘라야 하는 다른 이유는 없으므로 작업의 반을 분할하는 100을 선택하는 것이 합리적이라고 볼 수 있다.

3.6. 기본 임의 변수와 최적화된 변수의 성능 비교



최초 프로그램을 작성했을 때는 서버와 클라이언트에 필요한 중요한 인자를 아무런 이유 없이 임의로 넣어 프로그램을 작성한 뒤 실행하였다. (기본; 클라이언트 개수 **10개**, 입력 배열 크기 **50**, 작업 스레드 **4개**, 배열 분할 크기 **20개**) 하지만, 이 변수들의 변화에 따른 Throughput과 latency를 측정하여 완벽은 아니더라도 (각 독립 변수간 상호작용은 크게 고려하지 않았기 때문) 어느 정도 최적화된 입력값을 찾아낼 수 있었다. (최적화; 클라이언트 개수 **13개**, 입력 배열 크기 **200**, 작업 스레드 **3개**, 배열 분할 크기 **100개**) 최적화된 입력값은 밀리세컨드당 **25120개**의 정수를 소수인지 판별하여, 기본 밀리세컨드당 **7457개** 판별보다 3.3배 가량 높은 Throughput을 뽑아냈다. 그러면서도 레이턴시는 기본 1341밀리초에서 최적화 1592밀리초로 251밀리초 정도만 더 지연될 뿐이라 개인 사용자는 큰 성능 하락을 느끼지 못할 것이다.

4. 결론

단일 스레드 서버에서와 달리, 멀티스레드 서버는 기본적인 기능 이외에 여러 스레드를 동시에 실행하기 위해 사용되는 구조적/기술적인 요소가 필요하다. 본 과제에서는 acceptor 스레드, waiting 큐, worker 스레드로 서버를 구성하여 다중 클라이언트에 대한 처리를 효율적으로 나누어 가질 수 있도록 했다. 또, 추가적인 서버의 throughput 확장을 위해 작업이 끝난 worker 스레드에서 아직 작업이 남은 waiting 큐의 작업을 stealing 하는 기법을 구현하였다. 구현 도중 각 waiting queue의 상태를 확인할 때 사용되는 mutex lock/unlock이 서로 deadlock 현상을 일으키지 않도록 특정 횟수를 확인하고 나면 일정시간 대기하는 알고리즘을 사용해보았다. 하지만 mutex를 점유하는 구간이 너무 많아져 오히려 서버의 응답속도가 느려지거나 지나친 접속시간으로 강제로 서버와의 연결이 끊기게 되어 기대하는 결과를 얻을 수 없어 최종적인 작업물에는 포함하지 않았다.

설계한 멀티스레딩 서버는 워커스레드가 많이 존재할수록 다수의 클라이언트에 대한 평균 응답시간이 짧아지는 것을 목표로 했지만, 예상한 결과와 달리 스레드를 늘리는 경우 오히려 응답시간이 길어지는 결과가 있었다. 그 이유 중 하나는 스레드가 작업 자체는 나누어 갖지만, 자신의 큐가 비어있는지를 수시로 확인하기 위해 mutex lock을 사용하기 때문이다. 또한 인스턴스를 늘린다고 해서 그것이 CPU에 병렬적으로 할당되는 환경은 아니기 때문이다. 스레드에서 큐의 상태를 확인하는 동안은 critical section으로 설정했기 때문에 이 사이동안은 해당 큐에 배열 데이터가 삽입될 수 없으며, 이 경우가 모든 큐에 대해 존재하므로 생기는 serial 한 부분이 멀티스레딩 작업으로 인해 생기는 시간 단축보다 크기 때문에 이러한 결과를 얻었다고 할 수 있다. 멀티스레딩으로 인해 생기는 작업의 병렬 처리와, 그 반대로 mutual exclusion을 보장하기 위해 생기는 일련적인 작업을 적절히 분배하는 것이 궁극적으로 speed up을 얻을 수 있는 방법임을 알 수 있었다.