M1522.000800 System Programming
Fall 2018

System Programming
ProxyLab Report

Kim Bochang
2014-16757

1. <lab> Proxy Lab

Proxy Lab 의 목적은 직접 http 서버와 proxy 서버를 만들어 보면서, 소켓 라이브러리와 시스템 수준 I/O 함수, 그리고 Posix thread 라이브러리의 사용법을 익히면서 전반적인 시스템 함수들을 활용하는 법을 배운다. 또한 http 프로토콜에서의 request와 response를 처리하는 방식을 익히고, 통신 과정중 발생할 수 있는 여러가지 에러들을 핸들링 하면서 실제 서버와 클라이언트의 의사소통이 어떻게 이루어지는지를 배운다.

2. Implementation

<*http.c*>

내가 구현한 http 서버의 전체적인 구조는 다음과 같다. 포트번호를 받아 프로그램이 실행되면, 포트번호를 이용해 listen file descriptor를 연다음, accept()함수로 클라이언트가 연결하기를 기다린다. 만약 클라이언트가 연결하면, accept()는 connection file descriptor 를 리턴하게 되고, 이제 이 descriptor를 이용해 통신을 진행하게 된다. doit()함수를 이용한다.

doit() : Http 서버는 처음으로 request line을 받고, request line의 형식이 알맞은지를 확인한다. GET방식으로 들어온 요청인지 확인한 다음, url을 파싱하여 파일 이름을 알아낸뒤, 파일 상태를 stat()함수를 통해 알아내게 된다. 만약 파일이 존재하지 않거나, 파일을 여는데 실패했거나, 파일이 디렉토리거나, 서버가 파일에 대한 권한을 가지고 있지 않다면 해당하는 오류코드를 클라이언트에게 전송하게 된다. 요청받은 파일이 적절한 파일이라면, 서버는 파일 전송을 시작한다. serve_static 함수를 이용한다.

Serve_static() : 서버는 클라이언트에게 적당한 http response line과 response header를 전송하고, 해당하는 파일을 열고, mmap()을 통해 파일을 가상 주소공간에 매핑한뒤, rio_writen함수를 이용해 클라이언트에 파일을 보내게 된다. 그 후, 파일 디스크립터를 닫고, 매핑한 파일을 munmap()함수로 다시 해제한 뒤, 루틴을 종료한다.

루틴이 종료되면 서버는 connection file descriptor를 닫아 통신이 끝났음을 클라이언트에 알려준다. 그리고 새로운 입력을 기다린다.

<*proxy.c, cache.c*>

내가 구현한 프록시 서버의 전체적인 구조는 다음과 같다.

메인로직에서는 캐시와 프록시서버에서 필요한 세마포어등을 초기화하고, 오류 처리를 위한 시그널 핸들러를 지정해준뒤 클라이언트에서 요청을 받기 위한 listen file descriptor를 생성한다. 그 뒤, accept()함수를 통해 client로부터의 요청을 기다린 뒤, 요청이 오면 새로운 쓰레드를 생성한 뒤, 그쪽에 클라이언트의 connection file descriptor를 넘겨주어 쓰레드가

클라이언트로부터의 요청을 처리하게 한다. 그리고 메인 로직은 다시 새로운 연결요청을 기다린다.
쓰레드 루틴에서는 connection file descriptor를 받아서 클라이언트의 요청을 처리하게 된다.
클라이언트의 request line을 받아서, 만약 request line에 들어있는 url에 대한 데이터가 캐시되어있다면 캐시에서 데이터를 꺼내서 클라이언트에 보내준다.
그렇지 않다면, 실제 서버와 새로운 connection을 만든 다음, 클라이언트가 요청한 정보를 서버로부터 받아와 다시 클라이언트로 전송해주고, 받아온 데이터가 최대 캐시블록 크기보다 작다면 콘텐츠를 캐시에 새로 넣어주고, 클라이언트와의 connection file descriptor를 닫고 쓰레드 루틴을 종료한다.
캐시는 linked list를 사용하고, replacement policy로 FIFO (first in, first out)을 사용한다.
또한 캐시의 최대 크기와, 최대 캐시 가능한 개수를 지정해서 메모리 사용량과 캐시의 성능이 특정수치 이하로 떨어지지 않게 하였다.

HTTP 서버를 구현할때는 concurreny가 필요할만큼 복잡한 기능을 가지는 server가 아니었으므로, sequential하게 구현하게 되었다. 서버를 구현할때, stat()함수를 통해 파일의 정보를 확인하게 되었는데, 이때 stat()함수를 통해 나온 sbuf 구조체의 mode 멤버를 이용하여 사용자가 서버가 접근하면 안되는 파일 (핵심 시스템 파일등)에 접근을 시도할 경우, 에러메시지를 보내도록 하였다.

프록시 서버를 구현할때, 동시에 여러 요청이 올 수 있으므로 concurrency를 가지는 server를 만들어야 했다. concurrency를 구현하는 방법으로는 process, i/o multiplexing, thread를 이용한 세가지 방법이 있었는데, process를 사용하는 방법은 가장 깔끔하지만 캐시를 다른 프로세스간에 공유하지 못하기 때문에 제외했고, i/o multiplexing은 대부분의 서버에선 멀티코어를 사용하는데, 이를 온전히 활용하지 못하므로 제외했다. 따라서 쓰레드를 이용해서 동시성 서버를 구현하게 되었다.
쓰레드를 이용한 구현에서는 여러가지 신경써줘야할것이 많았다. 일단 쓰레드가 죽었을때 자동으로 reaping 되도록 pthread_detach()함수를 사용해서 쓰레드가 종료될때 스스로 자원을 회수하게 하였다. 또한 클라이언트의 요청을 처리하던 도중에 connection이 끊기거나 해서 broken pipe가 발생하면, SIGPIPE 시그널이 도착해서 프로세스 전체를 죽여버리는 경우가 생겼기 때문에, sigpipe 시그널을 받은 쓰레드만 죽도록 sigpipe_handler() 함수를 만들게 되었다. 또한 예측하지 못한 방식으로 연결이 끊어지거나 하는 에러가 발생했을때, csapp.c에 있는 robust_io 함수들의 wrapper가 unix_error()함수를 호출하고, 이 함수 내부에 exit() 함수가 있어 프로세스 전체가 죽는일이 발생했기 때문에, 나만의 wrapper 함수를 만들어서 문제가 발생한 쓰레드만 죽이도록 하였다.
쓰레드를 이용함으로써 공유된 변수에 접근할때 race가 발생할수도 있었는데,

쓰레드를 생성할때 내부에서 while loop이 계속해서 돌고있기 때문에 인자로 변수를 전달할때 다음 루프에서 값이 변할 수 있는 main thread의 지역변수를 전달하는게 아니라, malloc()으로 힙에 새로운 공간을 할당하고 인자를 전달한후 쓰레드 루틴에서 free()하는 방식을 사용하였고, 모든 쓰레드가 접근가능한 공용 자원인 캐시를 접근할때는 세마포어를 이용해 한 쓰레드에 의해 캐시가 변경되고 있을때 다른 쓰레드에 의해 캐시가 참조되는일이 없도록 하였다. 서버가 많은 클라이언트에게 빨리 정보를 제공해야 하기 때문에 캐시에 무언가를 쓰는 작업보다는 읽어오는작업의 우선순위가 높다고 생각했고, 따라서 이를 해결하기 위해 reader-writer problem에서 reader-preference 모델을 사용했다. 이를 캐시에 접근할때마다 적용하여 캐시에서 손상된 데이터를 꺼내가는 일을 막았다.
캐시를 구현할때는 linked-list방식을, replacement policy로는 FIFO (first-in-first-out)방식을 사용했다. linked-list방식이 비록 cache에서 무언가를 검색할때는 리스트의 크기에 비례하는 시간이 걸리지만, cache에서 원소를 삭제하거나 추가할때는 상수시간이 걸린다는 장점이 있었고, 구현한 프록시 서버 정책상 캐시 크기가 크지 않은데다가, 캐시에 무언가를 추가하는 작업이 매우 빈번하게 일어났기때문에 이 방식을 선택했다.
또한 proxy서버 메인 루틴에서 malloc()과 mmap() 함수를 이용해 메모리 공간을 할당받고, 할당받은 공간에 컨텐츠를 받은 뒤 데이터들의 포인터를 캐시에 넣어주고, free()와 unmap()은 캐시에서 사라질때 사용하는 방식을 택해서, 캐시에서 검색을 하거나 삽입, 삭제를 할때 실제 데이터를 통째로 옮길 필요가 없도록 하였다.
그 외에, 통신을 할때 connection refused by server 문제가 발생해서 서버와 연결된 file descriptor를 얻어오지 못했을때, 다시 연결을 시도해서 정상적으로 데이터를 받아올 수 있게 하였다.

3. Conclusion

랩을 진행하면서, 실제 서버와 클라이언트 사이에 오가는 http request와 response문장의 구조, 문장을 구성하고 파싱하는법을 알게되었고, 실제 통신이 이루어지는 과정에 대해 생각해 볼 수 있었다. 또한 concurrency를 가지는 서버를 만들면서, concurrency를 가지는 프로그램을 만들때는 고려해야 하는것이 정말 많다는 것도 알 수 있었다. 그리고 통신 과정에서 정말 많은 에러가 발생하고, 그 에러를 다루는것이 무척 어렵다는것도 알게 되었다.
이번랩이 에러 핸들링을 하기에 가장 어려웠던 랩중 하나라 생각한다. Http 리퀘스트를 보내고 처리하는 과정에서 \r\n을 빼먹는다던가 하는 자잘한 문제들이 발생하거나, chunked format을 처리할때도 \r\n을 빼먹거나, 파싱할때 문제가 생기는등의 어려움이 있었다.하지만 가장 어려웠던 문제는 통신에 관련되서 생기는 문제들이었다. 통신하다 broken pipe에러가 났을때, 프로세스로 SIGPIPE 시그널이 도착하게 되는데, 이때의 기본동작이 프로세스가 죽어버리는것이기 때문에 아무런 에러메시지도 없이 프로세스가 죽어버려서 프록시 서버가

다운되는 경우가 생기는 경우도 있었다. 또한 특별한 이유가 없는데도 서버쪽에서 connection을 refuse해서 proxy에서 자동으로 reconnecting을 하게 만들어 줘야 한다거나, 똑같은 명령어로 통신을 하는데 connection reseted by peer/server 문제가 발생한다거나 하는 당혹스러운 경우가 너무 많았다. TCP 프로토콜에서 handshake를 하지못하면 발생하는 에러라던데, 똑같은 상황에서 어떨땐 발생하고 어떨땐 발생 안하니 에러 원인을 찾기가 너무 힘들었다. 이러한 점들 때문에 랩을 진행하는데에 있어 애로사항이 많았다.

랩을 진행하면서 아쉬운 점도 있었다. 내 코드에서는 에러가 나면 비정상적인 루틴(sighandler나 사용자 래퍼함수를 거치는경우)으로 쓰레드를 종료하는 경우가 발생하는데, 만약 mmap()되거나 malloc()된 데이터가 free()되거나, munmap()되거나, 캐시에 저장되기 전에 에러가 발생하면 이들을 회수하지 못해서 계속해서 메모리 누수가 생기게 된다. 이를 해결하기 위해서는 서버에서 만들 수 있는 최대 쓰레드 개수를 정해놓고, 각 쓰레드마다 자료형을 담을 수 있는 전역변수 혹은 static 배열을 두고, 오류가 발생했을때 일일히 쓰레드 전용의 변수들을 체크해서 만약 free()나 munmap()되지 않았다면 일일히 free()나 munmap()을 해주는 과정이 필요한데, 이러한 것을 구현하려면 루틴이 너무 복잡해져서 구현하지 못했다. 각 쓰레드가 자신의 쓰레드 id를 갖는 배열에 접근하려면 쓰레드 id → 배열 index에 매핑되는 일정한 규칙이 필요한데, Thread id를 담는 pthread_t변수가 file descriptor처럼 0,1,2...와 같은 규칙으로 할당되는 것이 아니라, linux나 window에서는 임의의 integer형을 가지고, mac os 등에서는 pthread 구조체의 포인터형이라서 이 모든걸 고려하면서 구현하기에는 너무 루틴이 복잡해져서 구현하지 못했다. 계속해서 서버가 돌아갈때, 에러가 발생하면 메모리 누수가 발생할 수도 있다는점에서 아쉬운점이 남았다.