Assignment #4 Mini Binder

김재학

(소프트웨어학과, 202220757)

R0. 코드 실행

1. 실행 결과

```
[13002.422043] minibinder: device closed
[13006.462724] minibinder: device opened
[13006.462730] minibinder: ATTACH -> handle=1
[13006.462756] minibinder: mmap 65536 bytes OK
[13006.467108] SEND: handle=1 code=3
[13006.467429] SEND: handle=10 code=1
[13006.467437] RECV -> handle=10 code=1 offset=0
[13006.467762] SEND: handle=10 code=1
```

```
[server] registered 'caculate service' (token="caculate_service", handle=10)
[server] waiting for requests...
[server] ADD: 7 + 3 = 10
```

```
kimjaehak@ubuntu24:~/Downloads/minibinder/minibinder$ sudo ./.build/client
[client] bound to 'caculate service' (token="caculate_service", handle=10)
[client] TRSNACTION_ADD request: a=7, b=3
[client] TRSNACTION_ADD reply: result=10
```

그림 1. 코드 실행 결과

ReadMe를 참고하여 코드를 실행하였을 때 위와 같은 결과를 얻을 수 있었다. 위의 그림은 서버 실행 후 클라이언트를 실행하였을 때의 커널 로그와 각 실행 화면이다. 그림을 통해서 클라이언트가 실행될 때 mini binder로 접근한 로그와 접근 후 보낸 code, 서버가 받은 내역 등을 확인할 수 있었다. 이에 대한 분석은 아래 **R1. 코드 분석**에서 진행한다.

추가적으로 클라이언트에서 **그림 2**와 같이 코드를 추가하여 연속된 트랜잭션을 하였을 때의 결과는 **그림 3**과 같다.

```
do_calc(remote, TRSNACTION_ADD, a, b, "TRSNACTION_ADD");
do_calc(remote, TRSNACTION_SUB, a, b, "TRSNACTION_SUB");
do_calc(remote, TRSNACTION_MUL, a, b, "TRSNACTION_MUL");
```

그림 2. 여러 번의 트랜잭션

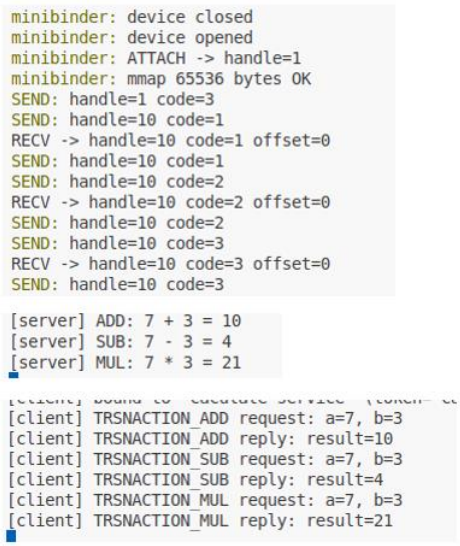

그림 3. 여러 번의 트랜잭션 결과

의도된 결과를 볼 수 있었고, 각 트랜잭션마다 해당되는 code의 넘버가 변한 것을 확인할 수 있었다.

2. 실행 중 문제점과 원인 추측

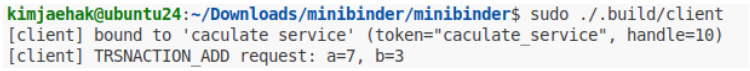

그림 4. 응답 없는 트랜잭션의 클라이언트 로그

코드를 처음 실행할 때 **그림 4**와 같이 클라이언트가 응답을 받지 못하였다. 서버를 먼저 실행한 상황임에도 응답을 받지 못한 블로킹이 발생한 것처럼 보였다. 클라이언트를 반복해서 재실행 한 후에 정상적인 결과를 볼 수 있었으며, 이후의 실행에서도 간헐적으로 응답을 받지 못하였다.

처음엔 이 문제에 대한 원인으로 컴퓨터의 성능 문제를 의심하였다. 현재 사용중인 컴퓨터에서 VM 실행이 원활하지 못한 상황이기 때문이다. 컴퓨터의 성능이 좋지 못한 이유로 VM이 부드럽게 실행되지 못하는 상황에서 서버와 클라이언트를 동시에 실행하였을 때 키보드 입력에 문제가 생기는 수준으로 코드가 실행되고 있었다. 이 문제에 더해서 서버가 초기화 되는 중이거나 하는 등의 제대로 된 서버의 응답이 없어서 레이스 컨디션 문제로 인해 클라이언트가 블로킹이 된 것처럼 보인다고 생각하였다. 하지만 다른 이유가 있을 수도 있다고 생각하고 있으며, 정확한 원인을 파악하진 못했다.

그림 2과 같이 여러 번 트랜잭션을 연속으로 진행할 때는 응답 없는 블로킹이 발생하는 것처럼 보이는 현상이 자주 관찰되었다. 이는 커널에서 트랜잭션을 하나밖에 관리하지 못하기 때문인 이유로 인한 부수적 효과로 보인다. 이에 대해선 아래의 R1의 2번 문항에서 간단히 설명하였다.

R1. 코드 분석

1. 호출 스택 분석

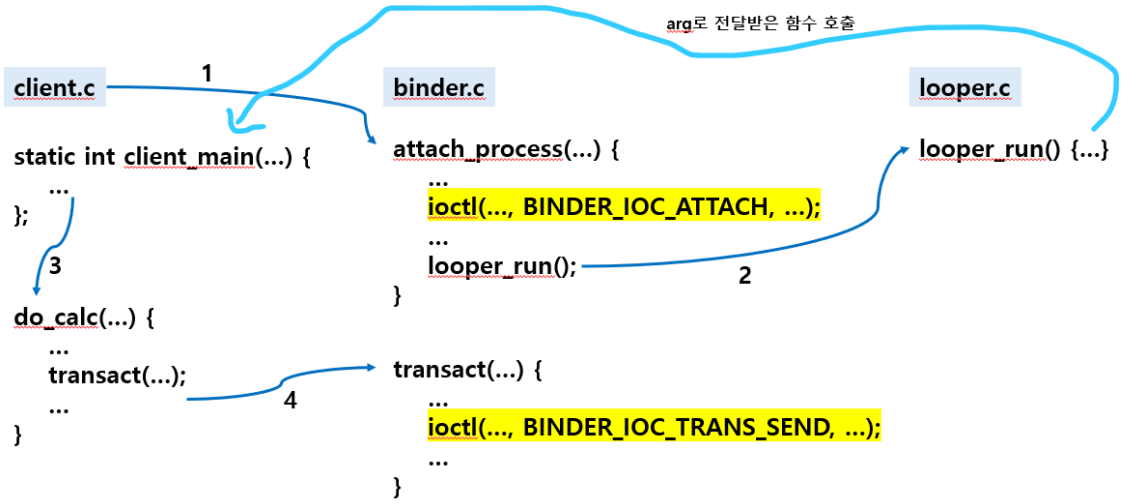

그림 5. 클라이언트의 함수 호출 순서

클라이언트는 첫 실행부터 RPC 요청까지 총 두 번의 커널 접근을 진행한다. 첫번째는 attach_process 함수 호출을 통해서 ioctl을 호출하여 커널로부터 핸들을 받아오게 된다. 이렇게 bind를 진행하게 되고, mmap으로 공유 버퍼를 매핑하게 되며 클라이언트는 RPC를 진행할 준비를 마친다. 이후 looper에서 클라이언트의 client_main 코드를 실행하게 되고 이 코드에선 do_calc 함수를 통해 트랜잭션을 시행한다. 이때 원하는 code를 ioctl로 요청하게 된다.

서버 또한 마찬가지로 첫 실행에서 attach_process 함수 호출을 통해 커널로부터 핸들을 받아와 bind를 하고 공유 버퍼를 매핑한다. 이후 binder 스레드를 만들어서 루프를 돌며 요청이 도착할 때까지 대기한다. 요청이 오게 되면 커널은 어떤 code인지 공유 버퍼 offset을 전달하고 서버는 버퍼의 offset위치에서 요청 데이터를 읽는다. 이후 code를 바탕으로 요청 parcel을 이용해 연산 로직을 처리하고 reply parcel에 결과를 저장한다. 그 후 ioctl을 다시 실행하여 커널로 SEND를 보내게 된다.

커널은 서버와 클라이언트가 ioctl로 attatch를 하였을 때 해당 프로세스를 IPC의 구성으로 등록하고 핸들을 반환한다. 이후 mmap 호출에 대응하여 공유 버퍼를 유저에게 매핑한다. 클라이언트가 SEND를 호출하였을 때 parcel을 커널로 가져와서 offset과 함께 대기중인 서버에 요청을 전달한다. 이후 서버가 다시 응답을 하면 클라이언트에 반환한다.

2. IPC 관점에서의 메커니즘 분석

기본적으로 ioctl을 통해서 유저와 커널 사이의 제어에 대한 내용이 오가게 된다. Attach와 SEND, RECV가 이에 해당한다. 이후 데이터에 대해선 유저의 mmap을 통한 공유 버퍼를 이용해서 진행되며 이는 유저의 서로 다른 프로세스의 데이터 전달을 위해 사용된다. 이들은 ioctl을 통한 copy_from_user와 copy_to_user의 호출에 의해 데이터를 주고받으며 진행된다.

요청과 응답에 대해서 유저에서 커널로 복사되는 부분은 copy기반이다. 다시 말해 요청과 응답의 parcel, binder_ipc는 커널로 copy_from_user와 copy_to_user로 복사되는 것이다.

커널은 요청과 응답을 공유버퍼에 저장해두게 된다. 이에 대해서 서버와 클라이언트는 copy없이 공유버퍼에서 그대로 읽어오게 된다. 이때는 copy가 일어나지 않기 때문에 zero-copy이다. 다시 말해 클라이언트와 서버 간의 데이터 공유는 공유버퍼로 전달되기 때문에 추가적인 복사가 일어나지 않는다.

커널은 요청에 대해 어떤 서버로 보내야 할 지를 결정한다. 이에 대해선 올바른 라우팅이 필요하다. 서버는 사람이 식별가능한 토큰으로 stub을 등록하는데, 서버가 서비스를 등록하였을 때 커널은 라우팅 테이블에 토큰과 핸들을 저장하고 클라이언트가 bind를 진행할 때 핸들을 반환한다. 이후 토큰이 아닌 핸들로 RPC 요청이 진행되게 된다. 이후 동기화를 위해서 mutex를 이용해 대기 상태를 관리한다. 서버는 요청이 오기까지 대기하게 되며 클라이언트는 서버의 응답을 기다리는 동안 대기하게 된다. 이 과정에서 waitqueue를 이용하며 이와 같은 동기화 방식으로 유저의 프로세스간 통신이 진행된다. 이 과정에서 성능을 이유로 zero-copy를 제공하게 되는데, 커널이 만든 공유메모리를 유저의 프로세스가 참조하면서 요청과 응답 데이터에 대해 유저의 서로 다른 프로세스는 추가적인 복사 없이 직접 데이터를 사용할 수 있다. 이 전반적인 내용에 대해 커널은 요청과 응답에 대해서 관리를 하며 현재 구현에서는 하나의 트랜잭션에 대해서 대응하는 것으로 보인다.

R2. Discussion

1. Binder는 빠른 IPC 성능을 보인다

Binder는 다른 IPC 시스템들과 비교해 비교적 더 빠른 성능을 보인다고 생각한다. 이유는 현재 과제에 사용된 코드를 보더라도 공유메모리를 이용해 zero-copy를 최대한으로 구현하여 IPC를 진행하였는데, 이는 실제 데이터의 copy를 줄여서 시간에 대해 단순한 message-passing 보다 성능적으로 우월함을 갖고 있다고 생각되기 때문이다. 하지만 Message-passing 방식에서도 공유 메모리를 혼합해서 사용한다면 높은 성능을 기대할

수 있을 것이라 생각되고, 경우에 따라 socket, pipe 등을 적절히 사용하는 사례가 있는 것으로 보아 상황마다 적절한 IPC 방식이 존재한다고 생각된다.

2. 4바이트 정렬의 이유

현대 컴퓨터는 대부분 4바이트의 워드를 가지고 있다. 그렇기에 항상 4바이트의 입력을 보장해 준다면 cpu는 추가적인 연산 없이 단순한 한번의 메모리 접근으로 데이터를 가져올 수 있게 된다. 즉, 추가적인 메모리 접근에 대한 연산이 적어지는 것이다.

3. 양방향 RPC에 대하여

과제의 코드는 양방향 RPC를 지원하지 못한다. 이는 커널에서 한 번에 트랜잭션에 대해서만 관리하기 때문이다. 이 상태에서는 여러 RPC를 처리할 수 없으며 당연히 양방향 RPC를 처리할 수 없다. 따라서 현재 과제의 코드에서는 단방향으로 RPC를 요청하는 단순한 서버와 클라이언트의 구조를 갖는다.

양방향 RPC를 위해서는 설계의 개선이 필요하다. 서버는 하나의 스레드만 가져서는 안 된다. 진행중인 요청 외에 다른 요청에 대해서도 반응할 수 있어야 되기 때문에 여러 개여 스레드를 생성하여 관리할 수 있어야 한다. 클라이언트 또한 요청만 보내는 것이 아닌 요청을 받을 수도 있어야 하므로 별도의 수신 스레드를 생성하여 관리할 수 있어야 한다.

또한 요청에 대해서 인터럽트로 관리되어야 한다. 실행중인 스레드가 단순히 block된다면 요청에 대한 처리를 할 수 없기 때문이다. 다른 runnable 한 스레드로 대응할 수 있게 하거나 block 상태에서 runnable 상태로의 전이를 관리할 수 있어야 한다.

4. Zero-copy의 구간에 대해서

현재 과제의 코드는 요청과 응답에 대해 커널이 중간에서 관리를 하는 형태이다. 요청에 대해 커널은 공유 메모리로 요청의 데이터를 작성하고, 응답에 대해 커널이 공유메모리로 데이터를 작성하고 있다. 이 경우에는 당연하게도 copy가 발생하고 있다고 말할 수 있다. 이 코드에서 zero-copy로 볼 수 있는 부분은 공유메모리로부터 유저가 데이터를 읽어올 때이다. 이때는 커널로부터 공유메모리의 어떤 부분을 참조하면 될지 전달받고 직접 공유메모리를 참조하여 데이터를 사용하기 때문에 copy가 일어나지 않는다.

이와 같은 구조에서 단순한 copy 기반 IPC보다 장점을 갖는 부분은 명확하다. 먼저 위의 설명과 같이 copy과정을 최소화하고 공유메모리를 직접 유저가 참조하기 때문에 성능적인 부분의 이점이 있다. 또한 커널 내부로 copy한 데이터를 공유메모리로 작성하여 유저가 읽는 형태이기 때문에 요청을 처리하는 도중 메모리가 변경되거나 다른 유저가 메모리를 접근하여 직접 변경하는 등의 문제, race condition 문제들을 방지할 수 있다.

하지만 커널이 이를 관리해야 하는 부담을 생각한다면 IPC에 있어서 커널의 자원 소비를 고려해서 설계해야 한다는 단점이 있다고 생각한다.

5. 제로카피의 철학과 모순되는 점에 대해서

위의 4번 토론에도 명시하였듯이 요청에 대한 부분을 커널이 직접 전달받아 공유메모리를 관리하는 방식은 안정성에서 이점이 있다. 이 구조에서 공유메모리에 직접적인 쓰기를 하는 것은 커널이고 유저는 읽기만 하는 것으로 보인다. 이는 유저가 공유메모리에 직접 접근하지 못한다는 것을 나타내는데, 어느 요청에 대해서 처리중인 데이터에 대해 유저가 직접적으로 접근하지 못하게 하면서 데이터 변조에 대응할 수 있다고 생각한다. 이 부분은 성능적인 부분보다 안정성에 대한 부분에 초점을 맞춘 것으로 zero-copy를 통한 성능향상과 안정성 모두를 고려한 설계의 결과라고 생각한다.

정리하자면, 이러한 zero-copy의 철학과 모순된 구조가 성능 향상 측면에서 의미가 사라질 수 있지만, 데이터를 안전하게 관리하는 방법을 생각하며 zero-copy의 원리를 일부 적용하면서 성능적인 부분을 고려하였다고 판단되므로 실용적인 성능향상과 안정성을 노린 뛰어난 설계라고 생각된다. 따라서 성능향상 측면에서도 여전히 유의미하다고 생각한다.

참고자료에 대한 설명

해당 보고서 작성에 있어 고영배 교수님의 시스템프로그래밍 강의자료를 많이 참고하였으며, 과제 출제 자료의 예제 코드에 대한 설명을 참고하였습니다.