UNIX시스템 프로그래밍

설계 과제물 최종보고서

파일 공유 시스템

2024년 12월 4일

7팀

2020212020 유강인

2022150029 조진영

2022150028 윤태훈

**목차**

1. 문제 정의 및 요구사항 분석
   1. 문제 정의 및 설계 과제 목표
   2. 요구사항 분석
2. 시스템 설계
   1. 소프트웨어 설계도
   2. 쓰레드 설계
   3. 공유 자원과 동기화 기법
3. 시스템 구현 및 데모
   1. 소스코드 주요 부분 설명
   2. 데모 시나리오
   3. 프로그램 실행 결과 캡처 및 설명
4. 현실적 제한 요소 및 성능 측정
   1. 현실적 제한 요소의 반영 내용
   2. IPC 기법의 성능 분석
5. 팀워크
   1. 팀원의 역할과 작업 내용
6. 결론 및 총평
7. 부록: 프로그램 소스

**1. 문제정의 및 요구사항 분석**

**1.1. 문제 정의 및 설계 과제 목표**

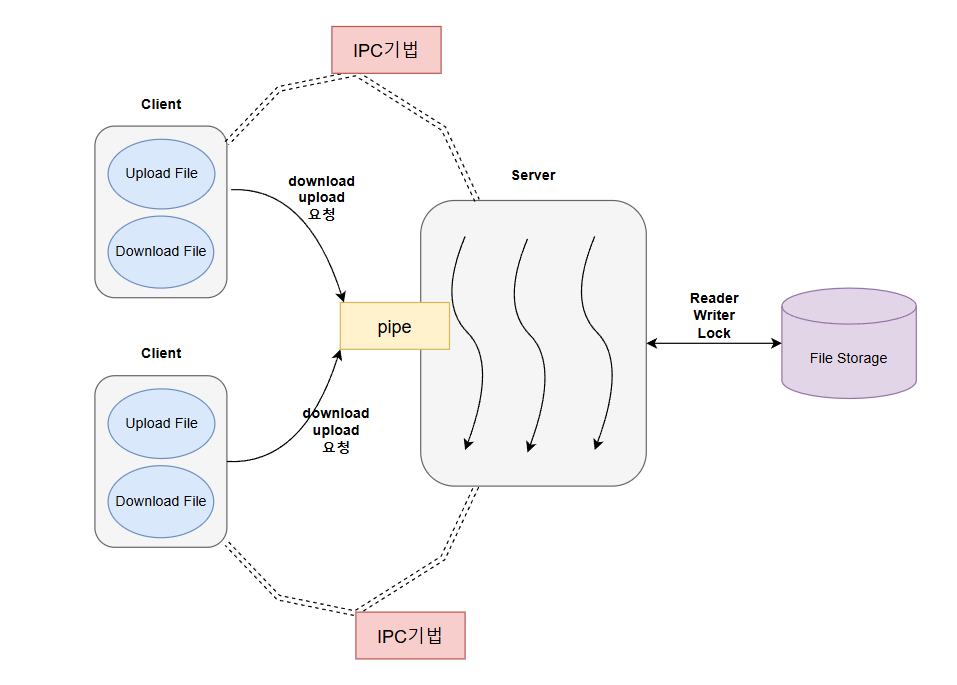
* 문제 정의: 클라이언트와 서버 사이에 다양한 IPC 기법을 활용하여 파일 업로드, 다운로드를 할 수 있는 파일 공유 서비스를 구현한다.
* 설계 과제 목표
  + IPC(Inter-Process Communication)를 활용하여 다중 클라이언트와 서버 간 파일 공유 서비스 구현한다
  + 이때 프로세스가 최소 3개 이상의 스레드를 활용하여 병렬 처리를 수행하고 스레드에서 발생할 수 있는 경쟁 상태를 막기 위해 동기화 기법을 적용한다.
  + 서로 다른 IPC 기법 (pipe, shared memory, message passing)의 성능 차이를 분석한다.

**1-2. 요구사항 분석**

* 클라이언트는 파일을 Upload 또는 Download할 수 있다.
* 서버는 클라이언트의 파일 Upload/Download 요청을 최대 10개 동시처리 가능해야한다.
* 서버는 파일의 데이터 무결성을 클라이언트에게 제공해야한다.
* 성능 분석은 clock\_gettime() 함수로 실행 시간을 측정한다.

**2. 시스템 설계**

**2-1. 소프트웨어 설계도**



<소프트웨어 설계도 이미지>

클라이언트는 서버에게 파일 upload/download 요청을 pipe로 보내고 서버는 클라이언트의 요청 하나당 하나의 스레드를 생성하여 upload/download 요청을 수행한다. 이때 스레드와 클라이언트간에 독립적인 IPC기법(pipe, shared memory, message passing)이 할당된다. 스레드는 파일 스토리지의 개별 파일에 read & writer lock을 선점하여 클라이언트의 요청을 수행한다.

**2-2. 쓰레드 설계**

클라이언트는 서버에게 Upload 또는 Download의 연산을 요청할 수 있다. 이때 서버는 클라이언트의 요청을 받아서 쓰레드를 생성하고 한 쓰레드가 한 클라이언트의 요청을 처리한다.

* 스레드 생성 함수: pthread\_create(&thread, NULL, client\_handler, IPC);
  + &thread: 생성된 스레드ID를 저장할 변수이다.
  + NULL: 쓰레드 속성을 지정하는 변수로 기본속성을 사용했다.
  + client\_hander: 스레드가 실행할 함수이다.
  + IPC: 스레드가 실행할 함수에 전달할 매개변수이다.
    - 스레드와 클라이언트별로 소통할 독립적인 IPC기법(pipe, shared memory, massage passing)의 키 또는 경로를 넘기고 파일이름과 명령(upload 또는 download)를 넘긴다. [총 3개를 문자열로 넘기게된다. 공백으로 구분됨].
* 스레드 해제 함수: pthread\_detach(thread);
  + 스레드의 종료를 기다리지 않고 리소스를 해제하는 함수로써 스레드가 종료되면 리소스를 자동으로 회수할 수 있다.
  + thread: 해제할 스레드 ID
* 스레드 종료함수: pthread\_exit(NULL);

**2-3. 공유 자원과 동기화 기법**

* **공통**

공유 자원은 서버에서 접근하는 파일 시스템(storage 디렉터리)에 있는 개별적인 파일들이다. 서버는 파일 시스템에 클라이언트의 파일을 Upload, Download를 하는데 Upload 중에 Download를 하거나 같은 파일을 동시 접근하여 Upload하는 경우와 같은 상황이 발생하면 데이터 무결성이 무너지게 된다. 따라서 각각의 파일은 공유 자원이기에 동기화기법을 적용해야 한다. 이때 동기화기법은 Reader and Writer Lock을 이용한다. 파일 시스템에서 일반적으로 Upload보단 Download 요청이 더 많고 Download 연산끼리 데이터 무결성을 해치지 않기 때문이다.

따라서 file\_locks라는 파일 락을 관리하는 구조체 배열을 만들어 공유 자원을 사용한다. 이때 파일 별로 mutex 락을 이용하여 파일 락 배열의 접근을 제어한다. 따라서 배열에 새로운 파일 락을 추가하거나 검색할 때, 상호 배제를 보장한다.

-rwlock 초기화함수: pthread\_rwlock\_init(&file\_locks[file\_lock\_count].lock, NULL);

* 첫번째 매개변수는 pthread\_rwlock\_t 변수로 read write lock의 동기화를 위해 필요한 정보가 저장된 구조체이다. 우리 구현에서는 파일별로 락을 획득해야하므로 배열로 이루어진 모습을 볼 수있다.
* 두번째 매개변수는 락 속성 지정 매개변수인데 기본값 사용을 위해 NULL을 주었다.

-rwlock 제거함수: pthread\_rwlock\_destroy(&file\_locks[i].lock);

-read lock 획득 함수: pthread\_rwlock\_rdlock(&file\_lock->lock);

-write lock 획득 함수: pthread\_rwlock\_wrlock(&file\_lock->lock);

-lock 해제 함수: pthread\_rwlock\_unlock(&file\_lock->lock);

* **shared memory**

추가적으로 shared memory에서는 세마포어를 이용하여 생산자 소비자 동기화를 설계한다. shared memory기법의 공유메모리는 프로세스가 먼저 write해야 read할 수 있기 때문이다. 따라서 공유메모리를 공유자원으로 두고 프로세스의 동작 순서를 제어하기 위해 데이터 쓰기/읽기 작업 앞뒤에 세마포어 기법을 사용한다.

- semget(key, SEM\_COUNT, 0666)

* 첫번째 매개변수 key는 유니크한 값으로 key가 같으면 같은 세마포어 집합을 사용한다.
* 두번째 매개변수는 세마포어 집합의 세마포어 개수이다.
* 세번째는 플래그 설정으로 접근권한/세마포어집합 생성옵션 등을 줄 수 있다.

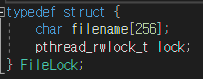
-세마포어 값 변경 함수: semop(semid, &sb, 1)

* 첫번째 매개변수는 값을 변경시킬 세마포어집합의 ID이다.
* 두번째 매개변수는 sembuf라는 구조체타입인데 해당 구조체를 통해 세마포어 집합중에서 세마포어를 지정, 값 변경 연산, 플래그를 지정할 수 있다.
* 세번째 매개변수는 두번째 세마포어집합의 크기를 뜻하는 것으로 만약 1이면 단일 세마포어에 대해 수행되고 1보다 크면 여러 세마포어에 대해 순차적으로 수행된다.

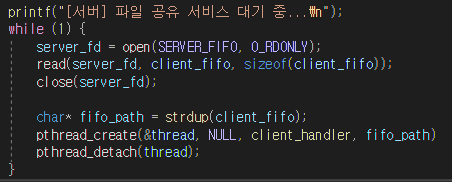
**3. 시스템 구현 및 데모**

**3-1. 주요 소스코드 설명(공유자원과 동기화 기법 위주)**

* **주요 코드**
  + **서버**



스토리지 내의 파일별 락을 관리하는 구조체이고 파일 별로 read & write lock을 획득할 수 있다. 이를 통해 공유 변수를 관리한다.





서버는 main문에서 무한 루프를 돌면서 SERVER\_FIFO 파이프를 통해 클라이언트의 요청을 수신한다. 이때 파이프에는 3가지 정보가 들어있는 배열이 들어오는데 각각

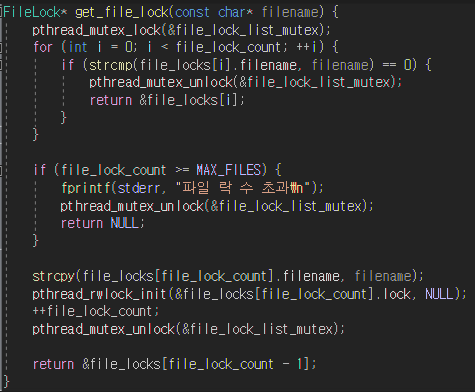
(1) command(upload/download),

(2) 대상파일이름,

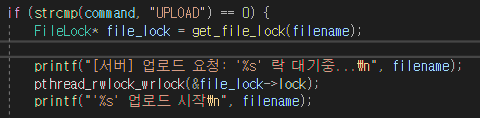
(3) IPC자원의 키 또는 파이프 경로이다.

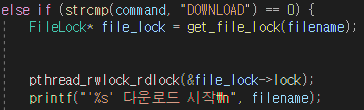
요청이 수신되면 스레드를 생성하여 client\_handler 함수로 요청을 처리한다. client\_handler에 주어진 3가지 정보가 매개변수로 주어진다.

client\_handler는 download/upload를 구분하여 처리한다.



다음은 파일 락을 확인하는 함수이다. 특정 파일에 대한 락 정보를 검색하고 없다면 생성한다. 이때 락 구조체 배열의 접근은 pthread\_mutex\_lock으로 보호한다. 그렇지않으면 하나의 파일에 대해 중복 락이 생성될 수 있기 때문이. 즉, 파일 락 구조체는 공유변수이다.





get\_file\_lock으로 파일 락 구조체 배열에서 파일 락을 얻은 뒤에 명령이 upload인지 download인지에 따라 read lock 또는 write lock을 얻게 된다. 이를 통해 같은 파일에 대하여 download 연산끼리는 병렬적으로 처리될 수 있다. 읽기+쓰기 또는 쓰기 + 쓰기연산이 같은 파일에 대해 발생하는 경우 순차적으로 처리된다.





명령이 완료되었으면 락을 해제하고 스레드는 종료된다.

* + **클라이언트**







사용하는 IPC기법별로 스레드와 소통할 IPC자원을 생성하고 서버파이프를 통해 서버에게 명령/파일이름/IPC자원을 보내 UPLOAD/DOWNLOAD를 요청한다.

이후 DOWNLOAD/UPLOAD를 구분하여 파일을 IPC를 통해 보내거나 받는 단순한 구조이다. 이때 IPC 기법에 따라 사용되는 함수들이 다르다.

* **shared memory 주요 코드**

앞서 설명한 주요 코드는 세 가지 IPC기법에 대해서 공통적인 주요코드이다. 이후 세 가지 기법의 차이점은 IPC자원에 쓰기 읽기 함수가 다를 뿐이다. 그런데 shared memory는 추가적으로 세마포어를 통해 동기화를 해야 하므로 따로 설명한다.

Pipe와 mssage passing은 파이프나 메시지 큐에 데이터가 없을 때 읽거나 데이터가 꽉 찾을 때 쓰기 작업을 하면 blocking되는 특성을 가지지만 shared memory는 non-blocking으로 동작하므로 동기화 기법이 필요하다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

semop함수를 통해 semwait/semsignal 함수를 만들어 세마포어 값을 증가 및 감소시키며 읽기와 쓰기 작업을 동기화시킨다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

위에 보이는 캡처는 fread로 파일의 내용을 읽고 shared\_data->data에 저장한다. 이러한 연산 위아래에 세마포어 연산을 사용하여 쓰기 동작전에는 읽기 동작을 못하게 한다. 반대로 읽기동작은 쓰기동작후에 이루어지도록 동기화시킨다.

**3-2. 데모 시나리오**

서버는 실행 시 무한 루프로 항상 요청을 대기한다. 서버는 클라이언트 요청마다 스레드를 생성하고 하나의 스레드는 하나의 클라이언트에 대한 요청을 처리한다. 따라서 한 쌍의 스레드와 클라이언트는 독립적인 IPC기법을 할당 받아 사용한다. 스레드는 1개부터 최대 10개까지 만들어질 수 있다. 따라서 한 순간에 10개의 클라이언트 요청을 처리할 수 있다. 여기서는 프로젝트 조건의 최소 쓰레드인 3개의 클라이언트 요청을 가정한다.

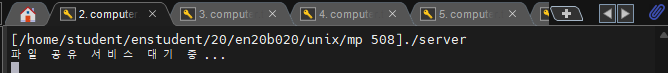
**1) write lock (upload)**

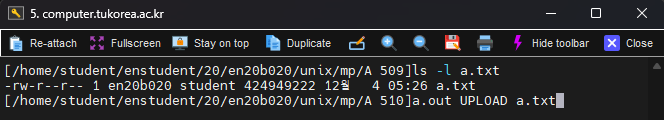
* 서버 터미널 1개와 클라이언트 터미널 3개를 생성한다.
* 서버를 실행시키고 3개의 클라이언트가 거의 동시에 서버에게 upload요청을 한다.
* 서버는 요청된 순서대로 스레드를 생성하여 upload를 수행하는데 스레드는 file별로 write lcok을 얻어 upload를 수행한다.
* 따라서 같은 파일에 대해서는 순차적으로 처리되는 모습을 확인할 수 있고 다른 파일에 대해서는 병렬적으로 처리되는 모습을 확인할 수 있다.
* 스토리지에 파일이 정상적으로 upload됨을 확인한다.

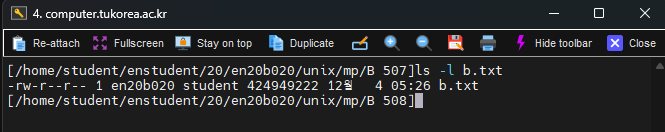
**2) read lock (download)**

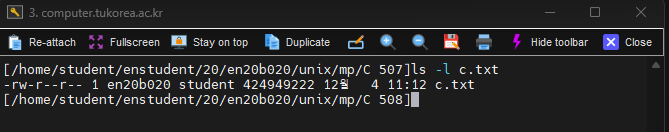
* 서버가 실행된 상태에서 3개의 클라이언트가 거의 동시에 서버에게 download file요청을 한다.
* Upload연산과 다르게 같은 파일이든 다른 파일이든 상관없이 세 개의 요청이 병렬적으로 처리되어 훨씬 빨리 끝나는 모습을 확인할 수 있다.
* Download를 수행한 현재 작업 디렉터리에 파일이 정상적으로 download 된 모습을 확인한다.

**3-3. 프로그램 시행 결과 캡쳐 및 설명**



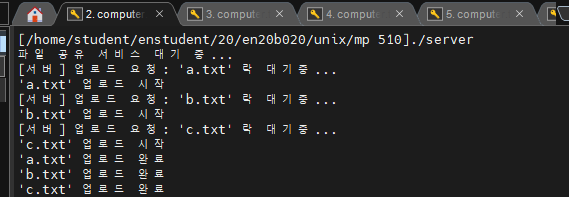


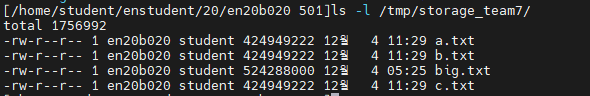




데모를 위해 서버를 실행시킬 1개의 터미널과 클라이언트를 위한 3개의 터미널을 생성한다. 첫 터미널은 server을 실행시킬 터미널이고 나머지 터미널은 client이다. 클라이언트에서 서로 다른 txt파일을 UPLOAD하는 연산을 작성한다. 이후 3개의 터미널에 순차적으로 엔터를 입력하여 동시에 파일을 업로드 하는 상황을 연출한다.

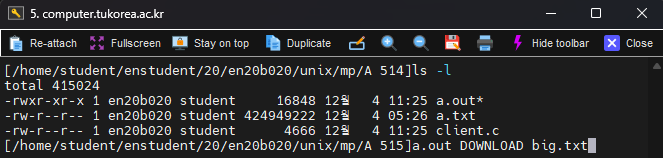
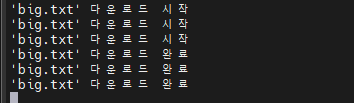
[위 캡처를 보면 server가 실행되는 작업디렉터리는 unix/mp이고 client가 실행되는 터미널의 작업 디렉터리는 unix/mp/A, B, C로 모두 다르다.]

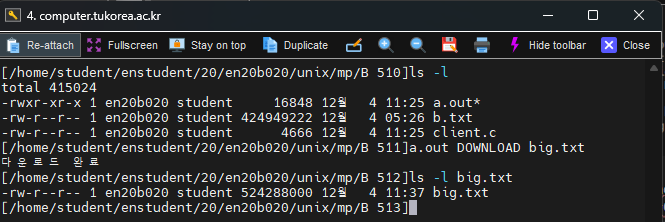


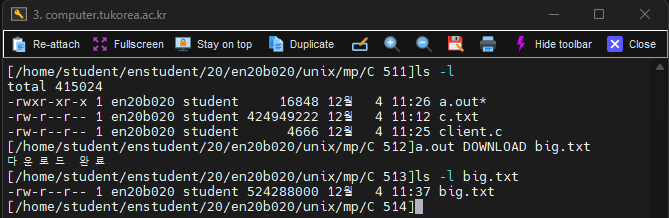


a, b, c순으로 upload 요청을 시행하였고 각 요청에 따라 서버에서는 스레드를 생성하여 파일을 upload한다. 이때 스레드들은 write lock을 선점하지만 read & write lock은 각 파일 별로 구분되기 때문에 병렬적으로 처리되는 것을 확인할 수 있다.

이제 storage에 존재하는 big.txt를 각 A/B/C에서 동시에 다운로드 요청을 시행한다.

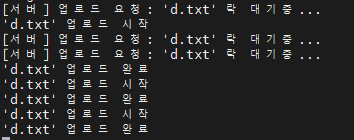






서버의 세 개의 스레드가 스토리지에 있는 big.txt에 대하여 read lock을 얻어서 병렬적으로 수행됨을 확인할 수 있다. 작업디렉터리 A, B, C를 보면 잘 다운받아진 것을 확인할 수 있다

이제 서로 다른 A, B, C에서 스토리지에 같은 파일(d.txt)을 업로드 해볼 것이다.



앞서 보인 두 개의 데모와는 다르게 동작됨을 확인할 수 있다. 처음 요청된 클라이언트에 대한 스레드가 d.txt에 대해 write lock을 선점하여 upload를 수행하면 나머지 스레드들은 wrtie lock을 얻을 수 없어 blocking된다. 이후 write lock이 해제되면 나머지 두 스레드가 순차적으로 write lock을 얻어 실행됨을 확인할 수 있다.

이렇게 upload연산(write lock)과 download연산(read lock)에서 언제 병렬적으로 처리되고 언제 순차적으로 처리되는 지 확인해보았다. 데모 캡처사진에는 같은 파일에 대해 upload연산과 download연산을 수행한 장면이 없는데 페이지가 너무 길어져 생략하였다. 같은 파일에 대해 upload와 download가 동시에 요청되면 먼저 요청된 순서부터 순차적으로 처리된다.

read & writer lock에서는 read lock간에서만 병렬적으로 처리될 수 있기 때문이다.

**4. 시스템 구현 및 데모**

**4-1. 현실적 제한 요소 반영 내용**

우리 팀에서 구현한 설계는 파일 공유 시스템으로 한 스토리지(디렉터리)에 파일을 올리고 받을 수 있는 시스템이었다. 클라이언트는 서버에게 upload/download 연산을 요청할 수 있고 서버는 받은 요청을 순차적으로 처리하게 된다. 이때 서버는 클라이언트 요청 하나에 스레드를 하나 생성하여 스레드와 클라이언트가 IPC기법으로 communication하게 된다. 따라서 특정시간에 서버에 요청되어 반환되지 않은 클라이언트수가 스레드 수가 된다.

스레드는 클라이언트에게 파일을 받아서 스토리지에 올리거나 스토리지에 있는 파일을 클라이언트에게 보내게 된다. 따라서 스토리지가 공유 변수가 되었다. 수업시간에 배운 내용으로 대부분의 소프트웨어는 write연산보다 read연산이 많다는 것을 적용하여 read & write lock을 동기화 기법으로 채택하였다. 이렇게 구현을 해보았는데 스토리지에 A파일을 upload하고 B파일을 download하는 경우에는 데이터 무결성이 깨지지 않음에도 스토리지가 공유 변수가 되어 순차 처리되는 현상을 확인하였다. 이것을 해결하기위해 스토리지 내에 있는 개별 파일들을 각각의 공유 변수로 두어 개별 파일들에게 read & write lock을 적용시켜 성능을 향상시켰다.

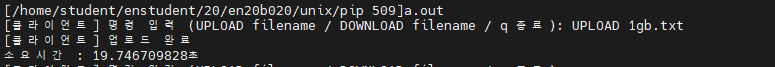
성능을 높일 수 있는 방안으로는 서버와 클라이언트가 파일을 주고받을 때 한 번에 쓰고 읽는 파일 사이즈를 크게 하는 것이었다. 읽고 쓰는 파일 사이즈가 커진다면 전체적으로 읽기 연산, 쓰기 연산, 시스템 콜 등의 횟수가 적어지기 때문이다. 하지만 우리 팀은 읽고 쓰는 파일사이즈의 최대 크기를 1024로 채택하였다. 여러 클라이언트로 서버에 upload download를 요청하여 read & write lock에 따른 순차 처리, 병렬 처리를 쉽게 보기 위해서는 파일 upload download 연산 속도를 의도적으로 늦추는게 좋다고 판단했기 때문이다.

**4-2. IPC 기법의 성능 분석**

현재 시스템에서 세 개의 IPC 기법은 서버의 한 스레드와 한 클라이언트 쌍이 파일을 업로드/다운로드할 때 이용된다. 따라서 클라이언트가 서버에 업로드/다운로드 요청을 하고나서 start 시간을 측정하고 “업로드완료”/”다운로드완료”가 출력된 후 end 시간을 측정하여 성능을 분석하였다.

이때 IPC기법 간 명확한 차이를 보기 위해서 적당히 큰 1GB의 파일을 업로드 해보았다.









현재 작업 디렉터리를 확인해보면 위에서부터 pipe, shared memory, massage passing기법을 활용한 것이다. 모두 1gb크기의 txt를 업로드하는 연산이었고 소요시간은 각각 19초대17초대/22초대가 나왔다.

shared memory가 가장 빠른 성능을 보여준 것은 프로세스간 데이터를 메모리 공간에 직접 공유하여, 커널의 도움이 없이도 빠른 읽기/쓰기가 가능하기 때문이다. 하지만 shared memory는 생산자/소비자 동기화기법을 적용해야해서 다른 두 기법보다 구현의 어려움이 존재한다는 단점이 있다.

pipe도 괜찮은 성능을 보여주었다. 사실 파이프가 가장 느리고 System V계열의 IPC기법이 더 빠를 것이라고 예상하였는데 그러지 않았다. 그 이유는 파이프는 파일 디스크립터를 사용하지만 실제로는 커널의 메모리 버퍼에서 읽기 쓰기 연산이 이용되기 때문에 나쁘지 않은 성능을 보여주게 되었다. 실제 파일에 읽기/쓰기 연산을 하는 것이 아니라 1440만 cpu cycle이 걸리는 것은 아니었다.

message passing기법이 가장 좋지 않은 성능을 보여주었다. message passing도 커널을 통해 메시지를 전달하지만 pipe보다 느린 이유가 존재했다. pipe는 단순한 버퍼구조이지만 message passing은 메시지 큐 구조로 좀 더 구조화된 데이터 교환 방식이었다. 따라서 우리 시스템과 같이 단순 데이터 전송방식에서는 파이프가 좋은 성능을 내게 된 것이었다. 만약 구조화된 데이터 교환방식이 필요한 시스템에서는 message passing을 고려해보면 되겠다.

**5. 팀워크**

팀원 모두 어느 기법을 사용해도 상관없다는 의견을 통해 사다리 타기로 IPC 기법을 정했다. 사다리 타기를 통해 유강인이 pipe 조진영이 message passing 윤태훈이 shared memory기법을 맡았다. 팀 프로젝트 중, 조진영과 윤태훈 팀원이 설계 구현에 어려움을 겪어 유강인이 message passing, shared memory 구현에 도움을 주었다.

**팀역할**

유강인이 최종보고서 작성과 발표를 맡았다.

조진영이 최종보고서 수정 및 보완을 맡았고 make 파일 작성을 맡았다.

윤태훈이 발표 ppt 및 소프트웨어 설계도제작을 맡았다.

**6. 결론 및 총평**

결론적으로 IPC 기법 중 shared memory 기법이 가장 좋은 성능을 보여 주었고 특이한 점은 pipe가 message passing보다 빨랐다는 점이다. 하지만 shared memory는 생산자/소비자 동기화기법을 적용해야해서 다른 두 기법보다 구현의 어려움이 존재했다. pipe는 파일 디스크립터로 관리되지만 실제 파일을 읽고 쓰는 1400만 cpu cycle이 걸리지 않는 점을 확인했다. pipe는 실제로 커널의 메모리 버퍼에 읽기 쓰기 연산에 이용된다. message passing은 pipe와 다르게 단순한 데이터구조가 아닌 메세지 큐를 사용한다. 따라서 단순 데이터 전달은 조금 느릴 수 있어도 구조화된 데이터 전달에는 이점이 있음을 확인했다.

팀프로젝트를 통해 수업에서 실습한 IPC기법 코드보다 조금 더 어려운 설계를 구현해보며 시행착오를 거치면서 수업시간에 배운 내용을 더 정확히 이해할 수 있었다. 또한 스레드의 공유 자원에 대한 동기화 기법을 사용해보며 데드락, 경쟁 상태 등에 대해서도 더 깊은 이해를 할 수 있었다. 스레드 동기화뿐 아니라 shared memory의 생산자 소비자 동기화를 사용해보며 프로세스간 순서 보장 동기화에 대한 깊은 이해를 할 수 있었다.

Copilot사용: shared memory에서는 다른 기법과는 다르게 빈 memory를 읽거나 꽉 찬 memory를 쓰면 에러가 나서 생산자/소비자 동기화를 해야 했는데 해당 동기화 부분의 구현에서 도움을 받았다. 또한 프로그램의 완성도를 높이기 위해서 예외 사항을 처리하는 코드에 대해 도움을 받았다.

**7. 프로그램 소스**

**PIPE server**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  #define STORAGE\_DIR "/tmp/storage\_team7/" // 파일 저장소 디렉터리[공유]  #define MAX\_FILES 10  // 파일 락 구조체 정의  typedef struct {  char filename[256];  pthread\_rwlock\_t lock;  } FileLock;  FileLock file\_locks[MAX\_FILES];  int file\_lock\_count = 0;  pthread\_mutex\_t file\_lock\_list\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  // 파일 락을 얻거나 새로 생성하는 함수  FileLock\* get\_file\_lock(const char\* filename) {  pthread\_mutex\_lock(&file\_lock\_list\_mutex);  for (int i = 0; i < file\_lock\_count; ++i) {  if (strcmp(file\_locks[i].filename, filename) == 0) {  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[i];  }  }  if (file\_lock\_count >= MAX\_FILES) {  fprintf(stderr, "[서버] 파일 락 수 초과\n");  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return NULL;  }  strcpy(file\_locks[file\_lock\_count].filename, filename);  pthread\_rwlock\_init(&file\_locks[file\_lock\_count].lock, NULL);  ++file\_lock\_count;  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[file\_lock\_count - 1];  }  // 클라이언트 요청 처리  void\* client\_handler(void\* arg) {  struct timespec start, end;  double elapsed;  char client\_fifo[256];  char buffer[MAX\_SIZE];  int fd;  strcpy(client\_fifo, (char\*)arg);  free(arg);  // 클라이언트 요청 읽기  fd = open(client\_fifo, O\_RDONLY);  if (fd < 0) {  perror("[서버] 클라이언트 FIFO 열기 실패");  pthread\_exit(NULL);  }  read(fd, buffer, MAX\_SIZE);  close(fd);  // 요청 처리  char command[10], filename[256];  sscanf(buffer, "%s %s", command, filename);  // 시작 시간 기록  clock\_gettime(CLOCK\_MONOTONIC, &start);  if (strcmp(command, "UPLOAD") == 0) {  // 업로드 요청 처리  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  perror("[서버] 파일 락 생성 실패");  pthread\_exit(NULL);  }  printf("[서버] 업로드 요청: '%s' 락 대기중...\n", filename);  pthread\_rwlock\_wrlock(&file\_lock->lock);  printf("[서버] '%s' 업로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  int file\_fd = open(filepath, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);  if (file\_fd < 0) {  perror("[서버] 파일 저장 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }    fd = open(client\_fifo, O\_RDONLY);  if (fd < 0) {  close(file\_fd);  perror("[서버] 클라이언트 FIFO 열기 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }  ssize\_t bytes\_read;  while ((bytes\_read = read(fd, buffer, MAX\_SIZE)) > 0) {  write(file\_fd, buffer, bytes\_read);  }  close(fd);  close(file\_fd);  pthread\_rwlock\_unlock(&file\_lock->lock);  fd = open(client\_fifo, O\_WRONLY);  write(fd, "업로드 완료", strlen("업로드 완료") + 1);  close(fd);  printf("[서버] 파일 '%s' 업로드 완료\n", filename);  } else if (strcmp(command, "DOWNLOAD") == 0) {  // 다운로드 요청 처리  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  perror("[서버] 파일 락 생성 실패");  pthread\_exit(NULL);  }  printf("[서버] 다운로드 요청: '%s' 락 대기중...\n", filename);  pthread\_rwlock\_rdlock(&file\_lock->lock);  printf("[서버] '%s' 다운로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  FILE\* file = fopen(filepath, "rb");  if (!file) {  perror("[서버] 파일 열기 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }  fd = open(client\_fifo, O\_WRONLY);  if (fd < 0) {  fclose(file);  perror("[서버] 클라이언트 FIFO 열기 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }    size\_t bytes\_read;  while ((bytes\_read = fread(buffer, 1, MAX\_SIZE, file)) > 0) {  write(fd, buffer, bytes\_read);  }  fclose(file);  close(fd);  pthread\_rwlock\_unlock(&file\_lock->lock);  printf("[서버] 파일 '%s' 다운로드 완료\n", filename);  }  // 종료 시간 기록 및 소요 시간 계산  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (double)(end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  //printf("소요시간 : %.9f초\n",elapsed);  // unlink(client\_fifo);  pthread\_exit(NULL);  }  int main() {  pthread\_t thread;  int server\_fd;  char client\_fifo[256];  // 저장소 디렉터리 생성  if (access(STORAGE\_DIR, F\_OK) != 0) { // 경로가 존재하지 않으면  if (mkdir(STORAGE\_DIR, 0700) < 0) { // 디렉토리 생성  perror("mkdir 실패");  return 1; // 오류 발생 시 종료  }  }  // 서버용 FIFO 생성  if (access(SERVER\_FIFO, F\_OK) != 0) { // 파일이 없으면  if (mkfifo(SERVER\_FIFO, 0600) < 0) {  perror("mkfifo 실패");  return 1; // 오류 시 종료  }  }  printf("[서버] 파일 공유 서비스 대기 중...\n");  while (1) {  server\_fd = open(SERVER\_FIFO, O\_RDONLY);  read(server\_fd, client\_fifo, sizeof(client\_fifo));  close(server\_fd);  char\* fifo\_path = strdup(client\_fifo);  pthread\_create(&thread, NULL, client\_handler, fifo\_path);  pthread\_detach(thread);  }    // 종료 시 모든 락 해제  pthread\_mutex\_lock(&file\_lock\_list\_mutex);  for (int i = 0; i < file\_lock\_count; ++i) {  pthread\_rwlock\_destroy(&file\_locks[i].lock);  }  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  unlink(SERVER\_FIFO);  return 0;  } |

**PIPE client**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  int main() {  char client\_fifo[256];  char buffer[MAX\_SIZE];  int server\_fd, client\_fd;  struct timespec start, end;  double elapsed;  sprintf(client\_fifo, "/tmp/client\_fifo\_%d", getpid());  mkfifo(client\_fifo, 0600);  while (1) {  printf("[클라이언트] 명령 입력 (UPLOAD filename / DOWNLOAD filename / q 종료): ");  fgets(buffer, MAX\_SIZE, stdin);  buffer[strcspn(buffer, "\n")] = 0; // 개행 문자 제거  // 'q' 입력 시 종료 처리  if (strcmp(buffer, "q") == 0) {  printf("[클라이언트] 클라이언트 파이프 해제 및 종료합니다.\n");  unlink(client\_fifo);  break;  }  if (strcmp(buffer, "exit") == 0) {  printf("[클라이언트] 종료합니다.\n");  unlink(client\_fifo);  break;  }  char command[10], filename[256];  sscanf(buffer, "%s %s", command, filename);  // 서버에 클라이언트 FIFO 경로 전달  server\_fd = open(SERVER\_FIFO, O\_WRONLY);  write(server\_fd, client\_fifo, strlen(client\_fifo) + 1);  close(server\_fd);  // 서버에 명령 전달  client\_fd = open(client\_fifo, O\_WRONLY);  write(client\_fd, buffer, strlen(buffer) + 1);  close(client\_fd);    // 시작 시간 기록  clock\_gettime(CLOCK\_MONOTONIC, &start);  if (strcmp(command, "UPLOAD") == 0) {  // 파일 읽어 전송  FILE\* file = fopen(filename, "rb");  if (!file) {  printf("[클라이언트] 파일 '%s' 열기 실패\n", filename);  continue;  }  client\_fd = open(client\_fifo, O\_WRONLY);  while (!feof(file)) {  size\_t bytes\_read = fread(buffer, 1, MAX\_SIZE, file);  if (bytes\_read > 0) {  write(client\_fd, buffer, bytes\_read);  }  }  fclose(file);  close(client\_fd);  // 완료 응답 대기  client\_fd = open(client\_fifo, O\_RDONLY);  read(client\_fd, buffer, MAX\_SIZE);  close(client\_fd);  printf("[클라이언트] %s\n", buffer);  } else if (strcmp(command, "DOWNLOAD") == 0) {  // 서버로부터 데이터 수신  client\_fd = open(client\_fifo, O\_RDONLY);  FILE\* file = fopen(filename, "wb");  if (!file) {  printf("[클라이언트] 파일 '%s' 저장 실패\n", filename);  close(client\_fd);  continue;  }  ssize\_t bytes\_read;  while ((bytes\_read = read(client\_fd, buffer, MAX\_SIZE)) > 0) {  fwrite(buffer, 1, bytes\_read, file);  }  fclose(file);  close(client\_fd);  printf("[클라이언트] 파일 '%s' 다운로드 완료\n", filename);  }  }  // 종료 시간 기록 및 소요 시간 계산  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  printf("소요시간 : %.9f초\n", elapsed);  unlink(client\_fifo);  return 0;  } |

**SHARED MEMORY server**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <errno.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  #define STORAGE\_DIR "/tmp/storage\_team7/"  #define MAX\_FILES 10  #define SEM\_PERMS 0600  typedef struct {  char filename[256];  pthread\_rwlock\_t lock;  } FileLock;  union semun {  int val;  struct semid\_ds\* buf;  unsigned short\* array;  };  typedef struct {  char data[MAX\_SIZE];  size\_t size;  } SharedData;  enum {  SEM\_WRITE = 0,  SEM\_READ = 1,  SEM\_COUNT = 2  };  FileLock file\_locks[MAX\_FILES];  int file\_lock\_count = 0;  pthread\_mutex\_t file\_lock\_list\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  FileLock\* get\_file\_lock(const char\* filename) {  pthread\_mutex\_lock(&file\_lock\_list\_mutex);  for (int i = 0; i < file\_lock\_count; ++i) {  if (strcmp(file\_locks[i].filename, filename) == 0) {  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[i];  }  }  if (file\_lock\_count >= MAX\_FILES) {  fprintf(stderr, "파일 락 수 초과\n");  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return NULL;  }  strcpy(file\_locks[file\_lock\_count].filename, filename);  pthread\_rwlock\_init(&file\_locks[file\_lock\_count].lock, NULL);  ++file\_lock\_count;  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[file\_lock\_count - 1];  }  void\* client\_handler(void\* arg) {  struct timespec start, end;  double elapsed;  char\* buffer = (char\*)arg;  key\_t key;  char command[10], filename[256];  sscanf(buffer, "%d %s %s", &key, command, filename);  free(buffer);  int shmid = shmget(key, sizeof(SharedData), 0666);  if (shmid == -1) {  perror("shmget failed");  pthread\_exit(NULL);  }  int semid = semget(key, SEM\_COUNT, 0666);  if (semid == -1) {  perror("semget failed");  pthread\_exit(NULL);  }  SharedData\* shared\_data = (SharedData\*)shmat(shmid, NULL, 0);  if (shared\_data == (void\*)-1) {  perror("shmat failed");  pthread\_exit(NULL);  }  // 시작 시간 기록  clock\_gettime(CLOCK\_MONOTONIC, &start);  if (strcmp(command, "UPLOAD") == 0) {  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  shmdt(shared\_data);  pthread\_exit(NULL);  }  printf("[서버] 업로드 요청: '%s' 락 대기중...\n", filename);  pthread\_rwlock\_wrlock(&file\_lock->lock);  printf("'%s' 업로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  FILE\* file = fopen(filepath, "wb");  if (!file) {  perror("파일 생성 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  shmdt(shared\_data);  pthread\_exit(NULL);  }  int continue\_reading = 1;  struct sembuf sops;  sops.sem\_flg = 0;  while (continue\_reading) {  sops.sem\_num = SEM\_READ;  sops.sem\_op = -1;  if (semop(semid, &sops, 1) == -1) {  if (errno == EINVAL) {  break;  }  perror("semop P failed");  break;  }  if (shared\_data->size == 0) {  sops.sem\_num = SEM\_WRITE;  sops.sem\_op = 1;  semop(semid, &sops, 1);  continue\_reading = 0;  break;  }  fwrite(shared\_data->data, 1, shared\_data->size, file);  sops.sem\_num = SEM\_WRITE;  sops.sem\_op = 1;  if (semop(semid, &sops, 1) == -1) {  if (errno != EINVAL) {  perror("semop V failed");  }  break;  }  }  fclose(file);  pthread\_rwlock\_unlock(&file\_lock->lock);  printf("'%s' 업로드 완료\n", filename);  }  else if (strcmp(command, "DOWNLOAD") == 0) {  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  shmdt(shared\_data);  pthread\_exit(NULL);  }  pthread\_rwlock\_rdlock(&file\_lock->lock);  printf("'%s' 다운로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  FILE\* file = fopen(filepath, "rb");  if (!file) {  perror("파일 열기 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  shmdt(shared\_data);  pthread\_exit(NULL);  }  size\_t bytes\_read;  int continue\_writing = 1;  struct sembuf sops;  sops.sem\_flg = 0;  while (continue\_writing) {  sops.sem\_num = SEM\_WRITE;  sops.sem\_op = -1;  if (semop(semid, &sops, 1) == -1) {  if (errno == EINVAL) {  break;  }  break;  }  bytes\_read = fread(shared\_data->data, 1, MAX\_SIZE, file);  shared\_data->size = bytes\_read;  sops.sem\_num = SEM\_READ;  sops.sem\_op = 1;  if (semop(semid, &sops, 1) == -1) {  if (errno != EINVAL) {  }  break;  }  if (bytes\_read < MAX\_SIZE) {  continue\_writing = 0;  }  }  fclose(file);  // EOF 표시  sops.sem\_num = SEM\_WRITE;  sops.sem\_op = -1;  if (semop(semid, &sops, 1) != -1) {  shared\_data->size = 0;  sops.sem\_num = SEM\_READ;  sops.sem\_op = 1;  semop(semid, &sops, 1);  }  pthread\_rwlock\_unlock(&file\_lock->lock);  printf("'%s' 다운로드 완료\n", filename);  }  // 종료 시간 기록 및 소요 시간 계산  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  //printf("소요시간 : %.9f초\n",elapsed);  shmdt(shared\_data);  pthread\_exit(NULL);  }  int main() {  pthread\_t thread;  int server\_fd;  char buffer[MAX\_SIZE];  if (access(STORAGE\_DIR, F\_OK) != 0) {  if (mkdir(STORAGE\_DIR, 0700) < 0) {  perror("mkdir 실패");  return 1;  }  }  if (access(SERVER\_FIFO, F\_OK) != 0) {  if (mkfifo(SERVER\_FIFO, 0600) < 0) {  perror("mkfifo 실패");  return 1;  }  }  printf("파일 공유 서비스 대기 중...\n");  while (1) {  server\_fd = open(SERVER\_FIFO, O\_RDONLY);  read(server\_fd, buffer, sizeof(buffer));  close(server\_fd);  char\* thread\_buffer = strdup(buffer);  pthread\_create(&thread, NULL, client\_handler, thread\_buffer);  pthread\_detach(thread);  }  return 0;  } |

**SHARED MEMORY client**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <errno.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  #define SEM\_PERMS 0600  union semun {  int val;  struct semid\_ds\* buf;  unsigned short\* array;  };  typedef struct {  char data[MAX\_SIZE];  size\_t size;  } SharedData;  enum {  SEM\_WRITE = 0,  SEM\_READ = 1,  SEM\_COUNT = 2  };  void sem\_wait(int semid, int sem\_num) {  struct sembuf sb = { sem\_num, -1, 0 };  if (semop(semid, &sb, 1) == -1) {  perror("semop P failed");  exit(1);  }  }  void sem\_signal(int semid, int sem\_num) {  struct sembuf sb = { sem\_num, 1, 0 };  if (semop(semid, &sb, 1) == -1) {  perror("semop V failed");  exit(1);  }  }  int main(int argc, char\* argv[]) {  if (argc != 3) {  printf("Usage: %s <UPLOAD/DOWNLOAD> <filename>\n", argv[0]);  return 1;  }  struct timespec start, end;  double elapsed;  char\* command = argv[1];  char\* filename = argv[2];  int server\_fd;  key\_t key = ftok("/tmp", getpid());  // 공유 메모리 생성  int shmid = shmget(key, sizeof(SharedData), IPC\_CREAT | 0666);  if (shmid == -1) {  perror("shmget failed");  exit(1);  }  // 세마포어 생성  int semid = semget(key, SEM\_COUNT, IPC\_CREAT | SEM\_PERMS);  if (semid == -1) {  perror("semget failed");  exit(1);  }  // 세마포어 초기화  if (errno != EEXIST) {  union semun arg;  arg.val = 1;  if (semctl(semid, SEM\_WRITE, SETVAL, arg) == -1) {  perror("semctl SETVAL SEM\_WRITE failed");  exit(1);  }  arg.val = 0;  if (semctl(semid, SEM\_READ, SETVAL, arg) == -1) {  perror("semctl SETVAL SEM\_READ failed");  exit(1);  }  }  // 공유 메모리 연결  SharedData\* shared\_data = (SharedData\*)shmat(shmid, NULL, 0);  if (shared\_data == (void\*)-1) {  perror("shmat failed");  exit(1);  }  // 서버에 명령어 전달  char msg[MAX\_SIZE];  sprintf(msg, "%d %s %s", key, command, filename);  server\_fd = open(SERVER\_FIFO, O\_WRONLY);  if (write(server\_fd, msg, strlen(msg) + 1) == -1) {  perror("Failed to write to server");  goto cleanup;  }  close(server\_fd);  // 시작 시간 기록  clock\_gettime(CLOCK\_MONOTONIC, &start);  if (strcmp(command, "UPLOAD") == 0) {  FILE\* file = fopen(filename, "rb");  if (!file) {  printf("파일 '%s' 열기 실패\n", filename);  goto cleanup;  }  size\_t bytes\_read;  do {  sem\_wait(semid, SEM\_WRITE);  bytes\_read = fread(shared\_data->data, 1, MAX\_SIZE, file);  shared\_data->size = bytes\_read;  sem\_signal(semid, SEM\_READ);  } while (bytes\_read == MAX\_SIZE);  fclose(file);  // 전송 완료 표시  sem\_wait(semid, SEM\_WRITE);  shared\_data->size = 0;  sem\_signal(semid, SEM\_READ);  printf("업로드 완료\n");  }  else if (strcmp(command, "DOWNLOAD") == 0) {  FILE\* file = fopen(filename, "wb");  if (!file) {  printf("파일 '%s' 저장 실패\n", filename);  goto cleanup;  }  do {  sem\_wait(semid, SEM\_READ);  if (shared\_data->size == 0) {  sem\_signal(semid, SEM\_WRITE);  break;  }  fwrite(shared\_data->data, 1, shared\_data->size, file);  sem\_signal(semid, SEM\_WRITE);  } while (1);  fclose(file);  printf("다운로드 완료\n");  }  // 종료 시간 기록 및 소요 시간 계산  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  printf("소요시간 : %.9f초\n", elapsed);  cleanup:  // 리소스 정리  shmdt(shared\_data);  shmctl(shmid, IPC\_RMID, NULL);  semctl(semid, 0, IPC\_RMID);  return 0;  } |

**MESSAGE PASSING server**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <pthread.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/msg.h>  #include <errno.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  #define STORAGE\_DIR "/tmp/storage\_team7/"  #define MAX\_FILES 10  // 메시지 타입 정의  #define MSG\_DATA 1  #define MSG\_EOF 2  #define MSG\_ERROR 3  #define MSG\_SUCCESS 4  typedef struct {  char filename[256];  pthread\_rwlock\_t lock;  } FileLock;  // 메시지 큐에서 사용할 구조체  typedef struct {  long mtype;  char mdata[MAX\_SIZE];  size\_t size;  } MsgBuffer;  FileLock file\_locks[MAX\_FILES];  int file\_lock\_count = 0;  pthread\_mutex\_t file\_lock\_list\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  FileLock\* get\_file\_lock(const char\* filename) {  pthread\_mutex\_lock(&file\_lock\_list\_mutex);  for (int i = 0; i < file\_lock\_count; ++i) {  if (strcmp(file\_locks[i].filename, filename) == 0) {  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[i];  }  }  if (file\_lock\_count >= MAX\_FILES) {  fprintf(stderr, "파일 락 수 초과\n");  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return NULL;  }  strcpy(file\_locks[file\_lock\_count].filename, filename);  pthread\_rwlock\_init(&file\_locks[file\_lock\_count].lock, NULL);  ++file\_lock\_count;  pthread\_mutex\_unlock(&file\_lock\_list\_mutex);  return &file\_locks[file\_lock\_count - 1];  }  void\* client\_handler(void\* arg) {  struct timespec start, end;  double elapsed;  char\* buffer = (char\*)arg;  key\_t key;  char command[10], filename[256];  sscanf(buffer, "%d %s %s", &key, command, filename);  free(buffer);  // 클라이언트의 메시지 큐 접근  int msgid = msgget(key, 0666);  if (msgid == -1) {  perror("msgget failed");  pthread\_exit(NULL);  }  MsgBuffer msg;  clock\_gettime(CLOCK\_MONOTONIC, &start);  // 클라이언트에게 준비 완료 신호 전송  msg.mtype = MSG\_SUCCESS;  msg.size = 0;  if (msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0) == -1) {  perror("Failed to send ready signal");  pthread\_exit(NULL);  }  if (strcmp(command, "UPLOAD") == 0) {  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  pthread\_exit(NULL);  }  printf("[서버] 업로드 요청: '%s' 락 대기중...\n", filename);  pthread\_rwlock\_wrlock(&file\_lock->lock);  printf("'%s' 업로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  FILE\* file = fopen(filepath, "wb");  if (!file) {  perror("파일 생성 실패");  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }  while (1) {  if (msgrcv(msgid, &msg, sizeof(msg) - sizeof(long), 0, 0) == -1) {  perror("msgrcv failed");  break;  }  if (msg.mtype == MSG\_EOF) {  break;  }  fwrite(msg.mdata, 1, msg.size, file);  // 클라이언트에게 수신 확인  msg.mtype = MSG\_SUCCESS;  msg.size = 0;  if (msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0) == -1) {  perror("Failed to send acknowledgment");  break;  }  }  fclose(file);  pthread\_rwlock\_unlock(&file\_lock->lock);  printf("'%s' 업로드 완료\n", filename);  }  else if (strcmp(command, "DOWNLOAD") == 0) {  FileLock\* file\_lock = get\_file\_lock(filename);  if (!file\_lock) {  pthread\_exit(NULL);  }  pthread\_rwlock\_rdlock(&file\_lock->lock);  printf("'%s' 다운로드 시작\n", filename);  char filepath[512];  sprintf(filepath, "%s%s", STORAGE\_DIR, filename);  FILE\* file = fopen(filepath, "rb");  if (!file) {  perror("파일 열기 실패");  msg.mtype = MSG\_ERROR;  strcpy(msg.mdata, "파일을 열 수 없습니다.");  msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0);  pthread\_rwlock\_unlock(&file\_lock->lock);  pthread\_exit(NULL);  }  size\_t bytes\_read;  while ((bytes\_read = fread(msg.mdata, 1, MAX\_SIZE, file)) > 0) {  msg.mtype = MSG\_DATA;  msg.size = bytes\_read;  if (msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0) == -1) {  perror("msgsnd failed");  break;  }  // 클라이언트의 수신 확인 대기  if (msgrcv(msgid, &msg, sizeof(msg) - sizeof(long), MSG\_SUCCESS, 0) == -1) {  perror("Failed to receive acknowledgment");  break;  }  }  // EOF 메시지 전송  msg.mtype = MSG\_EOF;  msg.size = 0;  msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0);  fclose(file);  pthread\_rwlock\_unlock(&file\_lock->lock);  printf("'%s' 다운로드 완료\n", filename);  }  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  //printf("소요시간 : %.9f초\n", elapsed);  pthread\_exit(NULL);  }  int main() {  pthread\_t thread;  int server\_fd;  char buffer[MAX\_SIZE];  if (access(STORAGE\_DIR, F\_OK) != 0) {  if (mkdir(STORAGE\_DIR, 0700) < 0) {  perror("mkdir 실패");  return 1;  }  }  if (access(SERVER\_FIFO, F\_OK) != 0) {  if (mkfifo(SERVER\_FIFO, 0600) < 0) {  perror("mkfifo 실패");  return 1;  }  }  printf("파일 공유 서비스 대기 중...\n");  while (1) {  server\_fd = open(SERVER\_FIFO, O\_RDONLY);  read(server\_fd, buffer, sizeof(buffer));  close(server\_fd);  char\* thread\_buffer = strdup(buffer);  pthread\_create(&thread, NULL, client\_handler, thread\_buffer);  pthread\_detach(thread);  }  return 0;  } |

**MESSAGE PASSING client**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/msg.h>  #include <errno.h>  #include <time.h>  #define MAX\_SIZE 1024  #define SERVER\_FIFO "/tmp/server\_fifo\_team7"  // 메시지 타입 정의  #define MSG\_DATA 1  #define MSG\_EOF 2  #define MSG\_ERROR 3  #define MSG\_SUCCESS 4  // 메시지 큐에서 사용할 구조체  typedef struct {  long mtype;  char mdata[MAX\_SIZE];  size\_t size;  } MsgBuffer;  int main(int argc, char\* argv[]) {  if (argc != 3) {  printf("Usage: %s <UPLOAD/DOWNLOAD> <filename>\n", argv[0]);  return 1;  }  struct timespec start, end;  double elapsed;  char\* command = argv[1];  char\* filename = argv[2];  int server\_fd;  // 클라이언트 고유 키 생성 (PID 기반)  key\_t key = ftok("/tmp", getpid());  // 메시지 큐 생성  int msgid = msgget(key, IPC\_CREAT | 0666);  if (msgid == -1) {  perror("msgget failed");  exit(1);  }  // 서버에 명령 전달  char msg[MAX\_SIZE];  sprintf(msg, "%d %s %s", key, command, filename);  server\_fd = open(SERVER\_FIFO, O\_WRONLY);  if (write(server\_fd, msg, strlen(msg) + 1) == -1) {  perror("Failed to write to server");  goto cleanup;  }  close(server\_fd);  MsgBuffer msgbuf;  clock\_gettime(CLOCK\_MONOTONIC, &start);  // 서버 준비 응답 대기  if (msgrcv(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), MSG\_SUCCESS, 0) == -1) {  perror("Failed to receive server ready signal");  goto cleanup;  }  if (strcmp(command, "UPLOAD") == 0) {  FILE\* file = fopen(filename, "rb");  if (!file) {  printf("파일 '%s' 열기 실패\n", filename);  goto cleanup;  }  size\_t bytes\_read;  while ((bytes\_read = fread(msgbuf.mdata, 1, MAX\_SIZE, file)) > 0) {  msgbuf.mtype = MSG\_DATA;  msgbuf.size = bytes\_read;  if (msgsnd(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), 0) == -1) {  if (errno == EIDRM) {  printf("서버와의 연결이 끊어졌습니다.\n");  }  else {  perror("msgsnd failed");  }  fclose(file);  goto cleanup;  }  // 서버 응답 대기  if (msgrcv(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), MSG\_SUCCESS, 0) == -1) {  if (errno == EIDRM) {  printf("서버와의 연결이 끊어졌습니다.\n");  }  else {  perror("msgrcv failed");  }  fclose(file);  goto cleanup;  }  }  // EOF 메시지 전송  msgbuf.mtype = MSG\_EOF;  msgbuf.size = 0;  msgsnd(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), 0);  fclose(file);  printf("업로드 완료\n");  }  else if (strcmp(command, "DOWNLOAD") == 0) {  FILE\* file = fopen(filename, "wb");  if (!file) {  printf("파일 '%s' 저장 실패\n", filename);  goto cleanup;  }  while (1) {  if (msgrcv(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), 0, 0) == -1) {  if (errno == EIDRM) {  printf("서버와의 연결이 끊어졌습니다.\n");  }  else {  perror("msgrcv failed");  }  fclose(file);  goto cleanup;  }  if (msgbuf.mtype == MSG\_EOF) {  break;  }  else if (msgbuf.mtype == MSG\_ERROR) {  printf("서버에서 오류가 발생했습니다: %s\n", msgbuf.mdata);  fclose(file);  goto cleanup;  }  fwrite(msgbuf.mdata, 1, msgbuf.size, file);  // 데이터 수신 확인 메시지 전송  msgbuf.mtype = MSG\_SUCCESS;  msgbuf.size = 0;  if (msgsnd(msgid, &msgbuf, sizeof(msgbuf) - sizeof(long), 0) == -1) {  perror("Failed to send acknowledgment");  fclose(file);  goto cleanup;  }  }  fclose(file);  printf("다운로드 완료\n");  }  clock\_gettime(CLOCK\_MONOTONIC, &end);  elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / (double)1000000000L;  printf("소요시간 : %.9f초\n", elapsed);  cleanup:  // 메시지 큐 제거  msgctl(msgid, IPC\_RMID, NULL);  return 0;  } |