5. 프로세스 간 통신 (IPC)

5.1 파이프 (`pipe()`, `mkfifo()`)

1. 개요

파이프(pipe)는 **프로세스 간 통신(IPC, Inter-Process Communication)**을 위한 가장 간단하고 효율적인 메커니즘 중 하나다.

- **한 프로세스에서 데이터를 쓰면, 다른 프로세스가 그 데이터를 읽을 수 있다.**
- 파이프는 **커널 버퍼**를 이용해 데이터를 임시 저장한다.
- **단방향 통신** 구조이다. (양방향으로 쓰고 싶으면 파이프 두 개 사용)

파이프는 크게 2가지로 나눌 수 있다:

종류	특징
익명 파이프 (`pipe()`)	부모-자식 프로세스 간 통신에 주로 사용됨
명명된 파이프 (`mkfifo()`)	서로 관계 없는 독립적인 프로세스 간 통신 가능 (파일시스템 경로로 접근 가능)

---

2. `pipe()` — 익명 파이프

2.1 함수 원형

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- `pipefd[0]` → **읽기용** file descriptor (read end)
- `pipefd[1]` → **쓰기용** file descriptor (write end)

반환값

- 성공 시 0, 실패 시 -1

2.2 기본 사용 패턴

```
Parent creates pipe()
fork()

Parent:
    writes to pipefd[1]

Child:
    reads from pipefd[0]
```

- fork 후 부모와 자식이 **각각 다른 쪽의 파이프 끝만 사용**하도록 설정 → 나머지 fd는 닫아주는 게 좋음.

2.3 예제

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    int pipefd[2];
    char buffer[128];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // Child process → read
        close(pipefd[1]);  // write end 닫기
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(pipefd[0]);
    } else {
        // Parent process → write
        close(pipefd[0]);  // read end 닫기
        const char *msg = "Hello from parent!";
        write(pipefd[1], msg, strlen(msg) + 1);
        close(pipefd[1]);
    }

    return 0;
}
```

실행 결과

```
Child received: Hello from parent!
```

---

3. `mkfifo()` — 명명된 파이프 (FIFO)

3.1 개념

- 파일 시스템 상에 **FIFO special file**을 생성
- 이름을 통해 서로 다른 프로세스가 FIFO를 통해 통신 가능
- 커널 내부에서 파이프처럼 동작 (메모리 버퍼 사용)

3.2 함수 원형

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

- `pathname`: 생성할 FIFO 파일 경로
- `mode`: 접근 권한 (ex. `0666`)

반환값

- 성공 시 0, 실패 시 -1

3.3 사용 패턴

```
mkfifo myfifo
```

```
# 터미널1
cat myfifo

# 터미널2
echo "hello via fifo" > myfifo
```

→ cat이 **FIFO에서 데이터 읽음** → 출력됨

3.4 C 코드 예제

writer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

int main() {
    const char *fifo_path = "myfifo";
```

```
    mkfifo(fifo_path, 0666);

    int fd = open(fifo_path, O_WRONLY);
    const char *msg = "Hello via FIFO!";
    write(fd, msg, strlen(msg) + 1);
    close(fd);

    return 0;
}
```

reader.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main() {
    const char *fifo_path = "myfifo";

    int fd = open(fifo_path, O_RDONLY);
    char buffer[128];
    read(fd, buffer, sizeof(buffer));
    printf("Reader received: %s\n", buffer);
    close(fd);

    return 0;
}
```

실행 순서

```
gcc writer.c -o writer
gcc reader.c -o reader

./reader    # 먼저 실행 → read 대기
./writer    # 실행 → writer가 msg 쓰고 종료
```

결과

```
Reader received: Hello via FIFO!
```

---

4. `pipe()` vs `mkfifo()`

구분	`pipe()`	`mkfifo()`
생성 대상	프로세스 간 fd 배열	파일 시스템 내 FIFO 파일

구분	`pipe()`	`mkfifo()`
통신 범위	부모-자식 등 **fork()로 연결된 프로세스**	**완전히 독립적인 프로세스** 간 가능
지속성	프로세스 종료 시 사라짐	FIFO 파일로 계속 존재
사용 예	간단한 IPC (쉘 파이프라인, 프로세스 간 통신)	데몬, 서버-클라이언트 IPC 등

---

5. 결론 🚀

- `pipe()`는 **프로세스 간 빠른 단방향 통신**에 유용.
- `mkfifo()`는 **독립 프로세스 간 IPC**에 유용하며 **쉘에서 사용하기 쉽다**.
- `pipe()`와 `mkfifo()`는 커널이 제공하는 **버퍼 기반 스트림**이며, 사용 시 **정확한 fd 관리**가 중요하다.

5.2 메시지 큐, 세마포어 (`System V`)

1. 개요

System V IPC (Inter-Process Communication)는 리눅스에서 제공하는 **전통적인 IPC 메커니즘** 중 하나다.
가장 대표적인 3가지 기능은:

기능	목적
Message Queue	**프로세스 간 메시지 교환**
Semaphore	**프로세스 간 동기화 및 상호 배제**
Shared Memory	**프로세스 간 메모리 영역 공유**

이번에는 그중 **Message Queue, Semaphore** 를 다룬다.
(Shared Memory는 5.3에서 보게 될 내용)

📌 Part 1: Message Queue (메시지 큐)

2. Message Queue란?

- 커널 내부에 **메시지 큐 객체**를 만들고, 프로세스끼리 **메시지 구조체** 단위로 데이터를 주고받음.
- 각 메시지에 **형(type)**이 있어 메시지 분류 가능.

장점

✅ 비동기 전송 가능 (sender와 receiver 독립 실행 가능)
✅ 메시지의 구조화 가능 (`struct msgbuf` 사용)
✅ 커널이 버퍼 관리

---

3. 관련 함수

```
#include <sys/ipc.h>
#include <sys/msg.h>
```

함수	역할
`msgget()`	메시지 큐 생성 / 접근
`msgsnd()`	메시지 보내기
`msgrcv()`	메시지 받기
`msgctl()`	메시지 큐 제어 (삭제 등)

---

4. 메시지 구조체

```
struct msgbuf {
    long mtype;        // 메시지 타입 (양수)
    char mtext[128];   // 메시지 본문
};
```

- `mtype` → 메시지 종류 식별용 → 수신 시 특정 type만 받을 수 있음.
- `mtext` → 메시지 내용.

---

5. 예제: 메시지 큐 송/수신

송신 프로그램 (sender.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msgbuf {
    long mtype;
    char mtext[128];
};

int main() {
    key_t key = ftok("msgqueuefile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msgbuf message;
    message.mtype = 1;
    strcpy(message.mtext, "Hello via Message Queue!");

```

```
    msgsnd(msgid, &message, sizeof(message.mtext), 0);

    printf("Sent: %s\n", message.mtext);

    return 0;
}
```

수신 프로그램 (receiver.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf {
    long mtype;
    char mtext[128];
};

int main() {
    key_t key = ftok("msgqueuefile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msgbuf message;
    msgrcv(msgid, &message, sizeof(message.mtext), 1, 0);

    printf("Received: %s\n", message.mtext);

    // 메시지 큐 삭제
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

실행 순서

```
touch msgqueuefile
gcc sender.c -o sender
gcc receiver.c -o receiver

./sender
./receiver
```

결과 예시

```
Sent: Hello via Message Queue!
Received: Hello via Message Queue!
```

📌 Part 2: Semaphore (세마포어)

1. Semaphore란?

- **프로세스 간 동기화/상호배제**를 위한 카운터.
- 커널 내에 존재하며 프로세스들이 원자적으로 값을 조작 가능.

사용 예

✅ 임계구역 보호 (Critical Section)
✅ 생산자-소비자 문제 해결
✅ 프로세스 간 실행 순서 제어

---

2. 관련 함수

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

함수	역할
`semget()`	세마포어 세트 생성 / 접근
`semop()`	세마포어 조작 (P/V 연산)
`semctl()`	세마포어 상태 제어 (값 설정, 삭제 등)

---

3. 기본 구조

- `semget()` → 세마포어 ID 획득
- `semctl()` → 초기값 설정
- `semop()` → P 연산(잠금), V 연산(해제) 실행

---

4. 예제: 세마포어 동기화

공통 헤더

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
```

```
};
```

producer.c (V 연산: unlock)

```
int main() {
    key_t key = ftok("semfile", 75);
    int semid = semget(key, 1, 0666 | IPC_CREAT);

    union semun u;
    u.val = 0;
    semctl(semid, 0, SETVAL, u);  // 초기값 0

    printf("Producer doing work...\n");
    sleep(3);

    // V 연산 (sem +1)
    struct sembuf op = {0, 1, 0};
    semop(semid, &op, 1);

    printf("Producer signaled!\n");

    return 0;
}
```

consumer.c (P 연산: lock)

```
int main() {
    key_t key = ftok("semfile", 75);
    int semid = semget(key, 1, 0666);

    printf("Consumer waiting for semaphore...\n");

    // P 연산 (sem -1 → 대기)
    struct sembuf op = {0, -1, 0};
    semop(semid, &op, 1);

    printf("Consumer proceeding!\n");

    // 세마포어 삭제
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```

실행 순서

```
touch semfile
gcc producer.c -o producer
gcc consumer.c -o consumer

./consumer    # consumer 먼저 → 세마포어 대기
./producer    # producer 실행 → 세마포어 signal → consumer 진행
```

결과

```
Consumer waiting for semaphore...
Producer doing work...
Producer signaled!
Consumer proceeding!
```

📌 결론 🚀

기능	핵심 특징
Message Queue	프로세스 간 메시지 교환 (구조화된 메시지 전달 가능)
Semaphore	프로세스 간 동기화 / 임계구역 보호 (원자적 P/V 연산)

System V IPC는 고전적이지만 여전히 강력한 IPC 메커니즘이며:

- `pipe()`, `mkfifo()` → 단순 스트림 전달
- `Message Queue` → 구조화된 메시지 기반 통신
- `Semaphore` → 동기화 및 상호배제
- `Shared Memory` → 고속 메모리 공유

로 각자 목적에 맞춰 사용하면 된다.
POSIX IPC (mq_open, sem_open 등)는 System V의 현대화된 대안으로도 사용된다.

5.3 공유 메모리 (`shmget`, `shmat`)

1. 개요

공유 메모리(Shared Memory)는 리눅스에서 **가장 빠른 IPC(Inter-Process Communication)** 방법 중 하나다.
다른 IPC 방법들은 **커널 버퍼를 거쳐 데이터를 전달**하지만, 공유 메모리는 **여러 프로세스가 동일한 메모리 영역을 직접 참조**한다.

특징

✅ 매우 빠름 (메모리 접근 비용 수준)
✅ 대용량 데이터 공유 가능
✅ 명시적 동기화 필요 (별도로 Semaphore 등 사용 권장)

---

2. 관련 함수

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

함수	역할
`shmget()`	공유 메모리 세그먼트 생성 또는 접근
`shmat()`	공유 메모리를 프로세스 주소 공간에 연결 (attach)
`shmdt()`	공유 메모리 연결 해제 (detach)
`shmctl()`	공유 메모리 정보 조회, 삭제 등 제어

---

3. 기본 흐름

```
1️⃣ Producer:
    shmget() → shmat() → 메모리 쓰기 → shmdt()

2️⃣ Consumer:
    shmget() → shmat() → 메모리 읽기 → shmdt()

3️⃣ 마무리: shmctl(IPC_RMID) → 공유 메모리 삭제
```

---

4. 공유 메모리 구조

```
key_t key = ftok("shmfile", 65);  // 키 생성
int shmid = shmget(key, size, IPC_CREAT | 0666);  // 공유 메모리 생성
void *ptr = shmat(shmid, NULL, 0);  // 프로세스에 attach
```

- 공유 메모리는 **키(key)**를 기준으로 접근한다 (`ftok()` 사용 가능).
- `shmat()`는 공유 메모리를 **가상 주소 공간으로 매핑**한다 → 일반 포인터처럼 사용 가능.

---

5. 예제: 공유 메모리를 통한 프로세스 간 데이터 전달

Producer (writer.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    key_t key = ftok("shmfile", 65);  // 키 생성
```

```
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);  // 공유 메모리 생성

    char *str = (char *)shmat(shmid, NULL, 0);  // attach
    printf("Write Data: ");
    fgets(str, 1024, stdin);  // 사용자 입력을 공유 메모리에 저장

    shmdt(str);  // detach
    return 0;
}
```

Consumer (reader.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
    key_t key = ftok("shmfile", 65);  // 동일한 키 사용
    int shmid = shmget(key, 1024, 0666);  // 기존 공유 메모리에 접근

    char *str = (char *)shmat(shmid, NULL, 0);  // attach

    printf("Data read from memory: %s\n", str);

    shmdt(str);  // detach
    shmctl(shmid, IPC_RMID, NULL);  // 공유 메모리 삭제

    return 0;
}
```

실행 순서

```
touch shmfile
gcc writer.c -o writer
gcc reader.c -o reader

./writer    # 먼저 실행 → 입력 데이터 저장
./reader    # 이후 실행 → 데이터 읽기 + 공유 메모리 삭제
```

결과 예시

```
Write Data: Hello Shared Memory!

Data read from memory: Hello Shared Memory!
```

---

6. 주의 사항

- 공유 메모리는 매우 빠르지만, **동기화가 필요**하다.
  → 여러 프로세스가 동시에 접근하면 **데이터 충돌 가능**
  → 일반적으로 **Semaphore** 등을 같이 사용
- `shmctl(IPC_RMID)` 호출로 공유 메모리를 **명시적으로 삭제**해야 한다.
  → 그렇지 않으면 시스템에 orphan된 공유 메모리가 남는다 (`ipcs -m` 명령으로 확인 가능).

---

7. `ipcs`, `ipcrm` 명령어

현재 공유 메모리 목록 확인

```
ipcs -m
```

공유 메모리 삭제 (id 확인 후 삭제)

```
ipcrm -m <shmid>
```

---

8. 결론 🚀

기능	특징
`shmget()`	공유 메모리 세그먼트 생성/획득
`shmat()`	공유 메모리 attach → 포인터처럼 사용 가능
`shmdt()`	detach
`shmctl()`	삭제, 정보 조회 등

공유 메모리는 **매우 빠르고 효율적인 IPC 수단**이며,
대용량 데이터 공유, **다중 프로세스 협력 시스템** 등에 필수적으로 사용된다.

단점은 **명시적 동기화 필요**(보통 Semaphore와 같이 사용).

5.4 시그널과 `kill()`, `signal()`

1. 개요

시그널(Signal)은 리눅스/유닉스에서 **프로세스 간 비동기적 이벤트 통지**를 위한 고전적인 IPC 메커니즘이다.

- 시그널은 커널이 프로세스에게 **"무언가가 발생했다"**는 알림을 보내는 방식이다.
- 예를 들어, 프로세스가 **Ctrl+C**로 종료될 때 내부적으로 **SIGINT 시그널**이 전송된다.

시그널 사용 예시

✅ 프로세스 종료 요청 (kill 명령어)
✅ 알람/타이머 시그널
✅ 사용자 정의 시그널 → 프로세스 제어/이벤트 통지
✅ IPC 구현에서 사용 가능

---

2. 주요 시그널 종류

시그널	의미
`SIGINT`	인터럽트 (Ctrl+C)
`SIGTERM`	정상 종료 요청
`SIGKILL`	강제 종료 (무조건 종료)
`SIGSTOP`	프로세스 일시 정지
`SIGCONT`	정지된 프로세스 재개
`SIGALRM`	알람 타이머 시그널
`SIGUSR1`, `SIGUSR2`	사용자 정의 시그널 (IPC 등에서 많이 사용됨)

→ `man 7 signal` 명령어로 전체 시그널 목록 확인 가능.

---

3. 관련 함수

`kill()` — 시그널 보내기

```c
#include <signal.h>

int kill(pid_t pid, int sig);
```

- `pid`: 대상 프로세스 ID
- `sig`: 보낼 시그널 번호

`signal()` — 시그널 핸들러 설정

```c
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

- `signum`: 설정할 시그널 번호
- `handler`: 시그널 발생 시 호출될 함수 포인터

---

4. 예제: 시그널 핸들링

4.1 시그널 핸들러 등록

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
    printf("Received signal %d\n", signo);
}

int main() {
    signal(SIGINT, sig_handler);  // Ctrl+C 처리
    signal(SIGUSR1, sig_handler); // 사용자 정의 시그널

    printf("PID: %d\n", getpid());

    while (1) {
        printf("Waiting for signal...\n");
        sleep(2);
    }

    return 0;
}
```

4.2 실행 예시

```
gcc signal_test.c -o signal_test
./signal_test
```

```
PID: 12345
Waiting for signal...
Waiting for signal...
```

→ 다른 터미널에서:

```
kill -SIGUSR1 12345
kill -SIGINT 12345
```

→ 출력 예시:

```
Received signal 10
Received signal 2
```

---

5. `kill` 명령어 사용법

프로세스 종료

```
kill -SIGTERM <pid>   # 정상 종료 요청 (기본값)
kill -9 <pid>         # SIGKILL (무조건 종료)
kill -SIGSTOP <pid>   # 프로세스 정지
kill -SIGCONT <pid>   # 프로세스 재개
```

---

6. 주의 사항

- `SIGKILL`은 프로세스가 **catch 불가 / block 불가 / ignore 불가** → 커널이 무조건 종료
- `SIGSTOP`도 마찬가지로 프로세스가 **반응 불가** → 강제 정지됨
- 일반적인 시그널들은 `signal()` 또는 `sigaction()`으로 핸들러 설정 가능

---

7. 고급: `sigaction()` (더 강력한 대안)

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- `sigaction()`은 `signal()`보다 더 안전하고 기능이 많음
- **재진입 안전**, **시그널 블록 설정**, **확장 기능 지원**

---

8. 결론 🚀

기능	특징
시그널	비동기적 이벤트 통지
`kill()`	시그널 전송 (다른 프로세스 또는 그룹에 보냄)
`signal()`	간단한 시그널 핸들러 설정
`sigaction()`	고급 시그널 핸들링 (추천)

사용 사례

✅ **프로세스 종료 요청**
✅ **비동기 이벤트 처리**
✅ **IPC에서 lightweight event 통지**
✅ **타이머 기반 이벤트 구현 (SIGALRM)**

5.5 UNIX 도메인 소켓

1. 개요

UNIX 도메인 소켓(Unix Domain Socket, UDS)은 **같은 머신 내 프로세스 간 통신(IPC)**을 위한 고성능 통신 방법이다.

- 프로세스 간 **스트림 또는 데이터그램 전송 가능**
- **TCP/IP 스택을 거치지 않고 커널 내부 메모리를 통해 직접 데이터 전송** → 매우 빠름
- 주로 **클라이언트-서버 구조 IPC**에 많이 사용됨 (ex: X 서버, DBMS, docker, systemd 내부 등)

장점

✅ 매우 빠른 IPC
✅ 신뢰성 있는 스트림 제공 가능 (SOCK_STREAM)
✅ 같은 호스트 내에서만 사용 → **보안성 높음**
✅ 파일 시스템 내의 경로를 통해 소켓 식별 가능

---

2. 주요 특징

항목	내용
주소 체계	파일 시스템 경로 기반
프로토콜	SOCK_STREAM, SOCK_DGRAM
속도	TCP/UDP보다 훨씬 빠름 (커널 내 복사만 수행)
사용 범위	동일 머신 내 프로세스 간 통신 전용

---

3. 관련 헤더 및 구조체

```
#include <sys/socket.h>
#include <sys/un.h>
```

- 주소 구조체:

```
struct sockaddr_un {
    sa_family_t sun_family;    // AF_UNIX
    char sun_path[108];        // 파일 경로 (NULL 종료 문자열 아님 주의)
};
```

---

4. 프로그래밍 인터페이스

함수	용도
`socket()`	소켓 생성

함수	용도
`bind()`	서버 측에서 소켓을 파일 시스템 경로에 바인딩
`listen()`	서버 측에서 클라이언트 연결 대기
`accept()`	서버 측에서 클라이언트 연결 수락
`connect()`	클라이언트 측에서 서버로 연결 요청
`send()` / `recv()`	데이터 송수신
`close()`	소켓 닫기
`unlink()`	소켓 파일 삭제 (서버 측에서 반드시 해야 함)

---

5. 예제: UNIX 도메인 소켓 스트림 통신

5.1 서버 프로그램 (server.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCKET_PATH "my_socket"

int main() {
    int server_fd, client_fd;
    struct sockaddr_un addr;
    char buffer[128];

    server_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        exit(1);
    }

    // 소켓 파일 존재 시 삭제
    unlink(SOCKET_PATH);

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, SOCKET_PATH);

    if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        perror("bind");
        close(server_fd);
        exit(1);
    }

    if (listen(server_fd, 5) == -1) {
```

```
        perror("listen");
        close(server_fd);
        exit(1);
    }

    printf("Server waiting for connection...\n");

    client_fd = accept(server_fd, NULL, NULL);
    if (client_fd == -1) {
        perror("accept");
        close(server_fd);
        exit(1);
    }

    printf("Client connected!\n");

    read(client_fd, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);

    write(client_fd, "Hello from server!", 19);

    close(client_fd);
    close(server_fd);
    unlink(SOCKET_PATH);  // 소켓 파일 삭제

    return 0;
}
```

5.2 클라이언트 프로그램 (client.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCKET_PATH "my_socket"

int main() {
    int sock_fd;
    struct sockaddr_un addr;
    char buffer[128];

    sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock_fd == -1) {
        perror("socket");
        exit(1);
    }

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, SOCKET_PATH);
```

```
    if (connect(sock_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        perror("connect");
        close(sock_fd);
        exit(1);
    }

    write(sock_fd, "Hello from client!", 19);

    read(sock_fd, buffer, sizeof(buffer));
    printf("Received from server: %s\n", buffer);

    close(sock_fd);

    return 0;
}
```

---

6. 실행 절차

컴파일

```
gcc server.c -o server
gcc client.c -o client
```

실행

1️⃣ 서버 먼저 실행:

```
./server
```

```
Server waiting for connection...
```

2️⃣ 클라이언트 실행:

```
./client
```

3️⃣ 결과 예시

서버 측:

```
Client connected!
Received: Hello from client!
```

클라이언트 측:

```
Received from server: Hello from server!
```

---

7. 주의 사항

- 서버 측에서 소켓 파일(`my_socket`)을 **반드시 unlink()로 삭제**해줘야 한다.
    - 그렇지 않으면 다음에 bind() 시 `Address already in use` 오류 발생.
- **AF_UNIX 주소 체계**는 **로컬 파일 시스템 경로 기반**이다 → 외부 네트워크로는 통신 불가.
- 통신 성능은 TCP보다 훨씬 빠르며, 대규모 IPC 시스템에서 많이 사용됨.

---

8. 결론 🚀

기능	특징
UNIX 도메인 소켓	커널 내부에서 매우 빠른 프로세스 간 통신
SOCK_STREAM	TCP와 같은 **연결지향 스트림 통신** 지원
SOCK_DGRAM	UDP와 같은 **비연결 데이터그램 통신**도 가능
파일 기반 주소	파일 시스템 경로를 주소로 사용 (AF_UNIX)
사용 사례	DBMS, systemd, Docker, X Window System 등

📎 실습

명명된 파이프를 통한 채팅 시뮬레이터

1. 목표

- 서로 다른 **터미널 2개에서 채팅** 가능하도록 구현
- **명명된 파이프(`mkfifo`)** 사용
- A → B 방향용 FIFO, B → A 방향용 FIFO 총 2개 사용 (양방향 통신 구현)

구성:

```
FIFO1 (fifo_AtoB) : A → B 방향
FIFO2 (fifo_BtoA) : B → A 방향
```

양쪽 프로세스는 각각 **쓰기용 FIFO, 읽기용 FIFO**를 사용해서 채팅 수행.

---

2. 준비 단계

FIFO 생성

```
mkfifo fifo_AtoB
mkfifo fifo_BtoA
```

또는 코드 내에서 `mkfifo()` 호출로 생성 가능.

3. 코드 예제

3.1 chat_A.c (A 유저용 프로세스)

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_SEND "fifo_AtoB"
#define FIFO_RECV "fifo_BtoA"

int main() {
    int fd_send, fd_recv;
    char buffer[128];

    // FIFO open
    fd_send = open(FIFO_SEND, O_WRONLY);
    fd_recv = open(FIFO_RECV, O_RDONLY);

    if (fd_send == -1 || fd_recv == -1) {
        perror("open");
        return 1;
    }

    printf("=== Chat A ready ===\n");

    while (1) {
        // 사용자 입력 → 전송
        printf("A: ");
        fgets(buffer, sizeof(buffer), stdin);
        write(fd_send, buffer, strlen(buffer));

        // 상대방 메시지 수신
        int n = read(fd_recv, buffer, sizeof(buffer) - 1);
        if (n > 0) {
            buffer[n] = '\0';
            printf("B: %s", buffer);
        }
    }

    close(fd_send);
    close(fd_recv);

    return 0;
}
```

3.2 chat_B.c (B 유저용 프로세스)

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_SEND "fifo_BtoA"
#define FIFO_RECV "fifo_AtoB"

int main() {
    int fd_send, fd_recv;
    char buffer[128];

    // FIFO open
    fd_send = open(FIFO_SEND, O_WRONLY);
    fd_recv = open(FIFO_RECV, O_RDONLY);

    if (fd_send == -1 || fd_recv == -1) {
        perror("open");
        return 1;
    }

    printf("=== Chat B ready ===\n");

    while (1) {
        // 사용자 입력 → 전송
        printf("B: ");
        fgets(buffer, sizeof(buffer), stdin);
        write(fd_send, buffer, strlen(buffer));

        // 상대방 메시지 수신
        int n = read(fd_recv, buffer, sizeof(buffer) - 1);
        if (n > 0) {
            buffer[n] = '\0';
            printf("A: %s", buffer);
        }
    }

    close(fd_send);
    close(fd_recv);

    return 0;
}
```

4. 실행 순서

1️⃣ FIFO 준비

```
mkfifo fifo_AtoB
mkfifo fifo_BtoA
```

2️⃣ 컴파일

```
gcc chat_A.c -o chat_A
gcc chat_B.c -o chat_B
```

3️⃣ 실행 (2개 터미널 사용)

- 터미널1:

```
./chat_A
```

- 터미널2:

```
./chat_B
```

---

5. 사용 예시

```
터미널1:

=== Chat A ready ===
A: Hello B!
B: Hello A!

터미널2:

=== Chat B ready ===
B: Hello A!
A: Hello B!
```

---

6. 주의 사항

- **양방향 통신 시 FIFO 2개 필요** → 한 FIFO는 **읽기용**, 다른 하나는 **쓰기용**으로 사용
- `read()` 호출은 **블록(blocking) 호출** → 상대방이 먼저 write할 때까지 대기
- `fgets()`는 개행 문자까지 입력받으므로 출력 시 줄바꿈 자동 포함됨

---

7. 정리 🚀

요소	구현 내용
FIFO 사용	`mkfifo()`, `open()`, `read()`, `write()`
양방향 통신	FIFO 2개 필요 (한쪽은 write, 다른쪽은 read)
채팅 시뮬레이터	간단한 IPC 데모로 유용

명명된 파이프는 이렇게 간단한 채팅 프로그램 외에도:

✅ 로그 전송용
✅ 이벤트 통지용
✅ 가벼운 클라이언트-서버 IPC 등에 널리 사용된다.

공유 메모리 기반 계산기

1. 목표

- **Producer 프로세스(사용자)**가 계산할 데이터를 **공유 메모리에 쓰고**
- **Consumer 프로세스(계산기)**가 공유 메모리에서 데이터를 읽어서 계산 후 결과를 다시 공유 메모리에 씀
- 사용자는 결과를 다시 읽음

→ 아주 간단한 **계산기 클라이언트-서버 구조 IPC** 예제

---

2. 설계

공유 메모리 구조 설계

```
struct shm_data {
    int operand1;
    int operand2;
    char operator;
    int result;
    int ready;    // producer → consumer 전달 flag (1: data ready)
    int done;     // consumer → producer 결과 완료 flag (1: done)
};
```

통신 흐름

```
1️⃣ Producer (client):
    operand1, operand2, operator → 공유 메모리에 기록
    ready = 1 설정 → consumer가 읽도록 알림
    done == 1 될 때까지 대기 → 결과 읽기

2️⃣ Consumer (server):
    ready == 1 될 때까지 대기 → 데이터 읽기
    계산 후 result 기록
    done = 1 설정 → producer에 결과 알림
```

---

3. 코드 예제

3.1 shared.h (공통 헤더)

```c
#ifndef SHARED_H
#define SHARED_H

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#define SHM_KEY 0x1234

struct shm_data {
    int operand1;
    int operand2;
    char operator;
    int result;
    int ready;
    int done;
};

#endif
```

3.2 Producer (calculator_client.c)

```c
#include "shared.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int shmid = shmget(SHM_KEY, sizeof(struct shm_data), 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }
```

```
    struct shm_data *data = (struct shm_data *)shmat(shmid, NULL, 0);

    while (1) {
        printf("Enter operand1 operator operand2 (ex: 3 + 5), or 0 q 0 to quit:\n");
        scanf("%d %c %d", &data->operand1, &data->operator, &data->operand2);

        if (data->operator == 'q') {
            data->ready = 0;
            data->done = 0;
            break;
        }

        data->ready = 1;
        data->done = 0;

        // Wait for result
        while (data->done == 0) {
            usleep(1000);  // sleep 1ms
        }

        printf("Result: %d\n", data->result);
    }

    shmdt(data);

    return 0;
}
```

3.3 Consumer (calculator_server.c)

```
#include "shared.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int shmid = shmget(SHM_KEY, sizeof(struct shm_data), 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    struct shm_data *data = (struct shm_data *)shmat(shmid, NULL, 0);

    printf("Calculator server started.\n");

    while (1) {
        // Wait for data ready
        while (data->ready == 0) {
            usleep(1000);  // sleep 1ms
```

```
        }

        if (data->operator == 'q') {
            printf("Quit signal received. Exiting.\n");
            break;
        }

        printf("Received: %d %c %d\n", data->operand1, data->operator, data->operand2);

        switch (data->operator) {
            case '+':
                data->result = data->operand1 + data->operand2;
                break;
            case '-':
                data->result = data->operand1 - data->operand2;
                break;
            case '*':
                data->result = data->operand1 * data->operand2;
                break;
            case '/':
                if (data->operand2 != 0)
                    data->result = data->operand1 / data->operand2;
                else
                    data->result = 0;  // Error handling: divide by zero
                break;
            default:
                data->result = 0;
                break;
        }

        data->ready = 0;
        data->done = 1;
    }

    shmdt(data);
    shmctl(shmid, IPC_RMID, NULL);  // 공유 메모리 삭제

    return 0;
}
```

4. 실행 순서

컴파일

```
gcc calculator_client.c -o client
gcc calculator_server.c -o server
```

실행

- 터미널1:

```
./server
```

- 터미널2:

```
./client
```

예시 동작

```
터미널2 (client):

Enter operand1 operator operand2 (ex: 3 + 5), or 0 q 0 to quit:
3 + 5
Result: 8

Enter operand1 operator operand2 (ex: 3 + 5), or 0 q 0 to quit:
7 * 9
Result: 63
```

```
터미널1 (server):

Calculator server started.
Received: 3 + 5
Received: 7 * 9
```

---

5. 주의 사항

- **ready / done 플래그를 이용한 단순 동기화** → 다중 프로세스 환경에서는 **Semaphore**로 개선 필요
- 이 예제는 매우 기본 구조 → 실전에서는 반드시 **race condition 방지** 고려해야 함
- **shmctl(IPC_RMID)** 호출로 서버 종료 시 공유 메모리 삭제 수행 (안 하면 시스템에 남아 있음 → `ipcs -m`으로 확인 가능)

---

6. 결론 🚀

구성 요소	역할
Shared Memory	계산 데이터 및 결과 공유
ready flag	producer → consumer 알림
done flag	consumer → producer 알림
동작 방식	client는 계산 요청 / server는 결과 응답

이런 방식으로 **고속 IPC 기반 계산기 서비스**를 쉽게 구현 가능하다.

`signal()`로 인터럽트 처리 프로그램

1. 목표

- **시그널 핸들러를 등록**해서 **Ctrl+C (SIGINT)** 시그널을 처리
- 기본적으로 **Ctrl+C 누르면 프로세스가 종료**되지만,
  `signal()`을 사용하면 우리가 원하는 동작을 수행할 수 있다.
- 예제에서는 SIGINT 수신 시 **"인터럽트 발생!" 메시지 출력** 후 계속 동작하도록 만들어볼게.

---

2. 기본 구조

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int signo) {
    printf("\n인터럽트 발생! (SIGINT)\n");
}

int main() {
    // SIGINT 핸들러 등록
    signal(SIGINT, sigint_handler);

    printf("프로그램 실행 중... (Ctrl+C로 인터럽트 시그널 테스트)\n");

    while (1) {
        printf("작업 수행 중...\n");
        sleep(2);
    }

    return 0;
}
```

---

3. 동작 설명

- `signal(SIGINT, sigint_handler);` → **SIGINT 발생 시 sigint_handler() 호출하도록 등록**
- 사용자 Ctrl+C 입력 → 커널이 프로세스에 SIGINT 전달 → sigint_handler() 실행
- 프로그램은 종료되지 않고 계속 동작 (원래는 Ctrl+C 누르면 바로 종료됨)

---

4. 실행

컴파일

```
gcc sigint_example.c -o sigint_example
```

실행

```
./sigint_example
```

터미널 출력 예시

```
프로그램 실행 중... (Ctrl+C로 인터럽트 시그널 테스트)
작업 수행 중...
작업 수행 중...
작업 수행 중...

^C
인터럽트 발생! (SIGINT)
작업 수행 중...
작업 수행 중...
...
```

→ Ctrl+C 눌러도 프로그램 종료 안 됨 → 대신 핸들러 동작

---

5. 확장: 여러 번 인터럽트 시 종료

예를 들어 **처음 2번은 메시지만 출력**, **3번째 Ctrl+C 시 종료**하도록 만들 수도 있다:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t count = 0;

void sigint_handler(int signo) {
    count++;
    printf("\n인터럽트 발생! (SIGINT), count = %d\n", count);

    if (count >= 3) {
        printf("3번 인터럽트 수신 → 프로그램 종료\n");
        _exit(0);  // 안전한 종료
    }
}

int main() {
    signal(SIGINT, sigint_handler);

    printf("프로그램 실행 중... (Ctrl+C로 인터럽트 시그널 테스트)\n");
```

```
21
22      while (1) {
23          printf("작업 수행 중...\n");
24          sleep(2);
25      }
26
27      return 0;
28  }
```

동작 예시

```
1  사작업 수행 중...
2  작업 수행 중...
3  ^C
4  인터럽트 발생! (SIGINT), count = 1
5  작업 수행 중...
6  ^C
7  인터럽트 발생! (SIGINT), count = 2
8  작업 수행 중...
9  ^C
10 인터럽트 발생! (SIGINT), count = 3
11 3번 인터럽트 수신 → 프로그램 종료
```

---

6. 정리 🚀

요소	설명
`signal()`	시그널 핸들러 등록
`SIGINT`	Ctrl+C → SIGINT 발생
핸들러 함수	시그널 수신 시 커널이 호출해줌
응용	안전한 인터럽트 처리, 사용자 정의 동작 수행 등

`signal()` **+ SIGINT 핸들링**은:

- ✅ 안전한 프로그램 종료
- ✅ 프로세스 상태 저장 후 종료
- ✅ "정지 금지" 기능 구현 (임베디드 등에서 많이 사용)
- ✅ 인터랙티브 프로그램 제어 등에서 매우 유용하다.