Lecture 13

리눅스 프로세스 간의 통신

리뷰

- 스레드(thread)
  - 프로세스 내에서 실행되는 흐름의 단위
  - 하나의 프로세스는 하나 이상의 스레드를 가질 수 있음
  - 각 스레드는 동일한 프로세스 내의 다른 스레드와 코드, 데이터, 힙 영역을 공유함
  - POSIX 스레드(pthread) 라이브러리를 사용하여 구현할 수 있음
    - 스레드 생성과 종료: `pthread_create()`와 `pthread_exit()`
    - 스레드 동기화: 뮤텍스(mutex), 조건 변수
    - 스레드 관리: `pthread_join()`와 `pthread_detach()`
    - 스레드 속성 제어
  - GCC와의 컴파일

```
gcc infile.c -o outfile -pthread
```

리뷰

- 동기화: 안전적인 공유 자원 접근 방법
  - 뮤텍스(mutex): 초기화 → 자원 접근 전 잠금 → 자원 접근 후 해제 → 사용 종료 시 뮤텍스 해제
  - 조건 변수
    - 공유 자원 접근 전 뮤텍스 잠금
    - 원하는 조건이 만족하지 않으면 뮤텍스 해제 후 대기
    - 다른 스레드가 조건을 만족시키면 대기 중인 스레드를 깨움
    - 깨어난 스레드는 다시 뮤텍스를 획득하고 조건 검사 후 작업 실행
  - 세마포어(semaphore)
    - 초기화 → 자원 접근 전 잠금 → 자원 접근 후 해제 → 사용 종료 시 세마포어 해제
    - 뮤텍스와 달리 동시에 여러 스레드가 동시에 공유 자원을 접근할 수 있음

리뷰

- 멀티스레싱 관련 이슈
  - 스레드 안전성(thread safety)
    - 스레드 안전하지 않은 함수는 동시에 호출 시 레이스 컨디션, 잘못된 결과 발생 가능
    - 스레드 안전성을 위해 재진입 함수(reentrant function) 사용 권장
  - 교착 상태(deadlock)
    - 두 개 이상의 스레드/프로세스가 서로 가진 자원을 기다리면서 무한 대기 상태에 빠지는 문제
    - 자원 획득 순서 규칙, 타임아웃 설정 등 해결 가능
  - 기아 상태(starvation)
    - 특정 스레드/프로세스 자원을 요청했는데, 다른 스레드를 때문에 계속 우선순위에서 밀려나 실행 기회를 못 얻는 문제
    - 공정한 스케줄링 정책 적용, 세마포어나 뮤텍스에서 fair lock 사용 등 해결 가능

리뷰

- 소켓(socket)
  - 네트워크 통신을 위한 엔드포인트
  - 파일 읽기/쓰기와 동일한 방식으로 I/O가 가능함
- 비동기식 I/O(asynchronous I/O)
  - 프로그램이 I/O를 기다리며 멈추지 않고, 다른 작업을 계속 수행할 수 있는 방식
  - CPU 시간을 낭비하지 않고 효율적 처리 가능

IPC(Inter-Process Communication)

- 각 프로세스는 독립적인 자원을 가지고 있어서 스레드처럼 통신이 안 됨
- 프로세스 간의 통신(Inter-Process Communication, IPC)
  - 시그널(signal)
  - 파이프(pipe)
  - FIFO 큐(FIFO queue)
  - 공유 메모리
  - 소켓(socket)

Signal

- 커널 또는 다른 프로세스가 비동기적으로 프로세스에 **이벤트를 전달**하는 메커니즘
- 일종의 **소프트웨어 인터럽트**로 볼 수 있음
- 특징
  - 단방향 통신이며, 간단한 알림(이벤트 전달) 용으로 사용함
  - 데이터 전달보다는 특정 동작 요청(종료, 중단, 재시작 등)에 사용함

시그널	의미	기본 동작
SIGKILL	강제 종료	즉시 종료
SIGSTOP	일시 중단	중지
SIGCONT	중단 이후 재개	실행 재개
SIGUSR1, SIGUSR2	사용자 정의	프로그래머가 정의 가능

Signal

- 시그널 사용의 일반적인 흐름
  - 시그널 동록
    - `signal()` 또는 `sigaction()`을 사용하여 프로세스가 시그널을 받을 때 실행할 함수(핸들러, handler)를 등록함
  - 시그널 발생
    - 사용자 입력, 다른 프로세스로부터 시그널 전송, 타이머 등 시스템 이벤트로 자동 전달 등
  - 시그널 대기 및 수신
    - 커널은 시그널을 대기열에 전달하고, 프로세스가 실행 가능한 시점에 처리함
    - 시그널이 블록되어 있다면 즉시 처리하지 않고 보류 상태가 됨
  - 시그널 처리
    - 커널이 등록된 시그널 핸들러를 호출함
    - 핸들러가 등록되지 않으면, 기본 동작(종료, 무시, 중단 등)이 수행됨

Signal

- `void (*signal(int sig, void(*handler)(int)))(int);`
  - 특정 시그널 `sig`에 대해, 해당 시그널이 발생했을 때 실행할 핸들러 `handler`를 등록함
  - 성공 시 설정되어 있던 시그널 핸들러의 함수 포인터, 실패 시 `SIG_ERR`를 리턴함
- `int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);`
  - 시그널에 대한 동작(핸들러, 옵션 등)을 정밀하게 지정할 수 있음
  - `act`는 새로 설정할 시그널 동작 구조체 포인터, `oldact`는 이전 시그널 동작을 저장할 포인터
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

Signal

- `int kill(pid_t pid, int sig);`
  - `pid`로 지정된 프로세스(또는 프로세스 그룹)에 시그널을 보냄
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int raise(int sig);`
  - 자기 자신(현재 프로세스)에 시그널을 보냄
  - `kill(getpid(), sig)`와 동일함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int pause();`
  - 프로세스를 일시 중단하고, 시그널이 도착할 때까지 대기함
  - 시그널이 처리된 후 `pause()`는 항상 `-1`을 리턴함

Signal

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

// 시그널 처리 함수 정의
void signal_handler(int sig) {
    printf("\nSignal %d occurs! (SIGINT: Ctrl+C)\n", sig);
    printf("Program stopped.\n");
    exit(0); // 정상 종료
}

int main() {
    // SIGINT 시그널 발생 시 실행할 핸들러 등록
    if (signal(SIGINT, signal_handler) == SIG_ERR) {
        perror("signal error");
        return 1;
    }
    printf("Program running... (Press Ctrl+C for stop)\n");
    // 무한 루프로 대기 (시그널을 기다림)
    while (1) {
        printf("Running...\n");
        sleep(2); // 2초 대기
    }
    return 0;
}
```

```
Program running... (Press Ctrl+C for stop)
Running...
Running...
Running...
Running...
Running...
Running...
Running...

Signal 2 occurs! (SIGINT: Ctrl+C)
Program stopped.
```

Pipe

- 부모-자식 관계의 프로세스 간에 단방향 통신 채널을 제공
- 파일처럼 읽기/쓰기 가능하지만, 메모리에 존재하는 버퍼를 사용함
- 특징
  - 단방향 통신(한쪽이 쓰면, 다른 한쪽이 읽음)
  - 관련된 프로세스(부모-자식) 간에만 사용 가능
  - FIFO 구조(먼저 쓴 데이터가 먼저 읽힘)

Pipe

- 파이프 사용의 일반적인 흐름
  - **파이프 생성**
    - `pipe()` 함수를 호출하여 파이프 생성
    - 커널은 읽기 용 및 쓰기 용 두 개의 파일 디스크립터를 준비해줌
  - **프로세스 생성**
    - `pipe()`로 만든 파이프는 `fork()` 이 후 자식 프로세스에 복사되므로, 부모와 자식 프로세스 모두 사용할 수 있게 됨
  - **사용하지 않는 끝을 닫기**
    - 부모와 자식은 각각 자신의 역할에 맞는 끝을 닫음
      - 데이터를 보내는 쪽은 읽기 끝을 닫음
      - 데이터를 받는 쪽은 쓰기 끝을 닫음
    - 끝을 닫지 않으면 무한 대기에 걸릴 수 있음
  - **데이터 송수신**
    - `read()/write()` 함수로 데이터 읽기/쓰기 수행
    - 파이프가 가득 차면 대기 됨
  - **모든 데이터 전송 후 파이프 닫기**
    - 송신 측에서 더 이상 데이터를 쓸 필요가 없어지면 `close()` 호출
    - 수신 측은 `read()`가 `0`을 리턴하면 `EOF`로 간주하고 종료
    - 모든 파일 디스크립터를 닫아 자원 회수

Pipe

- `int pipe(int fd[2]);`
  - 프로세스 간 단방향 통신을 위한 파이프를 생성하고, 일기 용(`fd[0]`)과 쓰기 용(`fd[1]`) 파일 디스크립터를 배열 `fd`에 채워 넣음
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `pid_t fork(void);`
  - 현재 프로세스를 복제하여 자식 프로세스를 생성함
  - 파이프는 `fork()` 후 부모와 자식 프로세스가 같은 파이프를 공유하므로, 이 두 프로세스 간 통신이 가능해짐
  - 부모 프로세스에서는 자식의 `PID`(양수), 자식 프로세스에서는 `0`, 실패 시 `-1`을 리턴함

Pipe

- `ssize_t write(int fd, const void *buf, size_t count);`
  - 버퍼 `buf`에 저장된 데이터를 `count` 바이트만큼 파이프의 쓰기 끝(`fd`)에 기록함
  - 성공 시 실제로 쓰여진 바이트 수, 실패 시 `-1`을 리턴함
- `ssize_t read(int fd, void *buf, size_t count);`
  - 파이프의 읽기 끝(`fd`)에서 데이터를 읽어 버퍼 `buf`에 저장함
  - 성공 시 읽은 바이트 수, 쓰기 끝이 닫혔고 읽을 데이터가 없을 때 `0`, 실패 시 `-1`을 리턴함
- `int close(int fd);`
  - 주어진 파일 디스크립터 `fd`를 닫고, 해당 자원을 해제함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

Pipe

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

int main() {
    int fd[2];
    pid_t pid;
    char msg[] = "Hello from parent";
    char buffer[100];

    pipe(fd); // 파이프 생성
    pid = fork(); // 자식 프로세스 생성

    if (pid == 0) { // 자식 프로세스
        close(fd[1]); // 쓰기 끝 닫기
        read(fd[0], buffer, sizeof(buffer)); // 파이프의 데이터를 읽음
        printf("Children side (received message): %s\n", buffer);
        close(fd[0]); // 읽기 끝 닫기
    } else { // 부모 프로세스
        close(fd[0]); // 읽기 끝 닫기
        write(fd[1], msg, strlen(msg) + 1); // 메시지 전송
        close(fd[1]); // 쓰기 끝 닫기
    }

    return 0;
}
```

```
dat@dat-VirtualBox:~/Downloads/test_C_pipe$ gcc -Wall test.c -o test
dat@dat-VirtualBox:~/Downloads/test_C_pipe$ ./test
Children side (received message): Hello from parent
```

FIFO queue

- 파이프의 확장판으로, 이름(파일 시스템 상의 경로)을 가진 파이프
- 서로 무관한 프로세스 간에도 통신 가능
- 특징
  - `mkfifo()`로 생성
  - 데이터는 FIFO 구조로 전달
  - 파일 시스템에 경로가 존재하므로 명시적 식별자 사용 가능

FIFO queue

- FIFO 큐 사용의 일반적인 흐름
  - **FIFO 생성**
    - `mkfifo()` 또는 `mknod()` 시스템 콜을 통해 파일 시스템 상에 파이프 파일을 생성함
    - FIFO 파일일 실제로 존재하는 경로를 통해 다른 프로세스가 연결할 수 있음
  - **쓰기 측과 읽기 측 실행**
    - 서로 다른 두 프로세스가 동일한 FIFO 파일 경로를 사용하여 통신을 시작
    - `open()` 함수로 파일을 열고 나서 `write()/read()` 통해 데이터 전송/수신
  - **FIFO 파일 열기**
    - FIFO 파일은 일반 파일처럼 `open()`를 통해 접근하지만, 읽기와 쓰기 동작이 상대방이 준비될 때까지 대기 됨
  - **데이터 송수신**
    - 쓰기 측 프로세스는 `write()`를 호출하여 FIFO에 데이터 쓰기
    - 읽기 측 프로세스는 `read()`를 호출하여 FIFO에서 순서대로 데이터 읽기
  - **FIFO 닫기**
    - 통신을 마친 후 각 프로세스는 열었던 FIFO 파일을 닫음
    - FIFO를 닫는다고 파일 자체가 삭제되지 않음
  - **FIFO 제거**
    - 시스템에 FIFO 파일을 더 이상 남기고 싶지 않다면 `unlink()`를 사용하여 삭제함

FIFO queue

- `int mkfifo(const char *pathname, mode_t mode);`
  - 파일 시스템 경로 `pathname`에 FIFO 특수 파일을 생성함
  - `mode`를 통해 FIFO의 접근 권한을 지정함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int open(const char *pathname, int flags);`
  - 지정된 `pathname`에 위치한 FIFO 파일을 열어 읽기 또는 쓰기 동작을 준비함
  - `flag`를 통해 열기 모드 지정
    - `O_RDONLY`: 읽기 전용
    - `O_WRONLY`: 쓰기 전용
    - `O_NONBLOCK`: 비차단 모드
  - 성공 시 열린 파일의 디스크립터, 실패 시 `-1`을 리턴함

FIFO queue

- `ssize_t write(int fd, const void *buf, size_t count);`
  - FIFO의 쓰기 끝(`fd`)에 버퍼 `buf`에 저장된 데이터를 `count` 바이트만큼 씀
  - 성공 시 실제로 쓰여진 바이트 수, 실패 시 `-1`을 리턴함
- `ssize_t read(int fd, void *buf, size_t count);`
  - FIFO의 읽기 끝(`fd`)에서 `count` 바이트만큼 데이터를 읽어와 `buf`에 저장함
  - 성공 시 읽은 바이트 수, 쓰기 측이 닫히고 더 이상 데이터 없을 때 `0`, 실패 시 `-1`을 리턴함

FIFO queue

- `int close(int fd);`
  - 파일 디스크립터 `fd`를 닫아, 열려 있던 FIFO의 읽기/쓰기 끝을 종료함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int unlink(const char *pathname);`
  - 파일 시스템에서 지정된 FIFO 파일을 제거함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

FIFO queue

쓰기 측

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

int main() {
    int fd;
    char *fifo = "/tmp/myfifo";
    mkfifo(fifo, 0666); // FIFO 파일 생성

    char msg[] = "Hello FIFO!";
    // 쓰기 모드로 열기 (읽기 쪽이 열릴 때까지 대기)
    fd = open(fifo, O_WRONLY);
    write(fd, msg, sizeof(msg)); // 데이터 전송
    close(fd); // 파일 닫기

    return 0;
}
```

읽기 측

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

int main() {
    int fd;
    char *fifo = "/tmp/myfifo";
    char buffer[100];

    fd = open(fifo, O_RDONLY); // 읽기 모드로 열기
    read(fd, buffer, sizeof(buffer)); // FIFO에서 데이터 수신
    printf("Read data: %s\n", buffer);
    close(fd); // 파일 닫기
    unlink(fifo); // FIFO 파일 삭제

    return 0;
}
```

```
dat@dat-VirtualBox:~/Downloads/test_C_FIFO$ ./writer | ./reader
Read data: Hello FIFO
```

공유 메모리

- 여러 프로세스가 공통된 메모리 영역을 직접 접근할 수 있게 함
- 가장 빠른 IPC 방식
- 특징
  - 커널이 관리하는 공유 메모리 세그먼트를 매핑하여 사용
  - 빠르지만, 동기화 문제가 일어날 수 있음
  - 일반적으로 세마포어와 함께 사용함

공유 메모리

- 공유 메모리 사용의 일반적인 흐름
  - 공유 메모리 식별자 생성
    - 공유 메모리 영역을 커널에 요청함
  - 프로세스 주소 공간에 연결
    - 생성된 공유 메모리 영역을 프로세스의 주소 공간에 연결함
  - 데이터 접근 및 조작
    - 공유 메모리 주소를 통해 프로세스 간 데이터를 주고받음
    - 공유 메모리를 사용하는 여러 프로세스는 동일한 주소 영역을 참조하므로, 쓰기/읽기 즉시 반영됨
  - 공유 메모리 분리
    - 작업이 끝나면 연결한 공유 메모리 주소를 프로세스 주소공간에서 분리함
  - 공유 메모리 제거
    - 더 이상 필요하지 않으면 공유 메모리를 커널에서 완전히 제거함

공유 메모리

- `key_t ftok(const char *pathname, int proj_id);`
  - IPC를 위한 고유 키값을 생성하며, 이 키는 공유 메모리나 세마포어, 메시지 큐에서도 공통 식별자로 사용됨
  - 성공 시 생성된 고유 키값, 실패 시 `(key_t)-1`을 리턴함
- `int shmget(key_t key, size_t size, int shmflg);`
  - 커널로부터 공유 메모리 세그먼트를 생성하거나, 기존 세그먼트를 가져옴
  - 성공 시 공유 메모리 식별자(`shmid`), 실패 시 `-1`을 리턴함
- `void *shmat(int shmid, const void *shmaddr, int shmflg);`
  - 공유 메모리 세그먼트를 현재 프로세스의 주소 공간에 연결함
  - 성공 시 프로세스 주소 공간 중 공유 메모리가 연결된 주소 포인터, 실패 시 `(void*)-1`을 리턴함

공유 메모리

- `int shmdt(const void *shmaddr);`
  - 프로세스 주소 공간에 연결된 공유 메모리를 분리함
  - 성공 시 0, 실패 시 -1을 리턴함
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
  - 공유 메모리 세그먼트에 대해 제어 명령을 수행함
  - 세그먼트 정보 조회, 제거, 퍼미션 변경 등을 수행할 수 있음
  - 성공 시 0, 실패 시 -1을 리턴함

공유 메모리

쓰기 측

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

int main() {
    key_t key = ftok("shmfile", 65); // 키 생성
    // 공유 메모리 생성
    int shmid = shmget(key, 1024, 0666|IPC_CREAT);
    // 메모리 연결
    char *data = (char*)shmat(shmid, NULL, 0);

    // 데이터 쓰기
    strcpy(data, "Hello Shared Memory");
    printf("Writer: data saved\n");

    shmdt(data); // 연결 해제
    return 0;
}
```

읽기 측

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

int main() {
    key_t key = ftok("shmfile", 65); // 동일 키 생성
    // 기존 공유 메모리 접근
    int shmid = shmget(key, 1024, 0666);
    // 연결
    char *data = (char*)shmat(shmid, NULL, 0);

    // 데이터 읽기
    printf("Reader: Read data -> %s\n", data);

    shmdt(data); // 연결 해제
    shmctl(shmid, IPC_RMID, NULL); // 커널에서 삭제

    return 0;
}
```

```
dat@dat-VirtualBox:~/Downloads/test_C_shared_memory$ ./writer
Writer: data saved
dat@dat-VirtualBox:~/Downloads/test_C_shared_memory$ ./reader
Reader: Read data -> Hello Shared Memory
dat@dat-VirtualBox:~/Downloads/test_C_shared_memory$
```

세마포어

- 공유 자원에 대한 접근을 동기화하기 위한 메커니즘
- 임계 구역(critical section)에 동시 접근을 제어
- 특징
  - 초기값으로 허용 가능한 접근 횟수 지정
  - `P()` (wait, down): 자원 잠금
  - `V()` (signal, up): 자원 해제

세마포어

- 세마포어 사용의 일반적인 흐름
  - 세마포어 식별자 생성
    - 세마포어 집합을 커널로부터 요청하거나 기존 세마포어를 자져옴
  - 초기값 설정
    - 세마포어를 새로 만든 경우, 자원을 제어할 초기값(대기 가능 횟수)을 설정함
  - 자원 사용 전 획득
    - 공유 자원을 사용하기 전에 세마포어를 감소(P 연산) 시킴
    - 세마포어 값이 0이면 다른 프로세스가 자원을 사용 중이므로 대기
  - 자원 사용
    - 한 프로세스만 자원에 접근하도록 함
  - 자원 사용 후 해제
    - 자원 사용이 끝나면 사마포어를 증가(V 연산) 시켜 다른 프로세스가 접근할 수 있게 함
  - 세마포어 제거
    - 모든 작업이 끝난 뒤, 세마포어를 커널에서 원전히 제거함

세마포어

- `key_t ftok(const char *pathname, int proj_id);`
  - 세마포어나 공유 메모리, 메시지 큐에서 사용할 고유 IPC 키를 생성함
  - 프로세스들이 같은 경로(`pathname`)와 프로젝트 ID(`proj_id`)를 지정하면 동일한 키값이 생성되어, 같은 자원을 공유할 수 있음
  - 성공 시 생성된 고유 키, 실패 시 `(key_t)-1`을 리턴함
- `int semget(key_t key, int nsems, int semflg);`
  - 커널로부터 세마포어 집합을 생성하거나 기존 것을 가져옴
  - 성공 시 세마포어 식별 ID(`semid`), 실패 시 `-1`을 리턴함
- `int semctl(int semid, int semnum, int cmd, union semun arg);`
  - 세마포어의 속성을 제어하거나 상태를 확인하거나 제거함
  - 성공 시 명령에 따른 상수 값, 실패 시 `-1`을 리턴함

세마포어

- `int semop(int semid, struct sembuf *sops, size_t nsops);`
  - 세마포어의 값을 증가(V 연산) 하거나 감소(P 연산) 하여 자원 접근을 제어하는 실제 동작을 수행함
  - 성공 시 0, 실패 시 -1을 리턴함

세마포어

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

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

int main() {
   key_t key = ftok("semfile", 65);
   int semid = semget(key, 1, 0666 | IPC_CREAT);

   // 초기값 설정
   union semun arg;
   arg.val = 1;
   semctl(semid, 0, SETVAL, arg);

   struct sembuf p = {0, -1, 0}; // P 연산
   struct sembuf v = {0, +1, 0}; // V 연산

   while (1) {
      semop(semid, &p, 1); // 진입 (잠금)
      printf("Enter critical section: PID=%d\n", getpid());
      sleep(2); // 자원 사용 중
      printf("Leave critical section: PID=%d\n", getpid());
      semop(semid, &v, 1); // 해제 (잠금 해제)
      sleep(1);
   }

   semctl(semid, 0, IPC_RMID); // 세마포어 삭제
   return 0;
}
```

```
dat@dat-VirtualBox:~/Downloads/test_C_semaphore$ ./test
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
Enter critical section: PID=4147
Leave critical section: PID=4147
```

소켓

- 네트워크를 통한 프로세스 간 통신 수단
- 같은 시스템 내 또는 다른 시스템 간 모두 사용 가능
- 특징
  - 다양한 통신 프로토콜 지원
  - 프로세스 간 통신과 네트워크 통신이 모두 가능
  - 파일 디스크립터 기반으로 입출력 처리

소켓

- 소켓 사용의 일반적인 흐름
  - 서버 측
    - 소켓 생성: 서버 통신을 위한 소켓을 생성함
    - 주소 할당: 소켓에 IP 주소와 포트 번호를 연결함
    - 연결 대기: 클라이언트의 요청을 받을 수 있도록 대기 상태로 전환함
    - 연결 수락: 클라이언트의 연결 요청을 받아들이고, 통신 용 새 소캣을 생성함
    - 데이터 송수신: 클라이언트와 데이터를 조고받음
    - 소켓 종료: 통신이 끝나면 소켓을 닫음
  - 클라이언트 측
    - 소켓 생성: 클라이언트도 통신용 소켓을 생성함
    - 서버에 연걸: 서버의 IP 주소와 포트로 연결 요청을 보냄
    - 데이터 송수신: 연결이 성공하면 서버와 데이터를 주고받음
    - 소켓 종료: 통신이 끝나면 소켓을 닫음

소켓

- `int socket(int domain, int type, int protocol);`
  - 새로운 소켓을 생성하여 소켓 디스크립터를 반환함
  - `domain`는 IPv4, IPv6, **로컬** 등의 통신 도메인을 지정함
  - 성공 시 새로 생성된 소켓 디스크립터, 실패 시 `-1`을 리턴함
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - 소켓에 IP 주소와 포트를 연결함
  - 서버 소켓에서 필수 단계임
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int listen(int sockfd, int backlog);`
  - 소켓을 수동 대기 상태로 설정하여, 클라이언트 연결 요청을 받을 준비함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

소켓

- `int accept(int domain, struct sockaddr *addr, socklen_t *addrlen);`
  - 클라이언트의 연결 요청을 수락하고, 통신 용 소켓 디스크립터를 반환함
  - 성공 시 새로 생성된 통신 용 소켓 디스크립터, 실패 시 `-1`을 리턴함
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - 클라이어트 측에서 서버의 IP와 포트로 연결을 시도함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int send(int sockfd, const void *buf, size_t len, int flags);`
  - TCP 연결을 통해 데이터를 전송함
  - 전송할 데이터는 `buf` 버퍼에 저장되며, `len` 바이트만큼 전송함
  - 성공 시 실제 전송한 바이트 수, 실패 시 `-1`을 리턴함

소켓

- `int recv(int sockfd, void *buf, size_t len, int flags);`
  - TCP 연결을 통해 데이터를 수신함
  - 수신한 데이터를 `buf`에 저장하며, `len` 바이트만큼 수신함
  - 성공 시 수신한 바이트 수, 연결 종료 시 `0`, 실패 시 `-1`을 리턴함
- `int close(int sockfd);`
  - 사용이 끝난 소켓을 닫아 시스템 자원을 해제함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

요약

방식	통신 방향	통신 범위	속도	데이터 양	동기화	주요 용도
시그널	단방향	시스템 내	빠름	매우 적음	불필요	이벤트 전달
파이프	단방향	부모-자식 간	빠름	적음	불필요	단순 데이터 전달
FIFO 큐	단방향	시스템 내	보통	적음	불필요	독립 프로세스 간
공유 메모리	양방향	시스템 내	빠름	많음	필요	대용량 데이터 공유
세마포어	-	시스템 내	빠름	적음	필수	동기화
소켓	양방향	로컬/네트워크	느림	많음	선택적	네트워크 통신