7. 멀티프로세스와 멀티스레드 서버

7.1 `fork()` 기반 다중 접속 서버

📘 개요

`fork()`는 현재 프로세스를 복제하여 자식 프로세스를 생성하는 시스템 콜이다. 네트워크 서버에서 이를 이용하면, 클라이언트 접속 시마다 별도의 자식 프로세스를 생성하여 병렬 처리할 수 있다. 이는 Unix 전통의 다중 접속 처리 방식이며, 안정성은 높지만 프로세스 생성 비용이 크다는 단점이 있다.

⚙️ 구조 개요

- 메인 서버 프로세스는 `socket()`, `bind()`, `listen()`을 수행한 뒤, 클라이언트 접속을 감지하면 `accept()` 호출
- `accept()` 이후 `fork()`로 자식 프로세스를 생성하여 클라이언트 요청을 처리
- 자식은 처리 종료 후 종료하며, 부모는 계속해서 새 클라이언트 요청을 대기

📐 아키텍처 다이어그램

```
            [Main Server Process]
                   ↓ listen
              ┌─────────────┐
              | accept()    |←┐
              └─────┬───────┘ | 반복
                    ↓         |
              ┌─────────────┐ |
              | fork()      | |
              └─────┬───────┘ |
             parent |      child
            continue|       ┌──────────────┐
                    |       | client 처리 |
                    ↓       └──────┬───────┘
                다음 연결          ↓
                                종료 (exit)
```

📚 예제 코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

#define PORT 8080
```

```
10
11  void handle_client(int client_sock) {
12      char buffer[1024];
13      int n;
14      while ((n = read(client_sock, buffer, sizeof(buffer))) > 0) {
15          write(client_sock, buffer, n); // echo
16      }
17      close(client_sock);
18  }
19
20  int main() {
21      int server_sock = socket(AF_INET, SOCK_STREAM, 0);
22
23      struct sockaddr_in addr;
24      addr.sin_family = AF_INET;
25      addr.sin_port = htons(PORT);
26      addr.sin_addr.s_addr = INADDR_ANY;
27
28      bind(server_sock, (struct sockaddr*)&addr, sizeof(addr));
29      listen(server_sock, 5);
30
31      // 좀비 프로세스 방지
32      signal(SIGCHLD, SIG_IGN);
33
34      while (1) {
35          int client_sock = accept(server_sock, NULL, NULL);
36
37          pid_t pid = fork();
38          if (pid == 0) {
39              // 자식 프로세스
40              close(server_sock); // 자식은 서버 소켓 필요 없음
41              handle_client(client_sock);
42              exit(0);
43          } else {
44              // 부모 프로세스
45              close(client_sock); // 부모는 client 소켓 닫기
46          }
47      }
48
49      close(server_sock);
50      return 0;
51  }
```

🔍 주요 특징 요약

항목	설명
처리 단위	접속당 프로세스 1개 생성
자원 격리	각 프로세스는 독립된 주소 공간 사용

항목	설명
안정성	개별 접속이 충돌 시 전체 서버 영향 없음
단점	프로세스 생성/해제 비용이 큼
개선 방안	`prefork`, `thread pool`, `epoll` 구조 도입 가능

🧠 실무 팁

- 반드시 `SIGCHLD`를 처리하거나 `signal(SIGCHLD, SIG_IGN)` 설정으로 좀비 프로세스를 방지해야 한다.
- `fork()`로 생성된 자식은 불필요한 리소스(예: 서버 소켓, 파일 디스크립터)를 반드시 정리해야 한다.
- 다수의 동시 접속이 예상될 경우 `fork()`는 성능 한계가 있으므로 `thread pool` 또는 `epoll` 기반 처리로의 확장이 필요하다.

7.2 `pthread` 기반 멀티스레드 서버

📘 개요

`pthread`(POSIX thread)는 리눅스 및 유닉스 계열에서 사용하는 스레드 라이브러리다. 네트워크 서버에서 `pthread`를 사용하면, 클라이언트 접속마다 별도의 스레드를 생성하여 병렬 처리할 수 있다. 이는 `fork()`보다 리소스 사용이 적고 빠르며, 메모리를 공유하므로 통신 성능이 우수하다.

🧱 설계 구조 요약

- 메인 서버는 `accept()`를 통해 클라이언트 소켓을 획득
- 해당 소켓을 새로운 `pthread`로 넘겨 클라이언트 처리 전담
- 각 스레드는 독립적으로 수신/응답 처리
- 종료 시 스레드 종료 및 리소스 정리 수행

📐 구조 흐름도

```
            [Main Server Thread]
                    ↓ accept
          ┌──────────────────────┐
          | client socket 획득 |
          └──────────┬───────────┘
                     ↓
           [pthread 생성]
                     ↓
        [Thread Function 실행]
             ↓          ↓
          recv()     send()
             ↓          ↓
         close(fd)   pthread_exit()
```

🧪 예제 코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080

void* client_handler(void* arg) {
    int client_sock = *(int*)arg;
    free(arg);

    char buffer[1024];
    int n;
    while ((n = read(client_sock, buffer, sizeof(buffer))) > 0) {
        write(client_sock, buffer, n); // echo
    }

    close(client_sock);
    pthread_exit(NULL);
}

int main() {
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(server_sock, (struct sockaddr*)&addr, sizeof(addr));
    listen(server_sock, 10);

    while (1) {
        int* client_sock = malloc(sizeof(int));
        *client_sock = accept(server_sock, NULL, NULL);

        pthread_t tid;
        pthread_create(&tid, NULL, client_handler, client_sock);
        pthread_detach(tid); // 자동 자원 회수
    }

    close(server_sock);
    return 0;
}
```

🔍 구조적 특징 요약

항목	설명
처리 단위	접속당 스레드 1개 생성
성능	`fork()` 보다 빠르고 자원 효율적
메모리 모델	동일 프로세스 공간 공유
주의사항	동시성 제어 필요 (mutex 등)
자원 해제	`pthread_detach()` 로 메모리 누수 방지

🧠 실무 고려사항

- 동시 클라이언트 수가 수천 단위 이상이면 `pthread` 보다는 **스레드 풀** 또는 `epoll` 기반 구조가 더 적합하다.
- 스레드 안전한 설계가 필수이며, 공유 자원에는 `mutex`, `semaphore` 등을 적용해야 한다.
- `pthread_join()` 대신 `pthread_detach()` 를 사용하면 불필요한 대기 없이 자동 종료된다.
- 커널에 따라 스레드는 내부적으로 경량 프로세스로 구현되며, 커널 스케줄링 비용이 존재한다.

7.3 `thread pool` 개념 및 구현

📘 개요

`Thread Pool`은 서버 성능과 안정성을 고려하여 **미리 생성된 제한된 수의 스레드 집합**을 이용해 클라이언트 요청을 처리하는 구조다.
필요할 때마다 스레드를 생성하는 `pthread` 방식은 접속 수가 증가할수록 오버헤드가 커지므로, 이를 제어하기 위한 효과적인 방법이 `thread pool`이다.

🧱 설계 개념 요약

- 서버 시작 시 일정 수의 스레드를 미리 생성
- 클라이언트 요청이 도착하면 **작업 큐(task queue)**에 작업을 등록
- 유휴 스레드 중 하나가 큐에서 작업을 꺼내 실행
- 스레드는 종료되지 않고 대기 상태로 돌아간다

📐 구성 요소

구성 요소	설명
작업 큐 (Task Queue)	처리할 클라이언트 소켓을 저장하는 FIFO 큐
워커 스레드 (Worker Thread)	큐에서 작업을 가져와 실행하고 다시 대기
동기화 객체	`mutex`, `condition variable` 등으로 큐 접근 보호

구성 요소	설명
스레드 풀 관리자	스레드 생성, 종료, 에러 감시 기능 담당

🧪 간단한 구현 예제

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <netinet/in.h>
#include <string.h>

#define PORT 8080
#define THREAD_COUNT 4
#define QUEUE_SIZE 10

int task_queue[QUEUE_SIZE];
int front = 0, rear = 0, count = 0;
pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queue_not_empty = PTHREAD_COND_INITIALIZER;

void enqueue(int client_sock) {
    pthread_mutex_lock(&queue_mutex);
    while (count == QUEUE_SIZE); // blocking when full (간단 구현)
    task_queue[rear] = client_sock;
    rear = (rear + 1) % QUEUE_SIZE;
    count++;
    pthread_cond_signal(&queue_not_empty);
    pthread_mutex_unlock(&queue_mutex);
}

int dequeue() {
    pthread_mutex_lock(&queue_mutex);
    while (count == 0)
        pthread_cond_wait(&queue_not_empty, &queue_mutex);
    int client_sock = task_queue[front];
    front = (front + 1) % QUEUE_SIZE;
    count--;
    pthread_mutex_unlock(&queue_mutex);
    return client_sock;
}

void* worker_thread(void* arg) {
    while (1) {
        int client_sock = dequeue();
        char buffer[1024];
        int n;
        while ((n = read(client_sock, buffer, sizeof(buffer))) > 0) {
            write(client_sock, buffer, n); // echo
        }
```

```
        close(client_sock);
    }
}

int main() {
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(server_sock, (struct sockaddr*)&addr, sizeof(addr));
    listen(server_sock, 10);

    pthread_t threads[THREAD_COUNT];
    for (int i = 0; i < THREAD_COUNT; i++)
        pthread_create(&threads[i], NULL, worker_thread, NULL);

    while (1) {
        int client_sock = accept(server_sock, NULL, NULL);
        enqueue(client_sock); // 요청을 큐에 등록
    }

    close(server_sock);
    return 0;
}
```

🔍 구조적 특징 요약

항목	설명
처리 방식	제한된 수의 스레드가 순차적으로 요청 처리
메모리 효율	동적 생성보다 적음
CPU 효율	context switching 최소화 가능
안정성	과도한 스레드 생성 방지
동시성 제어 필요	큐 보호를 위한 `mutex`, `cond` 필수
한계	워커 수보다 클라이언트 수가 많으면 대기 발생

🧠 실무 고려사항

- 큐 크기와 워커 수는 서버 성능과 사용 시나리오에 따라 조정해야 한다.
- `epoll` + `thread pool` 구조를 조합하면 고성능 대규모 서버 구현이 가능하다.
- 대기 중인 워커가 없을 때 처리 지연이나 거절 로직(`backpressure`)을 고려해야 한다.

- 스레드 풀의 상태(작업 수, 대기 중 스레드 수 등)를 모니터링하는 로직도 중요하다.

7.4 race condition, mutex, semaphore 사용법

📘 개요

멀티스레드 네트워크 서버 환경에서 여러 스레드가 공유 자원(예: 작업 큐, 전역 변수, 파일 등)을 동시에 접근하면 예기치 않은 결과를 초래할 수 있다. 이를 **Race Condition(경쟁 상태)**이라고 하며, 적절한 동기화 기법이 필요하다.

이를 해결하기 위한 대표적인 동기화 도구는 다음과 같다:

- **Mutex (Mutual Exclusion)**: 상호 배제를 통해 한 번에 하나의 스레드만 자원에 접근 가능하도록 보장
- **Semaphore**: 개수 기반의 접근 제어로, 특정 자원에 대해 동시 접근 가능한 스레드 수를 제한

🧨 Race Condition의 예시

```
int counter = 0;

void* thread_func(void* arg) {
    for (int i = 0; i < 100000; i++)
        counter++; // 동시 접근으로 인해 결과 불일치 가능성
    return NULL;
}
```

- 위 코드는 여러 스레드가 `counter++`를 동시에 실행하면서 중복된 읽기/쓰기 발생
- 결과적으로 기대값보다 적은 값이 출력될 수 있음

🔐 Mutex 사용법

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* thread_func(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

- `pthread_mutex_lock()`: 락 획득 (다른 스레드는 대기)
- `pthread_mutex_unlock()`: 락 해제
- **주의**: 락을 걸고 나서 `return` 또는 `exit`되면 데드락 위험 존재

🔄 Semaphore 사용법

```c
#include <semaphore.h>

sem_t sem;

void* thread_func(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&sem);   // 세마포어 획득
        counter++;
        sem_post(&sem);   // 세마포어 해제
    }
    return NULL;
}

int main() {
    sem_init(&sem, 0, 1); // 세마포어 초기값 1 (binary semaphore)
    // ...
}
```

- `sem_wait()` : 세마포어 값이 0이면 대기, 1 이상이면 값을 1 감소시키고 진입
- `sem_post()` : 세마포어 값을 1 증가 (대기 스레드 중 하나를 깨움)
- **Binary Semaphore (값 0 또는 1)**은 Mutex와 유사한 역할

📊 비교 요약표

항목	Mutex	Semaphore
기본 목적	상호 배제	동시 접근 제한
초기값	없음 (무조건 1 스레드)	정수 (N개의 동시 접근 허용)
사용 함수	`pthread_mutex_lock/unlock`	`sem_wait/post`
데드락 위험	있음	있음 (관리 미숙시)
주 용도	임계 영역 보호	제한된 자원 접근 (예: DB 커넥션)

🧠 실무 적용 팁

- `mutex`는 **단일 자원 보호**에 적합하며, 간결하고 빠르다.
- `semaphore`는 **N개 자원 제한** 혹은 **생산자/소비자 패턴** 구현에 적합하다.
- `pthread_mutex_trylock()`으로 **논블로킹 락 시도** 가능.
- `pthread_rwlock_t`를 사용하면 **읽기-쓰기 분리 동기화** 구현 가능.
- 다중 자원 락 순서가 꼬이면 데드락이 발생할 수 있으므로, **락 획득 순서 고정** 필요.

7.5 프로세스/스레드 간 통신 (pipe, socketpair, message queue 등)

📘 개요

멀티프로세스 또는 멀티스레드 프로그램에서 **데이터 공유와 동기화**를 위해 통신이 필요하다. 특히 네트워크 서버에서는 로그 처리, 백엔드 작업 분산, 프로세스 간 이벤트 전달 등 다양한 목적을 위해 **IPC(Inter-Process Communication)** 기법이 사용된다.

📚 대표 IPC 방식

방식	대상	특징	양방향	비고
pipe	프로세스 간	단방향, 부모-자식 간 사용	❌	`fork()` 후 공유됨
socketpair	프로세스/스레드 모두	양방향, 로컬 소켓 사용	✅	UNIX 도메인 소켓 기반
message queue	프로세스 간	커널 기반 메시지 큐	❌	키로 식별, 시스템 V API 사용
shared memory	프로세스 간	빠른 데이터 교환 가능	양방향	별도 동기화 필요
mutex/condition	스레드 간	동기화 목적 (데이터 전달 X)	❌	`pthread` 기반 스레드 제한

1️⃣ pipe: 단방향 통신 (익명 파이프)

예시 코드 (부모-자식 간 문자열 전달)

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

int main() {
    int fd[2]; // fd[0]: read, fd[1]: write
    pipe(fd);

    if (fork() == 0) {
        // 자식 프로세스
        close(fd[1]); // 쓰기 닫기
        char buf[100];
        read(fd[0], buf, sizeof(buf));
        printf("자식 수신: %s\n", buf);
    } else {
        // 부모 프로세스
        close(fd[0]); // 읽기 닫기
```

```
        char msg[] = "Hello from parent";
        write(fd[1], msg, strlen(msg) + 1);
    }
}
```

2️⃣ socketpair: 양방향 통신 (로컬 전용)

예시 코드

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

int main() {
    int sv[2];
    socketpair(AF_UNIX, SOCK_STREAM, 0, sv);

    if (fork() == 0) {
        close(sv[0]);
        char buf[100];
        read(sv[1], buf, sizeof(buf));
        printf("자식 수신: %s\n", buf);
        write(sv[1], "pong", 5);
    } else {
        close(sv[1]);
        write(sv[0], "ping", 5);
        char reply[100];
        read(sv[0], reply, sizeof(reply));
        printf("부모 수신: %s\n", reply);
    }
}
```

- `AF_UNIX`은 로컬 시스템 내 통신용 소켓 도메인
- TCP 소켓 없이도 안정적인 IPC 가능

3️⃣ System V 메시지 큐

특징

- `msgget`, `msgsnd`, `msgrcv` API 사용
- 메시지를 구조체 형태로 송수신
- 키 기반 식별자 사용

구조체 정의

```
struct msgbuf {
    long mtype;
    char mtext[100];
};
```

사용 예

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

struct msgbuf msg = {1, "Hello"};

int msgid = msgget(1234, IPC_CREAT | 0666);
msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0);
```

4️⃣ 스레드 간 통신: 공유 변수 + 동기화

멀티스레드에서는 공통 변수에 데이터를 저장하고 `mutex`, `condition`으로 접근을 조율함.

```
int shared_data;
pthread_mutex_t lock;

void* producer(void*) {
    pthread_mutex_lock(&lock);
    shared_data = 100;
    pthread_mutex_unlock(&lock);
}
```

> 직접적인 메시지 전송은 없지만 공유 메모리를 통해 통신이 가능하다.
> Race condition 방지를 위해 락 필수.

📌 IPC 방식 선택 기준 요약

상황	추천 방식
부모-자식 간 간단한 메시지 전달	`pipe`
양방향 통신 필요, fork 사용	`socketpair`
별도 프로세스 간 큐 형태 전달	`message queue`
다량의 데이터, 고속 처리	`shared memory`
동일 프로세스 내 스레드	공유 변수 + `mutex`