11. 실전 프로젝트

11.1 C로 만든 Mini Shell

🛠️ Mini Shell 구현 목표

목표 기능	설명
명령어 입력 받기	사용자 입력 처리 (`fgets` 등 사용)
명령어 파싱	공백 기준으로 파라미터 분리 (`strtok` 등 사용)
자식 프로세스 생성	`fork()`
명령어 실행	`execvp()` 사용
부모 프로세스 동기화	`wait()` 사용해서 자식 종료 기다리기
종료 지원	`exit` 입력 시 셸 종료

📁 전체 흐름

```
Mini Shell 시작
→ 사용자 입력 대기
→ 명령어 해석
→ 프로세스 fork
→ 자식에서 execvp로 실행
→ 부모는 wait()로 종료 대기
→ 반복
```

📄 전체 예제 코드 (mini_shell.c)

```c
// mini_shell.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_LINE 1024
#define MAX_ARGS 64

void parse_command(char *line, char **args)
{
    int i = 0;
    args[i] = strtok(line, " \t\r\n");

    while (args[i] != NULL && i < MAX_ARGS - 1)
```

```
    {
        i++;
        args[i] = strtok(NULL, " \t\r\n");
    }

    args[i] = NULL;
}

int main()
{
    char line[MAX_LINE];
    char *args[MAX_ARGS];
    pid_t pid;
    int status;

    while (1)
    {
        printf("mini-shell> ");
        fflush(stdout);

        if (fgets(line, sizeof(line), stdin) == NULL)
        {
            break; // EOF (Ctrl+D) 처리
        }

        // 입력 공백 시 무시
        if (line[0] == '\n')
            continue;

        // "exit" 명령어 처리
        if (strncmp(line, "exit", 4) == 0)
        {
            printf("Exiting mini-shell\n");
            break;
        }

        parse_command(line, args);

        pid = fork();

        if (pid < 0)
        {
            perror("fork error");
            continue;
        }
        else if (pid == 0)
        {
            // 자식 프로세스
            if (execvp(args[0], args) == -1)
            {
                perror("execvp error");
            }
            exit(EXIT_FAILURE);
```

```
        }
        else
        {
            // 부모 프로세스
            do
            {
                waitpid(pid, &status, WUNTRACED);
            } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        }
    }

    return 0;
}
```

🗂️ 빌드 방법

```
$ gcc -o mini_shell mini_shell.c
```

🚀 실행

```
$ ./mini_shell
mini-shell> ls -l
... (ls 결과 출력)

mini-shell> pwd
... (pwd 결과 출력)

mini-shell> echo Hello World
Hello World

mini-shell> exit
Exiting mini-shell
```

✅ 주요 기능 설명

기능	코드 위치
명령어 입력	`fgets(line, sizeof(line), stdin)`
명령어 파싱	`parse_command()` 함수
자식 프로세스 생성	`fork()`
명령어 실행	`execvp()`
자식 종료 기다림	`waitpid()`
종료 지원	`"exit"` 명령어 처리

🔍 확장 아이디어

기능	설명
백그라운드 실행 지원 (`&`)	`ls -l &` 형태 지원
파이프 (`|`) 지원	
리다이렉션 지원 (`>`, `<`)	`ls > out.txt`
history 기능 추가	입력 기록 저장
tab 자동완성	readline 라이브러리 사용

✅ Mini Shell 실습 체크리스트

- 명령어 입력 → 실행 성공
- 자식 프로세스 → fork/exec 정상 동작 확인
- 부모 → wait 로 동기화
- exit 명령어로 정상 종료

11.2 파일 동기화 도구

🗺️ 전체 목표 흐름

```
[Source Directory] → 비교 → [Target Directory]

- 새 파일 → 복사
- 수정된 파일 → 덮어쓰기
- 삭제된 파일 → (옵션: 삭제)
```

🛠️ 기능 명세

기능	설명
디렉토리 순회	`opendir()`, `readdir()` 사용
파일 상태 확인	`stat()`, `lstat()` 사용
파일 복사	`read()` / `write()` 기반으로 직접 구현
파일 비교 기준	`mtime` (수정 시간)
옵션	삭제 동기화는 옵션으로 구현 가능

예제 코드 (sync_files.c)

> 이번 코드는 **단일 디렉토리 레벨**에서 동작하는 **기본 버전**이야.
> (하위 디렉토리 재귀 탐색은 이후에 추가 가능)

```c
// sync_files.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#define BUF_SIZE 4096

void copy_file(const char *src_path, const char *dst_path)
{
    int src_fd = open(src_path, O_RDONLY);
    if (src_fd < 0)
    {
        perror("open src");
        return;
    }

    int dst_fd = open(dst_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dst_fd < 0)
    {
        perror("open dst");
        close(src_fd);
        return;
    }

    char buf[BUF_SIZE];
    ssize_t bytes;

    while ((bytes = read(src_fd, buf, BUF_SIZE)) > 0)
    {
        write(dst_fd, buf, bytes);
    }

    close(src_fd);
    close(dst_fd);

    printf("Copied: %s -> %s\n", src_path, dst_path);
}

void sync_directories(const char *src_dir, const char *dst_dir)
{
    DIR *dir = opendir(src_dir);
    if (!dir)
```

```
    {
        perror("opendir");
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL)
    {
        // Skip . and ..
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;

        char src_path[1024], dst_path[1024];
        snprintf(src_path, sizeof(src_path), "%s/%s", src_dir, entry->d_name);
        snprintf(dst_path, sizeof(dst_path), "%s/%s", dst_dir, entry->d_name);

        struct stat src_stat, dst_stat;
        if (stat(src_path, &src_stat) < 0)
        {
            perror("stat src");
            continue;
        }

        int dst_exists = (stat(dst_path, &dst_stat) == 0);

        // 파일인 경우 처리
        if (S_ISREG(src_stat.st_mode))
        {
            int need_copy = 0;

            if (!dst_exists)
            {
                need_copy = 1;
                printf("New file: %s\n", src_path);
            }
            else if (src_stat.st_mtime > dst_stat.st_mtime)
            {
                need_copy = 1;
                printf("Updated file: %s\n", src_path);
            }

            if (need_copy)
            {
                copy_file(src_path, dst_path);
            }
        }
    }

    closedir(dir);
}

int main(int argc, char *argv[])
{
```

```
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <source_dir> <target_dir>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    sync_directories(argv[1], argv[2]);

    return 0;
}
```

🗂️ 빌드 방법

```
$ gcc -o sync_files sync_files.c
```

🚀 실행 예시

```
$ ./sync_files ./source_dir ./target_dir
```

출력 예시:

```
New file: ./source_dir/file1.txt
Copied: ./source_dir/file1.txt -> ./target_dir/file1.txt

Updated file: ./source_dir/file2.txt
Copied: ./source_dir/file2.txt -> ./target_dir/file2.txt
```

✅ 주요 기능 설명

기능	코드 위치
디렉토리 순회	`opendir()`, `readdir()`
파일 상태 확인	`stat()`
파일 복사	`copy_file()`
비교 기준	`st_mtime` (수정 시간) 비교
디버그 출력	`printf` 사용

📌 확장 아이디어

기능	설명
재귀 동기화	하위 디렉토리까지 자동 처리
삭제 동기화	source에 없는 파일 target에서 삭제
동기화 옵션 추가	`--delete`, `--verbose` 등 지원
다중 스레드 복사	대용량 복사 시 성능 향상
프로그레스바	진행 상태 표시 (`ncurses` 등 활용 가능)

✅ 실습 체크리스트

- `sync_files.c` 작성
- `gcc` 빌드 성공
- source/target 디렉토리 준비
- 새 파일 복사 성공 확인
- 수정된 파일 덮어쓰기 확인

🏁 결론

지금까지 만든 이 **Mini 파일 동기화 도구**는:

✅ *디렉토리 순회*
✅ *파일 비교*
✅ *복사 수행*

의 **기본적인 rsync 원리의 핵심 구조**를 다뤘다고 보면 돼.

→ 이 원리를 확장하면 **고성능 파일 백업기**, **rsync-like 동기화기**, **네트워크 파일 전송기** 같은 고급 툴도 만들 수 있어.

11.3 쓰레드 기반 압축 유틸리티

🗺️ 구현 목표

```
[파일] → 블록으로 분할 → 쓰레드로 병렬 압축 → 결과 파일로 저장
```

기본 설계

- 블록 단위로 파일 읽기 (`read`)
- 각 블록은 쓰레드가 **압축 처리** (간단히 RLE 또는 zlib 사용 가능)
- 쓰레드 결과는 메인 쓰레드가 모아서 **압축 파일**로 출력

🛠️ 구현 단계

단계	설명
1️⃣ 입력 파일 블록 분할	고정 블록 크기로 쪼갬
2️⃣ 쓰레드 풀 구성	`pthread_create()` 사용
3️⃣ 블록 압축 처리	(RLE 구현 or zlib 사용)
4️⃣ 결과 모음	쓰레드 결과를 순서 보장하며 저장
5️⃣ 파일 출력	압축된 결과물로 저장

📄 예제 코드 (thread_compress.c)

→ 여기서는 **단순 RLE 압축 알고리즘**을 사용해서 흐름을 쉽게 보여줄게.
(RLE = Run Length Encoding → 반복 문자 수 기록)

헤더 및 전역 설정

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

#define BLOCK_SIZE 4096
#define MAX_THREADS 4

typedef struct {
    int block_num;
    size_t input_size;
    char *input_data;
    char *output_data;
    size_t output_size;
} compress_task_t;

void *compress_block(void *arg)
{
    compress_task_t *task = (compress_task_t *)arg;
    char *src = task->input_data;
    char *dst = malloc(task->input_size * 2); // Worst case, no compression

    if (!dst) pthread_exit(NULL);

    size_t out_pos = 0;
```

```
    size_t i = 0;

    while (i < task->input_size)
    {
        char ch = src[i];
        size_t count = 1;
        while (i + count < task->input_size && src[i + count] == ch && count < 255)
        {
            count++;
        }
        dst[out_pos++] = ch;
        dst[out_pos++] = count;
        i += count;
    }

    task->output_data = dst;
    task->output_size = out_pos;

    printf("Block %d compressed: %zu bytes -> %zu bytes\n", task->block_num, task->input_size, task->output_size);

    pthread_exit(NULL);
}
```

메인 함수 (블록 분할 + 쓰레드 관리)

```
int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <input_file> <output_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int input_fd = open(argv[1], O_RDONLY);
    if (input_fd < 0)
    {
        perror("open input");
        exit(EXIT_FAILURE);
    }

    int output_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (output_fd < 0)
    {
        perror("open output");
        close(input_fd);
        exit(EXIT_FAILURE);
    }

    pthread_t threads[MAX_THREADS];
    compress_task_t tasks[MAX_THREADS];
```

```
    int block_num = 0;

    while (1)
    {
        ssize_t bytes_read;
        char *buffer = malloc(BLOCK_SIZE);
        if (!buffer) break;

        bytes_read = read(input_fd, buffer, BLOCK_SIZE);
        if (bytes_read <= 0)
        {
            free(buffer);
            break;
        }

        compress_task_t *task = &tasks[block_num % MAX_THREADS];
        task->block_num = block_num;
        task->input_size = bytes_read;
        task->input_data = buffer;

        pthread_create(&threads[block_num % MAX_THREADS], NULL, compress_block, task);

        // Wait for full batch if MAX_THREADS reached
        if ((block_num + 1) % MAX_THREADS == 0)
        {
            for (int i = 0; i < MAX_THREADS; i++)
            {
                pthread_join(threads[i], NULL);
                write(output_fd, tasks[i].output_data, tasks[i].output_size);
                free(tasks[i].input_data);
                free(tasks[i].output_data);
            }
        }

        block_num++;
    }

    // 남은 블록 처리
    int remaining = block_num % MAX_THREADS;
    for (int i = 0; i < remaining; i++)
    {
        pthread_join(threads[i], NULL);
        write(output_fd, tasks[i].output_data, tasks[i].output_size);
        free(tasks[i].input_data);
        free(tasks[i].output_data);
    }

    close(input_fd);
    close(output_fd);

    printf("Compression complete.\n");
    return 0;
}
```

🗂️ 빌드 방법

```
$ gcc -o thread_compress thread_compress.c -lpthread
```

🚀 실행 예시

```
$ ./thread_compress input.txt output.rle
```

출력 예시:

```
Block 0 compressed: 4096 bytes -> 1024 bytes
Block 1 compressed: 4096 bytes -> 2048 bytes
...
Compression complete.
```

✅ 주요 기능 설명

기능	코드 위치
블록 분할	`read()` 4096 byte 단위
쓰레드 생성	`pthread_create()`
RLE 압축	`compress_block()`
결과 모음	`pthread_join()` + `write()`
자원 해제	`free()` 처리 철저

📌 확장 아이디어

기능	설명
고성능 압축	RLE → zlib (`compress()` API)로 교체 가능
재귀적 쓰레드 관리	쓰레드 풀 구현 (pthread pool)
순서 안정성 개선	block_num 기준으로 순서 보장 큐 사용
파일 헤더 추가	block 정보, block size 기록해서 복원 가능하도록 설계
병렬 압축 성능 측정	CPU 사용률 비교 (taskset으로 CPU 코어 고정 후 측정)

✅ 실습 체크리스트

- `thread_compress.c` 작성
- `gcc -lpthread` 빌드 성공
- 블록 단위 멀티 쓰레드 압축 확인
- 쓰레드 병렬 처리 확인
- 압축된 파일 출력 성공

결론

이번 **쓰레드 기반 압축 유틸리티** 실습으로:

✅ 멀티 쓰레드 기반 **파일 처리 패턴** 학습
✅ `pthread_create()`, `pthread_join()` 사용
✅ 블록 기반 처리 알고리즘 경험
✅ **압축 알고리즘**의 구조적 원리 이해

까지 해볼 수 있었어.

다음 추천 흐름

1️⃣ **zlib 사용** → 실전 압축기 만들기
2️⃣ **pthread pool 적용** → 쓰레드 재사용
3️⃣ **I/O 최적화** → 비동기 I/O 적용
4️⃣ **압축/복원 프로그램 세트** 완성 (decompress 기능 추가)

11.4 IPC 기반 채팅 서버

🎯 목표: IPC 기반 채팅 시스템

- 방식: **Named Pipe (FIFO)** 또는 **System V 메시지 큐**
- 구조: **서버 ↔ 여러 클라이언트**
- 기능:
  - 클라이언트 → 서버로 메시지 전송
  - 서버가 모든 클라이언트에 메시지 브로드캐스트

✅ 이번 구현은: Named Pipe (`mkfifo`) 기반

🧱 전체 구조 요약

```
[클라이언트 1] ┐          ┌→ [클라이언트 1 수신]
[클라이언트 2] ├→ 서버 ─┼→ [클라이언트 2 수신]
[클라이언트 3] ┘          └→ [클라이언트 3 수신]

✓ 클라이언트 → 서버: /tmp/chat_in
✓ 서버 → 클라이언트: /tmp/chat_user_<pid>
```

---

1. 서버 코드 (chat_server.c)

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

#define SERVER_FIFO "/tmp/chat_in"
#define MAX_MSG 256
#define USER_FIFO_PREFIX "/tmp/chat_user_"

typedef struct {
    pid_t pid;
    char message[MAX_MSG];
} ChatMessage;

void send_to_all_clients(ChatMessage *msg) {
    DIR *dir;
    struct dirent *entry;
    char fifo_path[256];

    dir = opendir("/tmp");
    if (!dir) {
        perror("opendir");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        if (strncmp(entry->d_name, "chat_user_", 10) == 0) {
            snprintf(fifo_path, sizeof(fifo_path), "/tmp/%s", entry->d_name);
            int fd = open(fifo_path, O_WRONLY | O_NONBLOCK);
            if (fd >= 0) {
                write(fd, msg, sizeof(ChatMessage));
                close(fd);
            }
        }
    }
```

```
    closedir(dir);
}

int main() {
    mkfifo(SERVER_FIFO, 0666);
    int server_fd = open(SERVER_FIFO, O_RDONLY);

    ChatMessage msg;

    printf("💬 Chat server started. Waiting for messages...\n");

    while (1) {
        if (read(server_fd, &msg, sizeof(ChatMessage)) > 0) {
            printf("[PID %d] %s\n", msg.pid, msg.message);
            send_to_all_clients(&msg);
        }
    }

    close(server_fd);
    unlink(SERVER_FIFO);
    return 0;
}
```

2. 클라이언트 코드 (chat_client.c)

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

#define SERVER_FIFO "/tmp/chat_in"
#define MAX_MSG 256

typedef struct {
    pid_t pid;
    char message[MAX_MSG];
} ChatMessage;

char user_fifo[256];

void *reader_thread(void *arg) {
    int user_fd = open(user_fifo, O_RDONLY);
    ChatMessage msg;

    while (read(user_fd, &msg, sizeof(ChatMessage)) > 0) {
        printf("[PID %d] %s\n", msg.pid, msg.message);
```

```
    }

    return NULL;
}

int main() {
    pid_t pid = getpid();
    snprintf(user_fifo, sizeof(user_fifo), "/tmp/chat_user_%d", pid);
    mkfifo(user_fifo, 0666);

    pthread_t tid;
    pthread_create(&tid, NULL, reader_thread, NULL);

    int server_fd = open(SERVER_FIFO, O_WRONLY);
    ChatMessage msg;
    msg.pid = pid;

    printf("💬 Enter your messages (Ctrl+C to quit):\n");

    while (fgets(msg.message, MAX_MSG, stdin) != NULL) {
        write(server_fd, &msg, sizeof(ChatMessage));
    }

    close(server_fd);
    unlink(user_fifo);
    return 0;
}
```

🛠️ 빌드 방법

```
$ gcc -o chat_server chat_server.c
$ gcc -o chat_client chat_client.c -lpthread
```

🚀 실행 방법

서버 먼저 실행

```
$ ./chat_server
```

클라이언트 여러 개 실행 (다른 터미널에서)

```
$ ./chat_client
```

✅ 실습 체크리스트

- 서버 → 메시지 브로드캐스트 구현
- 클라이언트 → 서버 메시지 전송
- 클라이언트끼리 메시지 주고받기 성공
- Named FIFO 사용 확인 (`ls /tmp/chat_*`)
- 종료 시 FIFO 정리됨 확인

📌 확장 아이디어

기능	설명
사용자 이름 추가	`nickname` 필드 추가
퇴장 알림	"exit" 입력 시 알림 브로드캐스트
명령어 처리	`/who`, `/exit`, `/help` 등
채팅 로그 저장	서버에서 파일로 로그 기록
System V 메시지 큐 or UNIX 도메인 소켓 기반으로 전환	더 안정적인 IPC 구현

11.5 커널 로그 파서

🎯 구현 목표

기능	설명
로그 파일 읽기	`/var/log/kern.log` 또는 `dmesg` 출력 저장 파일
키워드 검색	사용자 입력 키워드에 해당하는 줄만 출력
로그 레벨 분석	KERN_xxx 레벨별 통계 출력
옵션	실시간 tail -f 스타일 구현 가능

🗺️ 구현 설계 흐름

```
1 1️⃣ 로그 파일 열기
2 2️⃣ 한 줄씩 읽기 → `fgets()`
3 3️⃣ 키워드 포함 여부 검사 → `strstr()`
4 4️⃣ 레벨별 카운트 분석 (선택)
5 5️⃣ 결과 출력
```

기본 예제 코드 (kernel_log_parser.c)

```c
// kernel_log_parser.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE 1024

void print_usage(const char *progname) {
    printf("Usage: %s <log_file> <keyword>\n", progname);
}

void parse_log(const char *log_file, const char *keyword) {
    FILE *fp = fopen(log_file, "r");
    if (!fp) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    char line[MAX_LINE];
    int match_count = 0;

    while (fgets(line, sizeof(line), fp) != NULL) {
        if (strstr(line, keyword)) {
            printf("%s", line);
            match_count++;
        }
    }

    fclose(fp);

    printf("\nTotal matched lines: %d\n", match_count);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        print_usage(argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *log_file = argv[1];
    const char *keyword = argv[2];

    parse_log(log_file, keyword);

    return 0;
}
```

🗂️ 빌드 방법

```
$ gcc -o kernel_log_parser kernel_log_parser.c
```

🚀 실행 예시

```
# dmesg 로그 파일로 저장
$ dmesg > dmesg.log

# 특정 키워드 검색
$ ./kernel_log_parser dmesg.log USB
```

출력 예시:

```
[    2.345678] usb 1-1: new high-speed USB device number 2 using xhci_hcd
[    2.456789] usb 1-1: Manufacturer: Generic USB Device
...

Total matched lines: 5
```

✅ 주요 기능 설명

기능	코드 위치
로그 파일 열기	`fopen()`
한 줄씩 읽기	`fgets()`
키워드 검색	`strstr()`
매칭 줄 출력	`printf()`
매칭 카운트	`match_count` 변수 사용

📌 확장 아이디어

기능	설명
로그 레벨 통계 출력	KERN_ERR, KERN_WARNING 등 통계
실시간 tail -f 구현	`inotify` 또는 `sleep+fseek` 사용
다중 키워드 검색 지원	`OR` 검색 기능 추가
색상 출력	`ANSI escape code` 로 KERN_ERR는 빨강 등
로그 날짜별 필터링	`grep` + `strptime()` 활용 가능

✅ 실습 체크리스트

- `kernel_log_parser.c` 작성
- `gcc` 빌드 성공
- 로그 파일 준비 (`dmesg > dmesg.log` 등)
- 키워드 검색 정상 동작 확인
- 매칭 라인 출력 및 카운트 확인

🚀 발전 방향

다음 단계로는 → **레벨별 통계 출력 기능** 추가해볼 수 있어.
예를 들어:

```
KERN_ERR: 5 lines
KERN_WARNING: 8 lines
KERN_INFO: 20 lines
```

이런 출력 추가하면 **커널 로그 상태 분석기** 수준까지 올라갈 수 있어.