3. 프로세스와 시스템 호출

3.1 프로세스란 무엇인가?

1. 개요

운영체제에서 **프로세스(Process)**란 **실행 중인 프로그램의 인스턴스**를 의미한다.
단순히 디스크에 저장된 프로그램(파일)은 **정적인 코드 덩어리**에 불과하지만, 이를 메모리로 올려 CPU가 실행하기 시작하면 **프로세스**가 된다.
운영체제는 **프로세스 단위로 자원을 관리**하고, 각 프로세스는 고유한 **주소 공간, 실행 상태, 리소스**를 가진다.

---

2. 프로그램과 프로세스의 차이

개념	설명
프로그램	정적인 코드(파일)
프로세스	실행 중인 프로그램 인스턴스

💡 예시

`vim`이라는 프로그램은 디스크에 `/usr/bin/vim`으로 존재 → 실행 시 **프로세스 생성** → 메모리로 올라가고 CPU에서 실행됨 → PID 할당됨.

---

3. 프로세스의 구성 요소

운영체제는 프로세스를 다음과 같은 구성 요소로 관리한다:

구성 요소	설명
프로그램 코드 (text section)	실행할 명령어(코드)
데이터 (data section)	전역 변수, 정적 변수 영역
힙 (heap)	동적 메모리 영역 (`malloc` 등으로 할당)
스택 (stack)	함수 호출 시의 지역 변수, 리턴 주소 저장
프로세스 제어 블록 (PCB)	운영체제가 프로세스를 관리하기 위한 정보 저장 (PID, 상태, 우선순위 등)

---

4. 프로세스 상태

프로세스는 **여러 가지 상태**를 가지며, 운영체제가 상태 전이를 관리한다.

상태	설명
New	생성 중

상태	설명
Ready	CPU 할당을 기다리는 상태
Running	CPU에서 실행 중
Waiting (Blocked)	입출력 등 이벤트 대기 중
Terminated	실행 종료됨

---

5. 프로세스 식별자 (PID)

운영체제는 각 프로세스를 고유한 **PID (Process ID)**로 식별한다.

- 부모-자식 관계가 존재 (fork 호출 시 부모가 자식 생성)
- `getpid()` → 현재 프로세스 PID 확인
- `getppid()` → 부모 프로세스 PID 확인

C 코드 예제: 프로세스 식별

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

int main() {
    printf("My PID: %d\n", getpid());
    printf("Parent PID: %d\n", getppid());
    return 0;
}
```

---

6. 프로세스 생성

리눅스/유닉스에서 **프로세스 생성의 기본 시스템 콜**은 `fork()`이다.

- `fork()` → 부모 프로세스가 새로운 자식 프로세스를 생성
- 부모와 자식은 **독립적인 실행 흐름을 가지지만 동일한 코드 복사본으로 시작**한다.
- 이후 `exec()` 계열 호출로 다른 프로그램을 실행할 수 있음.

프로세스 트리 예시

```
PID 1 (init/systemd)
 ├── PID 101 (bash)
 |    ├── PID 202 (vim)
 |    └── PID 203 (gcc)
```

---

7. 프로세스와 쓰레드

- 프로세스는 **독립적인 주소 공간**을 가진 실행 단위
- 하나의 프로세스는 여러 개의 **쓰레드(Thread)**를 가질 수 있으며, 쓰레드는 **프로세스 내부의 실행 흐름 단위**이다.

구분	프로세스	쓰레드
주소 공간	독립적	공유 (같은 프로세스 내에서)
생성 비용	높음	낮음
통신 방식	IPC 필요	메모리 공유

---

8. 결론

프로세스는 실행 중인 프로그램의 인스턴스이며, 운영체제에서 자원 할당과 스케줄링의 기본 단위이다.
효율적인 프로세스 관리는 운영체제의 핵심 기능 중 하나이며, 프로세스 기반으로 CPU, 메모리, 파일, 입출력 자원이 할당된다.
🚀

3.2 `fork()`, `exec*()`, `wait()` 함수

1. 개요

운영체제는 프로세스를 생성하고 제어하기 위해 여러 가지 시스템 호출을 제공한다.
리눅스 및 유닉스 계열 시스템에서 프로세스 관리를 위한 대표적인 호출은 다음과 같다:

함수	주요 기능
`fork()`	현재 프로세스를 복제하여 새로운 프로세스를 생성
`exec*()`	현재 프로세스를 새로운 프로그램으로 덮어씀
`wait()` / `waitpid()`	자식 프로세스의 종료를 기다림

이 함수들은 **프로세스 생성 및 실행 제어의 기본 구성 요소**이며, 이를 통해 부모-자식 프로세스 구조를 만들 수 있다.

---

2. `fork()` — 프로세스 생성

함수 원형

```
#include <unistd.h>

pid_t fork(void);
```

동작

- `fork()`는 호출한 프로세스를 **복제**하여 새로운 프로세스를 생성한다.
- 반환값:
  - 부모 프로세스에는 **자식 프로세스의 PID**가 반환된다.
  - 자식 프로세스에는 **0**이 반환된다.
  - 오류 시 **-1**이 반환된다.

예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 자식 프로세스
        printf("Child Process, PID: %d\n", getpid());
    } else {
        // 부모 프로세스
        printf("Parent Process, PID: %d, Child PID: %d\n", getpid(), pid);
    }

    return 0;
}
```

특징

- **자식 프로세스는 부모 프로세스의 복사본**으로 생성된다.
- 부모와 자식은 **별도의 주소 공간**을 가진다.
- 이후 두 프로세스는 독립적으로 실행된다.

---

3. `exec*()` — 새로운 프로그램 실행

주요 함수

함수	설명
`execl()`	인자를 나열하여 전달
`execp()`	경로 탐색을 지원
`execv()`	인자를 배열로 전달
`execvp()`	인자 배열 + 경로 탐색 지원

함수	설명
`execve()`	시스템콜 수준 함수 (가장 저수준)

원형 (대표적으로 `execvp()` 사용 예시)

```
#include <unistd.h>

int execvp(const char *file, char *const argv[]);
```

동작

- 현재 프로세스를 새로운 프로그램으로 덮어쓴다.
- 성공 시 **리턴하지 않음** (새로운 프로그램으로 대체됨).
- 실패 시 **-1** 반환.

예제

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

int main() {
    char *args[] = { "ls", "-l", NULL };
    execvp("ls", args);

    // 여기에 도달하면 exec 실패
    perror("execvp");
    return 1;
}
```

특징

- `exec` 계열 함수는 새로운 프로그램을 **현재 프로세스 공간에서 실행**한다.
- 기존 코드, 데이터, 힙, 스택 영역은 사라지고 새로운 프로그램으로 덮어씀.

---

4. `wait()` / `waitpid()` — 자식 프로세스 종료 대기

원형

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

동작

- 부모 프로세스가 **자식 프로세스의 종료를 기다림**.
- `wait()` 는 **임의의 종료된 자식**을 기다린다.
- `waitpid()` 는 **특정 자식**의 종료를 기다릴 수 있다.

예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 자식 프로세스
        printf("Child PID: %d\n", getpid());
        return 42;
    } else {
        // 부모 프로세스
        int status;
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited with status %d\n", WEXITSTATUS(status));
        }
    }

    return 0;
}
```

특징

- 자식 프로세스가 종료되면 부모는 이를 wait로 확인하고 **자원(PCB 등)을 회수**한다.
- `WIFEXITED(status)` → 정상 종료 여부 확인
- `WEXITSTATUS(status)` → 종료 코드 추출

---

5. 실행 흐름 예시 🚀

```
Parent (fork 호출) → Child 생성
Parent wait() 호출로 Child 종료 대기
Child → exec 호출로 ls 실행 → ls 실행 후 종료
Parent → wait() 결과 확인 → Child 종료 코드 출력
```

---

6. 결론

- `fork()`를 통해 프로세스는 **복제**되어 독립적인 실행 흐름을 가진다.
- `exec*()` 계열 함수를 통해 프로세스는 **새로운 프로그램으로 변신**할 수 있다.
- `wait()` / `waitpid()`를 통해 부모는 **자식 프로세스의 종료를 적절히 관리**한다.

이러한 시스템 호출을 통해 운영체제는 **프로세스 생성-실행-종료**라는 기본 사이클을 구축하고 관리한다. 🛠️

3.3 프로세스 ID, 부모-자식 관계

1. 개요

운영체제는 각 프로세스를 **고유하게 식별**하기 위해 **프로세스 식별자(Process ID, PID)**를 사용한다.
프로세스는 시스템 내에서 **부모-자식 관계(parent-child relationship)**를 가지며, 이를 통해 **프로세스 트리**가 형성된다.

이 구조를 이해하면 프로세스가 어떻게 생성되고 관리되는지, 그리고 프로세스 간에 어떤 관계가 형성되는지 명확히 알 수 있다. 🛠️

---

2. 프로세스 ID (PID)

정의

- **PID**는 운영체제가 프로세스를 고유하게 식별하기 위한 **정수값**이다.
- 시스템 부팅 시 초기 프로세스(init 또는 systemd)가 PID 1을 가지며, 이후 생성되는 모든 프로세스에 PID가 할당된다.

PID 확인

```
#include <unistd.h>

pid_t getpid(void);   // 현재 프로세스의 PID 반환
```

예제

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

int main() {
    printf("Current PID: %d\n", getpid());
    return 0;
}
```

---

3. 부모 프로세스 ID (PPID)

정의

- 프로세스는 **부모 프로세스(parent process)**로부터 생성된다.
- 부모 프로세스의 PID를 **PPID(Parent Process ID)**라고 한다.

PPID 확인

```
#include <unistd.h>

pid_t getppid(void);   // 부모 프로세스의 PID 반환
```

예제

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

int main() {
    printf("Parent PID: %d\n", getppid());
    return 0;
}
```

---

4. 부모-자식 관계 형성 과정

`fork()` 호출 시 흐름

1️⃣ 부모 프로세스가 `fork()`를 호출
2️⃣ 커널이 새로운 PID를 할당
3️⃣ 새로운 프로세스(자식)가 생성됨
4️⃣ 자식 프로세스의 PPID는 **부모 프로세스의 PID**로 설정됨

결과

```
Parent PID = X
 └─ Child PID = Y (Parent PID = X)
```

---

5. 예제: 부모-자식 관계 출력

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
```

```
    } else if (pid == 0) {
        // 자식 프로세스
        printf("Child Process: PID=%d, Parent PID=%d\n", getpid(), getppid());
    } else {
        // 부모 프로세스
        printf("Parent Process: PID=%d, Child PID=%d\n", getpid(), pid);
    }

    return 0;
}
```

실행 결과 예시

```
Parent Process: PID=12345, Child PID=12346
Child Process: PID=12346, Parent PID=12345
```

---

6. 프로세스 트리

리눅스 시스템에서는 프로세스가 **트리 구조**로 구성된다.

```
PID 1 (systemd)
 ├─ PID 1000 (gnome-terminal)
 |    ├─ PID 1020 (bash)
 |    |    ├─ PID 1050 (vim)
 |    |    └─ PID 1051 (gcc)
```

- 각 프로세스는 자신의 **부모(Parent)**가 있고,
- 새로운 프로세스를 생성할 때 그 **부모-자식 관계가 유지**된다.

---

7. 고아 프로세스와 좀비 프로세스

개념	설명
고아 프로세스	부모 프로세스가 먼저 종료되었을 때 자식 프로세스는 고아가 된다. OS가 이를 init/systemd로 **재부모화(reparenting)**한다.
좀비 프로세스	자식 프로세스가 종료되었지만 부모가 `wait()`를 호출하지 않아서 프로세스 테이블에 **종료 상태 정보만 남아 있는 프로세스**.

고아/좀비 프로세스를 이해하면 프로세스 관계 유지와 정리의 중요성을 알 수 있다. 🖌️

---

8. 결론

- 프로세스는 고유한 **PID**를 가진다.
- **부모-자식 관계**는 프로세스 생성 시 자동으로 형성된다.
- `getpid()`, `getppid()` 호출로 현재 프로세스와 부모 프로세스의 PID를 확인할 수 있다.
- 프로세스 트리는 시스템에서 **구조적이고 계층적으로** 형성되어 관리된다.

이러한 관계를 명확히 이해하면 **프로세스 생성, 종료, 관리**가 보다 직관적이고 체계적으로 이루어질 수 있다. 🚀

3.4 좀비/고아 프로세스 실습

1. 개요

리눅스에서 **프로세스 종료와 부모-자식 관계**의 관리가 잘못되면, 프로세스가 **좀비(Zombie) 프로세스** 또는 **고아(Orphan) 프로세스** 상태로 남게 된다.

상태	정의
좀비 프로세스	종료된 자식 프로세스의 **상태 정보가 부모 프로세스에 의해 수거되지 않아서** 커널 프로세스 테이블에 남아 있는 프로세스
고아 프로세스	부모 프로세스가 먼저 종료되어 **부모 없는 자식 프로세스**가 된 상태 (커널이 이를 `init` 또는 `systemd`에 재부모화함)

이번 실습에서는 두 상황을 직접 C 코드로 만들어보고, **프로세스 테이블에서 상태 변화를 관찰**해볼 것이다. 🔍

---

2. 좀비 프로세스 실습

원리

- 부모가 `fork()`로 자식을 만든 후 `wait()`를 호출하지 않고 일정 시간 동안 **sleep()**한다.
- 자식이 먼저 종료되면 부모는 여전히 자식의 종료 상태를 수거하지 않음 → **좀비 발생**.

코드 예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }

```

```
    if (pid == 0) {
        // 자식 프로세스
        printf("Child process PID: %d exiting...\n", getpid());
        exit(0);
    } else {
        // 부모 프로세스
        printf("Parent process PID: %d sleeping...\n", getpid());
        sleep(30);  // 이 동안 자식은 종료 후 좀비 상태로 남음
        printf("Parent now calling wait()...\n");
        wait(NULL);  // 여기서 좀비 수거
    }

    return 0;
}
```

실행 절차

1️⃣ 실행 후 다른 터미널에서 확인:

```
ps -o pid,ppid,stat,cmd | grep Z
```

- `Z` 상태가 보이면 → 좀비 프로세스
- `ps aux`에서도 `STAT` 컬럼에 `Z` 표시됨

2️⃣ `Parent now calling wait()` 이후에는 좀비가 사라짐.

---

3. 고아 프로세스 실습

원리

- 자식 프로세스가 부모보다 오래 실행됨.
- 부모가 먼저 종료되면, 자식은 커널에 의해 **init/systemd 프로세스로 재부모화**된다.

코드 예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // 자식 프로세스
```

```
        printf("Child process PID: %d, Parent PID: %d\n", getpid(), getppid());
        sleep(10);  // 부모 종료 후에도 살아있도록 충분히 sleep
        printf("Child process PID: %d, New Parent PID: %d\n", getpid(), getppid());
    } else {
        // 부모 프로세스
        printf("Parent process PID: %d exiting...\n", getpid());
        exit(0);  // 부모 먼저 종료
    }

    return 0;
}
```

실행 절차

1️⃣ 실행 후 자식은 10초간 sleep.
2️⃣ 자식 sleep 중에 부모가 먼저 종료됨 → **고아 프로세스 발생**.
3️⃣ `ps -o pid,ppid,stat,cmd` 명령으로 자식의 `PPID`가 `1(init/systemd)`로 바뀐 것을 확인.

---

4. 정리

상태	원인	결과	해결 방법
좀비	부모가 `wait()` 호출 안 함	커널 테이블에 남음 (`Z` 상태)	부모가 `wait()` 호출
고아	부모가 먼저 종료됨	커널이 `init/systemd`로 재부모화	자동 처리됨

좀비 프로세스는 **커널 자원 누수**의 원인이 되므로 반드시 **부모가 `wait()` 호출**로 수거해야 한다.
고아 프로세스는 운영체제가 안전하게 **재부모화** 처리를 자동으로 수행한다.

3.5 `getpid()`, `getppid()`, `getuid()`

1. 개요

운영체제는 프로세스와 사용자 정보를 관리하기 위해 **프로세스 ID(PID)**와 **사용자 ID(UID)**를 제공한다.
C 언어에서는 이러한 정보를 확인하기 위한 시스템 호출을 사용할 수 있다. 이 호출들은 프로세스 및 사용자 상태 확인, 로깅, 보안 검사 등에 널리 활용된다. 🛠️

---

2. 프로세스 식별 관련 함수

`getpid()` — 현재 프로세스 ID 확인

헤더 파일

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

원형

```
pid_t getpid(void);
```

설명

- 현재 프로세스의 **PID**(Process ID)를 반환한다.
- PID는 운영체제가 프로세스를 고유하게 식별하기 위해 사용하는 값이다.

사용 예

```
pid_t pid = getpid();
printf("Current PID: %d\n", pid);
```

`getppid()` — 부모 프로세스 ID 확인

헤더 파일

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

원형

```
pid_t getppid(void);
```

설명

- 현재 프로세스의 **부모 프로세스(Parent Process)**의 PID를 반환한다.
- 부모 프로세스가 먼저 종료되어 고아 프로세스가 된 경우, **init/systemd 프로세스(보통 PID 1)**가 새로운 부모가 된다.

사용 예

```
pid_t ppid = getppid();
printf("Parent PID: %d\n", ppid);
```

`getuid()` — 실제 사용자 ID 확인

헤더 파일

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

원형

```
uid_t getuid(void);
```

설명

- 현재 프로세스를 실행한 **실제 사용자(Real User)의 UID**를 반환한다.
- **UID(User ID)**는 시스템에서 사용자를 구별하는 값이며 `/etc/passwd` 파일에 등록되어 있다.
- 일반 사용자 UID: 보통 **1000 이상**
- 관리자(root) UID: **0**

사용 예

```
uid_t uid = getuid();
printf("User ID: %d\n", uid);
```

---

3. 종합 예제

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

int main() {
    pid_t pid = getpid();
    pid_t ppid = getppid();
    uid_t uid = getuid();

    printf("📄 Current PID  : %d\n", pid);
    printf("📄 Parent PID   : %d\n", ppid);
    printf("👤 User ID (UID): %d\n", uid);

    return 0;
}
```

실행 예시

```
📄 Current PID  : 4567
📄 Parent PID   : 1234
👤 User ID (UID): 1000
```

---

4. 활용 사례

함수	활용 상황
`getpid()`	프로세스 로깅, 디버깅, 고유 식별 정보 제공
`getppid()`	부모 프로세스 상태 확인, 프로세스 트리 분석
`getuid()`	접근 제어, 보안 검사, 실행 권한 확인

---

5. 결론

- `getpid()` 는 **현재 프로세스의 식별 정보**를 제공한다.
- `getppid()` 는 **프로세스의 계층 관계(부모 확인)**에 사용된다.
- `getuid()` 는 **실행 사용자 권한 확인**에 사용된다.

이러한 함수들은 **운영체제에서 프로세스와 사용자 상태를 확인하고 안전하게 프로세스를 제어**하기 위한 기본적인 도구로 매우 유용하다. 🚀

📎 실습

`fork()` 기반의 단순 쉘 구현

1. 개요

리눅스/유닉스의 **쉘(shell)**은 사용자로부터 명령어를 입력받아 **프로세스를 생성(fork)**하고, 해당 명령어를 실행(`exec`)하여 결과를 출력하는 프로그램이다.

쉘의 가장 기본적인 동작은 다음과 같은 흐름으로 구성된다:

```
1 1️⃣ 사용자 입력 → 2️⃣ fork() 호출 → 3️⃣ 자식 프로세스에서 exec() → 4️⃣ 부모 프로세스 wait()
```

이번 예제에서는 `fork()` + `execvp()` + `wait()` 를 이용하여 **단순 쉘**을 구현한다.
이를 통해 **프로세스 생성/실행/종료 관리의 기본 흐름**을 직접 경험할 수 있다. 🛠️

---

2. 필수 시스템 호출

함수	기능
`fork()`	새로운 프로세스(자식) 생성
`execvp()`	자식 프로세스를 새로운 프로그램으로 덮어씀
`wait()`	부모가 자식 프로세스의 종료를 기다림

---

3. 단순 쉘 구현 예제

코드: `simple_shell.c`

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

#define MAX_LINE 1024
#define MAX_ARGS 64
```

```c

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

    while (1) {
        // 프롬프트 출력
        printf("mysh> ");
        fflush(stdout);

        // 사용자 입력 읽기
        if (fgets(line, sizeof(line), stdin) == NULL) {
            break;  // EOF (Ctrl+D) 시 종료
        }

        // 개행 문자 제거
        line[strcspn(line, "\n")] = '\0';

        // 입력이 비어있으면 다시 입력받기
        if (strlen(line) == 0) {
            continue;
        }

        // "exit" 명령어 처리
        if (strcmp(line, "exit") == 0) {
            break;
        }

        // 입력 파싱 (공백 기준)
        int i = 0;
        char *token = strtok(line, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;  // execvp()는 마지막 인자로 NULL 필요

        // fork()로 자식 프로세스 생성
        pid_t pid = fork();

        if (pid == -1) {
            perror("fork");
            continue;
        } else if (pid == 0) {
            // 자식 프로세스: 명령어 실행
            execvp(args[0], args);
            // execvp 실패 시 아래 코드 실행됨
            perror("execvp");
            exit(1);
        } else {
            // 부모 프로세스: 자식 종료 대기
            int status;
            wait(&status);
```

```
        }
    }

    printf("Bye!\n");
    return 0;
}
```

---

4. 실행 방법

```
gcc simple_shell.c -o simple_shell
./simple_shell
```

사용 예시

```
mysh> ls -l
[ls 실행 결과 출력]

mysh> date
[date 실행 결과 출력]

mysh> exit
Bye!
```

---

5. 동작 흐름

```
1️⃣ 사용자 입력 → 2️⃣ fork() → 3️⃣ 자식 프로세스에서 execvp() 호출
        ↘ 실패 시 오류 출력 ↙
4️⃣ 부모는 wait()로 자식 종료 기다림 → 다시 프롬프트 출력
```

---

6. 주요 포인트

- `fork()` : 부모-자식 프로세스 구조 형성
- `execvp()` : PATH 환경변수를 활용하여 명령어 검색 후 실행
- `wait()` : 부모가 자식 종료까지 기다림으로써 좀비 프로세스 방지
- `exit` **명령어 구현** : 사용자가 쉘을 종료할 수 있도록 처리

---

7. 한계 및 확장 가능성 🚀

현재 단순 쉘의 한계:

기능	지원 여부
파이프 (`|`)	

기능	지원 여부
리디렉션 (`>`, `<`)	❌
백그라운드 실행 (`&`)	❌
명령어 이력(history)	❌
다중 명령어(;) 처리	❌

→ 이후 단계에서 `dup2()` **기반 리디렉션 지원**, **파이프 구현**, **백그라운드 프로세스 관리** 등으로 확장 가능하다.

---

8. 결론

이번 예제를 통해 **기본적인 쉘 동작 흐름**인
입력 → **fork** → **exec** → **wait** 를 이해하고 구현할 수 있었다.

이 구조는 모든 유닉스/리눅스 쉘의 **핵심 원리**이며, 시스템 프로그래밍의 기초가 된다. 🌟

`execvp()` 기반 명령어 실행기

1. 개요

`execvp()`는 **현재 프로세스의 메모리 공간을 새로운 프로그램으로 덮어써서 실행**하는 함수이다.
주로 **fork()와 함께 사용**되어 새로 생성된 자식 프로세스에서 원하는 명령어를 실행하는 데 사용된다.

exec 계열 함수 비교

함수	특징
`execl()`	인자를 나열하여 전달
`execv()`	인자 배열로 전달
`execvp()`	인자 배열 + `PATH` 검색 지원
`execve()`	환경 변수까지 직접 설정 (시스템콜 수준)

👉 `execvp()`는 **쉘처럼 명령어 이름만 주면 자동으로** `PATH`**에서 검색**해 실행해준다. 매우 실용적이다. 🚀

---

2. 함수 원형

```
#include <unistd.h>

int execvp(const char *file, char *const argv[]);
```

- `file`: 실행할 프로그램 이름
- `argv[]`: 인자 목록 (첫 번째는 관례상 프로그램 이름, 마지막은 반드시 NULL)

반환값

- 성공 시 **절대 반환되지 않음** (새 프로그램으로 메모리 덮어씀)
- 실패 시 **-1 반환**, `errno` 설정됨

---

3. 단순 명령어 실행기 예제

목표

- 사용자에게 **명령어 입력 받기**
- `fork()`로 **자식 프로세스 생성**
- 자식에서 `execvp()`로 명령어 실행
- 부모는 `wait()`로 자식 종료까지 대기

코드: `execvp_runner.c`

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

#define MAX_LINE 1024
#define MAX_ARGS 64

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

    while (1) {
        printf("run> ");
        fflush(stdout);

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

        // 개행 문자 제거
        line[strcspn(line, "\n")] = '\0';

        // 입력이 비어있으면 continue
        if (strlen(line) == 0) {
            continue;
        }

        // "exit" 명령어 처리
        if (strcmp(line, "exit") == 0) {
            break;
        }

```

```
        // 입력 파싱
        int i = 0;
        char *token = strtok(line, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;  // execvp()는 마지막에 NULL 필요

        // fork → execvp → wait 구조
        pid_t pid = fork();

        if (pid == -1) {
            perror("fork");
            continue;
        } else if (pid == 0) {
            // 자식 프로세스에서 명령어 실행
            execvp(args[0], args);

            // execvp 실패 시만 이 코드 실행됨
            perror("execvp");
            exit(1);
        } else {
            // 부모 프로세스는 자식 종료 대기
            int status;
            wait(&status);

            if (WIFEXITED(status)) {
                printf("Child exited with status %d\n", WEXITSTATUS(status));
            } else {
                printf("Child terminated abnormally.\n");
            }
        }
    }

    printf("Bye!\n");
    return 0;
}
```

4. 실행 예시

```
gcc execvp_runner.c -o execvp_runner
./execvp_runner
```

사용 예

```
run> ls -l
[ls 실행 결과 출력]

run> date
[date 실행 결과 출력]

run> uname -a
[uname 실행 결과 출력]

run> exit
Bye!
```

---

5. 작동 원리

```
while (1):
    사용자 입력 → 파싱 → fork()

    자식 프로세스:
        execvp() 호출 → 프로그램 실행 → execvp 성공 시 반환 안 됨
        execvp 실패 시 perror 출력 후 exit

    부모 프로세스:
        wait()로 자식 종료 대기 → 결과 출력
```

---

6. 주요 포인트

- `execvp()` 는 `PATH` **환경 변수 기반 검색** 지원 (ex: "ls" → /bin/ls)
- exec 호출 성공 시 **프로세스의 코드/데이터/스택 완전히 덮어쓰기**
- exec 호출 실패 시만 **perror** → **exit(1)** 경로로 진입

---

7. 확장 방향 🚀

현재는 기본적인 **단일 명령어 실행기**이며 다음 기능 추가가 가능하다:

기능	지원 여부
파이프 (`|`)	
리디렉션 (`>` , `<` , `>>`)	❌
백그라운드 실행 (`&`)	❌
명령어 history	❌

👉 이후 `dup2()` , `pipe()` , `setpgid()` , `signal()` 등을 통해 단계별로 확장 가능

8. 결론

이번 예제에서 `fork()` + `execvp()` + `wait()`를 사용하여
쉘과 유사한 구조의 명령어 실행기를 구현할 수 있었다.

이 구조는 시스템 프로그래밍 및 리눅스 프로세스 관리의 **기본 중의 기본**으로 매우 중요하다. 🛠️

부모-자식 동기화 테스트

1. 개요

기본적인 **fork() 기반 부모-자식 프로세스**는 별도의 독립적인 실행 흐름을 가지므로
명확한 순서 보장이 필요할 때는 동기화가 필요하다.

문제 상황

- 부모가 자식보다 먼저 출력하기를 원하는데 실제로는 **출력 순서가 랜덤**하게 나타날 수 있다.
- I/O 버퍼링과 **프로세스 스케줄링의 비결정성** 때문에 발생.

해결 방법

- `wait()` 를 통한 **단순 동기화**
- **파이프(pipe)**, **시그널(signal)** 등 활용한 고급 동기화도 가능 🚀

---

2. 기본 테스트: 동기화 없는 fork()

예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 자식 프로세스
        printf("Child: PID=%d\n", getpid());
    } else {
        // 부모 프로세스
        printf("Parent: PID=%d\n", getpid());
    }

    return 0;
}
```

실행 결과 예시

```
Parent: PID=12345
Child: PID=12346
```

혹은

```
Child: PID=12346
Parent: PID=12345
```

→ 출력 순서가 **비결정적**이다.

---

3. `wait()` 를 사용한 기본 동기화

개선 예제

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

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 자식 프로세스
        printf("Child: PID=%d\n", getpid());
    } else {
        // 부모 프로세스
        int status;
        wait(&status);  // 자식 종료 대기
        printf("Parent: PID=%d (after child)\n", getpid());
    }

    return 0;
}
```

결과 예시

```
Child: PID=12346
Parent: PID=12345 (after child)
```

→ `wait()` 호출 덕분에 **부모가 자식 종료 이후에 출력**한다.

---

4. 고급 동기화: 파이프(pipe) 사용하기

원리

- **파이프는 커널 수준 버퍼**를 제공.
- 부모가 파이프에서 **자식의 완료 신호를 읽은 후** 다음 단계 진행.

코드 예제

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

int main() {
    int pipefd[2];
    pipe(pipefd);

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 자식 프로세스
        close(pipefd[0]);  // 읽기 끝 닫기
        printf("Child: PID=%d\n", getpid());
        write(pipefd[1], "X", 1);  // 완료 신호 보내기
        close(pipefd[1]);
    } else {
        // 부모 프로세스
        close(pipefd[1]);  // 쓰기 끝 닫기
        char buf;
        read(pipefd[0], &buf, 1);  // 자식 완료까지 대기
        printf("Parent: PID=%d (after child)\n", getpid());
        close(pipefd[0]);
    }

    return 0;
}
```

결과

```
Child: PID=12346
Parent: PID=12345 (after child)
```

파이프를 사용한 명시적 동기화 성공 🎉

---

5. 동기화 방법 비교

방법	특징
`wait()`	**자식 프로세스 전체 종료 후 대기**
`pipe`	**구체적 시점(이벤트 단위) 동기화 가능**
`signal`	**비동기 이벤트 기반 동기화 가능**

---

6. 결론

- **fork() 만으로는 동기화되지 않음** → 순서 보장 필요 시 반드시 `wait()` 또는 별도 메커니즘 사용
- **wait()** → 가장 간단하고 신뢰성 높은 기본 동기화 방법
- **pipe, signal 등** → 복잡한 동기화에 사용 가능

동기화는 **프로세스 간 협력 구조 설계**에서 매우 중요한 요소이며, 다양한 상황에서 사용된다. 🛠️