Project

OS HW-1

모바일시스템공학과

32204288 조민욱

Contents

Introduction

Analyzing and planning

OS 는 기본적으로 프로세스를 관리하는 역할을 수행한다. 즉 이런 OS 의 특징을 C 코드로 구현함으로써 Simple Shell 을 완성할 수 있다.

- 프로세스의 생성과 종료
- 프로세스 상태 관리 (wait, exit)
- 프로세스 입출력 지정
- 프로세스 간 통신

Association idea with code

프로세스의 생성과 종료는 fork 를 통해 구현 가능하다. 프로그램의 실행과 동시에 부모 프로세스는 존재하며, fork 호출과 동시에 운영체제가 현재 프로세스를 복사하여 자식 프로세스를 생성한다. 부모는 자식의 PID 값을 받고, 자식은 0 의 값을 받게 된다. 보다 명확한 프로세스의 동작 과정을 이해하기 위해 직접 만든 PPT

슬라이드를 첨부한다. 기본적으로 제공된 fork 예제에 print 를 통해 조금 더 자세한 동작과정을 확인할 수 있도록 변형했다.

fork

```c
int main(int argc, char *argv[])
{
    // 프로세스 ID
    pid_t pid;
    int i;

    for (i = 0 ; i < 10 ; i++) {
        // 새로운 자식 프로세스를 생성
        pid = fork();
        printf("\n!!! : %d\n", pid);
        if (pid == -1) {
            // 호출이 실패
            perror("fork error");
            return 0;
        } else if (pid == 0) {
            // child
            printf("child process with pid %d (i: %d) \n", getpid(), i);
            exit(0);
        } else {
            // parent
            printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
            wait(0);
        }
    }
    return 0;
}
```

```
!!! : 62174
parent process with pid 62173 (i: 0)... wait for child

!!! : 0
child process with pid 62174 (i: 0)

!!! : 62175
parent process with pid 62173 (i: 1)... wait for child

!!! : 0
child process with pid 62175 (i: 1)

!!! : 62176
parent process with pid 62173 (i: 2)... wait for child

!!! : 0
child process with pid 62176 (i: 2)

!!! : 62177
parent process with pid 62173 (i: 3)... wait for child

!!! : 0
child process with pid 62177 (i: 3)

!!! : 62178
parent process with pid 62173 (i: 4)... wait for child
```

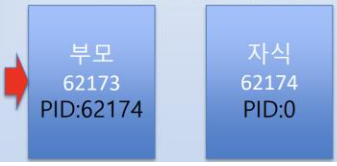
Figure 1) Fork example source code and result

Fork Flow example (i=0)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

PID 변수 값은 생성된 자식 프로세스의 PID=62174



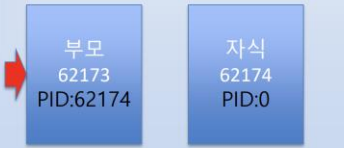

```
!!! : 62174
```

Fork Flow example (i=0)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

부모 프로세스 waiting...




```
parent process with pid 62173 (i: 0)... wait for child
```

Fork Flow example (i=0)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

자식 프로세스 진행 및 종료!


```
!!! : 0
child process with pid 62174 (i: 0)
```

Fork Flow example (i=1)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

부모 프로세스 진행... 변수 i값 증가

Fork Flow example (i=1)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

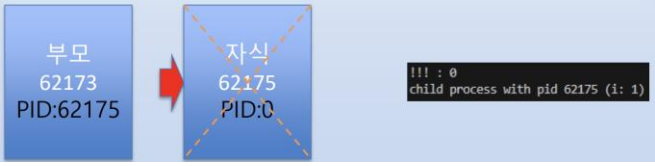
부모 프로세스 waiting...

Fork Flow example (i=1)

진행

```c
for (i = 0 ; i < 10 ; i++) {
    // 새로운 자식 프로세스를 생성
    pid = fork();
    printf("\n!!! : %d\n", pid);
    if (pid == -1) {
        // 호출이 실패
        perror("fork error");
        return 0;
    } else if (pid == 0) {
        // child
        printf("child process with pid %d (i: %d) \n", getpid(), i);
        exit(0);
    } else {
        // parent
        printf("parent process with pid %d (i: %d)... wait for child \n", getpid(), i);
        wait(0);
    }
}
```

자식 프로세스 진행 및 종료... 이후 반복



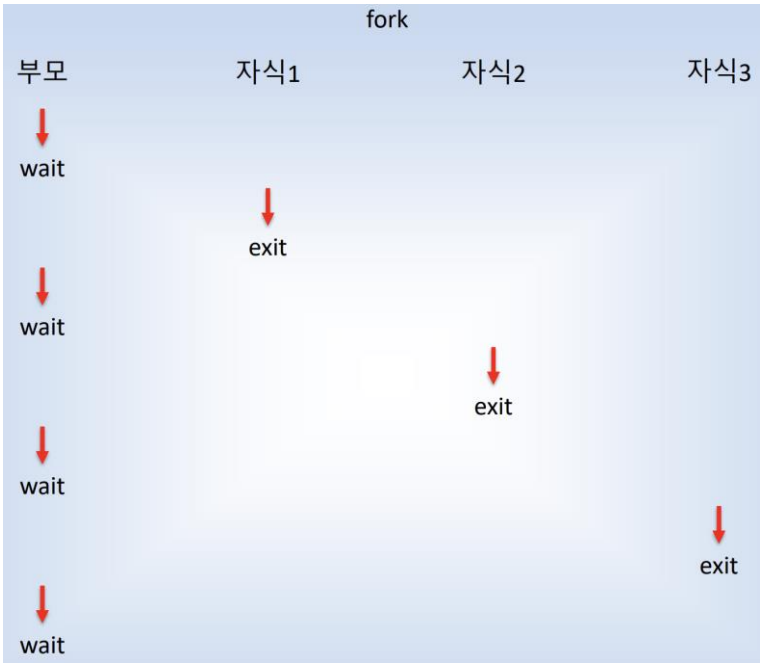


Figure 2) Fork logical flow

또한 getenv 를 통해 환경변수 경로를 받아온 다음 execve 의 인자로 전달하는 과정을 거쳐야 한다. PATH 를 받아온 문자열은 읽기 전용 문자열이므로 메모리 free 와 같은 과정을 거칠 수 없다. 따라서 문자열을 복제하고 조작하도록 한다. 또한 환경변수를 받은 후 호출하는 execve 함수는 실행과 동시에 현재 프로세스 자체를 대체하게 된다. 즉 이후의 코드들은 실행되지 않음에 주의해야 한다. 마찬가지로, exit 를 통해 굳이 자식 프로세스를 종료하지 않아도 무방하다.

Main explanation

Important concept

fgets 를 통해 사용자로부터 입력을 받은 뒤, 파이프 입력에 대하여(ls -al | grep fork) 명령어를 여러 커맨드로 분리한다. 즉 command[0]="ls -al", command[1]="grep fork" 와 같이 분리된다.

```
// "ls -al | grep fork"에 대해 파이프 처리
token = strtok(input, "|");
while (token != NULL && num_commands < MAX_PIPE_SIZE) {
    commands[num_commands++] = token;
    token = strtok(NULL, "|");
}
/*
commands[0] = ls -al
commands[1] = grep fork
*/
```

Figure 3) Separate by pipe ('|')

다음으로 각 명령어를 앞에서부터 하나씩 차근차근 실행하게 된다. 공백과 줄바꿈 문자 기준으로 문자열을 쪼갠 뒤 문자열 배열에 하나씩 값을 넣는다. 명령어를 실행할 함수 execve 는 명령어의 인자를 개별적으로 저장한 문자열 배열을 받아서 실행한다. 따라서 command 자체를 토큰화 하지 않고 넣을 시 문자열 통째로 넣을 시 오류가 난다.

```
arg_token = strtok(commands[i], " \n");
while (arg_token != NULL) {
    args[j++] = arg_token;
    arg_token = strtok(NULL, " \n");
}
args[j] = NULL;
/*
args[0] = ls
args[1] = -al
args[2] = NULL
*/
```

Figure 4) Separate by blank and new line

빈 커맨드라인 처리와, 쉘 종료, 디렉토리 변경에 대해서 우선적으로 처리를 해준다. execve 의 기능을 사용하지 않기에 fork 를 이용하여 프로세스를 생성하기 전에 일괄적으로 확인한다.

```
// 빈 커맨드 처리
if (j == 0) return;

// 쉘 종료 처리
if (strcmp(args[0], "quit") == 0) {
    exit(0);
}

// 디렉토리 변경 처리... "cd /home/~"
if (strcmp(args[0], "cd") == 0) {
    if (j < 2) {
        fprintf(stderr, "cd: missing argument\n");
    } else {
        if (chdir(args[1]) != 0) {
            perror("cd failed");
        }
    }
    return;
}
```

Figure 5) Null command, Shell quit, change directory

파이프에 관한 자세한 설명은 Unique features 에서 설명하도록 한다. 우선 하나의 단일 명령어만 들어왔다고 가정할 경우, 부모 프로세스가 자식 프로세스를 호출한 뒤 자식 프로세스가 명령어를 수행하고 부모 프로세스로 돌아간다. 즉 부모의 waiting 후, 자식 프로세스로 넘어갔을 때, PID 가 0 인 경우 execve 를 실행하게 된다. 특이하게도 현재 프로세스를 완전히 대체하며 명령어를 실행하는 기능을 가지고 있어, 메모리 누수를 신경 쓸 필요가 없다. execve 뒤에 배치함으로써 명령어 실행이 실패하는 경우에만 free 를 통해 메모리 관리를 해준다. 실행 가능한 명령어를 찾아서 가져오는 과정은 별도의 함수로 선언하였다. 절대 경로와 상대 경로를 이용한 실행 파일을 처음에

검증하고, 이후 PATH 를 통해 환경변수를 받는다. 이 문자열은 직접 수정할 수 없는 문자열로 복사해서 사용하는 과정이 필요하다.

```c
// 명령어 PATH 검색
char* command_path = find_command_path(args[0]);
if (command_path != NULL) {
    // 지정된 프로그램 실행
    // 현재 프로세스를 완전히 대체, 이후 자동 종료
    execve(command_path, args, environ);
    free(command_path);
    // 여기 코드는 실행되지 않음, 프로세스 대체
} else {
    fprintf(stderr, "Command not found: %s\n", args[0]);
    exit(1);
}
```

Figure 6) Child process command execution

```c
char* find_command_path(const char* command) {
    // 상대 경로와 절대 경로 처리... "./a.out"
    if (access(command, X_OK) == 0) {
        return strdup(command);
    }

    // 환경 변수 받기
    // getenv("PATH")로 얻은 문자열은 읽기 전용 문자열로, 직접 수정할 수 없음
    // 문자열을 복사한 후 수정, 이후 메모리 해제
    char* path = getenv("PATH");
    char* path_copy = strdup(path);
    char* dir = strtok(path_copy, ":");
    static char full_path[MAX_INPUT_SIZE];

    // 여러 환경 변수 경로들에 대해서 access 함수를 통해 실행 가능여부 확인
    while (dir != NULL) {
        snprintf(full_path, sizeof(full_path), "%s/%s", dir, command);
        if (access(full_path, X_OK) == 0) {
            free(path_copy);
            return strdup(full_path);
        }
        dir = strtok(NULL, ":");
    }

    free(path_copy);
    return NULL;
}
```

Figure 7) Finding command path

Unique features

파이프를 이용한 명령어 실행은 매우 독특한 구조로 이루어져 있다. 파이프는 한마디로 프로세스간 통신을 가능하게 해주는 매개체이다. "ls -al"을 통해 뽑힌 결과값을 "grep"하는 과정에서 input 과 output 을 조율해주는 게 필요한데, pipe() 명령어를 통해 한 쌍의 파이프를 생성하게 된다. 만약 커맨드가 남아있는 경우 파이프 쌍을 새롭게 생성하게 된다.

```
// 커맨드가 파이프로 이루어진 경우 파이프 생성
if (i < num_commands - 1) {
    if (pipe(fd) == -1) {
        perror("pipe failed");
        return;
    }
    // printf("----------\n");
    // printf("[%d]New Pipe generated... fd[%d], fd[%d]\n", i, fd[0], fd[1]);
}
```

Figure 8) Pipe generate

파이프는 fork 이후에 프로세스별로 다르게 활용된다. 먼저 자식 프로세스의 경우 이전에 넘겨받은 파이프_읽기(이전 명령어의 출력 값)가 있는지 체크하고 STDIN 으로 설정하게 된다. 또한 커맨드가 남아있는 경우 파이프_쓰기에 결과를 작성하고 아닐 경우 사용자 터미널 창에 출력하게 된다.

```
if (prev_fd != -1) {
    /*
    이전 프로세스에서 생성한 파이프의 읽기 디스크립터인 prev_fd를 현재 프로세스의 표준 입력(STDIN_FILENO)으로 복제
    이전 프로세스가 파이프를 통해 쓴 데이터를 현재 프로세스가 읽을 수 있도록 설정
    */
    // printf("[%d]Child process... prev_fd to STDIN\n", i);
    dup2(prev_fd, STDIN_FILENO);
    close(prev_fd);
}
```

```c
if (i < num_commands - 1) {
    /*
    현재 프로세스의 표준 출력(기본적으로 화면에 출력되는 내용)을 fd[1]로 변경
    현재 프로세스가 이 디스크립터를 통해 데이터를 쓸 수 있도록 설정
    */
    // printf("[%d]Child process... fd[1] to STDOUT\n", i);

    // 아래 코드에서 STDOUT을 리다이렉션하기에 프린트 문을 여기에 작성
    // printf("[%d]Executing... %s\n", i, commands[i]);
    dup2(fd[1], STDOUT_FILENO);
    close(fd[1]);
}
```

Figure 9) Child process STDIN, STDOUT setting

부모 프로세스의 경우 사용하지 않는 쓰기 파이프를 닫아주고(파이프 명령어가 남아있는 경우에 닫아도 되고 항상 닫아도 무방), 자식 프로세스를 위한 prev_fd 값을 위해 close 및 define 과정을 거친다. 이렇게 하면 반복문을 돌아가 새로운 커맨드에 대한 fork 와 pipe 를 하는 과정에서 이전 결과값을 무사히 자식 프로세스 input 으로 전달할 수 있게 된다.

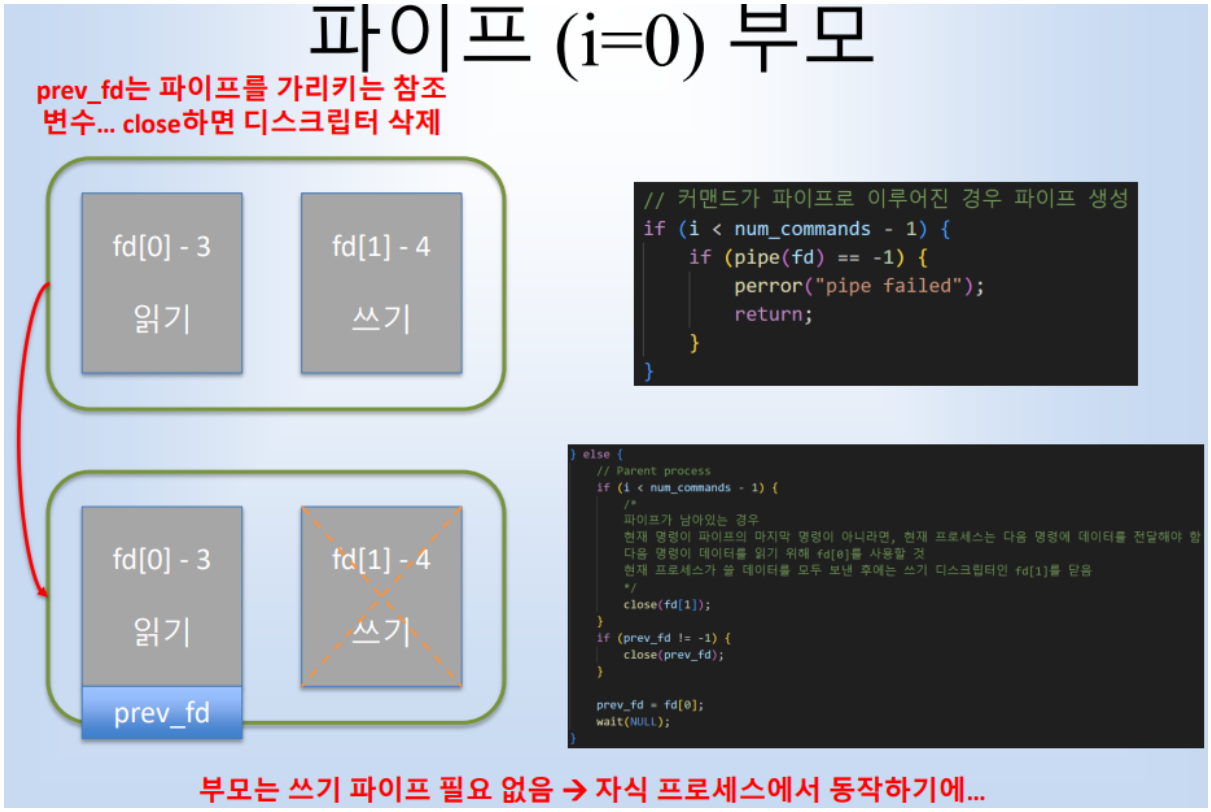
```c
} else {
    // Parent process
    // printf("[%d]Parent Process with %d, %d\n", i, fd[0], fd[1]);
    if (i < num_commands - 1) {
        /*
        파이프가 남아있는 경우
        현재 명령이 파이프의 마지막 명령이 아니라면, 현재 프로세스는 다음 명령에 데이터를 전달해야 함
        다음 명령이 데이터를 읽기 위해 fd[0]를 사용할 것
        현재 프로세스가 쓸 데이터를 모두 보낸 후에는 쓰기 디스크립터인 fd[1]를 닫음

        부모 프로세스는 fd[1]을 사용하지 않기에, if를 통해 체크하지 않고 닫아도 무방
        하지만 논리적으로 파이프가 남아있다면 fd[1]을 사용하지 않는게 이상적
        */
        close(fd[1]);
        // printf("[%d]Parent process... fd[1] closed\n", i);
    }

    if (prev_fd != -1) {
        // printf("[%d]Parent process... prev_fd closed\n", i);
        close(prev_fd);
    }
    // 이전 파이프에서 출력한 결과값 지정
    // printf("[%d]Parent process... prev_fd is defined, waiting\n\n", i);
    prev_fd = fd[0];
    wait(NULL);
}
```

```
else {
    if (prev_fd != -1) {
        close(prev_fd);
    }
    if (i < num_commands - 1) {
        close(fd[1]);
        prev_fd = fd[0];
    }
    wait(NULL);
}
```

Figure 10) Parent process previous pipe setting (original and modified version)

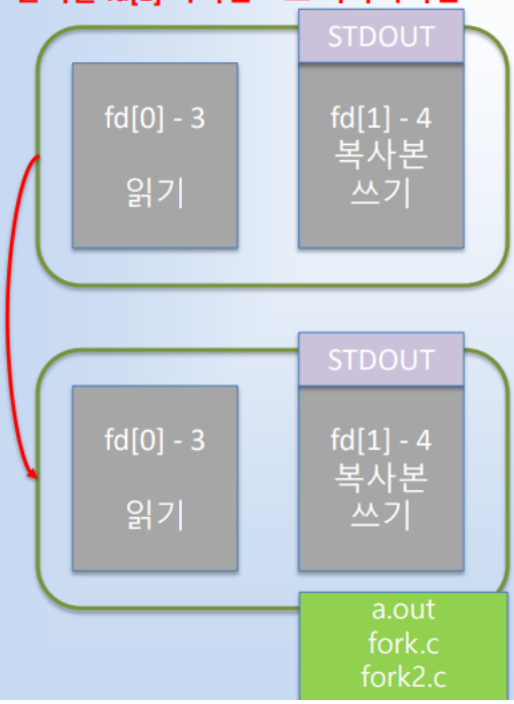
"ls -al | grep fork | sort" 명령어를 예시로, 이 일련의 동작 과정을 정리하면 다음과 같다. 부모는 항상 다음에 생성될 자식 프로세스를 위한 파이프 세팅을 하게 되고, 자식 프로세스는 설정된 값들을 바탕으로 명령어 STDIN, STDOUT 을 하게 된다.

- 부모 i=0, 자식 프로세스가 다다음에(i=1)일 때 사용할 읽기 파이프 세팅
- 자식 i=0, 결과값 쓰기 파이프에 저장, 이후 i=1 일 때 해당 값 사용



파이프 (i=0) 자식

자식 프로세스에서 발생하는 모든
출력을 fd[1] 복사본으로 리다이렉션

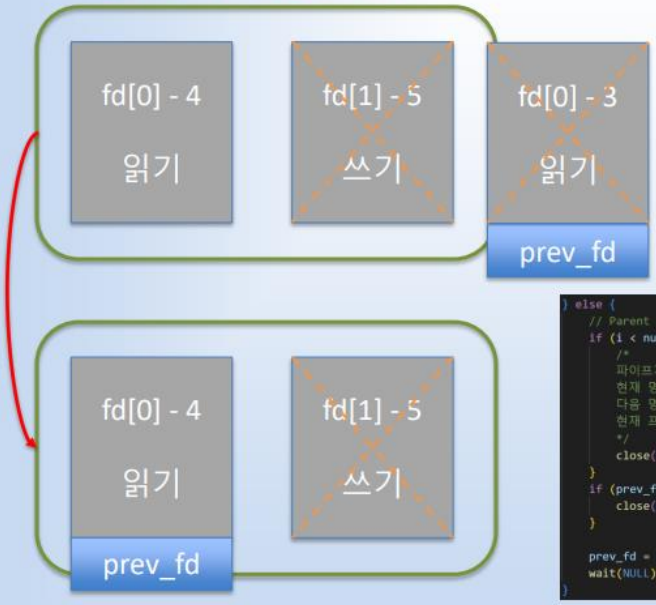



```c
// Child process
if (prev_fd != -1) {
    /*
    이전 프로세스에서 생성한 파이프의 읽기 디스크립터인 prev_fd를 현재 프로세스의 표준 입력
    이전 프로세스가 파이프를 통해 쓴 데이터를 현재 프로세스가 읽을 수 있도록 설정
    */
    dup2(prev_fd, STDIN_FILENO);
    close(prev_fd);
}

if (i < num_commands - 1) {
    /*
    현재 프로세스의 표준 출력(기본적으로 화면에 출력되는 내용)을 fd[1]로 변경
    현재 프로세스가 이 디스크립터를 통해 데이터를 쓸 수 있도록 설정
    */
    dup2(fd[1], STDOUT_FILENO);
    close(fd[1]);
}
```

```c
// 명령어 PATH 검색
char* command_path = find_command_path(args[0]);
if (command_path != NULL) {
    // 지정된 프로그램 실행
    // 현재 프로세스를 완전히 대체, 이후 자동 종료
    execve(command_path, args, environ);
    free(command_path);
} else {
    fprintf(stderr, "Command not found: %s\n", args[0]);
    exit(1);
}
```

파이프 (i=1) 부모

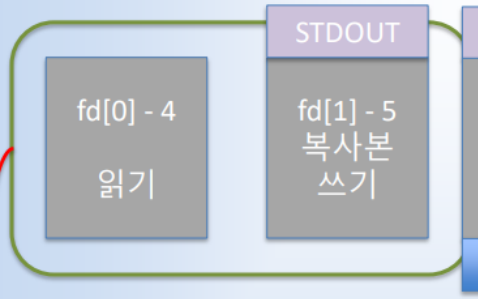
한쪽 스트림에 쓴 데이터는 다른 쪽
스트림에서 바로 읽을 수 있다.



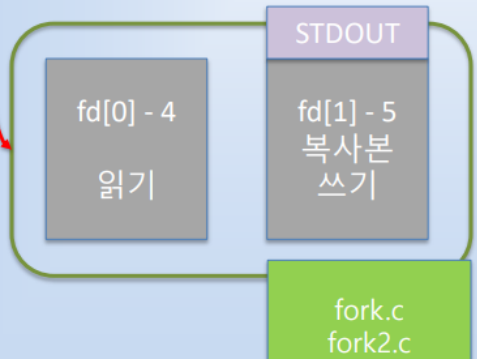

```c
// 커맨드가 파이프로 이루어진 경우 파이프 생성
if (i < num_commands - 1) {
    if (pipe(fd) == -1) {
        perror("pipe failed");
        return;
    }
}
```

```c
} else {
    // Parent process
    if (i < num_commands - 1) {
        /*
        파이프가 남아있는 경우
        현재 명령이 파이프의 마지막 명령이 아니라면, 현재 프로세스는 다음 명령에 데이터를 전달해야 함
        다음 명령이 데이터를 읽기 위해 fd[0]를 사용할 것
        현재 프로세스가 쓸 데이터를 모두 보낸 후에는 쓰기 디스크립터인 fd[1]를 닫음
        */
        close(fd[1]);
    }
    if (prev_fd != -1) {
        close(prev_fd);
    }

    prev_fd = fd[0];
    wait(NULL);
}
```

파이프 (i=1) 자식

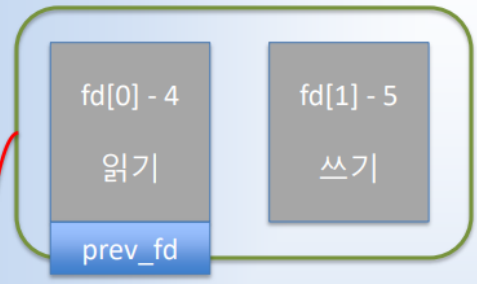



```
// Child process
if (prev_fd != -1) {
    /*
    이전 프로세스에서 생성한 파이프의 읽기 디스크립터인 prev_fd를 현재 프로세스의 표준 입력
    이전 프로세스가 파이프를 통해 쓴 데이터를 현재 프로세스가 읽을 수 있도록 설정
    */
    dup2(prev_fd, STDIN_FILENO);
    close(prev_fd);
}

if (i < num_commands - 1) {
    /*
    현재 프로세스의 표준 출력(기본적으로 화면에 출력되는 내용)을 fd[1]로 변경
    현재 프로세스가 이 디스크립터를 통해 데이터를 쓸 수 있도록 설정
    */
    dup2(fd[1], STDOUT_FILENO);
    close(fd[1]);
}
```

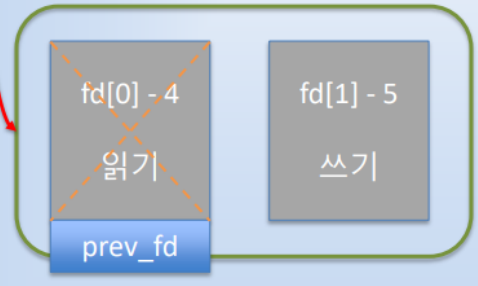



```
// 명령어 PATH 검색
char* command_path = find_command_path(args[0]);
if (command_path != NULL) {
    // 지정된 프로그램 실행
    // 현재 프로세스를 완전히 대체, 이후 자동 종료
    execve(command_path, args, environ);
    free(command_path);
} else {
    fprintf(stderr, "Command not found: %s\n", args[0]);
    exit(1);
}
```

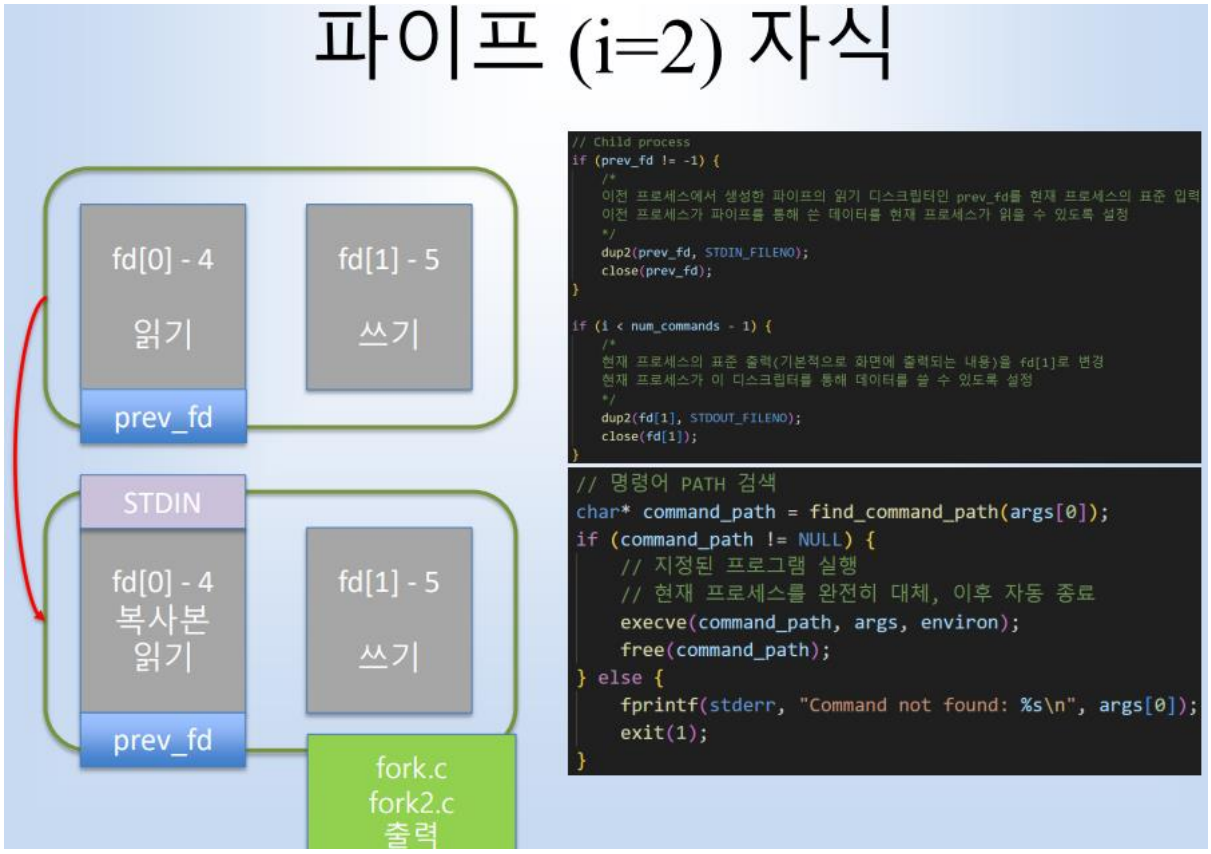
파이프 (i=2) 부모




```
// 커맨드가 파이프로 이루어진 경우 파이프 생성
if (i < num_commands - 1) {
    if (pipe(fd) == -1) {
        perror("pipe failed");
        return;
    }
}
```




```
} else {
    // Parent process
    if (i < num_commands - 1) {
        /*
        파이프가 남아있는 경우
        현재 명령이 파이프의 마지막 명령이 아니라면, 현재 프로세스는 다음 명령에 데이터를 전달해야 함
        다음 명령이 데이터를 읽기 위해 fd[0]를 사용할 것
        현재 프로세스가 쓸 데이터를 모두 보낸 후에는 쓰기 디스크립터인 fd[1]를 닫음
        */
        close(fd[1]);
    }
    if (prev_fd != -1) {
        close(prev_fd);
    }

    prev_fd = fd[0];
    wait(NULL);
}
```




```
SiSH (/home/solid/2024-os-hw1) >>> ls -al | grep fork | sort
----------
[0]New Pipe generated... fd[3], fd[4]
[0]Parent Process with 3, 4
[0]Parent process... fd[1] closed
[0]Parent process... prev_fd is defined, waiting

[0]Child Process with 3, 4
[0]Child process... fd[1] to STDOUT
[0]Executing... ls
----------
[1]New Pipe generated... fd[4], fd[5]
[1]Parent Process with 4, 5
[1]Parent process... fd[1] closed
[1]Parent process... prev_fd closed
[1]Parent process... prev_fd is defined, waiting

[1]Child Process with 4, 5
[1]Child process... prev_fd to STDIN
[1]Child process... fd[1] to STDOUT
[1]Executing... grep
[2]Parent Process with 4, 5
[2]Parent process... prev_fd closed
[2]Parent process... prev_fd is defined, waiting

[2]Child Process with 4, 5
[2]Child process... prev_fd to STDIN
-rw-rw-r-- 1 solid users   456 Sep 15 07:54 fork.c
-rw-rw-r-- 1 solid users  1842 Sep 15 07:54 fork2.c
```

Figure 11) Pipe logical flow with print statement

Outro

Environment

Linux environment with assam server. Also tested on Windows Ubuntu.

```
solid@solid-00001-58989694-7jl4n:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:        22.04
Codename:       jammy
```

```
adrd@ADALIV:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:       jammy
```

Screenshots

```
SiSH (/home/solid/2024-os-hw1) >>> ls
LICENSE  README.md  SiSH.c  a.out  fork.c  fork2.c  getenv.c  stat.c
SiSH (/home/solid/2024-os-hw1) >>> pwd
/home/solid/2024-os-hw1
```

Figure 12) Command result (ls, pwd)

```
SiSH (/home/solid/2024-os-hw1) >>> ls -al | grep fork | sort
-rw-rw-r-- 1 solid users   456 Sep 15 07:54 fork.c
-rw-rw-r-- 1 solid users  1842 Sep 15 07:54 fork2.c
SiSH (/home/solid/2024-os-hw1) >>> ls -al | grep fork | sort -r
-rw-rw-r-- 1 solid users  1842 Sep 15 07:54 fork2.c
-rw-rw-r-- 1 solid users   456 Sep 15 07:54 fork.c
```

Figure 13) Command result (pipe)

```
SiSH (/home/solid/2024-os-hw1) >>> cd ../test
SiSH (/home/solid/test) >>> gcc test1.c
SiSH (/home/solid/test) >>> ./a.out
hi
SiSH (/home/solid/test) >>> gcc test2.c
SiSH (/home/solid/test) >>> ./a.out
Type something... str : hi
Type something... num : 100

hi
100
```

Figure 14) Command result (cd, gcc)