

운영체제론 실습 4주차

정보보호연구실 @ 한양대학교

프로세스 (Process)

메모리에 적재된 프로그램 (실행 중 혹은 실행 가능)

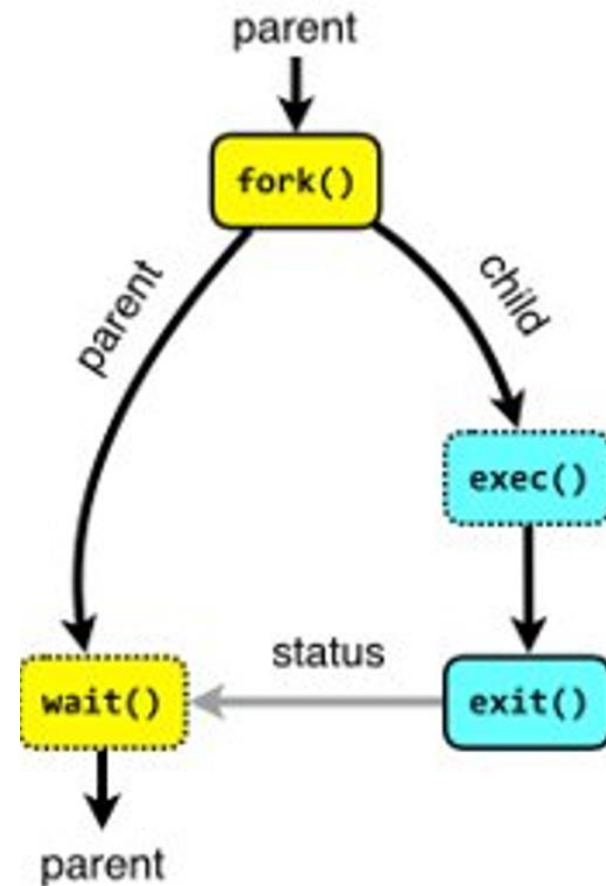
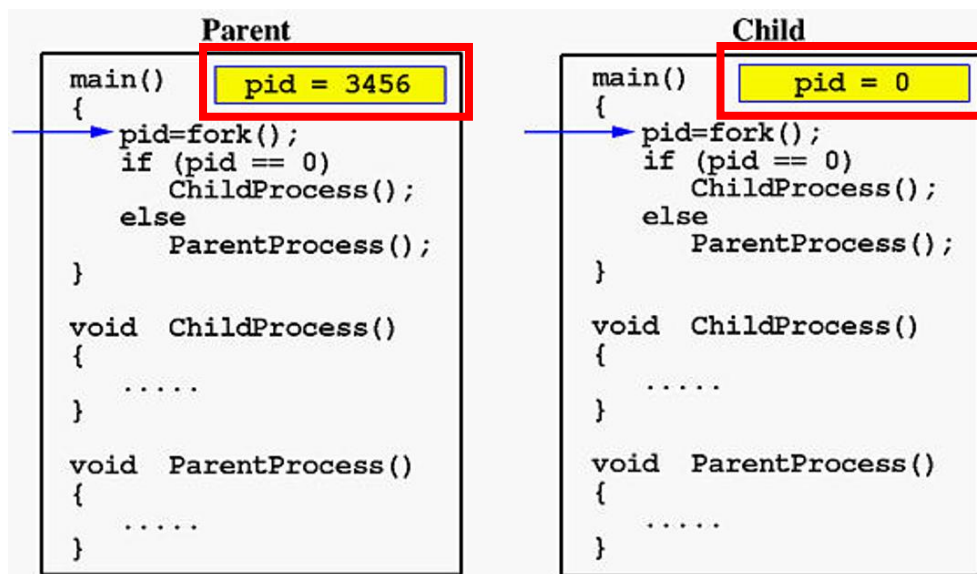
- 프로세스 제어 블록, PCB (Process Control Block)

- 프로세스의 정보를 저장하는 자료구조

- 프로세스의 식별번호 (Process ID, PID)
 - 프로세스의 **상태** (실행, 대기 등)
 - 다음 명령어의 포인터 (Program Counter, PC)
 - 스케줄링 **우선순위**
 - 자원 접근 **권한**
 - **부모** 프로세스 정보
 - 할당된 **자원**의 포인터
 - 프로세스 **실행** 문맥
 - ...

fork()

- `pid_t fork(void)`: 자식 프로세스 생성 함수
 - `fork()`를 기점으로 부모와 자식 프로세스가 갈라짐
 - 해당 프로세스가 부모인지, 자식인지는 `fork()` 함수의 반환값으로 알 수 있음



create_child.c

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

int main(){
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        return 1;
    }
    else if (pid == 0) {
        printf("=====\n");
        printf("CHILD: ls command\n");
        execlp("/bin/ls", "ls", NULL);
        printf("execlp call was unsuccessful\n");
    }
    else {
        wait(NULL);
        //waitpid(pid, NULL, 0);
        printf("-----\n");
        printf("PARENT: Child Complete\n");
        printf("-----\n");
    }
    return 0;
}
```

프로세스 생성 실패

자식 프로세스

부모 프로세스

\$./create_child

- execlp() 뒤 printf()
- wait()? waitpid()?

num_of_process.c

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

int main(){
    int i;

    for (i=0; i<4, i++) {
        fork();
    }

    sleep(120);

    return 0;
}
```

\$./num_of_process &

- 총 만들어지는 프로세스의 개수는?
(원 프로세스 포함)

- \$ pstree {pid}
- 어째서?

- 유용한 명령어들

- \$ ps aux | grep {process_name}
- \$ pgrep {process_name} | wc -l

num_of_process.c

```
dongmin1@dongmin1-VirtualBox:~/week5/ex$ pstree 9608
num_of_process--num_of_process--num_of_process--num_of_process--num_of_pro+
                                     |num_of_process
                                     |num_of_process
                                     |num_of_process
num_of_process--num_of_process--num_of_process
                                     |num_of_process
num_of_process--num_of_process
num_of_process

dongmin1@dongmin1-VirtualBox:~/week5/ex$ ps aux | grep num_of_process
dongmin1  9608  0.0  0.0  4372  736 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9609  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9610  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9611  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9612  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9613  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9614  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9615  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9616  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9617  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9618  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9619  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9620  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9621  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9622  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9623  0.0  0.0  4372   72 pts/0    S   11:24   0:00  ./num_of_process
dongmin1  9627  0.0  0.0 22820  968 pts/0    S+  11:25   0:00  grep --color=auto num_of_process
```

execv()

• `int execv(const char *path, char *const argv[]):`
프로그램 실행 함수

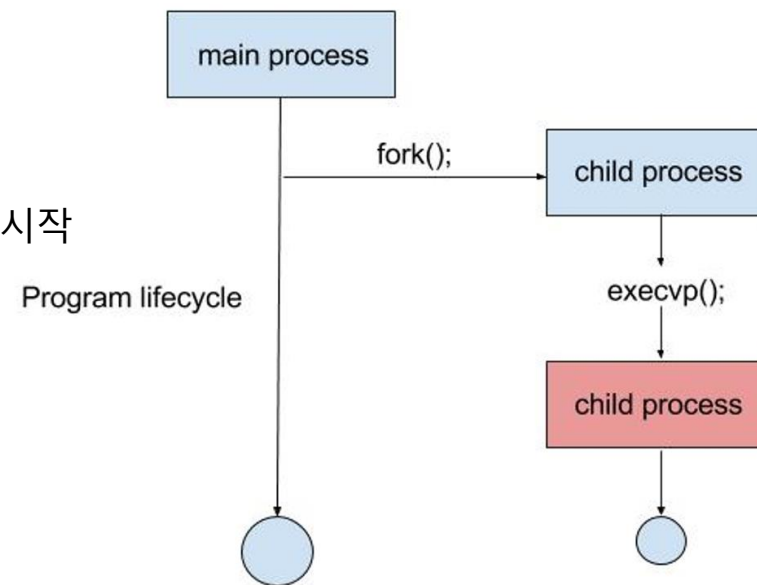
- 프로그램의 **경로, 이름**, 그리고 **옵션**을 인자로 받아 실행
- 현재의 프로세스를 **새로운** 프로세스로 **대체**

```
char *const paramList[] = {"/bin/ls", "-l", ".", NULL};
```

```
execv("/bin/ls", paramList);
```

인자 배열은 보통 프로그램 경로나 이름으로 시작

경로/명령어 인자 배열



wait(), waitpid()

- **pid_t wait(int *status)**

- fork() 되어 나온 자식 프로세스가 종료할 때까지 기다림
- 종료된 자식 프로세스의 pid를 반환 (실패 시, EOF 반환)
- status: 자식 프로세스가 종료할 때의 상태 정보를 저장할 위치

- **pid_t waitpid(pid_t pid, int *status, int options)**

- 인자로 받은 pid를 가진 자식 프로세스의 종료를 기다림
- 다수의 자식을 가질 경우 특정 자식을 지정 가능
(pid가 -1일 경우 어느 한 자식이라도 종료될 경우 대기 종료)
- 자주 쓰이는 options: WNOHANG, 0

wait_child.c

\$./wait_child

- 외부 입력이 없을 때

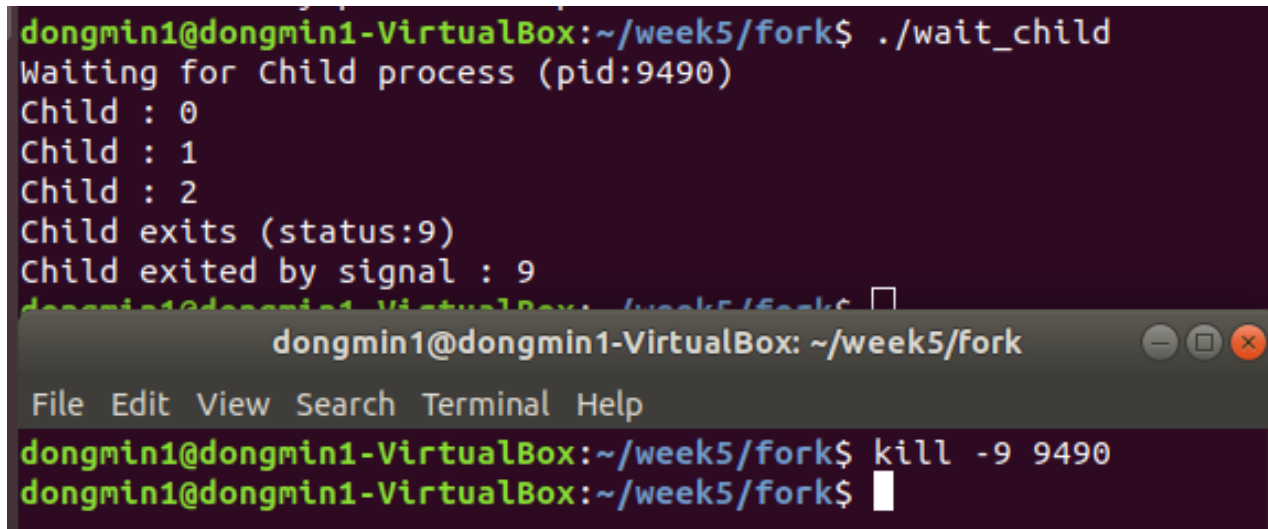
```
dongmin1@dongmin1-VirtualBox:~/week5/fork$ ./wait_child
Waiting for Child process (pid:9486)
Child : 0
Child : 1
Child : 2
Child : 3
Child : 4
Child : 5
Child : 6
Child : 7
Child : 8
Child : 9
Child exits (status:768)
Child exited by process completeion : 3
```

wait_child.c

\$./wait_child

- 시그널을 보냈을 때

- 파일을 실행 시, 자식 프로세스의 pid 출력
- 다른 터미널을 열어 \$ kill -9 {pid} 명령으로 프로세스를 강제 종료 (9)



```
dongmin1@dongmin1-VirtualBox:~/week5/fork$ ./wait_child
Waiting for Child process (pid:9490)
Child : 0
Child : 1
Child : 2
Child exits (status:9)
Child exited by signal : 9
dongmin1@dongmin1-VirtualBox:~/week5/fork$
```

The screenshot shows a terminal window titled 'dongmin1@dongmin1-VirtualBox: ~/week5/fork'. The first terminal session runs './wait_child', which prints 'Waiting for Child process (pid:9490)', followed by 'Child : 0', 'Child : 1', 'Child : 2', 'Child exits (status:9)', and 'Child exited by signal : 9'. A second terminal window is opened, showing the command 'kill -9 9490' being entered and executed.

wait(), waitpid()

- if (WIFEXITED(status))
 - 자식 프로세스가 정상적으로 종료되었을 경우
- if WIFSIGNALED(status)
 - 자식 프로세스가 어떠한 시그널을 받아 종료됨

```
if (retval > 0) {  
    if (WIFEXITED(status)) {  
        printf("Child exited by process completeion : %d\n", WEXITSTATUS(status));  
    }  
    if (WIFSIGNALED(status)) {  
        printf("Child exited by signal : %d\n", WTERMSIG(status));  
    }  
}
```

line_a.c

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

int value = 5;

int main(){
    pid_t pid;

    pid = fork()

    if (pid == 0) {
        value += 15;
        return 0;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("PARENT: value - %d\n", value);
        return 0;
    }

    return 0;
}
```

\$./line_a

- 출력될 value의 값은?

- value의 값이 바뀌지 않는 이유
- value를 바꾸고 싶다면?

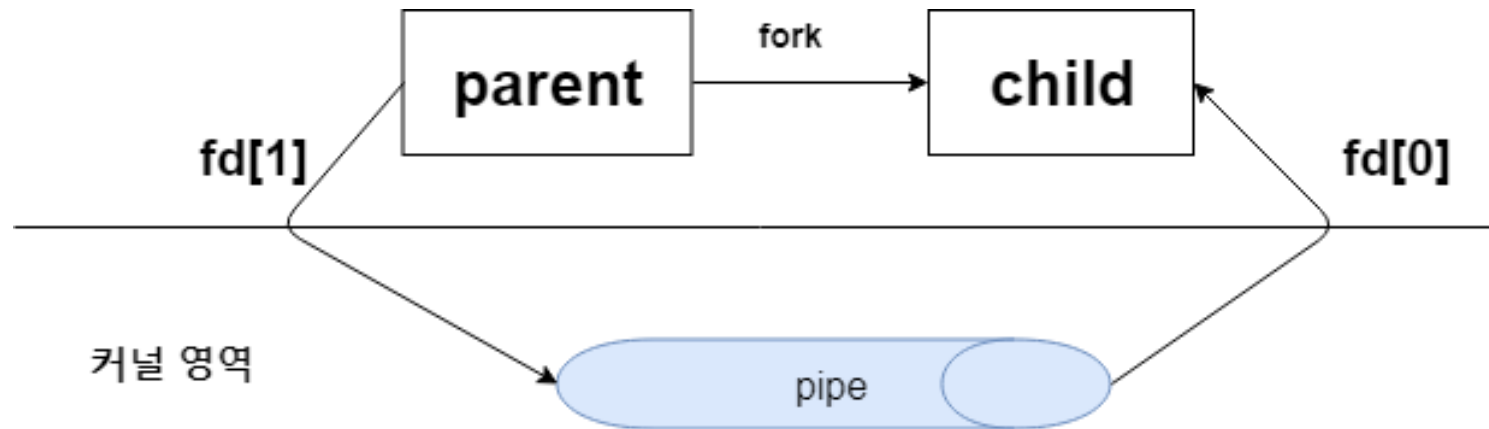
- \$ ls | grep line

- 명령어의 의미?
- | ← ?

pipe()

- `int pipe(int pipefd[2]):` 파이프 생성 함수

- 부모-자식 프로세스 간의 통신을 위한 채널
- `fd[1]`: **write** / `fd[0]`: **read**
- 해당 프로세스가 쓰지 않는 쪽은 반드시 **닫기** (일방향)



ordinary_pipe.c

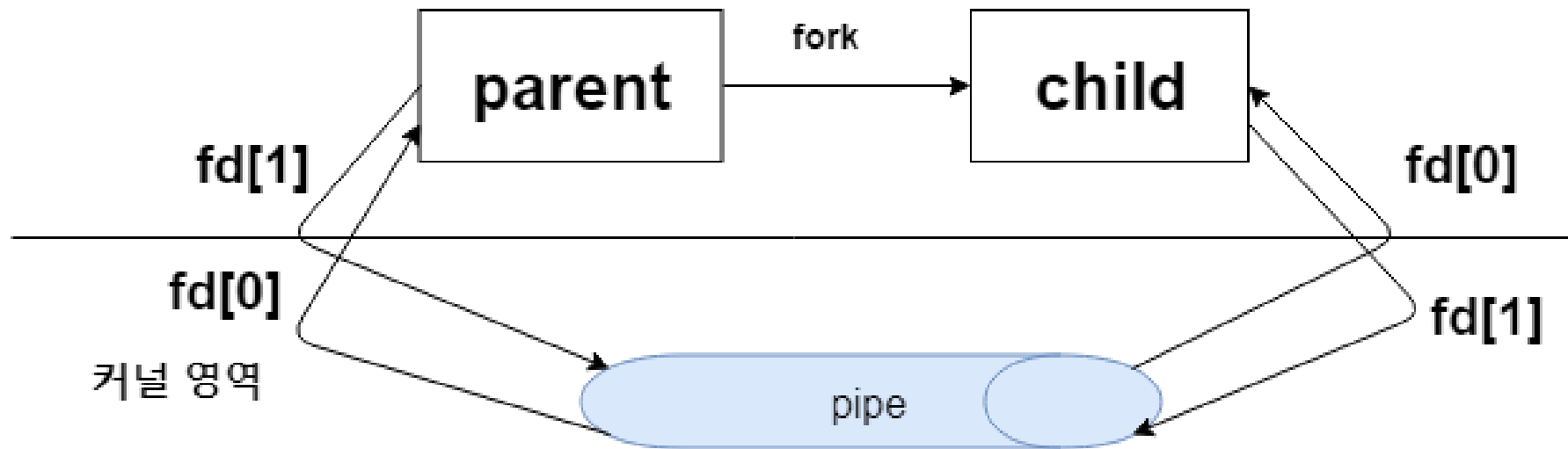
\$./ordinary_pipe

- 부모에게서 자식에게로 메시지 전달
 - 부모: write 를 담당하므로, READ_END 폐쇄
 - 자식: read 를 담당하므로, WRITE_END 폐쇄

```
yeeun@yeeun-virtual-machine:~/Desktop/test$ ./ordinary_pipe
PARENT PROCESS
CHILD PROCESS
Read buffer: "Greetings"
```

pipe()

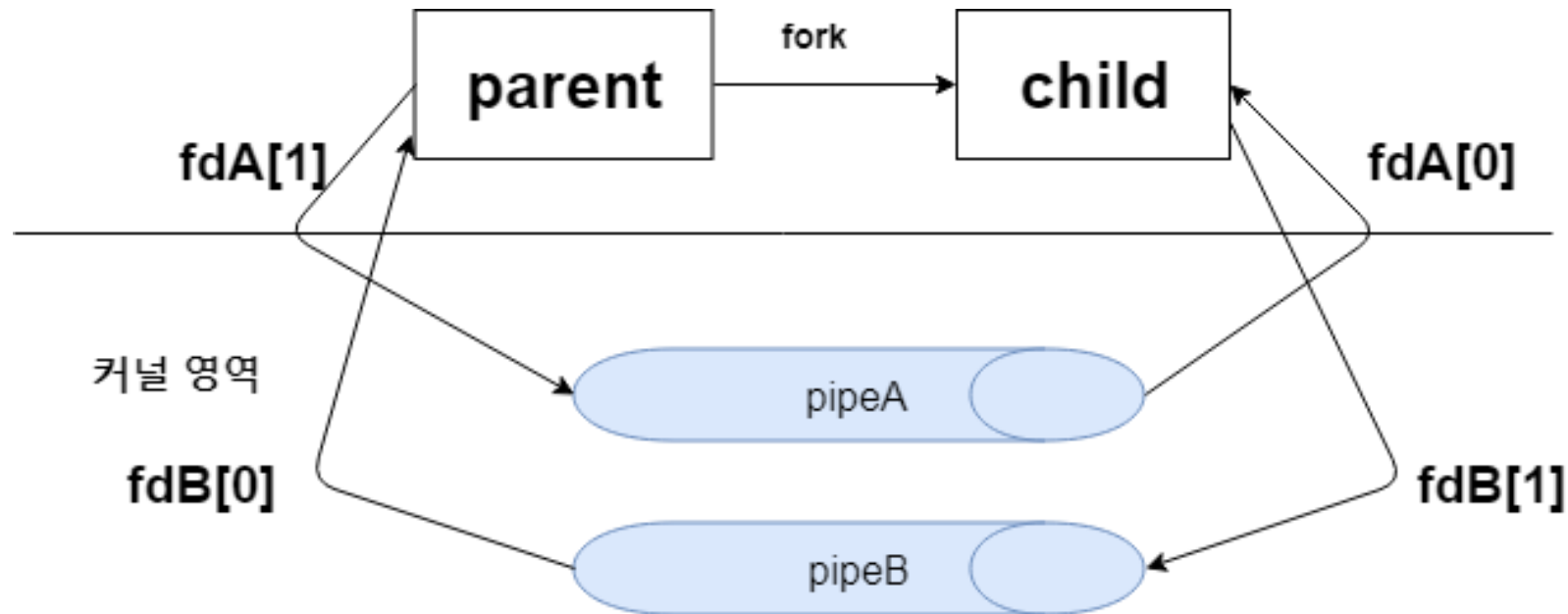
- 양방향으로 통신을 하고 싶다면
 - 쓰지 않는 쪽을 닫지 않고 열어둔다면?
 - 아래의 상황은 가능할까?



pipe()

• 양방향: 2개의 파이프

- 부모 → 자식으로 메시지를 보내는 파이프 (fdA)
- 자식 → 부모로 메시지를 보내는 파이프 (fdB)



pipe()

- 양방향 메시지 전달 프로그램 만들어보기

- line_a.c 파일을 의도하던 대로 바꿀 수 있나?
- `$ gcc -o {program_name} {source_file(.c)}`

```
yeeun@yeeun-virtual-machine:~/Desktop/test$ ./two_pipe

    child got message: parent 0
parent got message: child 10000
    child got message: parent 1
parent got message: child 10001
    child got message: parent 2
parent got message: child 10002
    child got message: parent 3
parent got message: child 10003
    child got message: parent 4
parent got message: child 10004
yeeun@yeeun-virtual-machine:~/Desktop/test$ ./line_a_2
PARENT: value = 20
```

오늘의 과제

- 양방향 메시지 전달 프로그램 만들어보기

- line_a_2.c

- two_pipe.c

- int count: **parent**는 0부터, **child**는 1000부터 시작
 - parent가 보낼 메시지: "parent {count}"
 - child가 보낼 메시지: "child {count}"
 - 메시지를 보낸 후 count에 +1 (count++)
 - 파이프를 통해 상대가 보낸 메시지를 읽고 출력
 - 1초 sleep