

# 시스템 프로그래밍

**Linux system programming**

프로세스 제어



# 프로세스 생성

- fork() 시스템 호출
  - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
  - 자기복제(自己複製)

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

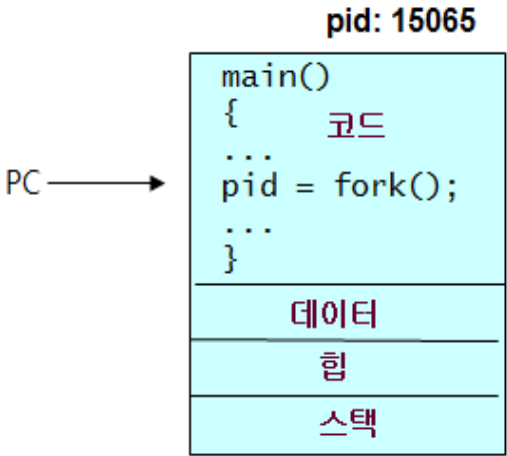
```
#include <unistd.h>
```

```
pid_t fork(void);
```

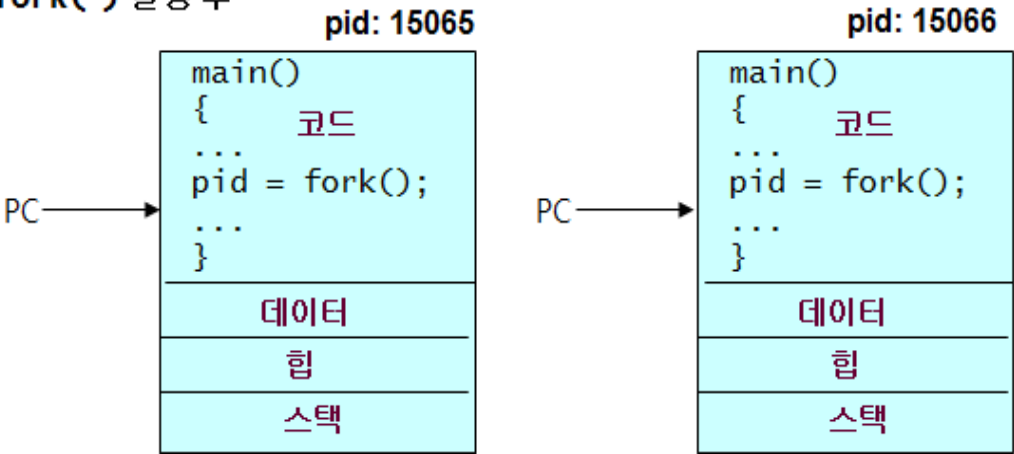
새로운 자식 프로세스를 생성한다. 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.



fork( ) 실행 전



fork( ) 실행 후



- `fork()`는 한 번 호출되면 두 번 리턴한다.
  - 자식 프로세스에게는 0을 리턴하고
  - 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.
- 부모 프로세스와 자식 프로세스는 병행적으로 각각 실행을 계속한다.



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

/* 자식 프로세스를 생성한다. */
int main()
{
    int pid;
    printf("[%d] 프로세스 시작 \n", getpid());
    pid = fork();
    printf("[%d] 프로세스 : 리턴값  %d\n", getpid(), pid);
}
```

fork1.c



# 부모 프로세스와 자식 프로세스 구분

- fork() 호출 후에 리턴값이 다르므로 이 리턴값을 이용하여
- 부모 프로세스와 자식 프로세스를 구별하고
- 서로 다른 일을 하도록 할 수 있다.

```
pid = fork();  
if ( pid == 0 )  
{ 자식 프로세스의 실행 코드 }  
else  
{ 부모 프로세스의 실행 코드 }
```



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

fork2.c

```
/* 부모 프로세스가 자식 프로세스를 생성하고 서로 다른 메시지를 프린트한다. */
int main()
{
    int pid;

    pid = fork();
    if(pid == 0){ // 자식 프로세스
        printf("[Child] : Hello, world pid=%d\n",getpid());
    }
    else { // 부모 프로세스
        printf("[Parent] : Hello, world pid=%d\n",getpid());
    }
}
```



## 두 개의 자식 프로세스 생성

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

/* 부모 프로세스가 두 개의 자식 프로세스를 생성한다. */
int main()
{
    int pid1, pid2;

    pid1 = fork();
    if (pid1 == 0) {
        printf("[Child 1] : Hello, world ! pid=%d\n",getpid());
        exit(0);
    }

    pid2 = fork();
    if (pid2 == 0) {
        printf("[Child 2] : Hello, world ! pid=%d\n",getpid());
        exit(0);
    }
}
```

fork3.c





# 프로세스 기다리기: wait()

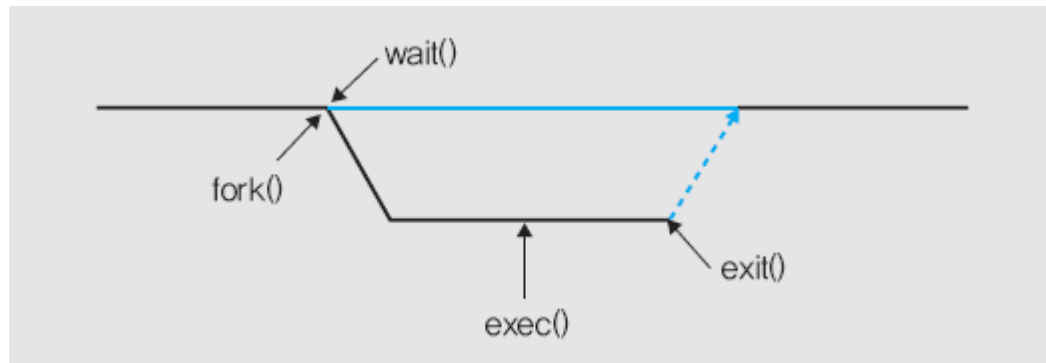
- 자식 프로세스 중의 하나가 끝날 때까지 기다린다.
  - 끝난 자식 프로세스의 종료 코드가 status에 저장되며
  - 끝난 자식 프로세스의 번호를 리턴한다.

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

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

```
pid_t wait(int *status);
```

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```



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

/* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
int main()
{
    int pid, child, status;

    printf("[%d] 부모 프로세스 시작 \n", getpid( ));
    pid = fork();
    if (pid == 0) {
        printf("[%d] 자식 프로세스 시작 \n", getpid( ));
        exit(1);
    }

    child = wait(&status); // 자식 프로세스가 끝나기를 기다린다.
    printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), child);
    printf("\t종료 코드 %d\n", status >> 8);
}
```



## waitPid.c

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

/\* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. \*/

```
int main()
{
    int pid1, pid2, child, status;

    printf("[%d] 부모 프로세스 시작 %dn", getpid( ));
    pid1 = fork();
    if (pid1 == 0) {
        printf("[%d] 자식 프로세스 #1 시작 %dn", getpid( ));
        sleep(1);
        printf("[%d] 자식 프로세스 #1 종료 %dn", getpid( ));
        exit(1);
    }
```

```
    pid2 = fork();
    if (pid2 == 0) {
        printf("[%d] 자식 프로세스 #2 시작 %dn", getpid( ));
        sleep(2);
        printf("[%d] 자식 프로세스 #2 종료 %dn", getpid( ));
        exit(2);
    }
```

```
    child = waitpid(pid1, &status, 0); // 자식 프로세스 #1의 종료를 기다린다.
    printf("[%d] 자식 프로세스 #1 %d 종료 %dn", getpid(), child);
    printf("%t종료 코드 %d%dn", status >> 8);
}
```

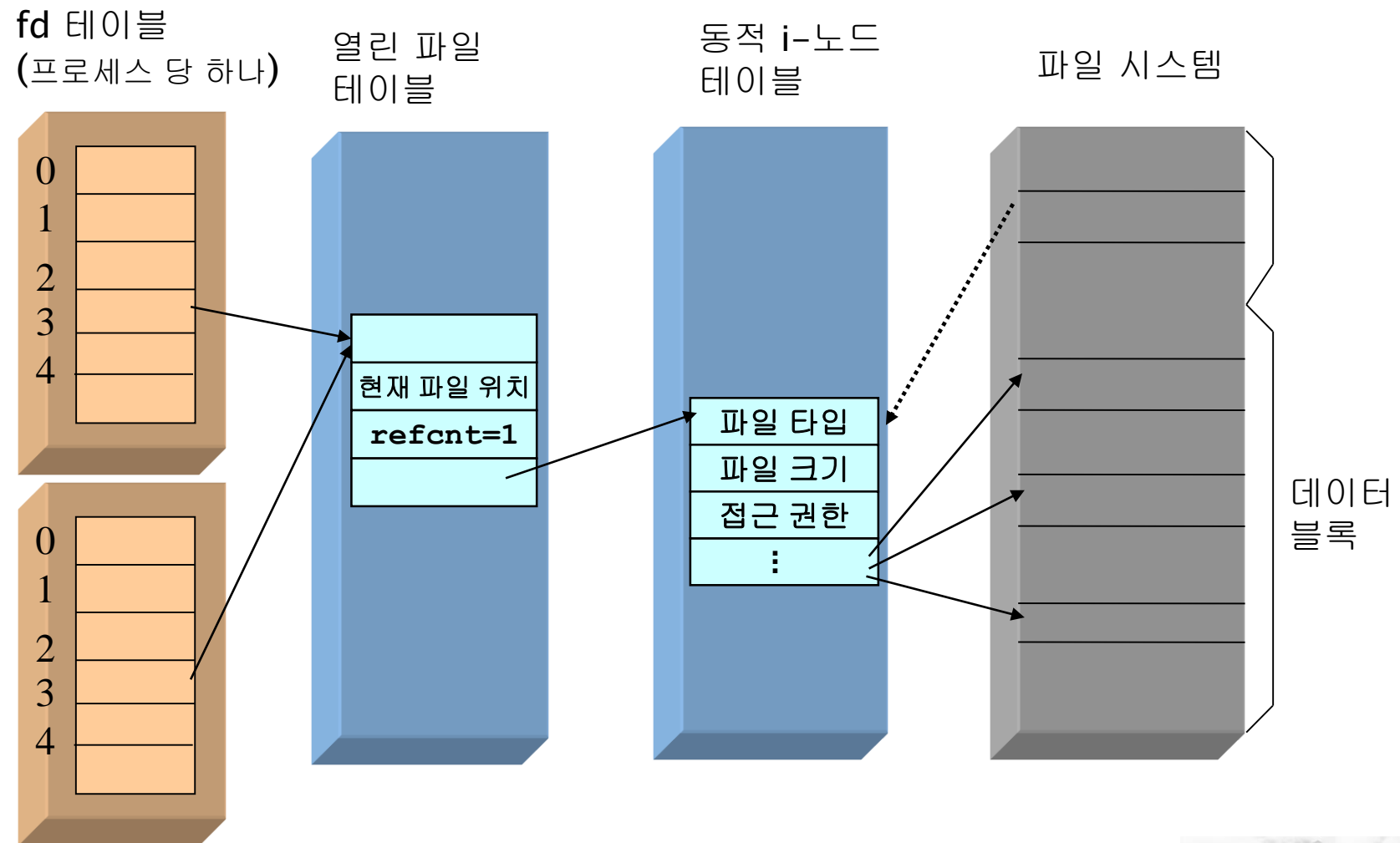


# fork() 후에 파일 공유

- 자식은 부모의 fd 테이블을 복사한다.
  - 부모와 자식이 같은 파일 디스크립터를 공유
  - 같은 파일 오프셋을 공유
  - 부모와 자식으로부터 출력이 서로 섞이게 됨
- 자식에게 상속되지 않는 성질
  - fork()의 반환값
  - 프로세스 ID
  - 파일 잠금
  - 설정된 알람과 시그널



# fork()



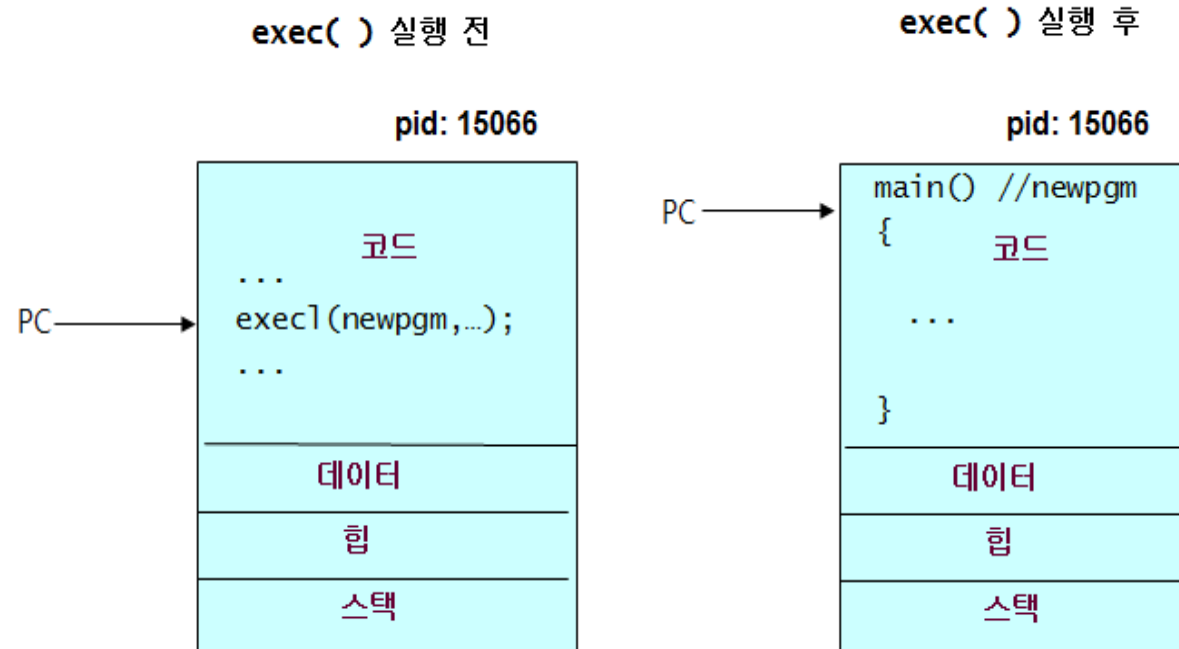
# 프로그램 실행

- fork() 후
  - 자식 프로세스는 부모 프로세스와 똑같은 코드 실행
- 자식 프로세스에게 새 프로그램 실행
  - exec() 시스템 호출 사용
  - 프로세스 내의 프로그램을 새 프로그램으로 대체
- 보통 fork() 후에 exec( )



# 프로그램 실행: exec()

- 프로세스가 exec() 호출을 하면,
  - 그 프로세스 내의 프로그램은 완전히 새로운 프로그램으로 대체
  - 자기대치(自己代置)
  - 새 프로그램의 main()부터 실행이 시작된다.



## 프로그램 실행: exec()

- exec() 호출이 성공하면 리턴할 곳이 없어진다.
- 성공한 exec() 호출은 절대 리턴하지 않는다.

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

```
int execl(char* path, char* arg0, char* arg1, ... , char* argn, NULL)
```

```
int execv(char* path, char* argv[ ])
```

```
int execlp(char* file, char* arg0, char* arg1, ... , char* argn, NULL)
```

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

호출한 프로세스의 코드, 데이터, 힙, 스택 등을 path가 나타내는 새로운 프로그램으로 대체한 후 새 프로그램을 실행한다.

성공한 exec( ) 호출은 리턴하지 않으며 실패하면 -1을 리턴한다.





# fork/exec

- 보통 fork() 호출 후에 exec() 호출
  - 새로 실행할 프로그램에 대한 정보를 arguments로 전달한다
  - .
- exec() 호출이 성공하면
  - 자식 프로세스는 새로운 프로그램을 실행하게 되고
  - 부모는 계속해서 다음 코드를 실행하게 된다.

```
if ((pid = fork()) == 0 ){  
    exec( arguments );  
    exit(1);  
}  
// 부모 계속 실행
```



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

exec1.c

```
/* 자식 프로세스를 생성하여 echo 명령어를 실행한다. */
int main( )
{
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl("/bin/echo", "echo", "hello", NULL);
        fprintf(stderr, "첫 번째 실패");
        exit(1);
    }
    printf("부모 프로세스 끝\n");
}
```



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

/* 세 개의 자식 프로세스를 생성하여 각각 다른 명령어를 실행한다. */
int main( )
{
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl("/bin/echo", "echo", "hello", NULL);
        fprintf(stderr,"첫 번째 실패");
        exit(1);
    }

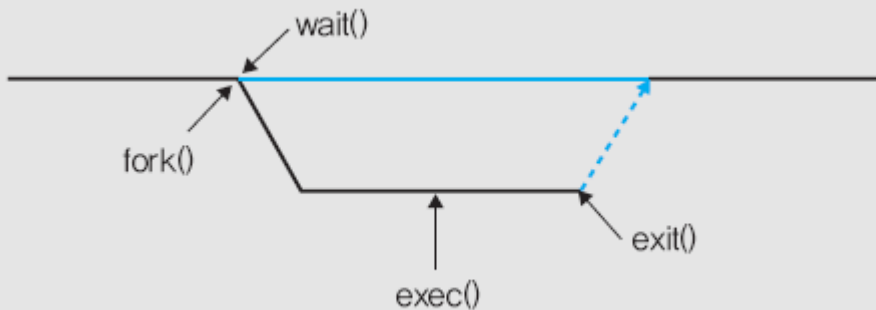
    if (fork( ) == 0) {
        execl("/bin/date", "date", NULL);
        fprintf(stderr,"두 번째 실패");
        exit(2);
    }

    if (fork( ) == 0) {
        execl("/bin/ls", "ls", "-l", NULL);
        fprintf(stderr,"세 번째 실패");
        exit(3);
    }
    printf("부모 프로세스 끝\n");
}
```



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

/* 명령줄 인수로 받은 명령을 실행시킨다. */
int main(int argc, char *argv[])
{
    int child, pid, status;
    pid = fork( );
    if (pid == 0) { // 자식 프로세스
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "%s:실행 불가\n", argv[1]);
    } else { // 부모 프로세스
        child = wait(&status);
        printf("[%d] 자식 프로세스 %d 종료\n", getpid(), pid);
        printf("종료 코드 %d\n", status >> 8);
    }
}
```



# system()

```
#include <stdlib.h>
```

```
int system(const char *cmdstring);
```

이 함수는 /bin/sh -c cmdstring를 호출하여 cmdstring에 지정된 명령어를 실행하며, 명령어가 끝난후 명령어의 종료코드를 반환한다.

- 자식 프로세스를 생성하고 /bin/sh로 하여금 지정된 명령어를 실행시킨다  
system("date > file");
- system() 함수 구현
  - fork(), exec(), waitpid() 시스템 호출을 이용
- 반환값
  - 명령어의 종료코드
  - -1 with errno: fork() 혹은 waitpid() 실패
  - 127 : exec() 실패



```
#include <stdio.h>
#include <stdlib.h>
int main() {
    system("pstree");
    system("ls -l");
    printf("system() 함수는 쉘 명령을 수행할 수 있습니다.\n");
    exit(0);
}
```

systemTest.c



## system() 함수 구현

system.c

```
#include <sys/types.h> /* system.c */
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int mysystem(const char *cmdstring)
{
    pid_t pid;    int status;

    if (cmdstring == NULL) /* 명령어가 NULL인 경우 */
        return(1);

    if ( (pid = fork()) < 0) {
        status = -1; /* 프로세스 생성 실패 */

    } else if (pid == 0) { /* 자식 프로세스 */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127); /* execl 오류 */
    } else { /* 부모 프로세스 */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* waitpid()로부터 EINTR외의 오류 */
                break;
            }
    }
    return(status);
}
```



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

systemCall.c

```
int main()
{
    int status;
    if ((status = system("date")) < 0)
        perror("mysystem() 오류");
    printf("종료코드 %d\n", WEXITSTATUS(status));

    if ((status = system("hello")) < 0)
        perror("mysystem() 오류");
    printf("종료코드 %d\n", WEXITSTATUS(status));

    if ((status = system("who; exit 44")) < 0)
        perror("mysystem() 오류");
    printf("종료코드 %d\n", WEXITSTATUS(status));
}
```





# 입출력 재지정

- 명령어의 표준 출력이 파일에 저장  
\$ 명령어 > 파일
- 출력 재지정 기능 구현
  - 파일 디스크립터 fd를 표준출력(1)에 dup2()  
fd = open(argv[1], O\_CREAT|O\_TRUNC|O\_WRONLY, 0600);  
dup2(fd, 1);

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

```
int dup(int oldfd);
```

oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 반환한다.

```
int dup2(int oldfd, int newfd);
```

oldfd을 newfd에 복제하고 복제된 새로운 파일 디스크립터를 반환한다.



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

redirect1.c

```
/* 표준 출력을 파일에 재지정하는 프로그램 */
int main(int argc, char* argv[])
{
    int fd, status;
    fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
    dup2(fd, 1); /* 파일을 표준출력에 복제 */
    close(fd);
    printf("Hello stdout !\n");
    fprintf(stderr, "Hello stderr !\n");
}
```



redirect2.c

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

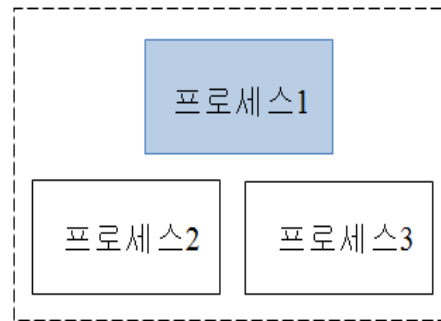
/* 자식 프로세스의 표준 출력을 파일에 재지정한다. */
int main(int argc, char* argv[])
{
    int child, pid, fd, status;

    pid = fork( );
    if (pid == 0) {
        fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
        dup2(fd, 1); /* 파일을 표준출력에 복제 */
        close(fd);
        execvp(argv[2], &argv[2]);
        fprintf(stderr, "%s:실행 불가\n",argv[1]);
    } else {
        child = wait(&status);
        printf("[%d] 자식 프로세스 %d 종료\n", getpid(), child);
    }
}
```



# 프로세스 그룹

- 프로세스 그룹은 여러 프로세스들의 집합이다.
- 보통 부모 프로세스(그룹 리더)가 생성하는 자손 프로세스들은 부모와 같은 프로세스 그룹에 속한다.
- 프로세스 그룹 리더: Process GID = PID



프로세스 그룹

- 프로세스 그룹은 **signal 전달** 등을 위해 사용됨.



- 프로세스 IDs

- 프로세스 ID(PID)
- 프로세스 그룹 ID(GID)
- 각 프로세스는 하나의 프로세스 그룹에 속함.
- 각 프로세스는 자신이 속한 프로세스 그룹 ID를 가지며 fork 시 물려받는다.

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

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

```
pid_t getpgrp(void);
```

호출한 프로세스의 프로세스 그룹 ID를 반환한다.



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

int main()
{
    int pid, gid;

    printf("PARENT: PID = %d GID = %d \n", getpid(), getpgrp());
    pid = fork();
    if (pid == 0) { // 자식 프로세스
        printf("CHILD: PID = %d GID = %d \n", getpid(), getpgrp());
    }
}
```

pgrp1.c



# 프로세스 그룹

- 프로세스 그룹 만들기
  - A process can create a new process group and become leader
  - `int setpgid(pid_t pid, pid_t pgid);`
- 프로세스 그룹 소멸
  - the last process terminates OR
  - joins another process group
  - (leader may terminate first)



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

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

```
int setpgid(pid_t pid, pid_t pgid);
```

프로세스 pid의 프로세스 그룹 ID를 pgid로 설정한다.

성공하면 0을 실패하면 -1를 반환한다.

- 새로운 프로세스 그룹을 생성하거나 다른 그룹에 멤버로 참여
  - `pid == pgid` → 새로운 그룹 리더가 됨.
  - `pid != pgid` → 다른 그룹의 멤버가 됨.
  - `pid == 0` → 호출자의 PID 사용
  - `pgid == 0` → 새로운 그룹 리더가 됨
- 호출자가 새로운 프로세스 그룹을 생성하고 그룹의 리더
  - `setpgid(getpid(), getpid());`
  - `setpgid(0,0);`





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

```
main()
{
```

```
    int pid, gid;
```

```
    printf("PARENT: PID = %d  GID = %d \n", getpid(), getpgrp());
```

```
    pid = fork();
```

```
    if (pid == 0) {
```

```
        setpgid(0, 0);
```

```
        printf("CHILD: PID = %d  GID = %d \n", getpid(), getpgrp());
```

```
    }
```

```
}
```

pgrp2.c



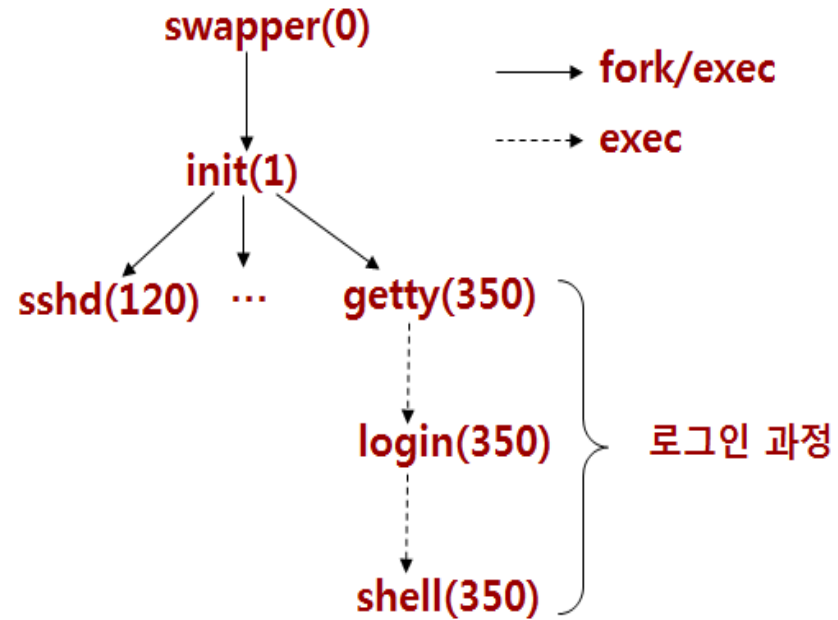
# 프로세스 그룹 사용

- 프로세스 그룹 내의 모든 프로세스에 시그널을 보낼 때 사용
  - \$ kill -9 pid
  - \$ kill -9 0
  - \$ kill -9 -pid
- `pid_t waitpid(pid_t pid, int *status, int options);`
  - `pid == -1` : 임의의 자식 프로세스가 종료하기를 기다린다.
  - `pid > 0` : 자식 프로세스 `pid`가 종료하기를 기다린다.
  - `pid == 0` : 호출자와 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다린다.
  - `pid < -1` : `pid`의 절대값과 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다린다.



# 시스템 부팅

- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.



- swapper(스케줄러 프로세스)
  - 커널 내부에서 만들어진 프로세스로 프로세스 스케줄링을 한다
- init(초기화 프로세스)
  - /etc/inittab 파일에 기술된 대로 시스템을 초기화
- getty 프로세스
  - 로그인 프롬프트를 내고 키보드 입력을 감지한다.
- login 프로세스
  - 사용자의 로그인 아이디 및 패스워드를 검사
- shell 프로세스
  - 시작 파일을 실행한 후에 쉘 프롬프트를 내고 사용자로부터 명령어를 기다린다.



# 핵심 개념

- 프로세스는 실행중인 프로그램이다.
- fork() 시스템 호출은 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성한다.
- exec() 시스템 호출은 프로세스 내의 프로그램을 새로운 프로그램으로 대체하여 새로운 프로그램을 실행시킨다.
- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.
- 시그널은 예기치 않은 사건이 발생할 때 이를 알리는 소프트웨어 인터럽트이다.



## Reference

- ✓ 리눅스 프로그래밍, 창병모, 생능출판
- ✓ <https://www.44bits.io/ko/post/wsl2-install-and-basic-usage>

