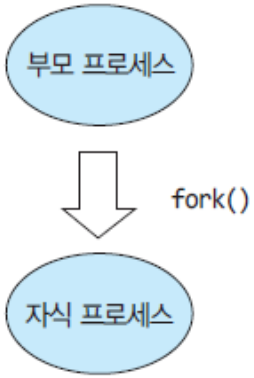
9장 프로세스 제어

9.1 프로세스 생성

프로세스 생성

- 프로세스 생성
  - 부모 프로세스가 자식 프로세스 생성

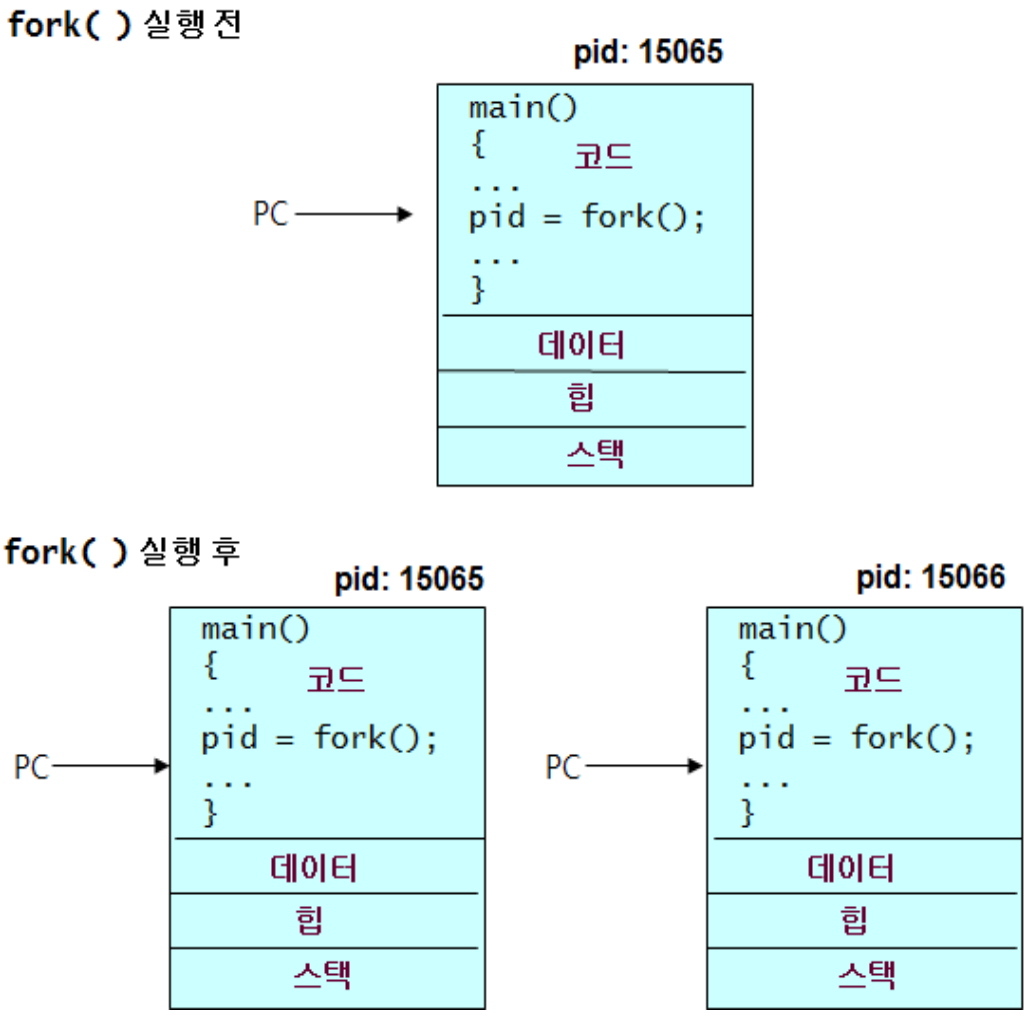


- fork() 시스템 호출
  - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
  - 자기복제(自己複製)

```
#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.

프로세스 생성


fork( ) 실행 전
pid: 15065
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택
fork( ) 실행 후
pid: 15065
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택
pid: 15066
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택

프로세스 생성

- 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.

프로세스 생성: fork1.c

```
#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
[15065] 프로세스 시작
[15065] 프로세스 : 반환값 15066
[15066] 프로세스 : 반환값 0
```

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 }
```

fork2.c

```
#include <stdlib.h>
#include <stdio.h>
/* 부모 프로세스가 자식 프로세스를 생성하고 서로 다른 메시지를 프린트 */
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());
    }
}
```

```
$ fork2
[Parent] Hello, world! pid=15799
[Child] Hello, world! pid=15800
```

두 개의 자식 프로세스 생성: fork3.c

```
#include <stdlib.h>
#include <stdio.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);
    }
    printf("[PARENT] : Hello, world ! pid=%d₩n",getpid());
}
```

두 개의 자식 프로세스 생성: fork3.c

- $ fork3

  [Parent] Hello, world! pid=15740

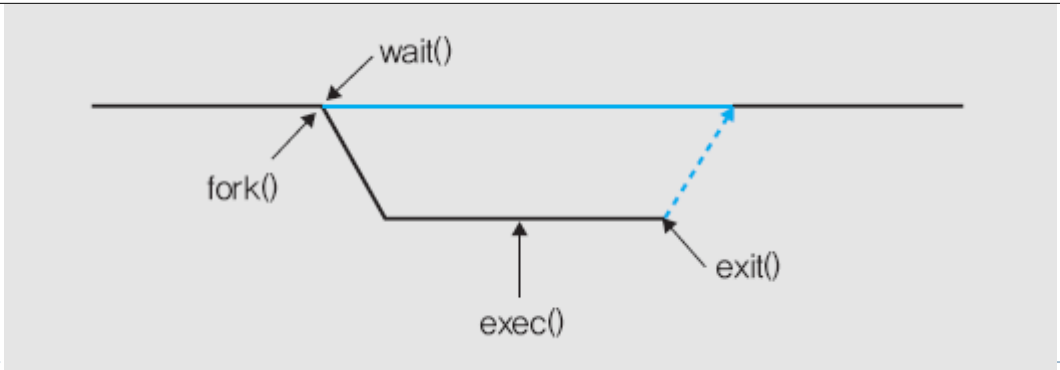
  [Child 1] Hello, world! pid=15741

  [Child 2] Hello, world! pid=15742

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);
```



프로세스 기다리기: forkwait.c

```
#include <sys/types.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);
}
```

프로세스 기다리기: forkwait.c

- $ forkwait
  [15943] 부모 프로세스 시작
  [15944] 자식 프로세스 시작
  [15943] 자식 프로세스 15944 종료
  종료코드 1

특정 자식 프로세스 기다리기: waitpid.c

```
1  #include <sys/types.h>
...
7   /* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
8   int main()
9   {
10      int pid1, pid2, child, status;
11
12      printf("[%d] 부모 프로세스 시작 ₩n", getpid( ));
13      pid1 = fork();
14      if (pid1 == 0) {
15          printf("[%d] 자식 프로세스[1] 시작 ₩n", getpid( ));
16          sleep(1);
17          printf("[%d] 자식 프로세스[1] 종료 ₩n", getpid( ));
18          exit(1);
19      }
```

waitpid.c

```
20
21          pid2 = fork();
22          if (pid2 == 0) {
23              printf("[%d] 자식 프로세스 #2 시작 ₩n", getpid( ));
24              sleep(2);
25              printf("[%d] 자식 프로세스 #2 종료 ₩n", getpid( ));
26              exit(2);
27          }
28          // 자식 프로세스 #1의 종료를 기다린다.
29          child = waitpid(pid1, &status, 0);
30          printf("[%d] 자식 프로세스 #1 %d 종료 ₩n", getpid( ), child);
31          printf("₩t종료 코드 %d₩n", status>>8);
32        }
```

특정 자식 프로세스 기다리기

- $ waitpid

  [16840] 부모 프로세스 시작

  [16841] 자식 프로세스[1] 시작

  [16842] 자식 프로세스[2] 시작

  [16841] 자식 프로세스[1] 종료

  [16840] 자식 프로세스[1] 16841 종료

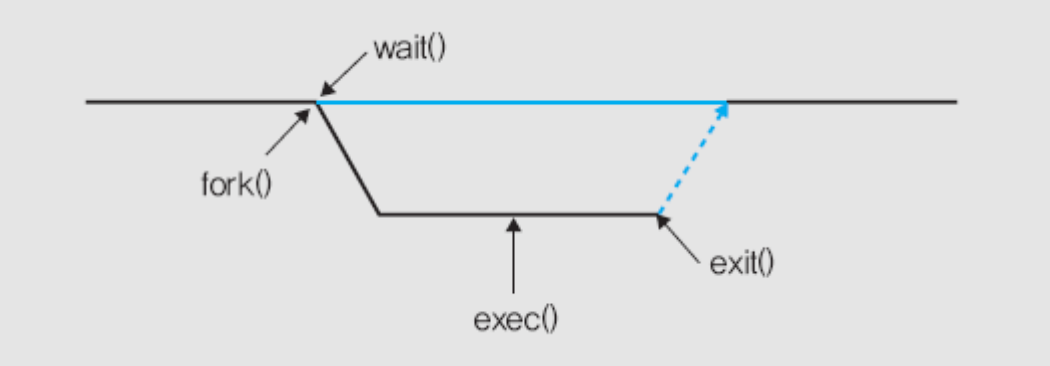
  종료코드 1

  [16842] 자식 프로세스[2] 종료

9.2 프로그램 실행

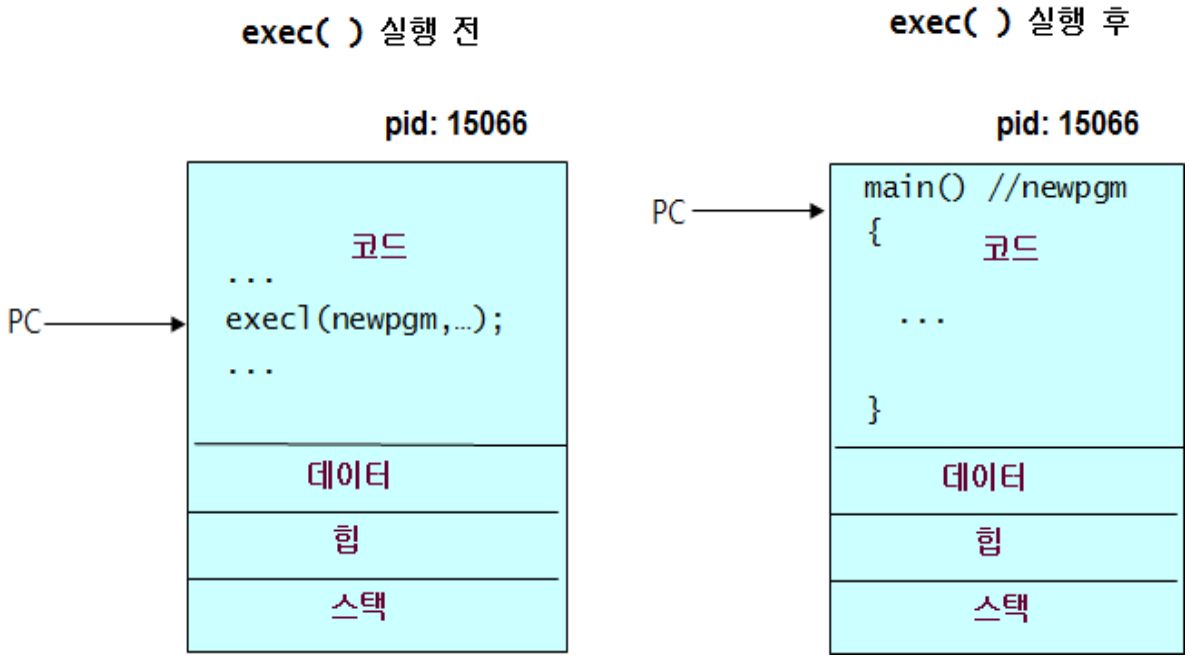
프로그램 실행

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



프로그램 실행: exec()

- 프로세스가 exec() 호출을 하면,
  - 그 프로세스 내의 프로그램은 완전히 새로운 프로그램으로 대치
  - 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 대치한 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
```

명령어 실행: execute1.c

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

/* 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");
}
```

```
$ exec1
부모 프로세스 시작
부모 프로세스 끝
hello
```

여러 개의 명령어 실행: execute2.c

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

/* 세 개의 자식 프로세스를 생성하여 각
   각 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");
}
```

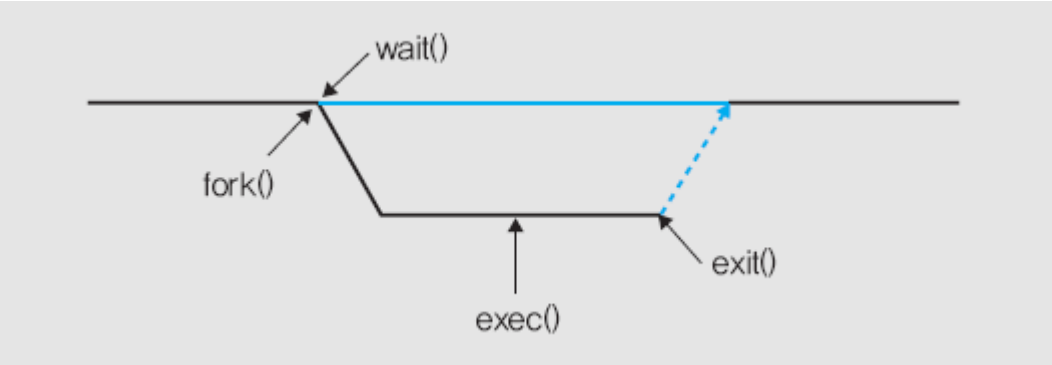
여러 개의 명령어 실행 예

- $ execute2
  부모 프로세스 시작
  부모 프로세스 끝
  hello
  2022. 03. 01. (화) 11:33:14 PST
  총 50
  -rwxr-xr-x 1 chang chang 24296 2월 28일 20:43 execute2
  -rw-r--r-- 1 chang chang 556 2월 28일 20:42 execute2.c
  …

명령줄 인수로 받은 명령어 실행: execute3.c

```
#include <stdio.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("₩t종료 코드 %d ₩n", status>>8);
    }
}
```

execute3.c




```
$ execute3 wc you.txt
25 68 556 you.txt
[26470] 자식 프로세스 26471 종료
종료코드 0
```

명령어 실행 함수 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

syscall.c

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

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

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

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

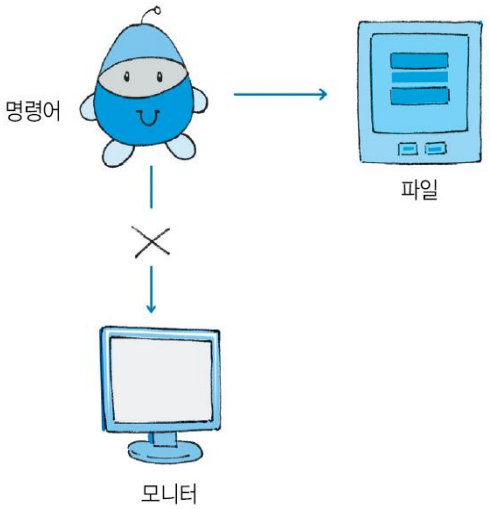
system() 함수 구현

```
int mysystem (const char *cmdstring) {
    int pid, status;
    if (cmdstring == NULL) /* 명령어가 NULL인 경우 */
        return 1;
    pid = fork();
    if (pid == -1) /* 프로세스 생성 실패 */
        return –1;
    if (pid == 0) {
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127); /* 명령어 실행 오류 */
    }
    do {
        if (waitpid(pid, &status, 0) == -1) {
            if (errno != EINTR) /* waitpid()로부터 EINTR 오류 외 */
            return -1;
        } else return status;
     } while(1);
}
```

9.3 입출력 재지정

입출력 재지정

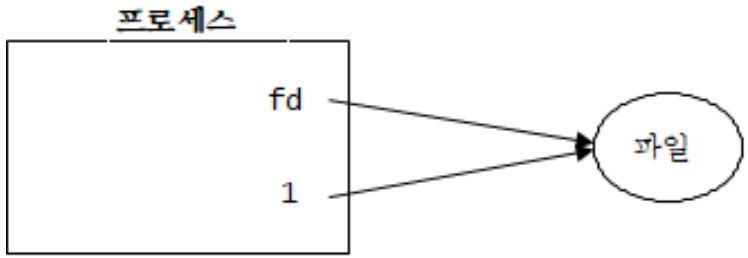
- 명령어의 표준 출력이 파일에 저장
  $ 명령어 > 파일



입출력 재지정

- 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.

출력 재지정: redirect1.c

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4
5 /* 표준출력을 파일에 재지정하는 프로그램 */
6 int main(int argc, char* argv[])
7 {
8       int fd, status;
9       fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
10      dup2(fd, 1); /* 파일을 표준출력에 복제 */
11     close(fd);
12     printf("Hello stdout !₩n");
13     fprintf(stderr,"Hello stderr !₩n");
14 }
```

```
$ redirect1 hi1.txt
Hello stderr !
$cat hi1.txt
Hello stdout !
```

명령어 출력 재지정: redirect2.c

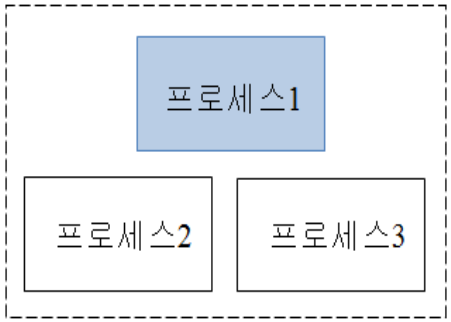
```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdio.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6
7 /* 자식 프로세스의 표준 출력을 파
   일에 재지정한다. */
8 int main(int argc, char* argv[])
9 {
10    int child, pid, fd, status;
11
12    pid = fork( );
13  if (pid == 0) {
14      fd = open(argv[1],O_CREAT|
              O_TRUNC| O_WRONLY, 0600);
15      dup2(fd, 1);   // 파일을 표준출력에 복제
16      close(fd);
17      execvp(argv[2], &argv[2]);
18      fprintf(stderr, "%s:실행 불가\n",argv[1]);
19    } else {
20       child = wait(&status);
21       printf("[%d] 자식 프로세스 %d 종료 \n",
                 getpid(), child);
22    }
23 }
```

```
$ redirect2 out wc you.txt
[26882] 자식 프로세스 26883 종료
$ cat out
25 68 556 you.txt
```

9.4 프로세스 그룹

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 그룹

- 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.

프로세스 그룹: pgrp1.c

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

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
[PARENT] PID = 17768 GID = 17768
[CHILD] PID = 17769 GID = 17768
```

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)

프로세스 그룹

```
#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);

프로세스 그룹: pgrp2.c

```
#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
[PARENT] PID = 17768 GID = 17768
[CHILD] PID = 17769 GID = 17769
```

프로세스 그룹 사용

- 프로세스 그룹 내의 모든 프로세스에 시그널을 보낼 때 사용
  - $ kill –9 pid
  - $ kill –9 0
  - $ kill –9 –pid

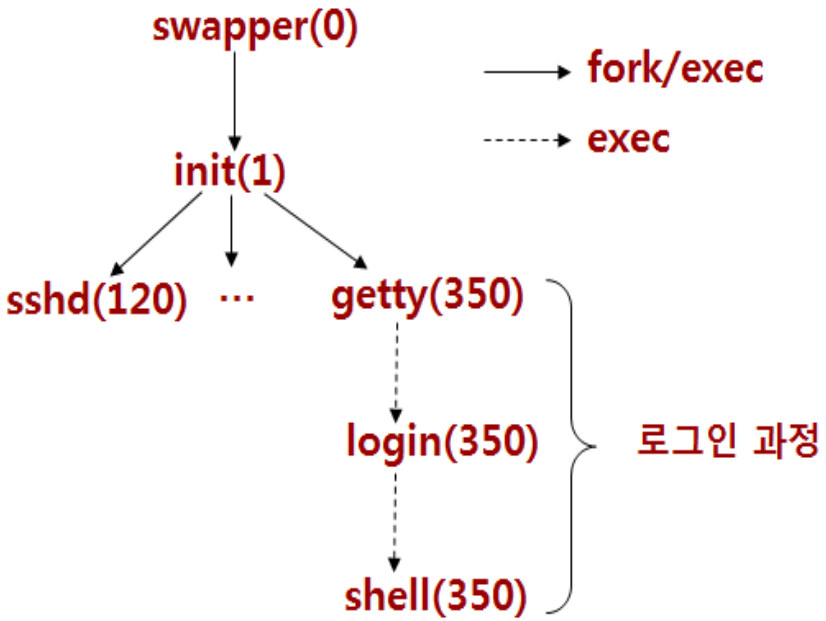
- pid_t waitpid(pid_t pid, int *status, int options);

  - pid == -1 : 임의의 자식 프로세스가 종료하기를 기다린다.
  - pid > 0 : 자식 프로세스 pid가 종료하기를 기다린다.
  - pid == 0 : 호출자와 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다린다.
  - pid < -1 : pid의 절대값과 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다린다.

9.5 시스템 부팅

시스템 부팅

- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.



시스템 부팅

- 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 대치하여 새로운 프로그램을 실행시킨다.
- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.
- 시그널은 예기치 않은 사건이 발생할 때 이를 알리는 소프트웨어 인터럽트이다.