Linux Programming
9장. 프로세스 제어

sisong@ut.ac.kr
한국교통대학교 컴퓨터공학전공
송석일

9.1 프로세스 생성

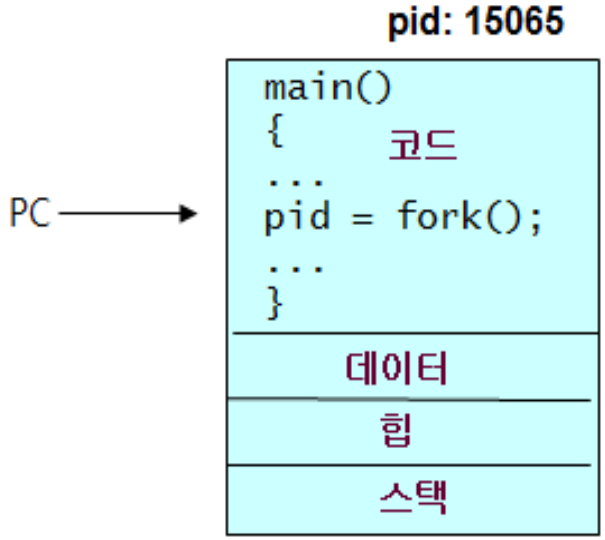
프로세스 생성 : fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```
•새로운 자식 프로세스를 생성
•자식 프로세스에게는 0을 반환하고, 부모 프로세스에게는 자식 프로세스 ID를 반환

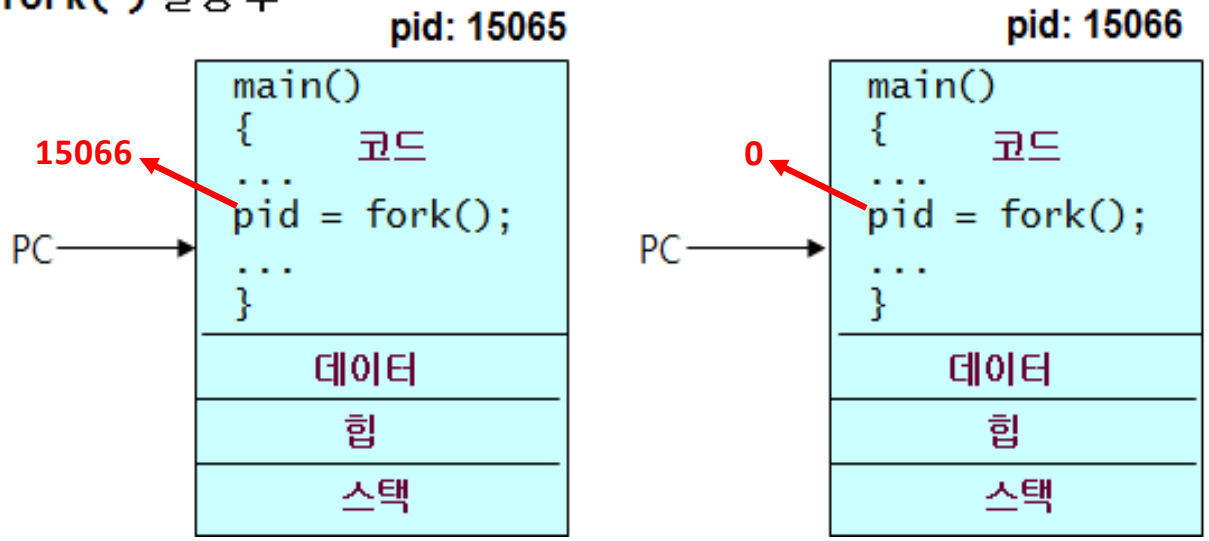
- 모든 프로세스는 부모 프로세스에 의해 생성됨
- 생성된 자식프로세스는 부모 프로세스의 이미지와 동일 (다음 슬라이드 참고)

프로세스 생성

fork( ) 실행 전



fork( ) 실행 후



부모 프로세스와 자식 프로세스는 병행 수행

프로세스 생성: fork1.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
    /* 자식 프로세스를 생성한다. */
4.  int main()
5.  {
6.    int pid;
7.    printf("[%d] 프로세스 시작 \n", getpid());
8.    pid = fork();
9.    printf("[%d] 프로세스 : 반환값 %d\n", getpid(), pid);
10. }
```

```
$ fork1
[15065] 프로세스 시작
[15065] 프로세스 : 반환값 15066
[15066] 프로세스 : 반환값 0
```

프로세스 스스로 부모/자식 판별

- fork() 시스템 호출의 반환 값을 확인

```
pid = fork();

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

fork2.c

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

/* 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());
  }
}
```

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

두 개의 자식 프로세스 생성: fork3.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  int main()
5.  {
6.    int pid1, pid2;
7.    pid1 = fork();
8.    if (pid1 == 0) {
9.      printf("[Child 1] : Hello, world ! pid=%d\n",getpid());
10.     exit(0);
11.   }
12.   pid2 = fork();
13.   if (pid2 == 0) {
14.     printf("[Child 2] : Hello, world ! pid=%d\n",getpid());
15.     exit(0);
16.   }
17.   printf("[PARENT] : Hello, world ! pid=%d\n",getpid());
18. }
```

```
$ fork3
[Parent] Hello, world! pid=15740
[Child 1] Hello, world! pid=15741
[Child 2] Hello, world! pid=15742
```

프로세스 기다리기: wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- 자식 프로세스 중 하나가 종료될때 까지 기다림
- 종료된 자식 프로세스의 종료 코드가 status에 저장됨
- 종료된 자식 프로세스의 번호를 반환

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

- 종료된 자식 프로세스의 종료 코드가 *statloc*에 저장됨
- options : 부모프로세스 대기 방법 (0: 자식 프로세스 종료 기다림)
  - WNOHANG : 자식 프로세스 종료 기다리지 않음. 단지 확인 만 수행

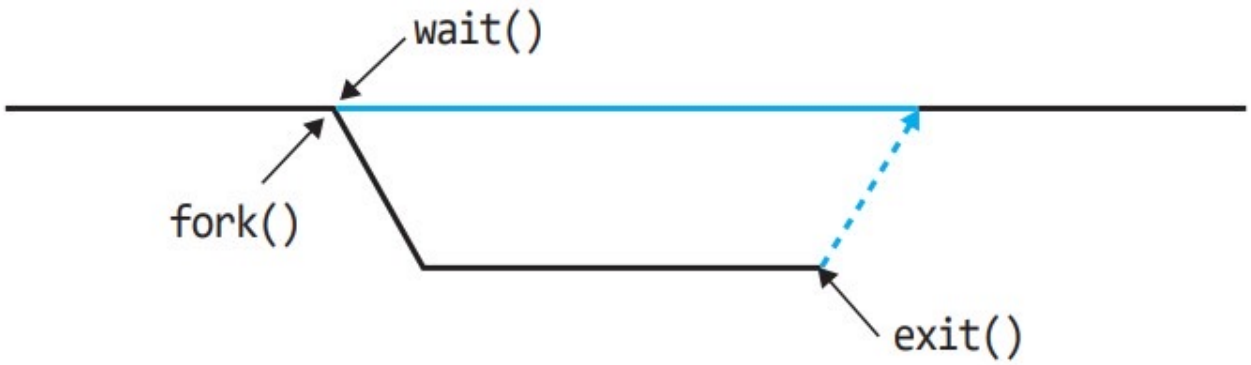
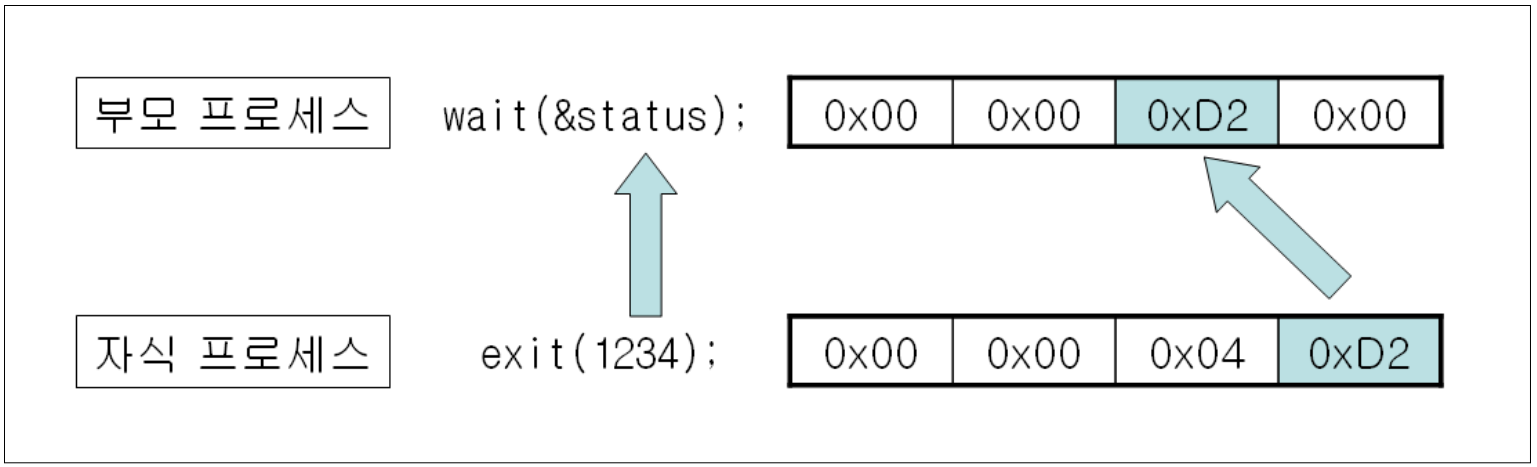


그림 9.3 프로세스 생성 및 기다리는 과정

프로세스 기다리기: wait()

- status
  - 자식 프로세스가 exit를 호출하면서 지정한 값을 부모 프로세스는 status 변수로 전달 받음
  - 자식 프로세스가 exit(n); 을 실행했을 때 부모 프로세스에게 전달되는 실제 값은 n의 하위 1바이트 뿐
  - 자식 프로세스가 전달한 1바이트 값은 부모 프로세스 쪽의 status 변수의 하위 두 번째 바이트에 저장



프로세스 기다리기: forkwait.c

```
1.  #include <sys/types.h>
2.  #include <sys/wait.h>
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <unistd.h>
6.  /* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
7.  int main()
8.  {
9.    int pid, child, status;
10.   printf("[%d] 부모 프로세스 시작 \n", getpid( ));
11.   pid = fork();
12.   if (pid == 0) {
13.     printf("[%d] 자식 프로세스 시작 \n", getpid( ));
14.     exit(1);
15.   }
16.   child = wait(&status); // 자식 프로세스가 끝나기를 기다린다.
17.   printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), child);
18.   printf("\t종료 코드 %d\n", status>>8);
19. }
```

```
$ 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>#include <sys/wait.h>#include <stdio.h>
2.  #include <stdlib.h>#include <unistd.h>
3.  int main()
4.  {
5.    int pid1, pid2, child, status;
6.    printf("[%d] 부모 프로세스 시작 \n", getpid( ));
7.    pid1 = fork();
8.    if (pid1 == 0) {
9.      printf("[%d] 자식 프로세스[1] 시작 \n", getpid( ));
10.     sleep(1);
11.     printf("[%d] 자식 프로세스[1] 종료 \n", getpid( ));
12.     exit(1);
13.   }
14.   pid2 = fork();
15.   if (pid2 == 0) {
16.     printf("[%d] 자식 프로세스[2] 시작 \n", getpid( ));
17.     sleep(2);
18.     printf("[%d] 자식 프로세스[2] 종료 \n", getpid( ));
19.     exit(2);
20.   }
21.   child = waitpid(pid1, &status, 0); // 자식 프로세스 #1의 종료를 기다린다.
22.   printf("[%d] 자식 프로세스[1] %d 종료 \n", getpid(), child);
23.   printf("\t종료 코드 %d\n", status>>8);
24.   exit(0);
25. }
```

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

프로세스가 새로운 프로그램 실행 (새로운 프로세스 생성)

- fork()
  - 생성된 자식 프로세스는 부모 프로세스와 똑같은 코드 실행
- 자식 프로세스가 새로운 프로그램 실행 ?
  - 예) bash 프로세스가 ls 명령어를 실행해서 프로세스 생성
- exec() 시스템 호출
  - 현재 프로세스를 다른 프로그램의 프로세스로 대치
  - 보통 fork() 후에 exec( )을 수행하여 원하는 프로세스 생성

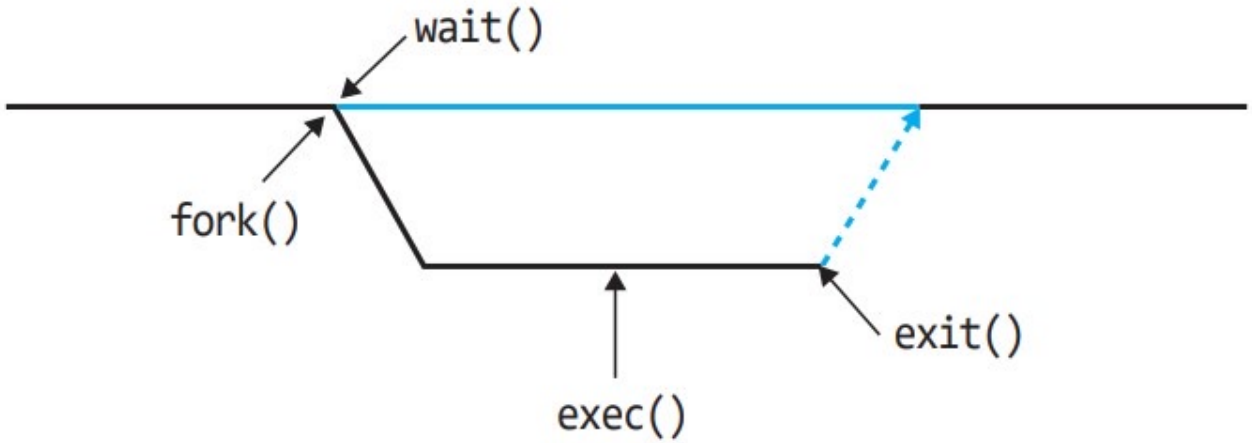
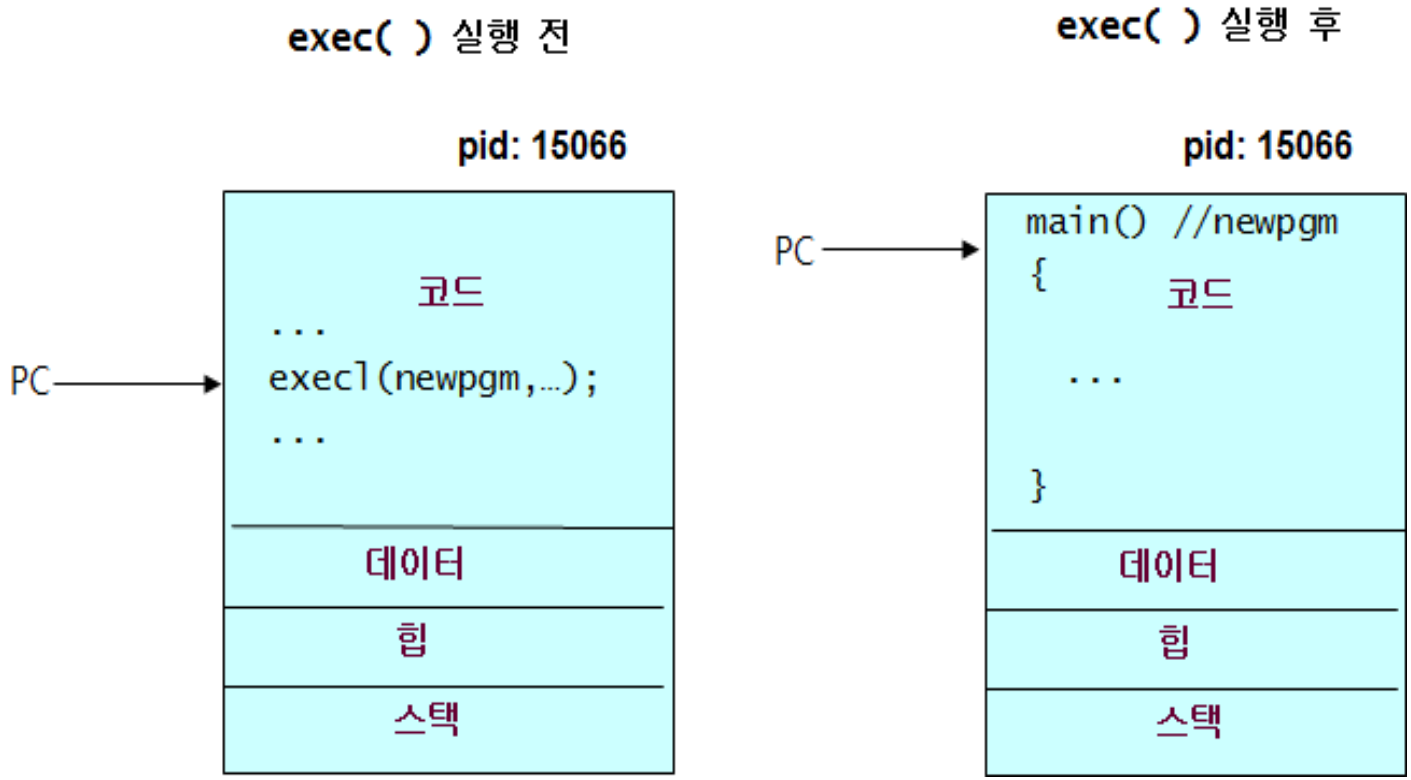


그림 9.5 프로그램 실행 및 기다리는 과정

프로그램 실행: exec()

- exec() 시스템 호출
  - 현재 프로세스는 사라지고 새로운 프로그램의 프로세스로 대치



프로그램 실행: exec()

```
#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 프로세스를 실행
- 성공한 exec( ) 호출은 반환하지 않으며 실패하면 -1을 반환

fork() & exec()

- 보통 fork() 호출 후에 exec() 호출
  - 새로 실행할 프로그램에 대한 정보를 arguments로 전달

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

- exec() 호출이 성공하면
  - 자식 프로세스는 새로운 프로그램을 실행
  - 부모는 계속해서 다음 코드를 실행

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

명령어 실행: execute1.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  int main( )
5.  {
6.      printf("부모 프로세스 시작\n");
7.      if (fork( ) == 0) {
8.          execl("/bin/echo", "echo", "hello", NULL);
9.          fprintf(stderr,"첫 번째 실패");
10.         exit(1);
11.     }
12.     printf("부모 프로세스 끝\n");
13. }
```

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

여러 개의 명령어 실행: execute2.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  int main( )
5.  {
6.      printf("부모 프로세스 시작\n");
7.      if (fork( ) == 0) {
8.          execl("/bin/echo", "echo", "hello", NULL);
9.          fprintf(stderr,"첫 번째 실패");
10.         exit(1);
11.     }
12.     if (fork( ) == 0) {
13.         execl("/bin/date", "date", NULL);
14.         fprintf(stderr,"두 번째 실패");
15.         exit(2);
16.     }
17.     if (fork( ) == 0) {
18.         execl("/bin/ls","ls", "-l",
19.                 NULL);
20.         fprintf(stderr,"세 번째 실패");
21.         exit(3);
22.     }
23.     printf("부모 프로세스 끝\n");
24. }
```

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

명령줄 인수로 받은 명령어 실행: execute3.c

```
1.  #include <sys/types.h>
2.  #include <sys/wait.h>
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <unistd.h>
6.  int main(int argc, char *argv[])
7.  {
8.      int child, pid, status;
9.      pid = fork( );
10.     if (pid == 0) {   // 자식 프로세스
11.         execvp(argv[1], &argv[1]);
12.         fprintf(stderr, "%s:실행 불가\n",argv[1]);
13.     } else {        // 부모 프로세스
14.         child = wait(&status);
15.         printf("[%d] 자식 프로세스 %d 종료 \n",
16.                 getpid(), pid);
17.         printf("\t종료 코드 %d \n", status>>8);
18.     }
19. }
```

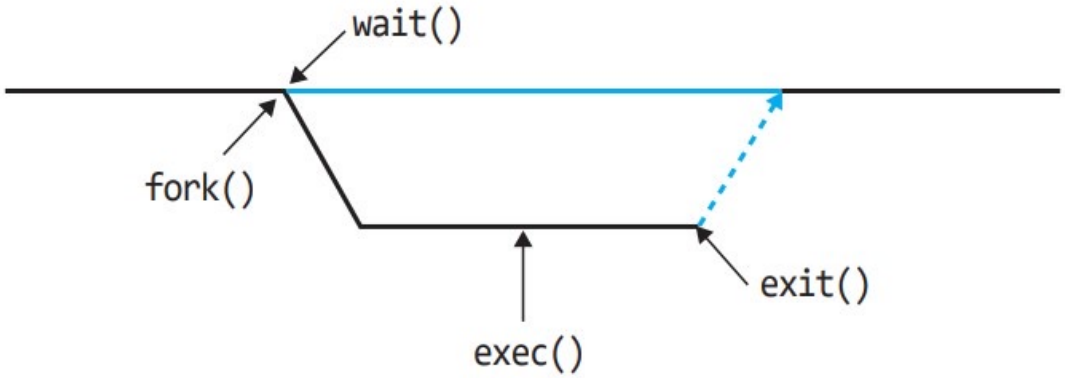


그림 9.5 프로그램 실행 및 기다리는 과정

```
$ 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);
```

- /bin/sh –c cmdstring를 호출하여 cmdstring에 지정된 명령어를 실행
- 명령어가 끝난후 명령어의 종료코드를 반환

- 자식 프로세스를 생성하고 /bin/sh 를 이용해서 지정된 명령어 실행
  - ex) 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

```
1.  #include <sys/wait.h>
2.  #include <stdio.h>
3.  int main()
4.  {
5.      int status;
6.      if ((status = mysystem("date")) < 0)
7.          perror("system() 오류");
8.      printf("종료코드 %d\n", WEXITSTATUS(status));

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

12.     if ((status = mysystem("who; exit 44")) < 0)
13.         perror("system() 오류");
14.     printf("종료코드 %d\n", WEXITSTATUS(status));
15. }
```

```
16. int mysystem (const char *cmdstring) {
17.     int pid, status;
18.     if (cmdstring == NULL)
19.         return 1;
20.     pid = fork();
21.     if (pid == -1)  return –1;
22.     if (pid == 0) {
23.         execl("/bin/sh", "sh", "-c",
24.                 cmdstring, (char *) 0);
25.         _exit(127); /* 명령어 실행 오류 */
26.     }
27.     do {
28.         if (waitpid(pid, &status, 0) == -1) {
29.             if (errno != EINTR)
30.                 return -1;
31.         } else return status;
32.      } while(1);
33. }
```

9.3 입출력 재지정

입출력 재지정

- 입출력 재지정 ?
  - $ 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);
```

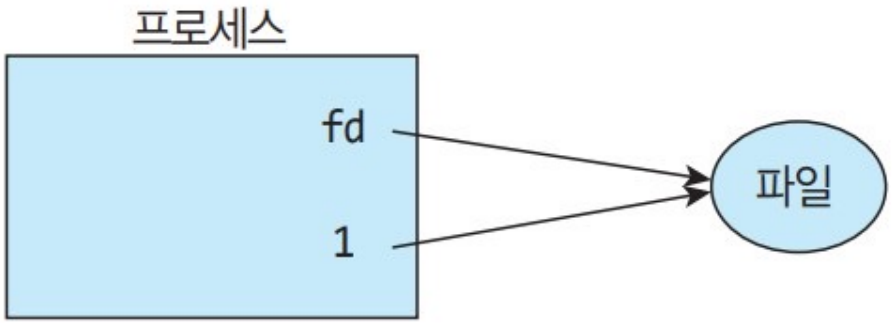


그림 9.7 dup2(fd, 1)를 이용한 출력 재지정 구현

입출력 재지정

```
#include <unistd.h>
int dup(int oldfd);
```

- oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 반환

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

- oldfd을 newfd에 복제하고 복제된 새로운 파일 디스크립터를 반환

출력 재지정: redirect1.c

```
1.  #include <stdio.h>
2.  #include <fcntl.h>
3.  #include <unistd.h>

4.  /* 표준 출력을 파일에 재지정하는 프로그램 */
5.  int main(int argc, char* argv[])
6.  {
7.      int fd, status;
8.      fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
9.      dup2(fd, 1);  /* 파일을 표준출력에 복제 */
10.     close(fd);
11.     printf("Hello stdout !\n");
12.     fprintf(stderr,"Hello stderr !\n");
13. }
```

```
$ 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.  /* 자식 프로세스의 표준 출력을 파일에 재지정한다. */
7.  int main(int argc, char* argv[])
8.  {
9.      int child, pid, fd, status;
10.     pid = fork( );
11.     if (pid == 0) {
12.         fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
13.         dup2(fd, 1);  /* 파일을 표준출력에 복제 */
14.         close(fd);
15.         execvp(argv[2], &argv[2]); /* wc you.txt */
16.         fprintf(stderr, "%s:실행 불가\n",argv[1]);
17.     } else {
18.         child = wait(&status);
19.         printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), child);
20.     }
21. }
```

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

9.4 프로세스 그룹

프로세스 그룹

- 프로세스 그룹 ?
  - 여러 프로세스들의 집합
  - 부모 프로세스(그룹 리더)가 생성하는 자식 프로세스들은 부모와 같은 프로세스 그룹에 속함
- 프로세스 그룹 리더: Process GID = PID
  - 프로세스 그룹은 signal 전달 등을 위해 사용됨.

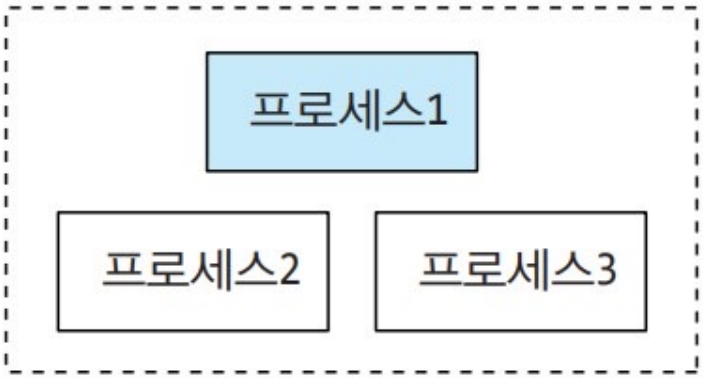


그림 9.8 프로세스 그룹

프로세스 그룹

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
```

- 호출한 프로세스의 프로세스 그룹 ID를 반환

- 프로세스 ID(PID)
- 프로세스 그룹 ID(GID)
- 각 프로세스는 하나의 프로세스 그룹에 속함
- fork된 프로세스는 부모 프로세스가 속한 프로세스 그룹 ID를 가짐

프로세스 그룹: pgrp1.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  int main()
4.  {
5.      int pid, gid;
6.      printf("PARENT: PID = %d GID = %d \n", getpid(), getpgrp());
7.      pid = fork();
8.      if (pid == 0) {  // 자식 프로세스
9.          printf("CHILD: PID = %d GID = %d \n", getpid(), getpgrp());
10.     }
11. }
```

```
$ pgrp1
[PARENT] PID = 17768 GID = 17768
[CHILD] PID = 17769 GID = 17768
```

프로세스 그룹

- 프로세스 그룹 생성
  - 프로세스가 새로운 프로세스 그룹을 만들고 그룹 리더가 될 수 있음
- 프로세스 그룹 소멸
  - 마지막 프로세스가 종료되거나 다른 프로세스 그룹과 결합될 때

프로세스 그룹

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

- 프로세스 pid의 프로세스 그룹 ID를 pgid로 설정
- 성공하면 0을 실패하면 -1를 반환

- 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

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

3.  int main()
4.  {
5.    int pid, gid;
6.    printf("PARENT: PID = %d  GID = %d \n", getpid(), getpgrp());
7.    pid = fork();
8.    if (pid == 0) {
9.       setpgid(0, 0);
10.      printf("CHILD: PID = %d  GID = %d \n", getpid(), getpgrp());
11.   }
12. }
```

```
$ pgrp2
[PARENT] PID = 17768 GID = 17768
[CHILD] PID = 17769 GID = 17769
```

프로세스 그룹 사용

- 그룹 내 모든 프로세스에 시그널을 보낼 때 사용
  - $ kill –9 **pid**
  - $ kill –9 **0** → 현재 프로세스가 속한 그룹의 모든 프로세스에게 신호 보냄
  - $ kill –9 **–pid** → pid가 속한 그룹의 모든 프로세스에게 신호 보냄

- waitpid 에서 사용

  Pid_t waitpid(pid_t pid, int *status, int options);
  - pid == -1 : 임의의 자식 프로세스가 종료하기를 기다림
  - pid > 0 : 자식 프로세스 pid가 종료하기를 기다림
  - pid == 0 : 호출자와 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다림
  - pid < -1 : pid의 절대값과 같은 프로세스 그룹 내의 어떤 자식 프로세스가 종료하기를 기다림

프로세스 그룹 사용

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);
pid_t setsid(void);
```

- *Pid :* 프로세스의 식별 번호이다.
- 호출이 성공하면 프로세스의 세션 식별 번호를 반환하고, 실패하면 –1을 반환

- 프로세스의 세션 식별 번호를 구하거나, 새로운 세션 생성
- 세션 (session)
  - 일반적으로 시스템과 연결된 하나의 제어 단말기를 포함한 단위
  - 식별 번호(id)가 부여됨
  - 세션 ⊃ 그룹 ⊃ 프로세스

getsid, setsid

- getsid
  - pid로 지정한 프로세스가 포함된 세션의 식별 번호 획득
  - pid = 0 일 경우 현재 프로세스 획득
  - 세션의 리더 (프로세스)
    - 자신의 PID = 자신의 PGID = 자신의 PSID 일 경우
- setsid
  - 호출하는 프로세스가 그룹의 리더가 아닌 경우 새로운 세션을 생성
  - 일반적인 경우에 셸 프로세스가 세션과 그룹의 리더
  - 호출이 성공하면 프로세스는 새로운 세션과 그룹의 리더가 됨
    - 제어 터미널을 가지지 않음
  - 세션과 그룹 내에서 유일한 프로세스가 됨

getsid, setsid

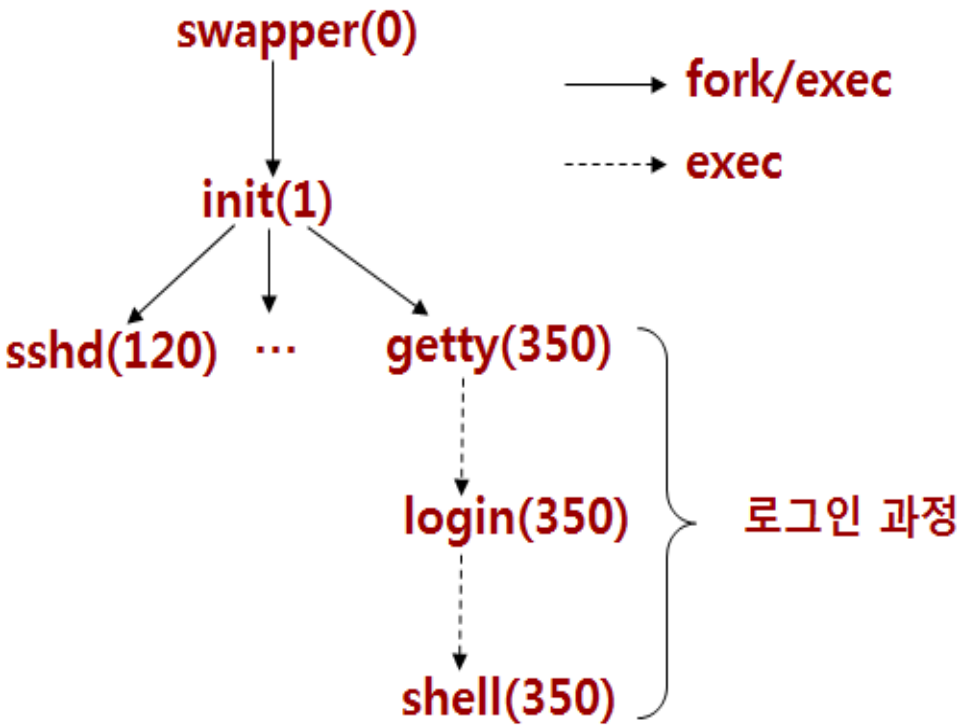
- 로그인 셸
  - 제어 터미널의 연결 상태를 가짐 (제어 터미널을 포함한 세션)
  - 세션과 그룹 내에서 리더
  - 셸 프롬프트 상에서 실행한 많은 프로세스가 존재

- 로그인 셸의 종료
  - 세션의 리더이기 때문에 연결이 끊어짐
  - 같은 세션에 있는 다른 프로세스도 종료
  - 후면(background) 실행 중인 프로세스도 종료

9.5 시스템 부팅

시스템 부팅

- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어짐



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