리눅스 시스템 프로그래밍

day5 프로세스 이해

고 강 태
010-8269-3535
james@thinkbee.kr
https://www.linkedin.com/in/thinkbeekr/
http://www.facebook.com/gangtai.goh

Process 이해

프로세스 다루기
프로세스 환경

프로세스 다루기

프로세스 생성 호출

프로세스를 생성하고 종료하는 시스템 호출/표준 라이브러리 함수

함수	의미
fork()	자신과 완전히 동일한 프로세스를 생성한다.
exec() 계열	지정한 실행 파일로부터 프로세스를 생성한다.
exit()	종료에 따른 상태 값을 부모 프로세스에게 전달하며 프로세스를 종료한다.
atexit	exit로 프로세스를 종료할 때 수행할 함수를 등록한다.
_exit	atexit로 등록한 함수를 호출하지 않고 프로세스를 종료한다.

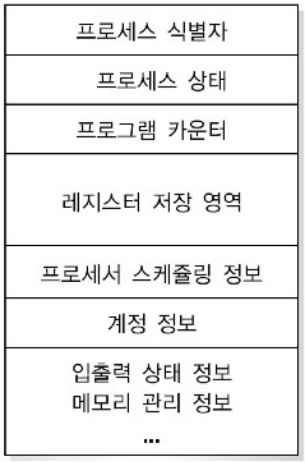
fork()

fork를 호출하는 쪽을 부모 프로세스라고 하고 새로 생성된 쪽을 자식 프로세스라고 한다.

- 부모 프로세스와 자식 프로세스는 서로 다른 프로세스이다.
- 프로세스 식별 번호 (PID)가 서로 다르다.
- 프로세스 식별 번호 (PPID)는 자신을 생성한 부모 프로세스가 된다.

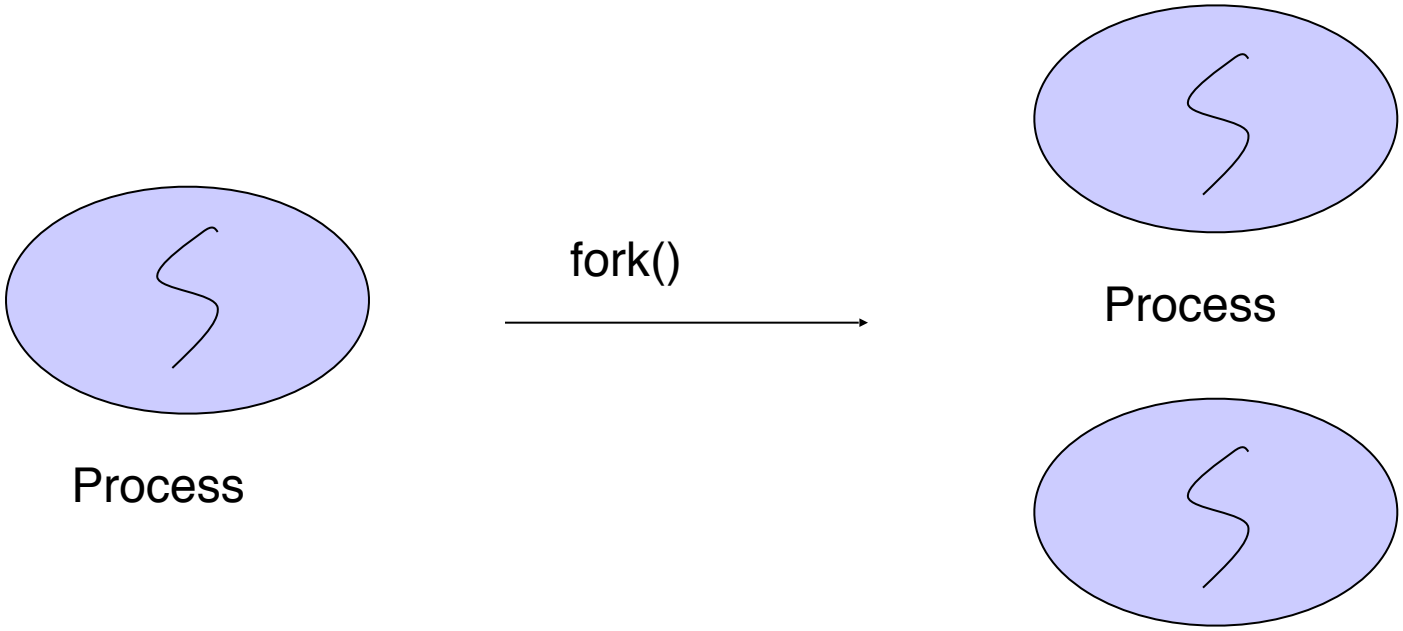
자식 프로세스는 부모 프로세스 시점의 상태를 그대로 물려받는다.

- PCB



fork 호출 이후에 부모와 자식 프로세스는 자신들의 나머지 프로그램 코드를 수행한다.

fork()



- 자기 자신을 복제
- 메모리 영역은 독립
- PID 는 다름
- 완전히 복제된 두 개의 Process 가 돌게 된다.

fork() - 프로세스 생성

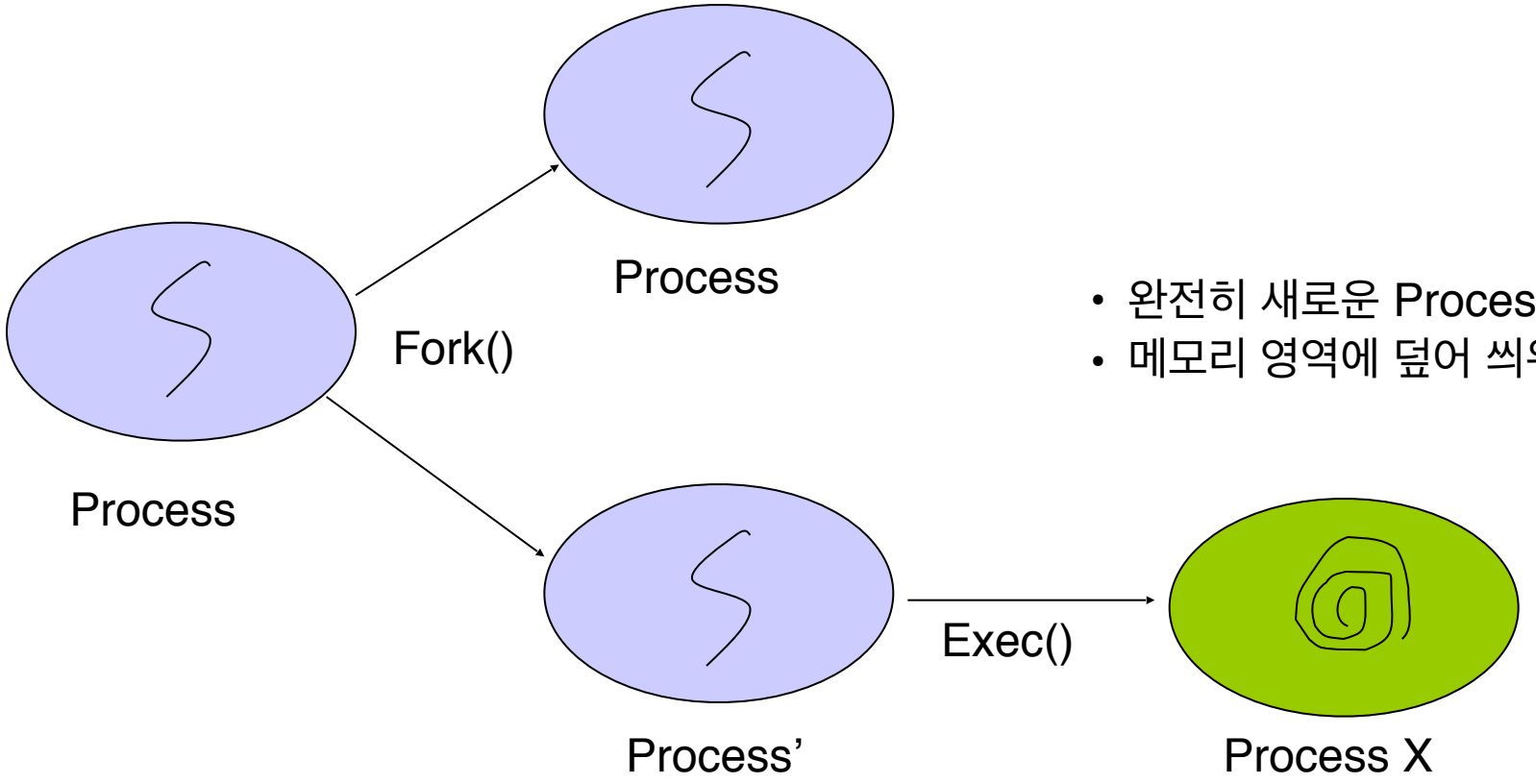
프로세스를 복제하여 완전히 동일한 프로세스를 생성한다.

- **fork()**를 사용하면 실행 중인 프로세스를 복제하여 새로운 프로세스를 생성할 수 있다.
- fork() 호출이 성공하여 자식 프로세스가 만들어지면 부모 프로세스에서는 자식 프로세스의 프로세스 ID가 반환되고 자식 프로세스에서는 0을 반환한다. fork 호출이 실패하여 자식 프로세스가 만들어지지 않으면 부모 프로세스에서는 -1이 반환된다.

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

pid_t fork(void);
  - return: 성공시 부모는 PID, 자식은 0, 실패시 -1
```

exec() 계열



- 완전히 새로운 Process !
- 메모리 영역에 덮어 씌워짐

exec 계열 - 프로세스 생성

경로 이름 또는 파일 이름으로 지정한 실행 파일을 실행하여 프로세스 생성한다.

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
  - path, file:실행 파일 경로(상대/절대 경로 가능)
  - arg: path/file 실행할 때 옵션과 인자이다.마지막 인자는 반드시 NULL 포인터로 지정해야 한다.
  - argv: arg와 같은 의미 배열의 마지막은 NULL 문자열로 끝나야 한다.
  - 반환값: 성공하면 호출하는 프로세스에서는 반환 값을 받을 수 없다. 실패시 -1
```

함수 이름에 p가 없으면 경로(path)로 실행 파일을 지정한다.

p가 있으면 실행 파일의 이름만 지정한다.

```
execl("/bin/ls", "ls", "-l", (char*)0);
```

exec 계열

Caller / Callee Process
- 호출 프로세스 (caller process): exec를 실행하는 프로세스
- 피호출 프로세스 (callee process): exec에 의해 생성되는 프로세스

exec를 성공적으로 호출한 결과
- 호출 프로세스는 종료된다.
- 호출 프로세스가 메모리 영역을 피호출 프로세스가 차지한다.
- 호출 프로세스의 PID를 피호출 프로세스가 물려받는다.

```
char *arg[] = {"ls", "-l", (char *)0};
printf("before executing ls -l\n");
execv("/bin/ls", arg);
printf("after executing ls -l\n");
```

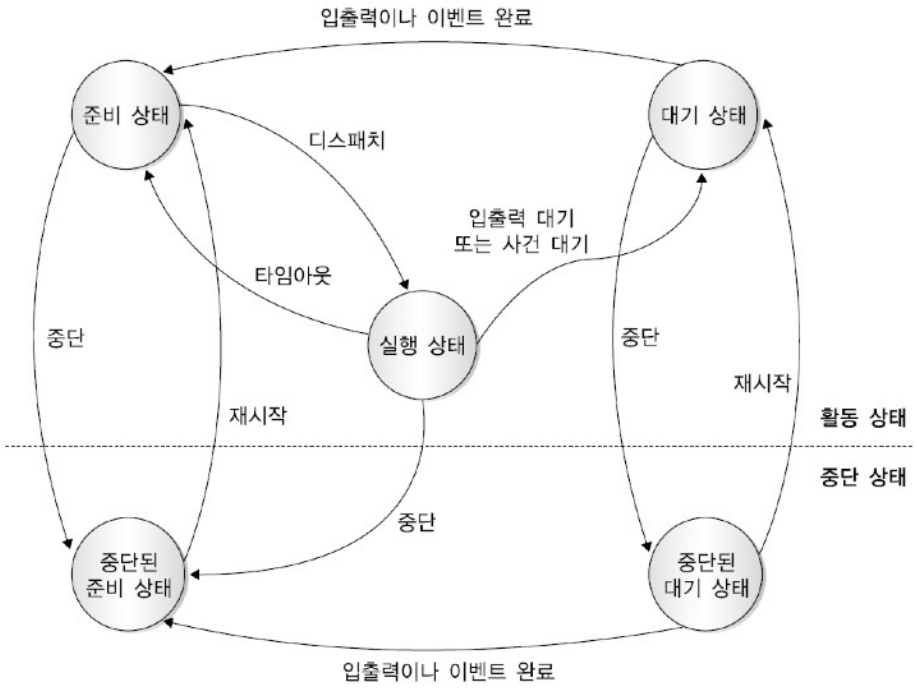
forv() vs. exec()

	fork	exec 계열
프로세스의 원본	부모 프로세스를 복제하여 새로운 프로세스를 생성한다.	지정한 프로그램(파일)을 실행하여 프로세스를 생성한다.
셸 명령줄의 프로그램 인자	새롭게 지정할 수 없고 부모 프로세스의 것을 그대로 사용한다.	필요할 경우 적용할 수 있다.
부모(또는 호출) 프로세스의 상태	자식 프로세스를 생성한 후에도 자신의 나머지 코드를 실행한다.	**호출이 성공할 경우 호출(Caller) 프로세스는 종료된다.**
자식(또는 피호출) 프로세스의 메모리 상의 위치	부모 프로세스와 다른 곳에 위치한다.	호출 프로세스가 있던 자리를 피호출 프로세스가 물려받는다.
프로세스 생성 후 자식(또는 피호출) 프로세스의 프로그램 코드의 시작 지점	fork 호출 이후부터 수행된다.	프로그램의 처음부터 수행된다.
프로세스 식별 번호 (PID)	자식 프로세스는 새로운 식별 번호를 할당받는다.	호출 프로세스의 식별 번호를 피호출 프로세스가 물려받는다.
프로세스의 원본인 파일에 대한 권한	부모 프로세스를 복제하므로 상관없다.	실행 파일에 대한 실행 권한이 필요하다.

프로세스 중단/재시작

시스템에 장애, 사용자에 의한 중지/재시작을 통해 실행 중인 프로세스는 잠시 중단했다가, 시스템이 기능을 회복했을 때 다시 재시작할 수 있다.

- 사용자는 실행 중인 프로세스를 중단 후 재시작, 정지 가능.
- 처리할 일이 너무 많아 시스템 부담이 크면 프로세스 몇 개를 중단했다가 시스템이 다시 정상 상태로 돌아왔을 때 재시작할 수 있다.



중단과 재시작을 추가한 프로세스 상태 변화

exit()

프로세스를 종료하면서 부모 프로세스에게 종료와 관련된 상태 값을 넘겨준다.

```
#include <stdlib.h>

void exit(int status);
 - status: 부모 프로세스에게 전달되는 상태 값. 임의의 0~255(1Byte)의 값.
```

status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.

프로세스 종료:

- exit는 프로세스를 의도적으로 종료시킨다.
- main 함수 내에서 return문을 수행할 때.

atexit()

프로세스가 exit를 호출하여 종료할 때 수행되는 함수 등록.

```
#include <stdlib.h>

int atexit(void (*function)(void));
 - function: 사용할 함수 이름
 - return: 호출이 성공하면 0을 반환하고, 실패하면 0이 아닌 값을 반환한다.
```

status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.

function:

- 함수는 void function(void) 형으로 정의되어야 한다.
- 종료 시 마무리 작업 (clean-up-action)
- 프로세스가 종료할 때 깔끔한 마무리를 위해 수행해야 하는 작업들 최대 32개까지 등록할 수 있다. (실제 실행 순서는 등록 순서의 역순)

_exit()

exit 함수와 같지만 clean-up-action을 수행하지 않는다

```
#include <stdlib.h>

void _exit(int status);
- status: 부모 프로세스에게 전달되는 상태 값. 임의의 0~255(1Byte)의 값.
```

status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.

atexit()로 clean-up-action에 해당하는 함수들을 등록해 놓았더라도 종료 할 때 이를 수행하지 않는다.

fork() 실습

lec03-process/fork_ex.c

```c
// fork
pid = fork();
if (pid > 0) /* 부모프로세스가 수행하는 부분*/
{
     atexit(cleanupaction);
}
else if (pid == 0) /* 자식프로세스가 수행하는 부분*/
{
     execl("/bin/ls", "ls", "-l", (char *)0);
}
```

```
$ ps
  PID TTY          TIME CMD
51620 ttys000    0:00.20 -bash
53038 ttys001    0:00.00 ./fork_ex
53039 ttys001    0:00.00 (ls)
```

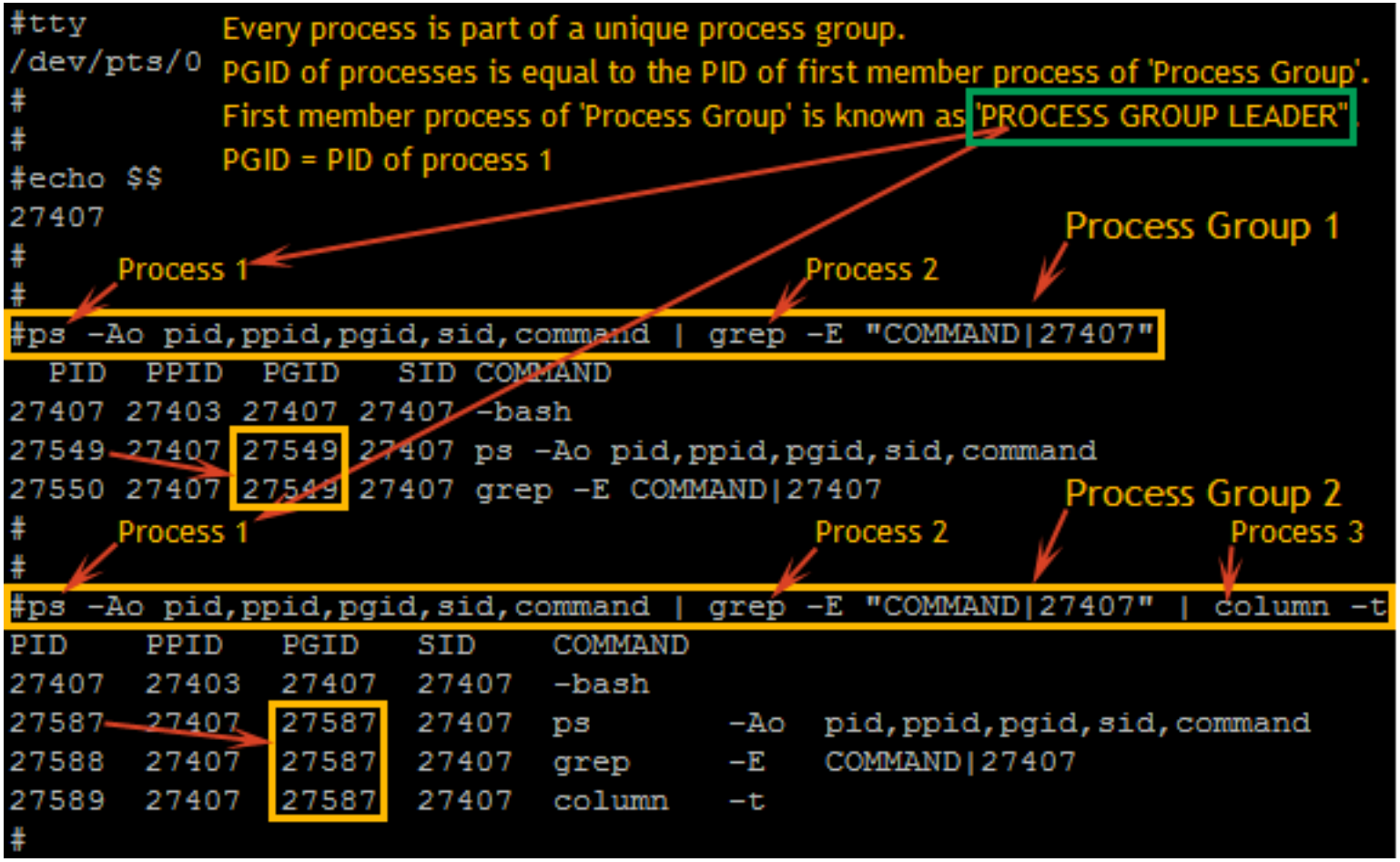
프로세스 환경

프로세스 상태 및 환경 호출

프로세스 동기화, 속성, 환경 변수관련 시스템 호출/표준 라이브러리 함수

함수	의미
wait()	자신의 자식 프로세스가 종료할 때까지 대기 상태가 된다.
waitpid()	지정한 자신의 자식 프로세스가 종료할 때까지 대기 상태가 된다.
getpid(), getppid()	자신(또는 부모)의 프로세스 식별 번호를 구한다.
getpgrp, setpgrp()	자신의 프로세스 그룹 식별 번호를 구하거나 변경한다.
getpgid, setpgid	지정한 프로세스의 그룹 식별 번호를 구하거나 변경한다.
getsid	지정한 프로세스의 세션 식별 번호를 구한다.
setsid	현재 프로세스가 새로운 세션을 생성한다.
getenv, putenv	환경 변수의 값을 구하거나, 새로운 환경 변수를 등록/변경한다.
setenv	새로운 환경 변수를 등록하거나 변경한다.
unsetenv	등록된 환경 변수를 삭제한다.

PID, PGID, SID




```
#tty
/dev/pts/0
#
#
#echo $$
27407
#
#
#ps -Ao pid,ppid,pgid,sid,command | grep -E "COMMAND|27407"
  PID  PPID  PGID   SID COMMAND
27407 27403 27407 27407 -bash
27549 27407 27549 27407 ps -Ao pid,ppid,pgid,sid,command
27550 27407 27549 27407 grep -E COMMAND|27407
#
#
#ps -Ao pid,ppid,pgid,sid,command | grep -E "COMMAND|27407" | column -t
PID    PPID   PGID   SID    COMMAND
27407  27403  27407  27407  -bash
27587  27407  27587  27407  ps       -Ao  pid,ppid,pgid,sid,command
27588  27407  27587  27407  grep     -E   COMMAND|27407
27589  27407  27587  27407  column   -t
#
```

getpid()

자신이나 부모의 프로세스 식별 번호(PID)를 구한다.

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

pid_t getpid(void);  // 현재 프로세스의 PID
pid_t getppid(void); // 부모 프로세스의 PID
 - return: 프로세스의 PID
```

pid=0 현재 프로세스에 대해 알아온다.

getenv()

환경 변수의 값을 알아오거나 새로운 값을 등록한다.

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(char *string);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
 - name, string: name=value 형식으로 구성된 문자열
 - overwrite: 환경변수 존재할 경우 덮어쓰기 여부. 0 덮어쓰기, 0이 아니면 덮어
쓰기를 하지 않는다.
 - return: getenv() 실패할 경우 NULL을 반환.
           putenv와 setenv 성공할 경우 0, 실패할 경우 -1을 반환.
```

PID 실습1

lec03-process/fork_ex2.c

```
 if(pid > 0) {                          /* 부모프로세스가 수행하는 부분*/
   printf("[parent]  PID: %d\n", getpid());
   printf("[parent] PPID: %d\n", getppid());
   printf("[parent]  GID: %d\n", getpgrp());
   printf("[parent]  SID: %d\n", getsid(0));
   waitpid(pid, &status, 0);
}
 else if(pid == 0) {                    /* 자식프로세스가 수행하는 부분*/
   printf("[child]  PID: %d\n", getpid());
   printf("[child] PPID: %d\n", getppid());
   printf("[child]  GID: %d\n", getpgid(0));
   printf("[child]  SID: %d\n", getsid(0));
   sleep(1);
 }
```

PID 실습2

lec03-process/fork_ex3.c

```
if( (pid = fork()) > 0) {                /* 부모프로세스가 수행하는 부분*/
  printf("[fork_ex3]  PPID:%d, PID:%d\n", getppid(), getpid());
  sleep(1);
}
else if(pid == 0) {                      /* 자식프로세스가 수행하는 부분*/
  printf("[fork_ex3]  PPID:%d, PID:%d\n", getppid(), getpid());
  execl("fork_ex3_sub", "fork_ex3_sub", (char* )0);
}
```

```
$ ./fork_ex3
[fork_ex3]  PPID:1865, PID:53362
[fork_ex3]  PPID:53362, PID:53363
[fork_ex3_sub]  PPID:53362, PID:53363
```

getpgrp/getpgid()

프로세스의 그룹 식별 번호를 구하거나 변경한다.

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

int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
int setpgrp(void);
pid_t getpgrp(void);
 - pid: 프로세스 식별 번호이다.
 - pgid: 프로세스 그룹의 식별 번호이다.
 - return: setpgid(),setpgrp() 호출 성공시 0, 실패 -1
     getpgid() 호출 성공 GID, 실패 -1. getpgrp() 프로세스 그룹 식별 번호
```

프로세스 식별번호 (process id): 프로세스 아이디
프로세스 그룹: 여러 프로세스가 속한 그룹
프로레스 그룹 식별번호: 프로세스 그룹의 아이디
  - 프로세스 그룹 리더는 그룹아이디와 동일한 프로세스 아이디이다.

getsid()

프로세스의 세션 식별 번호를 구하거나, 새로운 세션을 생성한다

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

pid_t getsid(pid_t pid);
pid_t setsid(void);
 - pid: 프로세스 식별 번호이다.
 - return: 호출 성공 SID, 실패 -1
```

pid=0 현재 프로세스에 대해 알아온다.

세션 (session):

- 일반적으로 시스템과 연결된 하나의 제어 단말기를 포함한 단위
- 식별 번호(id)가 부여되어 있다.
- 세션 > 그룹 > 프로세스
- 세션의 리더 (프로세스): 자신의 PID = PGID = PSID 일 경우

GID 실습

lec03-process/fork_ex4.c

```c
if( (pid = fork()) > 0) {                    /* 부모프로세스가 수행하는 부분*/
    printf("[parent] PPID:%d, PID:%d\n", getppid(), getpid());
    printf("[parent] GID:%d, PGID:%d, SID:%d\n",
                getgid(), getpgrp(), getsid(getpid()));
    printf("[parent] getpgid(0):%d, getpgid(getpid()):%d\n",
                getpgid(0), getpgid(getpid()));

    sleep(1);
}
else if(pid == 0) {                          /* 자식프로세스가 수행하는 부분*/
    printf("[child] PPID:%d, PID:%d\n", getppid(), getpid());
    printf("[child] GID:%d, PGID:%d, SID:%d\n",
           getgid(), getpgrp(), getsid(getpid()));
    printf("[child] getpgid(0):%d, getpgid(getpid()):%d\n",
           getpgid(0), getpgid(getpid()));
}
```

```
[parent] PPID:1865, PID:53612
[parent] GID:20, PGID:53612, SID:1863
[parent] getpgid(0):53612, getpgid(getpid()):53612
[child] PPID:53612, PID:53613
[child] GID:20, PGID:53612, SID:1863
[child] getpgid(0):53612, getpgid(getpid()):53612
```

wait()

자신의 자식 프로세스가 종료할 때까지 대기한다.

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

pid_t wait(int *status);
 - status: 부모 프로세스에게 전달되는 상태 값. 임의의 0~255(1Byte)의 값.
 -return:호출이 성공했을 경우 종료한 자식 프로세스의 식별 번호가 반환되고, 실
패할 경우 -1이 반환된다.
```

status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.

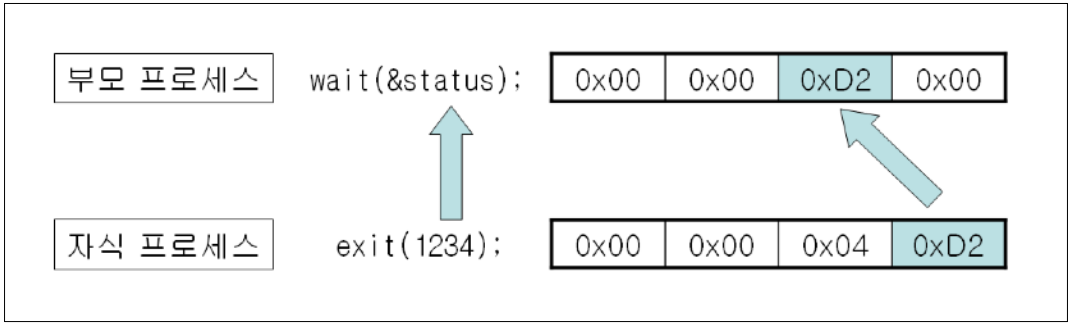
자식 프로세스를 가진 부모 프로세스가 wait를 호출하면

- 자식 프로세스가 종료할 때까지 실행이 중단된다. (대기 상태)
- 자식 프로세스가 종료하면 이를 처리한다.
  + wait 호출 이전에 자식 종료면 대기 상태가 되지 않고 처리

wait()

자식 프로세스가 exit를 호출하면서 지정한 값을 부모 프로세스는 status 변수로 받는다.

- 자식 프로세스가 exit(n); 을 실행했을 때 부모 프로세스에게 전달되는 실제 값은 n의 하위 1바이트 뿐이다.
- 자식 프로세스가 전달한 1바이트 값은 부모 프로세스 쪽의 status 변수의 하위 두 번째 바이트에 저장된다.



	8비트	8비트
정상 종료	프로세스 반환 값	0
비정상 종료	0	종료 시킨 시그널 번호

exit() 실습

lec03-process/fork_exit.c

```
 if(pid > 0) {                          /* 부모프로세스가 수행하는 부분*/
   printf("[parent] waiting...\n";
   wait(&status);
   printf("[parent] status: %d\n", status);
}
 else if(pid == 0) {                    /* 자식프로세스가 수행하는 부분*/
   sleep(1);
   printf("[child]  bye!\n");
   exit(1234); //
 }
```

자식 프로세스가 1234로 exit했다.
부모 프로세스가 실제로 받는 값은 얼마인가?

```
$ ./fork_ex3
[parent] waiting...
[child]  bye!
[parent] status: 53760
Bye Bye!
```

비정상 상태에 따른 분류

Zombie process

- 부모프로세스가 wait() 않고 있는 상태에서 자식이 종료.
- 자식프로세스 종료를 부모 프로세스가 처리해주지 않으면 자식 프로세스는 좀비 프로세스가 된다.
- 좀비 프로세스는 CPU, Memory 등의 자원을 사용하지 않으나, 커널의 작업 리스트에는 존재한다.

Orphan process

- 하나 이상의 자식 프로세스가 수행되고 있는 상태에서 부모가 먼저 종료

init process

- 좀비와 고아 프로세스의 관리는 결국 시스템의 init 프로세스로 넘겨진다.
- init 프로세스가 새로운 부모가 된다.

waitpid()

자신의 자식 프로세스가 종료할 때까지 대기한다.

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

pid_t waitpid(pid_t pid, int *status, int options);
 - pid: child pid.
 - status: 부모 프로세스에게 전달되는 상태 값. 임의의 0~255(1Byte)의 값.
 - options: 부모프로세스 대기방법. 보통 0.
 - return: 성공 child pid. 실패할 경우 -1. WNOHANG 옵션시 0.
```

status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.

wait와 waitpid의 차이점

- wait는 자식프로세스 중 가장 먼저 종료되는 것을 처리해주나,
- waitpid는 PID로 지정한 자식 프로세스의 종료만 처리해준다.

waitpid() 실습

lec03-process/fork_waitpid.c

```
pid1 = fork();
if(pid1 > 0) {
  pid2 = fork();
}

if( pid1 > 0 && pid2 > 0) {          /* 부모프로세스가 수행하는 부분*/

}
else if( pid1 == 0 && pid2 == -1 ) { /* child1 */

}
else if( pid1 > 0 && pid2 == 0) {    /* child2 */

}
```

waitpid() 실습

lec03-process/fork_waitpid2.c

waitpid()에 WNOHANG 사용

- PID 자식 프로세스가 종료하였는지 확인해서
  - 종료했으면 이를 처리하고
  - 종료하지 않았으면 자신의 일을 계속 수행한다.

```
pid_t pid;
int status;

if( (pid = fork()) > 0) {         /* 부모프로세스가 수행하는 부분*/
  while( !waitpid(pid, &status, WNOHANG)) {
    printf("[parent] status: %d\n", status++);
    sleep(1);
  }
  printf("[parent] child exit: %d\n", status);
}
else if( pid == 0 ) { /* child1 */
  sleep(5);
  printf("bye child!\n");
  exit(1);
}
```

환경변수 전달

새 프로세스 실행시 환경변수를 옵션으로 전달할 수 있다.
- putenv() 같이 환경변수 등록이 아니다.
- exec 계열의 새로운 함수를 사용: execle(), execve()

```
char *envlist[] = {"APPLE=BANANA", (char *)0};
...
execle("myprogram", "myprogram", (char *)0, envlist);
execve("myprogram", arglist, envlist);
```

마지막에 NULL

환경변수 받기:

```
extern char **environ;
while(*environ)
    printf("%s\n", *environ++);
```

```
main(int argc, char *argv[], char *envlist[]) {
    while(*envlist)
        printf("%s\n", *envlist++);
}
```

환경변수 전달 실습

lec03-process/envex.c

execle()에 환경변수 목록을 전달한다.

```
int main(int argc, char **argv)
{
  char *envlist[] = {"APPLE=0", "BANANA=1", (char *)0 };
  execle("envex_sub", "envex_sub", (char *)0, envlist);

  return 0;
}//end-of-main
```

호출 프로세스에서 환경변수를 활용한다.

```
extern char **environ;

int main(int argc, char **argv)
{
  while(*environ)
    printf("%s\n", environ++);
  return 0;
}//end-of-main
```

종료와 sid

로그인 셸
- 제어 터미널의 연결 상태를 가진다. (제어 터미널을 포함한 세션)
- 세션과 그룹 내에서 리더이다.
- 셸 프롬프트 상에서 실행한 많은 프로세스가 있다.
- 세션의 리더이기 때문에 연결이 끊어진다.
- 같은 세션에 있는 다른 프로세스도 종료된다.
- background 실행 중인 프로세스도 종료된다.

종료와 sid 실습

lec03-process/fork_ex5.c

차일드 프로세스에서 setsid() 를 실행하면 새 세션이 생성되며, 현재 프로세스가 프로세스 리더가 된다.

```
  if( (pid = fork()) > 0) {                          /* 부모프로세스가 수행하는 부분*/
    printf("[parent] PPID:%d, PID:%d\n", getppid(), getpid());
    printf("[parent] GID:%d, PGID:%d, SID:%d\n", getgid(), getpgrp(),
getsid(getpid()));
    printf("[parent] getpgid(0):%d, getpgid(getpid()):%d\n", getpgid(0),
getpgid(getpid()));

    sleep(1);
  }
  else if(pid == 0) {                    /* 자식프로세스가 수행하는 부분*/
    printf("[child] Current SID:%d\n", getsid(0));
    // 새 세션이 만들어 지며 현재 프로세스가 리더가 된다
    printf("[child] New SID:%d\n", setsid());
    sleep(600);
  }
```

시간

tms

리눅스 시스템에서는 프로세스의 수행 시간을 clock_t 단위로 계산하는 times 함수 제공, times 함수는 struct tms 형식 변수의 주소를 입력 인자로 받아 수행 시간을 채운다.

struct tms 형식은 User 모드 수행한 시간, Kernel 모드 수행한 시간, 종료한 자식 User 모드 수행한 시간과 Kernel 모드에서 수행한 시간을 기억하는 멤버로 구성.

```
#include <sys/times.h>
clock_t times(struct tms *buf);
struct tms {
    clock_t tms_utime; //user mode time, USER CPU time
    clock_t tms_stime; //kernel mode time, SYSTEM CPU time
    clock_t tms_cutime; //user mode time (terminated children)
    clock_t tms_cstime; //kernel mode time (terminated children)
};
```

실습-prc_runtime.c

lec03-process/prc_runtime.c

```c
int main() {
    srand((unsigned)time(0));

    int i = 0;
    for (i = 0; i < 5; i++)
        do_in_child(i + 1, rand() % 10);

    int rval = 0;    pid_t cpid = 0;
    while (i > 0) {
        cpid = wait(&rval);
        printf("cpid:%d exit status:%#x\n", cpid, rval);
        i--;
    }

    for (i = 0; i < 10000000; i++)       putchar('-');

    struct tms buf;    times(&buf);

    printf("USER CPU time:%d \n", buf.tms_utime);
    printf("SYSTEM CPU time:%d \n", buf.tms_stime);
    printf("Children's USER CPU time:%d \n", buf.tms_cutime);
    printf("Children's SYSTEM CPU time:%d \n", buf.tms_cstime);
    return 0;
}
```

5개의 자식 프로세스를 생성하여 표준 출력에 문자를 반복 출력하는 구문을 작성하게 하여 CPU 시간을 측정

실습-prc_runtime.c

lec03-process/prc_runtime.c

```c
void do_in_child(int seq, int rval) {
    pid_t cpid = fork();

    if (cpid == -1) {
        perror("error fork");
        return;
    }

    if (cpid > 0)
        printf("fork child pid:%u\n", cpid);
    else {
        printf("pid:%d sleep %dseconds\n", getpid(), rval);

        int i = 0;
        for (i = 0; i < 10000000; i++) {
            putchar('.');
        }

        exit(seq);
    }
}
```

실습-prc_runtime.c

lec03-process/prc_runtime.c

pid=0 현재 프로세스에 대해 알아온다.

```
$ ./prc_runtime
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
----------------------
USER CPU time:46
SYSTEM CPU time:5
Children's USER CPU time:237
Children's SYSTEM CPU time:165
```

실습-prc_runtime.c

lec03-process/prc_runtime.c

time 명령으로 실행시간 측정

```
$ time ./prc_runtime
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
----------------------
USER CPU time:11
SYSTEM CPU time:5
Children's USER CPU time:61
Children's SYSTEM CPU time:34

real  0m15.418s
user 0m0.727s
sys   0m0.401s
```

user: user영역에서 실행된 시간,
sys: 커널에서 실행된 시간,
real: 둘을 합산한 총 실행시간