프로세스 프로그래밍
Process Programming

- 프로세스 생성: fork, exec
- 프로세스 동기화: wait
- 프로세스 관리 함수

프로세스 관련 함수

❑ 프로세스 생성과 종료

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.
wait	자신의 자식 프로세스가 종료할 때까지 대기 상태가 된다.
waitpid	지정한 자신의 자식 프로세스가 종료할 때까지 대기 상태가 된다.

프로세스 관련 함수

❑ 프로세스 속성과 환경 변수

함수	의미
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.

fork

❑ 프로세스를 복제하여 완전히 동일한 자식 프로세스 생성

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

pid_t fork(void);
```

반환값	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.

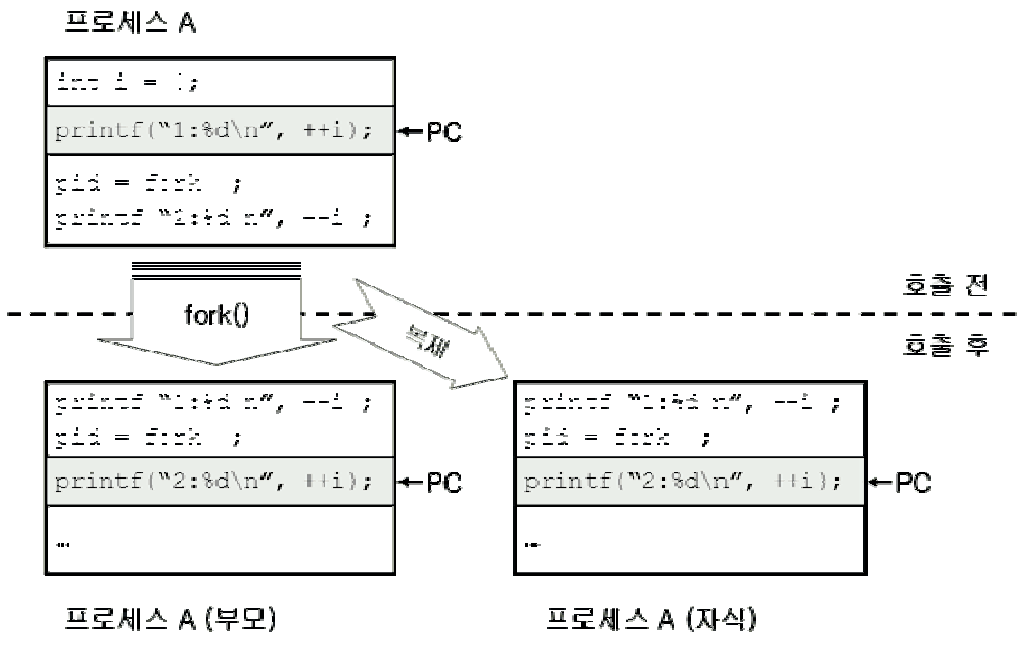
fork

❑ fork로 프로세스 생성하면…

- 생성하는 쪽이 부모(parent), 생성된 프로세스는 자식(child)
- 서로 다른 프로세스이다 → 서로 다른 PID
  - 자식의 PPID는 부모 프로세스의 PID
- 실행 상태는 똑같다.
  - fork를 호출하던 시점의 context(실행 상태)를 그대로 물려 받는다.
    - 프로그램 코드
    - 변수, 레지스터 값
    - 스택의 값 등등
  - fork 이후부터는 각자 자기 길을 간다!

fork

❑ fork로 프로세스 생성하면…



fork

❑ 사용 예

```
pid_t pid;

pid = fork();
if (pid > 0)          // 부모 프로세스가 수행할 부분
  …
else (pid == 0)     // 자식 프로세스가 수행할 부분
  …
else                  // fork 호출이 실패할 경우
  …
```

- 부모와 자식이 수행할 일이 다르다면 pid 반환값을 이용해 구분하여 코딩한다.

exec 계열

❑ 경로 또는 파일 이름으로 지정한 실행 파일을 실행하여 프로세스 생성

```
#include <unistd.h>

extern char **environ;

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	실행 파일의 경로로 상대 경로와 절대 경로 모두 사용 가능
file	경로 이름이 아닌 실행 파일의 이름
arg	실행 파일과 전달인자 리스트. 마지막 인자는 반드시 NULL
argv	arg와 같으나 문자열 포인터 배열 형태. 배열의 마지막 요소는 반드시 NULL 문자열.
반환값	호출이 성공하면 호출 프로세스는 종료되고 반환값을 받을 수 없다. 실패 시 -1 반환.

exec 계열

❑ 사용 예

```
execlp("ls", "ls", "-l", "subdir", NULL);
```

== *$ ls -l subdir*

실행파일이름 명령어와 옵션, 전달인자

whereis, which 활용

❑ exec 호출 결과

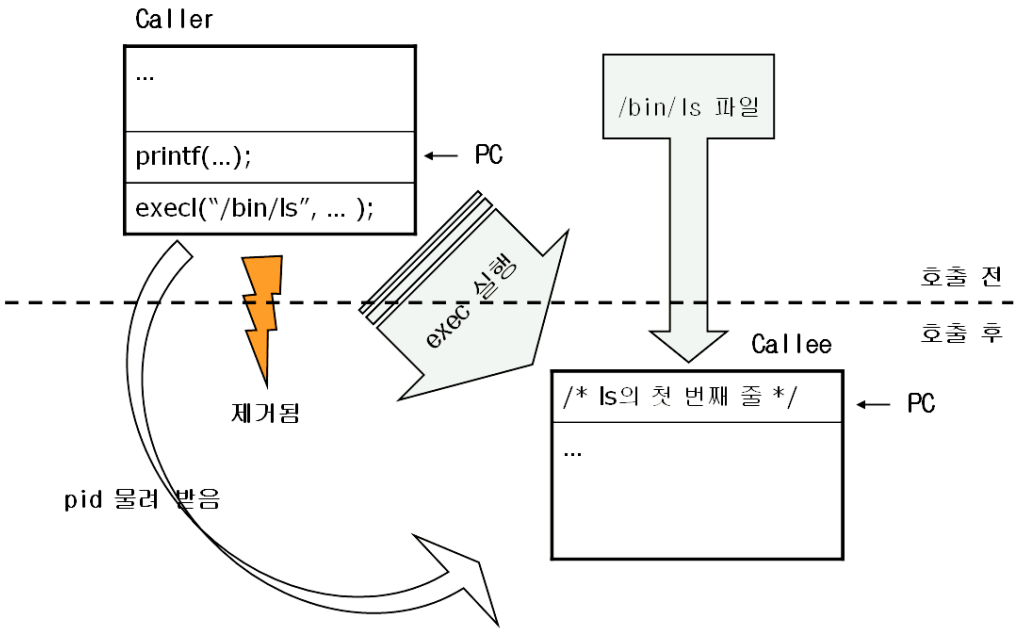
- caller - callee 관계 생성
- caller는 종료된다.
- caller의 메모리 영역을 callee가 차지하고 PID도 물려 받는다.

❑ 이름 구분

- l은 리스트 형태 인자, v는 문자 포인터 배열 형태 인자
- p가 없으면 경로를 직접 지정, p가 있으면 환경 변수 PATH 이용

exec 계열

❑ exec로 프로세스 생성하면...



fork & exec

❑ fork, exec 비교

	fork	exec 계열
프로세스의 원본	부모 프로세스를 복제	지정한 프로그램(파일)을 실행
명령행 인자	부모 프로세스 것을 그대로 사용	필요할 경우 새로 적용
부모(또는 호출) 프로세스의 상태	자식 프로세스를 생성한 후에도 자신의 나머지 코드를 실행	호출이 성공할 경우 호출(Caller) 프로세스 종료
자식(또는 피호출) 프로세스의 메모리 상의 위치	부모 프로세스와 다른 곳에 위치. 처음 내용은 같다.	호출 프로세스가 있던 자리를 피호출 프로세스가 물려받는다.
자식(또는 피호출) 프로세스의 실행 시작점	fork 호출 이후부터 수행	프로그램의 처음부터 수행
프로세스 식별 번호	자식에게 새로운 PID	Caller와 Callee PID 동일
프로세스의 원본인 파일에 대한 권한	권한도 복제	실행 파일에 대한 실행 권한이 필요

fork & exec

❑ fork와 exec 함께 사용

- 부모와 자식이 다른 작업을 하면서 둘 다 살아있게 하려면?
  - fork로 일단 복제
  - 자식 프로세스에서 exec를 호출하여 다른 프로그램 실행

```
pid_t pid;

pid = fork();
if (pid > 0) {
   // 부모 프로세스가 수행할 부분
   …
}
else (pid == 0)   // 자식 프로세스
   execl("…", "…", …);
else                    // fork 호출이 실패할 경우
   …
```

exit

❑ 프로세스를 종료하고 부모 프로세스에게 종료 상태값 전달

#include <stdlib.h> void exit (int status);	
status	부모 프로세스에게 전달되는 상태 값으로 0~255(1바이트)의 값이 사용된다.
반환값	없음

- 프로세스를 명시적으로 종료시킴
  - 암묵적 종료: 더 이상 수행할 문장이 없거나 return 문 수행
- status 값에는 정해진 의미가 없다!
  - 보통 0이 정상 종료

atexit

❑ exit 호출 시 종료 전에 수행할 함수들을 등록

#include <stdlib.h> int atexit (void (*function)(void));	
function	atexit로 등록할 함수의 이름
반환값	호출이 성공하면 0을 반환, 실패하면 0이 아닌 값을 반환

- 함수는 void function(void); 형으로 정의되어야 한다.
- 종료 시 마무리 작업 (clean-up-action)
  - 깔끔한 마무리를 위해 수행해야 하는 작업들
    - 파일 닫기
    - 자식 프로세스 종료 등
  - 최대 32개까지 등록 가능하며 등록 역순으로 실행된다

_exit

❑ exit와 같지만 마무리 작업이 없다.

- clean-up-action에 해당하는 작업을 등록해 두었더라도 수행하지 않는다.

```
#include <unistd.h>

void _exit (int status);
```

status	부모 프로세스에게 전달되는 상태 값으로 0~255(1바이트)의 값이 사용된다.
반환값	없음

예제 1

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

void clean_up_action(void);

int main (void)
{
    pid_t pid;
    int i;
    char *arg[] = {"ls", "-l", ""};

    for (i=0; i<3; i++) {
        printf("before fork [%d]₩n", i);
        sleep(1);
    }
    pid = fork();
    if (pid > 0) { // 부모 프로세스인 경우
        for ( ; i<7; i++) {
            printf("parent [%d]₩n", i);
            sleep(1);
        }
        atexit(clean_up_action);
    }
    else if (pid == 0) { // 자식 프로세스인 경우
        for ( ; i<5; i++) {
            printf("child [%d]₩n", i);
            sleep(1);
            execv("/bin/ls", arg);
        }
    }
    else
        printf("fail to fork child process₩n");
    exit(0);
```

실행 결과:

```
juyoon@ce:~/system/programming/process
[juyoon:process/22]$ ex1
before fork [0]
before fork [1]
before fork [2]
child [3]
parent [3]
parent [4]
합계 20
-rwxrwxr-x    1 juyoon   juyoon
-rw-rw-r--    1 juyoon   juyoon
parent [5]
parent [6]
Clean up action!
[juyoon:process/23]$
```

예제 2

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

int main (int argc, char* argv[])
{
    pid_t pid;
    char **arg;
    i

    p

    p

    a
    i

        }
    }
    else if (pid == 0) {
        sleep(2);
        execlp("ex2", "ex2", "child", NULL);
        printf("I'm dying...₩n");
    }
    else printf("Error in process %d!₩n", getpid());
    return 0;
}
```

절대로 linda에서 테스트하지 말 것!!!

실행 결과:

```
juyoon@ce:~/system/programming/process
[juyoon:process/18]$ ex2
before fork
                                                    ork
14398 pts/5    00:00:00 ex2
14399 pts/5    00:00:00 ex2
14400 pts/5    00:00:00 ps
[juyoon:process/20]$ 14398: child
before fork
14399: ex2
14399: child
before fork
14401: ex2
```

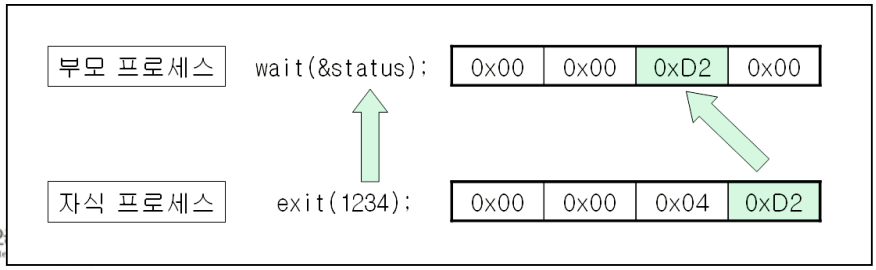

wait

❑ 자식 프로세스가 종료할 때까지 대기

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

pid_t wait (int *status);
```

status	자식 프로세스가 exit 함수로 종료하면서 전달하는 종료 상태 값으로 0에서 255 사이의 값을 가진다.
반환값	호출이 성공했을 경우 종료한 자식 프로세스의 식별 번호가 반환되고, 실패할 경우 -1이 반환된다.

- 이미 종료한 상태라면 대기하지 않고 처리
- status: 전달된 결과는 부모의 두번째 바이트에 저장



wait

❑ 특별한 상태의 프로세스

- 좀비 프로세스 (Zombie process)
  - == defunct process
  - CPU, memory 등의 자원을 사용하지는 않으나 커널의 작업 리스트에는 존재하는 프로세스
  - 부모가 wait하고 정상종료하면 죽을 수 있다.
- 고아 프로세스 (Orphan process)
  - 하나 이상의 자식 프로세스가 수행 중인 상태에서 부모가 먼저 종료
- 문제가 되나?
  - 지나치게 많을 경우 시스템 자원 고갈
  - init process (PID 1)가 거두어 처리해 준다.

waitpid

❑ PID로 지정한 자식 프로세스의 종료 대기

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

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

pid	자식 프로세스의 식별 번호
status	자식 프로세스가 exit로 종료하면서 전달하는 종료 상태 값
options	부모 프로세스의 대기 방법을 선택. 일반적으로 0이 사용된다.
반환값	호출이 성공했을 때 종료한 자식 프로세스가 있다면 자식 프로세스의 식별 번호가 반환, WNOHANG 옵션을 사용할 때 종료한 자식 프로세스가 없으면 0을 반환. 호출이 실패할 경우 -1을 반환.

- wait는 가장 먼저 종료되는 자식 프로세스 처리
- waitpid: 부모가 처리 순서를 결정할 수 있다.
  - -1이면 wait와 같다.
- 옵션에 따라서 자식 종료 시까지 대기 안 할 수도 있다.

waitpid

❑ 옵션

- WNOHANG
  - PID로 지정한 자식 프로세스가 종료하지 않았으면 자신의 일을 계속 수행

```
pid_t pid1;
if ((pid = fork()) > 0) {
   while (!waitpid(pid, &status, WNOHANG)) {
         printf("parent: %d\n", status++");
         sleep(1);
   }
}
else if ( pid == 0) {
   sleep (5);
   printf("bye!\n");
   exit(0);
}
…
```


예제 3

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

int main (void)
{
    pid_t pid1, pid2;
    int status;

    pid1 = pid2 = -1;
    pid1 = fork();
    if (pid1 > 0)                    // parent process
        pid2 = fork();
    if (pid1 > 0 && pid2 > 0) { // parent process
        waitpid (pid2, &status, 0);
        printf("parent: child2 - exit (%d)\n", status);
        waitpid (pid1, &status, 0);
        printf("parent: child1 - exit (%d)\n", status);
    }
    else if (pid1 == 0 && pid2 < 0) { // first child
        sleep(1);
        exit(1);
    }
    else if (pid1 > 0 && pid2 == 0) { // second child
        sleep(2);
        exit(2);
    }
    else printf("fail to fork\n");
    return 0;
}
```

실행 결과:

```
juyoon@linda: ~/test/system/programming/process
[juyoon@linda:process] waitpid
parent: child2 - exit (512)
parent: child1 - exit (256)
[juyoon@linda:process]
```

getpid, getppid

❑ 자신 또는 부모의 식별번호를 구한다.

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

pid_t getpid (void);
pid_t getppid (void);
```

반환값	프로세스의 식별 번호

- PID: 프로세스 식별 번호
  - 음이 아닌 정수
  - 한 시점에서 프로세스를 식별하는 유일한 번호

프로세스 그룹

❑ 프로세스 그룹의 식별 번호를 구하거나 변경

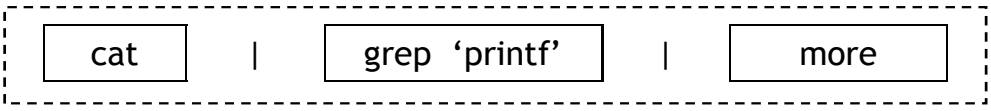
```
#include <sys/types.h>
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);
pid_t getpgid (pid_t pid);
int setpgrp (void);
pid_t getpgrp void);
```

pid	프로세스 식별 번호
pgid	프로세스 그룹의 식별 번호
반환값	setpgid와 setpgrp는 호출이 성공할 경우 0을 반환, 실패할 경우 -1을 반환 getpgid는 호출이 성공할 경우 프로세스의 그룹 식별 번호를 반환, 실패할 경우 -1을 반환 getpgrp은 항상 프로세스의 그룹 식별 번호를 반환

프로세스 그룹

❑ 프로세스 그룹

- 여러 프로세스들이 하나의 그룹에 속할 수 있음
- 시그널 처리를 동시에 할 수 있다.
- 그룹 식별번호를 별도로 가진다 - 그룹 리더의 PID



❑ 함수와 파라미터의 의미

- setpgid (pid, pgid)
  - pid 프로세스의 그룹ID를 pgid로 변경
  - pid = 0: 현재 프로세스에 적용
  - pgid = 0: pid를 pgid로 사용
- setpgrp, getpgrp
  - setpgrp == setpgid(0,0): 현재 프로세스를 리더로 하는 그룹설정
  - getpgrp == getpgid(0): 현재 프로세스의 그룹ID 구하기

세션 식별 번호

❑ 세션 (Session)

- 로그온 중인 하나의 단말을 포함하는 단위
- 세션 ⊃ 그룹 ⊃ 프로세스
- 세션 리더의 경우: PID == PGID == PSID

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

pid_t getsid (pid_t pid);
pid_t setsid (void);
```

pid	프로세스의 식별 번호. 0이면 현재 프로세스
반환값	성공하면 프로세스의 세션 식별 번호를 반환, 실패하면 -1을 반환

- setsid
  - 리더가 아닌 경우 새로운 세션 생성 (일반적으로 셸이 리더)
  - 호출한 프로세스가 세션과 그룹의 리더가 된다.

세션 식별 번호

❑ 종료와 세션

- 로그인 셸이 세션 리더
- 셸을 종료하면 다른 프로세스도 종료

juyoon@ce:~/system/programming/process

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

int main (int argc, char *argv[])
{
    pid_t pid;

    if (argc != 2)
        exit(1);
    pid = atoi(argv[1]);

    printf("shell..₩n");
    printf("PID: %d,PGID: %d, SID: %d₩n",
            pid, getpgid(pid), getsid(pid));
    printf("current process..₩n");
    printf("PID: %d,PGID: %d, SID: %d₩n",
           getpid(), getpgrp(), getsid(0));

}
```

juyoon@ce:~/system/programming/process

```
[juyoon:process/216]$ ps
  PID TTY          TIME CMD
14266 pts/5    00:00:00 bash
15390 pts/5    00:00:00 vim
15443 pts/5    00:00:00 ps
[juyoon:process/217]$ sid 15390
shell..
PID: 15390,PGID: 15390, SID: 14266
current process..
PID: 15444,PGID: 15444, SID: 14266
[juyoon:process/218]$
```


세션 식별 번호

❑ 데몬(daemon)

- 셸 종료 후에도 살아 있는 프로세스
  - init 프로세스의 자식
  - 특정 터미널에 연결되지 않아야 한다.
- 주로 서버(서비스 제공자)로 동작
- setsid로 만들 수 있다.

❑ 서버 데몬 만들기

- fork()로 자식을 만들고 부모는 종료(exit)한다.
- setsid()로 새 세션을 만든다.
- chdir()로 작업 디렉터리를 '/'로 바꾼다.
- 입출력을 재지정한다.
  - 열린 모든 파일을 닫고, 표준입출력을 /dev/null로 지정
- C 표준함수 daemon(0, 0)으로 호출하는 것과 같은 효과

세션 식별 번호

❑ 데몬 예제

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

int main (void)
{
    pid_t pid;

    if ((pid=fork()) > 0) {
        sleep(1);
        exit(1);
    }
    else if (pid == 0) {
        printf("old session id: %d₩n", getsid(0));
        printf("new session id: %d₩n", setsid());
        sleep(600);
    }
    return 0;
}
```

```
juyoon@ce:~/system/programming/process
[juyoon:process/235]$ ps -ef | grep juyoon
juyoon    14266 14264  0 15:36 pts/5    00:00:00 -bash
juyoon    15552     1  0 18:24 ?        00:00:00 demon
juyoon    15556 15555  0 18:25 pts/0    00:00:00 -bash
juyoon    15596 14266  0 18:26 pts/5    00:00:00 ps -ef
juyoon    15597 14266  0 18:26 pts/5    00:00:00 grep juyoon
[juyoon:process/236]$
```


환경 변수

❑ 환경 변수의 값을 알아 오거나 새로 설정

```c
#include <stdlib.h>

char *getenv (const char *name);
int setenv (const char *name, const char *value, int overwrite);
int putenv (char *string);
void unsetenv (const char *name);
```

name	환경 변수의 이름에 해당하는 문자열
value	환경 변수의 값에 해당하는 문자열
string	“name=value” 형식으로 구성된 문자열
overwrite	name 환경 변수가 이미 존재할 경우 덮어쓰기 여부를 결정한다. 0이면 덮어쓰기를 하고 0이 아니면 덮어쓰기를 하지 않는다.
반환값	getenv는 성공할 경우 name에 해당하는 환경 변수의 값에 대한 문자열 포인터를 반환, 실패할 경우 NULL을 반환 putenv와 setenv는 성공할 경우 0을 반환, 실패할 경우 -1을 반환

환경 변수

❑ 프로세스 간에 값 주고 받기

- exec 계열 함수로 프로세스 생성 시 환경 변수값 전달
  - setenv/getenv
  - envlist 추가하여 exec 함수 호출

- setenv – getenv

```
putenv("APPLE=RED");
execl("prog", "prog", NULL);
```

caller

```
if (getenv("APPLE") {
   printf("%s\n", getenv("APPLE"));
   unsetenv("APPLE");
}
```

callee (prog)

환경 변수

❑ 프로세스 간에 값 주고 받기

- envlist 추가하여 exec 함수 호출
  - exec 계열의 함수 execle, execve 사용

caller

```
char *envlist[] = {"APPLE=BANANA", NULL};
…
execle("prog", "prog", NULL, envlist);
execve("prog", arglist, envlist);
```

callee (prog)

```
extern char *envlist[];
int main (void) {
   …
   while (*envlist) { … }
}
```

또는

```
int main (int argc,
          char *argv[],
          char *envlist[]) {
   …
   while (*envlist) { … }
}
```

예제 4

juyoon@ce:~/system/programming/process

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

int main (void)
{
    pid_t pid;
    int status;

    pid = fork();
    putenv("APPLE=RED");
    if (pid >0) {       // 부모 프로세스
        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);
        printf("[parent] status is %d₩n", status);
        unsetenv("APPLE");
    }
    else if (pid == 0) {
        printf("[child] PID: %d₩n", getpid());
        printf("[child] PPID: %d₩n", getppid());
        printf("[child] GID: %d₩n", getpgrp());
        printf("[child] SID: %d₩n", getsid(0));
        sleep(1);
        printf("[child] APPLE=%s₩n", getenv("APPLE"));
        exit(1);
    }
    else printf("fail to fork₩n");
    return 0;
}
```

실행 결과:

juyoon@ce:~/system/programming/proc

```
[juyoon:process/129]$ ex3
[child] PID: 14957
[child] PPID: 14956
[child] GID: 14956
[child] SID: 14266
[parent] PID: 14956
[parent] PPID: 14266
[parent] GID: 14956
[parent] SID: 14266
[child] APPLE=RED
[parent] status is 256
[juyoon:process/130]$
```