리눅스시스템 및 응용

Week 13

Process Programming

학습목표

- 프로세스 생성 방법과 상호 비교
  - system()
  - exec_()
  - fork()
- 프로세스의 동기화
- 시그널의 생성과 처리

Process

- Process Identifier
  - 식별자 , PID
  - 2 ~ 32768
  - 1: init process

- Process의 기능
  - Stack Space : 지역변수, 함수호출 및 반환 등 제어
  - 환경변수를 위한 공간
  - 자체 프로그램 카운터(PC) 관리

- 프로세스 상태 보기 명령어
  - ps –af, ps –ax, ps –ux
  - top, gtop

- System Process
  - N, NN, NNN 자리수의 프로세스
  - 주로 부팅시 init 프로세스 혹은 init에 의해 활성화된 프로세스에 의해 생성
  - 시스템의 유지관리, 운영 프로세스

- 프로세스의 우선순위
  - Process Scheduler에 의해 프로세서(ALU)를 점유할 권리의 우선 순위
  - 기준값 : 10
  - nice : 기준값 10 + 로 변경
  - renice : 우선순위 변경 (by root)

Process

- Process 관련 시스템 call
  - system()
  - exec_()
  - fork()

system()

```
#include <stdlib.h>
int system(const char *string);
```

- *string으로 전달되는 실행가능 코드파일명을 실행시킨다
- 이는 쉘에서 명령어라인에 입력하는 것과 동일 !!

- 일반적으로 쉘에서 직접 명령어를 수행하는 것이 타당
- 단, 프로그램 속에서 실행이 필요한 경우 사용

system(): Ex

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

int main(){
        printf("Running ps with system() \n“);
        system("ps –ax ");
        printf("Done --------\n");
        exit(0);
}
```

exec_()

- 실행한 프로세스(caller process) 대체
- Caller process 에서 Callee process로 pid, 프로세스 환경등이 이전
- caller process의 프로그램 내용은 임무 끝
- 현재 프로세스를 path나 file 인수에 지정된 실행파일을 실행하기 위하여 새로운 프로세스로 대체

```
#include <unistd.h>
char **environ;
ini execl()
int execlp()
int execle()
int execv()
int execvp()
int execve()
```

- 프로세스 시작 방법과 인수(parameter)전달 방법에 따라 선택

- **____l(), ____l_()**
  - **execl(),execlp(), execle()**
  - 알수 없는 개수의 argument전달
  - Null pointer로 끝나는 갯수의 인수 전달 받음
- **____v(), ____v_()**
  - **execv(), execvp()**
  - argv 전달
- **____p()**
  - $PATH환경 변수 검색
- **____e()**
  - Environment 변수 전달 가능

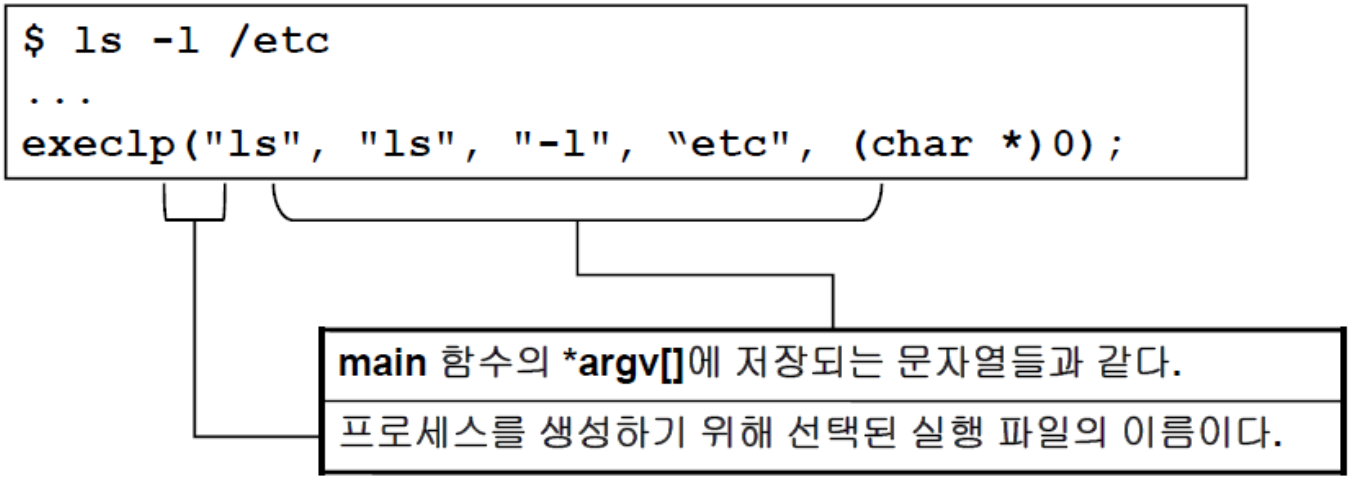
```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- path
  - 실행 파일의 경로로 상대 경로와 절대 경로 모두 사용할 수 있다
- file
  - 경로 이름이 아닌 실행 파일의 이름
- 반환값(return)
  - 호출이 성공하면 호출하는 프로세스에서는 반환 값을 받을 수없다.
  - 만약 함수 호출 후 -1이 반환되면 이는 함수 호출이 실패 했음을 의미

```
#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[]);
```

- arg
  - path나 file로 지정한 실행 파일을 실행할 때 필요한 명령어라인의 옵션과 인자이다.
  - 한 개 이상을 지정할 수 있으며 마지막 인자는 반드시 NULL 포인터로 지정해야 한다
- argv
  - arg와 같은 의미를 가지나 문자형 포인터의 배열로 형태가 다르다.
  - 배열의 마지막은 NULL 문자열로 끝나야 한다

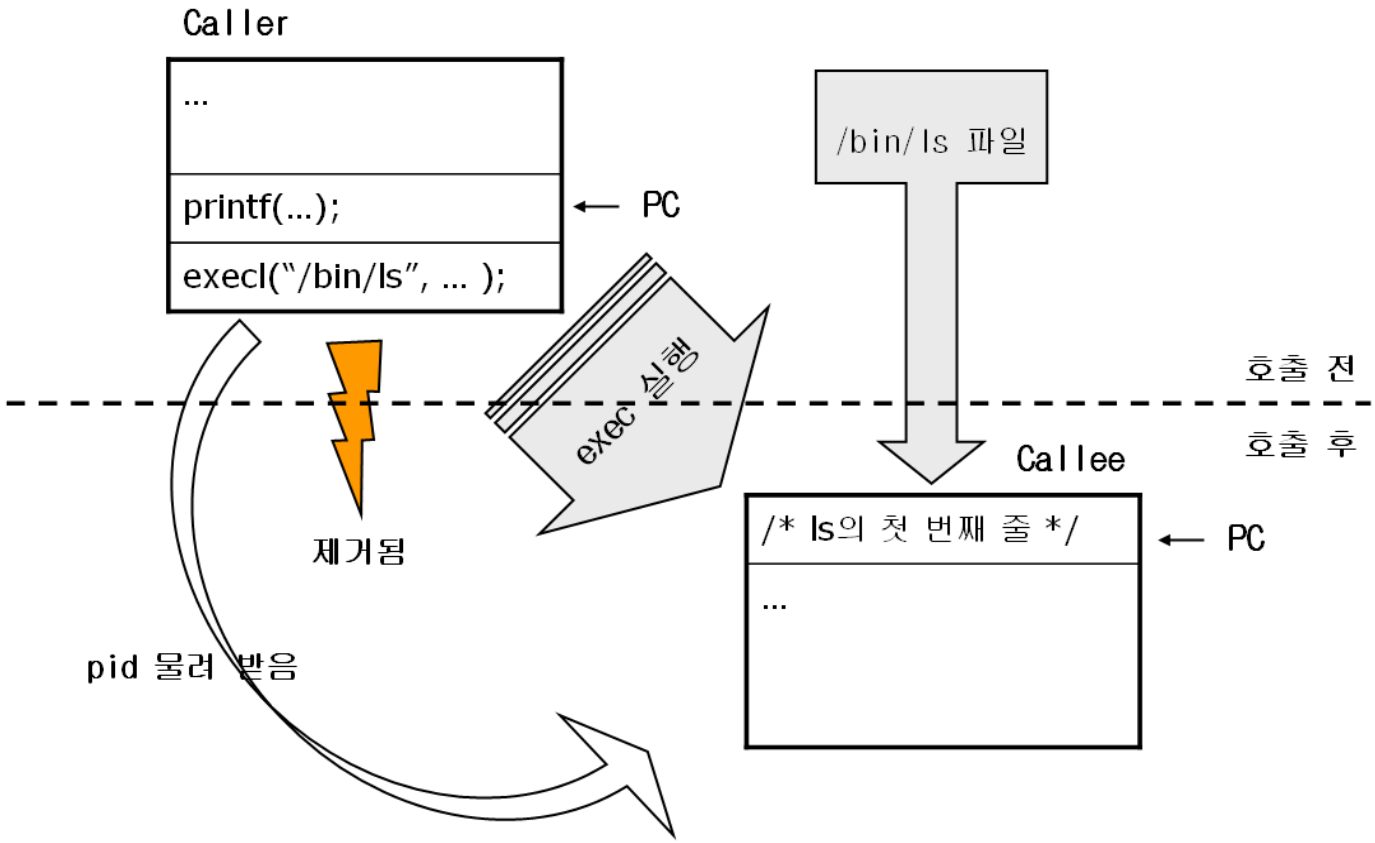
- exec() 계열의 함수는 지정한 실행 파일로부터 프로세스를 생성
  - 비교: fork는 실행 중인 프로세스로부터 새로운 프로세스를 생성
- exec() 계열 함수의 사용 예
  - Shell prompt 상에서 "ls"를 실행하는 것과 비교



- 함수 이름에 **p**가 있고 없고의 차이
  - **p**가 없으면
    - 경로**(path)**로 실행 파일을 지정한다**.**
  - **p**가 있으면
    - 실행 파일의 이름만 지정한다**.**
- 경로를 지정하는 경우 **(p**가 없을 경우**)**
  - 지정한 **(**상대**/**절대**)**경로에서 해당 파일을 찾는다**.**
- 파일의 이름만 지정하는 경우 **(p**가 있을 경우**)**
  - shell 환경 변수 PATH에서 지정한 디렉터리를 차례대로 검색하여 찾는다.
  - Ex: $ printenv PATH ← 환경 변수 $PATH의 값을 출력

- 호출 프로세스와 피호출 프로세스
  - 호출 프로세스 (caller process)
    - exec_() 를 실행하는 프로세스
  - 피호출 프로세스 (callee process)
    - exec_() 에 의해 생성되는 프로세스

- **exec()**를 성공적으로 호출한 결과
  - 호출 프로세스는 종료된다**.(**즉 프로세스 대체 **!)**
  - 호출 프로세스가 메모리 영역을 피호출 프로세스가 차지한다**.**
  - 호출 프로세스의 **PID**를 피호출 프로세스가 물려받는다**.**


Caller
...
printf(...);
execl("/bin/ls", ...);
PC
/bin/ls 파일
exec 실행
호출 전
호출 후
Callee
/* ls의 첫 번째 줄 */
...
PC
제거됨
pid 물려 받음

```
#include <unistd.h>
#include <stdio.h>
int main()
{
        printf("Running ps with execlp\n");
        execlp("ps", "ps", "-ax", 0);
        printf("Done.\n");
        exit(0);
}
```

```c
#include <unistd.h>

main()
{
    printf("before executing ls -l\n");
    execl("/bin/ls", "ls", "-l", (char *)0);
    printf("after executing ls -l\n");
}
```

인자의 나열이 끝났음을 의미한다.

exec 호출이 성공하면 실행되지 않는다. (될 수가 없다.)

```
$ ./a.out
before executing ls -l
-rwxr-xr-x    1 usp      student     13707 Oct 24 21:57
ex07-03
$
```

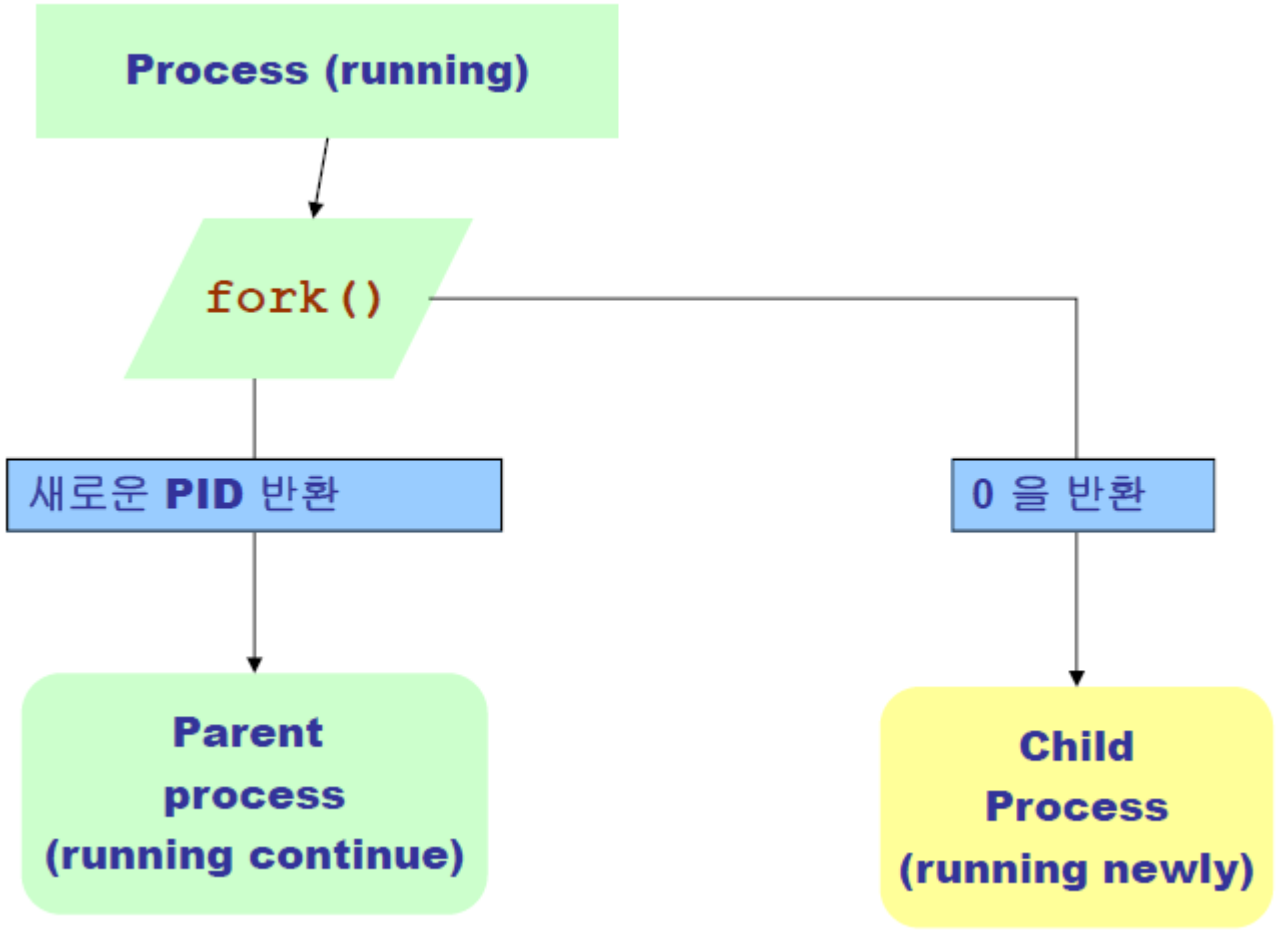
```
#include <unistd.h>

/* example parameters, argv[0] */
Const char *ps_argv[]={“ps”, “-ax”, 0};
Const char *ps_envp[]={“PATH=/bin:/usr/bin”,“TERM=xterm”, 0};

/examples of exec() calls */
execl(“/bin/ps”, “ps”, “-ax”, 0); /* PATH 제공 */
execlp(“ps”, “ps”, “-ax”, 0); /* /bin이 PATH에 존재 가정*/
execv(“/bin/ps”, ps_argv);
execvp(“ps”, ps_argv);
execve(“/bin/ps”, ps_argv, ps_envp);
```

fork()

- fork()
  - 하나 이상의 함수를 수행하는 프로세스 사용 가능
  - 새로운 프로세를 복제
    - parent process <> child process
  - Parent process와 속성을 많이 가지는 새로운 child procss 환경을 가짐
  - Child process는 paraent proces와 같은 코드 실행
  - 자체 데이타공간 / 환경 / file descriptor 가짐
  - Parent proces에서 fork() 후 새로운 pid 반환
    - chile process와 구분하기 위한 방법
  - Child process에는 0을 반환
  - 실패할 경우 -1
  - CHILD_MAX 에 의해 최대 개수 제한

- 프로세스를 생성하고 종료하는 시스템 호출/표준 라이브러리 함수

함수	의미
`fork()`	자신과 완전히 동일한 프로세스를 생성한다.
`wait()`	**Child process** 종료 기다리기
`exit`	종료에 따른 상태 값을 부모 프로세스에게 전달하며 프로세스를 종료한다.
`atexit`	**exit**로 프로세스를 종료할 때 수행할 함수를 등록한다.
`_exit`	**atexit**로 등록한 함수를 호출하지 않고 프로세스를 종료한다.


Process (running)
fork()
새로운 PID 반환
0 을 반환
Parent process (running continue)
Child Process (running newly)

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

pid_t fork(void);
```

- pid_t
  - fork 호출이 성공하여 자식 프로세스가 만들어지면 부모 프로세스에서는 자식 프로세스의 프로세스 ID가 반환되고
  - 자식 프로세스에서는0을 반환
  - fork 호출이 실패하여 자식 프로세스가 만들어지지 않으면 부모 프로세스에서는 -1이 반환

  - 프로세스는 실행 파일로 존재하는 프로그램으로부터 생성되는 것이 일반적이다.
  - 그러나, fork를 사용하면 실행 중인 프로세스를 복제하여 새로운 프로세스를 생성할 수 있다.

- **fork**를 호출하는 프로그램의 구조
  - **fork**를 호출하는 시점을 기준으로, **fork**를 호출한 이후에 부모 프로세스가 할 일과 자식 프로세스가 할 일을 구분한다.
  - **fork**의 반환 값으로 부모 프로세스와 자식 프로세스를 구분한다.

```
pid = fork();    /* fork 호출이 성공하면 자식 프로세스가 생성된다. */

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

```c
#include <unistd.h>
#include <sys/types.h>
int main(){
        pid_t pid;
        int i = 0;
        i++;
        printf("before calling fork(%d)\n",i);
        pid = fork();
        if(pid == 0)
           /* 자식 프로세스가 수행할 부분 */
           printf("child process(%d)\n",++i);
        else if(pid > 0)
           /* 부모 프로세스가 수행할 부분 */
           printf("parent process(%d)\n",--i);
        else
           /* fork 호출이 실패할 경우 수행할 부분 */
           printf("fail to fork\n");
}
```

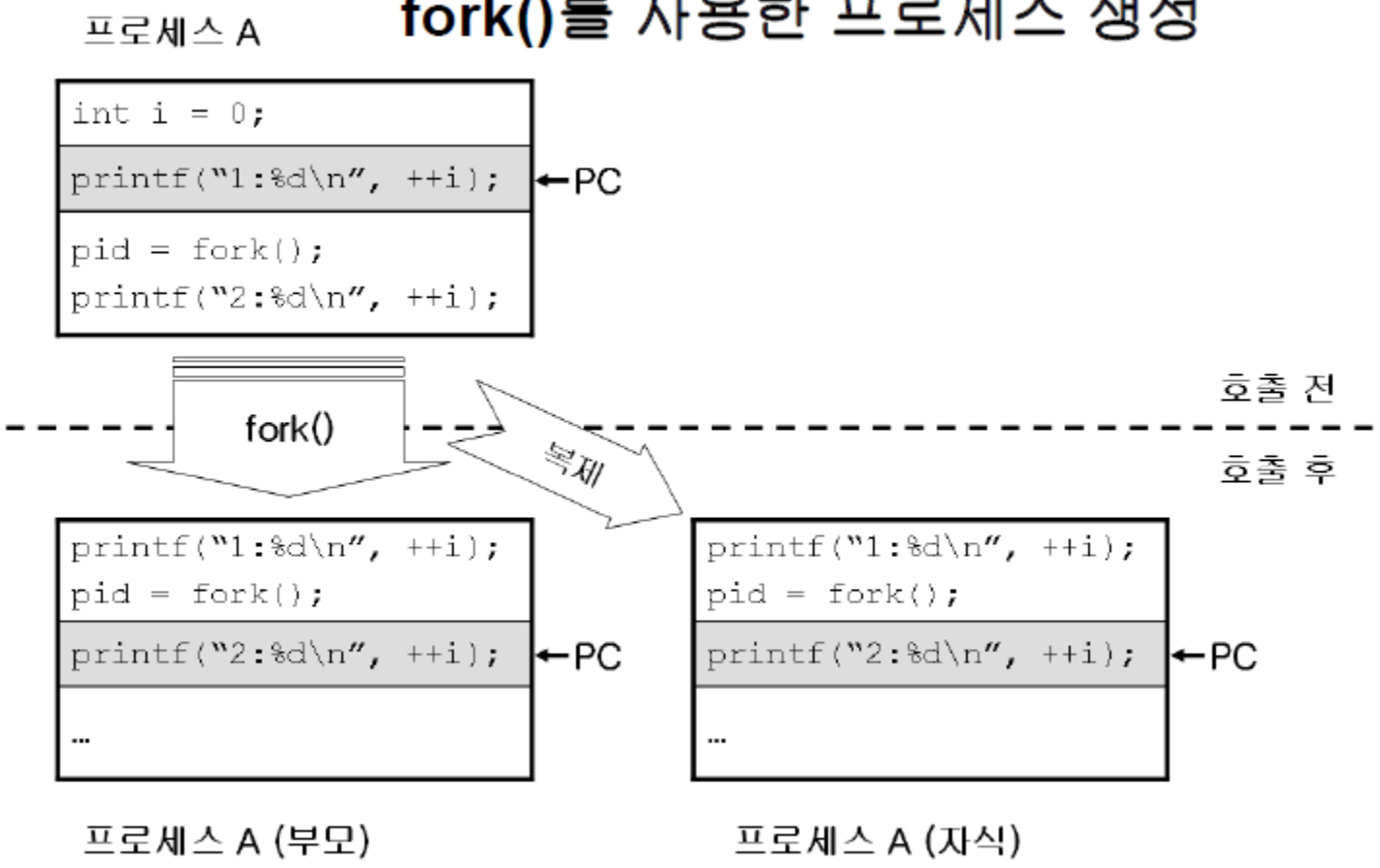
```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
   pid_t pid;
   char *message;
   int n;
   printf("fork program starting\n");
   pid = fork();
     switch(pid){
         case -1:
           perror("fork failed");
           exit(1);
         case 0:
           message = "This is the child";
           n = 5;
           break;
         default:
           message = "This is the parent";
           n = 3;
           break;
         }
   for(; n > 0; n--) {
     puts(message);
     sleep(5);
   }
   exit(0);
}
```

실행결과

```
# ./a.out
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
#
```

- 부모 (parent) 프로세스와 자식 (child) 프로세스
  - fork를 호출하여 새로운 프로세스를 생성할 때, 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.
    - 자식 프로세스의 부모 프로세스 식별 번호 (PPID)는 자신을 생성한 부모 프로세스가 된다.
  - 자식 프로세스는 부모 프로세스가 fork를 호출하던 시점의 상태를 그대로 물려받는다.
    - 프로그램 코드
    - 프로그램 변수에 저장되어 있는 데이터 값
    - 하드웨어 레지스터의 값
    - 프로그램 스택의 값 등등
  - fork 호출 이후에 부모와 자식 프로세스는 자신들의 나머지 프로그램 코드

fork()를 사용한 프로세스 생성



	fork()	exec()
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 메모리상의 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.

- **fork**와 **exec**의 비교
  - **fork**
    - 자신과 동일한 자식 프로세스만 생성할 수 있다.
      - 다른 종류의 프로세스를 생성할 수 없다.
    - 자식 프로세스를 생성하더라도 자신은 종료되지 않는다.
  - **exec**
    - 자신과 다른 종류의 프로세스를 생성할 수 있다.
    - 새로운 프로세스를 생성하면 자신은 종료된다.
- **fork**와 **exec**를 함께 사용하기
  - **fork**를 호출하여 자식 프로세스를 생성 한 후에 자식 프로세스가 **exec**를 호출하여 새로운 프로세스를 생성한다.
  - 결과적으로, 부모 프로세스는 종류가 다른 자식 프로세스를 생성하고 자신 역시 나머지 작업을 계속할 수 있다.

```
#include <unistd.h>
#include <sys/types.h>
int main(){
        pid_t pid;
        printf("hello!\n");
        pid = fork();
        if(pid > 0) { /* parent process */
          printf("parent\n");
          sleep(1);
        }
        else if(pid == 0) { /* child process */
          printf("child\n");
          execl("/bin/ls", "ls", "-l", (char *)0);
          printf("fail to execute ls\n");
        }
        else
          printf("parent : fail to fork\n");
          printf("bye!\n");
}
```

exit()

- 프로세스를 종료하면서 부모프로세스에게 종료와 관련된 값은 반환

```
#include <stdlib.h>

void exit(int status);
```

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

- exit()는 프로세스를 의도적으로 종료 시킴
- 이외 프로세스가 종료하는 경우는
  - 더 이상 수행할 문장이 없거나
  - main() 함수내에서 return()문을 수행할 때
- status 값은 0 ~ 255 사이 값
  - 프로그램 작성자가 임의로 그 의미를 부여해서 사용

atexit()

- 프로세스가 exit()를 호출해서 종료할 때 수행해야 할 작업을 등록하는 함수

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

function	**atexit**로 등록할 함수의 이름이다.
반환값	호출이 성공하면 **0**을 반환하고, 실패하면 **0**이 아닌 값을 반환한다.

- function()
  - 함수의 이름으로 함수는 void function(void) 형태
  - 종료시 마무리해야 할 작업 (clean-up-action)
    - 프로세스가 종료 될 때 임시파일 같은 것을 삭제 한다던지 마무리해야 할 일들을 미리 함수내에 정의
    - 최대 32개 함수를 등록 가능
    - 실행 순서는 등록된 순서

```
#include <stdlib.h>
#include <unistd.h>
#include <stdlib.h>
void func1(void);
void func2(void);
main(){
        printf(“atexit() Test…!\n");
        atexit(func1);
        atexit(func2);
        printf("bye!\n");
        exit(0);
}
void func1(void){
        printf(“This is func1 called by main()\n”);
}
void func2(void){
        printf(“This is func2 called by main()\n”);
}
```

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
void cleanupaction(void);

main(){
        pid_t pid;
        int i;
        for(i = 0; i < 3; i++){
           printf("before fork [%d]\n", i);
           sleep(1);
        }
        pid = fork();
        if(pid > 0) {
           for( ; i < 7; i++) {
           printf("parent [%d]\n", i);
           sleep(1);
        }
```

```
            atexit(cleanupaction);
        }
        else if(pid == 0) {
            for( ; i < 5; i++) {
              printf("child [%d]\n", i);
              sleep(1);
              execl("/bin/ls", "ls", "-l", (char *)0);
            }
        }
        else {
            printf("fail to fork child process\n");
        }
        exit(0);
} /* end of main */
void cleanupaction(void){
        printf("clean-up-action…\n");
}
```

<실행 결과의 예>

```
$ ./a.out
before fork [0]
before fork [1]
before fork [2]
parent [3]
child [3]
parent [4]
-rwxr-xr-x 1 usp student 14456 Oct 26 00:31 ex07-
01
parent [5]
parent [6]
clean-up-action
$
```

_exit()

- exit()와 같지만 atexit()로 등록한 함수를 수행하지 않는다

```
#include <unistd.h>

void _exit(int status);
```

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

```
#include <stdlib.h>
#include <unistd.h>
#include <stdlib.h>
void func1(void);
void func2(void);
main(){
        printf(“atexit() Test…!\n");
        atexit(func1);
        atexit(func2);
        printf("bye!\n");
        _exit(0);
}
void func1(void){
        printf(“This is func1 called by main()\n”);
}
void func2(void){
        printf(“This is func2 called by main()\n”);
}
```

wait()

- 자식 프로세스 종료 대기 : 동기화

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

pid_t wait(int *stat_loc);
```

pid_t	Child process 로 부터 반환 되는 PID
****stat_loc***	Child process의 종료시 전달되는 값(main() or exit()) sys/wait.h 참조

- Child process가 끝날 때 까지 parent proces가 기다림
- Parent process vs child process
- running time, racing condition
- child process가 종료될 때 까지 parent process는 대기
- return value: child process의 pid
- return status: *stat_loc -> sys/wait.h 에 상태정보에 대한 return value가 있음

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main(){
          pid_t pid;
          char *message;
          int n;
          int exit_code;
          printf("fork program starting\n");
          pid = fork();
             switch(pid){
                    case -1:
                    exit(1);
             case 0:
                    message = "This is the child";
                    n = 5;
                    exit_code = 37;
                    break;
```

```
    default:
      message = "This is the parent";
      n = 3;
      exit_code = 0;
      break;
  }
  for(; n > 0; n--) {
    puts(message);
    sleep(1);
  }
```

```
/* 아래 코드에서 parent process는 child
process가 종료될 때 까지 기다린다 */
          if(pid) {
             int stat_val;
             pid_t child_pid;
             child_pid = wait(&stat_val);
            printf("Child has finished: PID
               = %d\n", child_pid);
            if(WIFEXITED(stat_val))
             printf("Child exited with
               code %d\n",
               WEXITSTATUS(stat_val));
            else
              printf("Child terminated
                abnormally\n");
          }
          exit (exit_code);
}
```

waitpid()

- 특정 자식프로세스의 종료를 대기 : 동기화

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, init *stat_loc, int options);
```

pid_t	Child process 로 부터 반환 되는 PID
****stat_loc***	Child process의 종료시 전달되는 값(main() or exit()) sys/wait.h 참조

- 자식프로세스가 끝날 때 까지 부모프로세스가 대기
  - 특정 자식프로세스 PID를 대기
  - 다수 프로세스 상황에서 대기하고 후속 작업시 유용
  - options
    - WEXITED : 자식 프로세스가 종료될 때까지 기다림
    - WSTOPPED : 시그널을 받아 중단된 자식 프로세스를 기다림
    - WCONTINUED : 시그널을 받아 다시 수행 중인 자식 프로세스를 기다림
    - WNOHANG : waitpid()에서와 같음
    - WNOWAIT : 상탯값을 리턴한 프로세스가 대기 상태로 머물 수 있도록 함

Signal

- 시그널의 개요
  - 소프트웨어 인터럽트로 프로세스에 뭔가 발생했음을 알리는 간단한 메시지를 비동기적으로 보내는 것
  - 0으로 나누기처럼 프로그램에 예외적인 상황이 일어나는 경우나 프로세스가 함수를 사용해 다른 프로세스에 시그널을 보내는 경우에 발생
  - 보통 시그널로 전달되는 메시지는 무엇이 발생했는지를 표시하는 미리 정의된 상수를 사용
  - 시그널을 받은 프로세스는 시그널에 따른 기본 동작을 수행하거나, 시그널을 무시하거나, 시그널 처리를 위해 특별히 지정된 함수를 수행
- 발생 원인
  - 시그널은 소프트웨어 인터럽트
  - 시그널은 비동기적으로 발생하며, 리눅스 운영체제가 프로세스에 전달
  - 시그널은 다음과 같은 세 가지 경우에 발생
    - 0으로 나누기처럼 프로그램에서 예외적인 상황이 일어나는 경우
    - 프로세스가 kill() 함수와 같이 시그널을 보낼 수 있는 함수를 사용해 다른 프로세스에 시그널을 보내는 경우
    - 사용자가 Ctrl + C 같은 인터럽트 키를 입력한 경우

- 시그널 처리 방법
  - 기본 동작 수행
    - 대부분 시그널의 기본 동 작은 프로세스를 종료하는 것
    - 이 외에 시그널을 무시하거나 프로세스 수행 일시 중지/재시작 등을 기본 동작으로 수행
  - 시그널 무시
    - 프로세스가 시그널을 무시하기로 지정하면 시스템은 프로세스에 시그널을 전달하지 않음
  - 지정된 함수 호출
    - 프로세스는 시그널의 처리를 위해 미리 함수를 지정해놓고 시그널을 받으면 해당 함수를 호출해 처리
    - 시그널 핸들러: 시그널 처리를 위해 지정하는 함수
    - 시그널을 받으면 기존 처리 작업을 중지한 후 시그널 핸들러를 호출
    - 시그널 핸들러의 동작이 완료되면 기존 처리 작업을 계속 수행

- 시그널 종류 : signal.h

표 8-7 현재 정의된 시그널

시그널	번호	기본 처리	발생 요건
SIGHUP	1	종료	행업으로 터미널과 연결이 끊어졌을 때 발생
SIGINT	2	종료	인터럽트로 사용자가 Ctrl + C 를 입력하면 발생
SIGQUIT	3	코어 덤프	종료 신호로 사용자가 Ctrl + \ 를 입력하면 발생
SIGILL	4	코어 덤프	잘못된 명령 사용
SIGTRAP	5	코어 덤프	추적(trace)이나 브레이크 지점(break point)에서 트랩 발생
SIGABRT	6	코어 덤프	abort() 함수에 의해 발생
SIGIOT	6	코어 덤프	SIGABRT와 동일
SIGBUS	7	코어 덤프	버스 오류로 발생
SIGFPE	8	코어 덤프	산술 연산 오류로 발생
SIGKILL	9	종료	강제 종료로 발생
SIGUSR1	10	종료	사용자가 정의해 사용하는 시그널 1
SIGSEGV	11	코어 덤프	세그먼테이션 오류로 발생
SIGUSR2	12	종료	사용자가 정의해 사용하는 시그널 2
SIGPIPE	13	종료	잘못된 파이프 처리로 발생
SIGALRM	14	종료	alarm() 함수에 의해 발생
SIGTERM	15	종료	소프트웨어 종료로 발생
SIGSTKFLT	16	종료	보조 프로세서의 스택 오류로 발생(리눅스에서는 지원 안 함)

시그널	번호	기본 처리	발생 요건
SIGCHLD	17	무시	자식 프로세스의 상태가 바뀌었을 때 발생
SIGCONT	18	무시	중지된 프로세스를 재시작할 때 발생
SIGSTOP	19	중지	중지(stop) 시그널로, 이 시그널을 받으면 SIGCONT 시그널을 받을 때까지 프로세스 수행 중단
SIGTSTP	20	중지	사용자가 Ctrl + z 로 중지시킬 때 발생
SIGTTIN	21	중지	터미널 입력을 기다리기 위해 중지시킬 때 발생
SIGTTOU	22	중지	터미널 출력을 위해 중지시킬 때 발생
SIGURG	23	무시	소켓에 긴급한 상황이 생기면 발생
SIGXCPU	24	코어 덤프	CPU 시간 제한을 초과할 때 발생
SIGXFSZ	25	코어 덤프	파일 크기 제한을 초과할 때 발생
SIGVTALRM	26	종료	가상 타이머가 종료할 때 발생
SIGPROF	27	종료	프로파일 타이머가 종료할 때 발생
SIGWINCH	28	무시	윈도우 크기가 바뀌었을 때 발생
SIGIO	29	종료	비동기식 입출력 이벤트로 발생
SIGPOLL	SIGIO	종료	SIGIO와 동일
SIGPWR	30	무시	전원이 중단되거나 재시작할 때 발생
SIGSYS	31	코어 덤프	잘못된 시스템 호출로 발생
SIGUNUSED	31	무시	향후 사용을 위해 예약된 번호

- 시그널 보내기 : kill()

```
#include <sys/types.h>                                    [함수 원형]
#include <signal.h>

int kill(pid_t pid, int sig);
```

  - pid_t pid : 시그널을 받을 프로세스의 PID
  - int sig : pid로 지정한 프로세스에 보내는 시그널

- pid에 대응하는 프로세스에 sig로 지정한 시그널을 정송
- pid는 특정 프로세스 또는 프로세스 그룹을 의미한다.
- sig에 0(널 시그널)을 지정하면 실제로 시그널을 보내지 않고 오류를 확인, 예를 들면, pid가 정상인지 검사
- pid에 지정한 값에 따라 시그널을 어떻게 보낼 것인지를 결정

- 시그널 보내기 kill()

```
01 #include <sys/types.h>
02 #include <unistd.h>
03 #include <signal.h>
04 #include <stdio.h>
05
06 int main() {
07     printf("Before SIGCONT Signal to parent.₩n");
08     kill(getppid(), SIGCONT);
09
10     printf("Before SIGQUIT Signal to me.₩n");
11     kill(getpid(), SIGQUIT);
12
13     printf("After SIGQUIT Signal.₩n");
14 }
```

- 08행, 11행 08행에서 부모 프로세스에 SIGCONT 시그널을 전송하고, 11행에서 자신에게 SIGQUIT 시그널을 전송
- [표 8-7]에 따르면 SIGCONT 시그널의 기본 처리는 무시이므로 특별한 처리를 하지 않음
- 하지만 SIGQUIT 시그널은 코어 덤프를 발생시키며 프로세스를 종료
- 따라서 11행을 수행한 후 종료되므로 13행은 실행되지 않음

- 실행 결과 07행과 10행은 출력되었지만 13행은 출력되지 않고 코어 덤프를 생성한 뒤 종료했음을 알 수 있음

- 시그널 핸들러
  - 프로세스를 종료하기 전에 처리할 작업이 남아 있는 경우, 특정 시그널은 종료하지 않고자 하는 경우 받은 시그널을 처리하는 함수를 지정하는 것

기능	함수
시그널 핸들러 지정	sighandler_t signal(int signum, sighandler_t handler);
	sighandler_t sigset(int sig, sighandler_t disp);

  - handler 설정
    - **시그널 핸들러 주소**
    - **SIG_IGN** : 시그널을 무시하도록 지정
    - **SIG_DFL** : 시그널의 기본 처리 방법을 수행하도록 지정
  - 시스템별 signal() 동작
    - 시스템 V : 시그널을 처리한 후 시그널 처리 방법을 기본 처리 방법(SIG_DFL)으로 재설정, 따라서 시그널 처리를 계속하려면 signal() 함수를 호출해 시그널을 처리한 후 다시 signal() 함수를 설정해야 함
    - BSD : 시그널을 처리한 후 시그널 처리 방법을 기본 처리 방법(SIG_DFL)으로 재설정하지 않음, 따라서 시그널 핸들러가 계속 동작
    - 리눅스 : 커널의 signal(2) 시스템 콜은 시스템 V와 같은 방식으로 동작, 그러나 gcc의 glibc 2부터 signal(3) 함수는 호출하지 않고 sigaction(2)를 호출해 BSD 형식으로 동작

```
01 #include <unistd.h>
02 #include <signal.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 void sig_handler(int signo) {
07     printf("Signal Handler signum: %d₩n", signo);
08     psignal(signo, "Received Signal");
09 }
10
11 int main() {
12     void (*hand)(int);
13
14     hand = signal(SIGINT, sig_handler);
15     if (hand == SIG_ERR) {
16         perror("signal");
17         exit(1);
18     }
19
20     printf("Wait 1st Ctrl+C... : SIGINT₩n");
21     pause();
22     printf("After 1st Signal Handler₩n");
23     printf("Wait 2nd Ctrl+C... : SIGINT₩n");
24     pause();
25     printf("After 2nd Signal Handler₩n");
26 }
```

실행

```
$ ch8_2.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler signum: 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^CSignal Handler signum: 2
Received Signal: Interrupt
After 2nd Signal Handler
```

- **06~09행** 시그널 핸들러로 sig_handler() 함수를 정의
  psignal() 함수는 시그널 번호를 이름으로 바꿔서 출력하는 함수로 8절에서 다룸
- **12행** signal() 함수의 리턴값인 함수 포인터를 저장할 변수를 선언
- **14행** signal() 함수로 SIGINT 시그널의 시그널 핸들러를 지정
- **15~18행** signal() 함수가 오류를 리턴하면 처리
- **21행** pause() 함수는 시그널이 입력될 때까지 기다리는 함수로 8절에서 다룸
  여기서는 사용자가 Ctrl + C 를 입력하기를 기다림
- **22행** 21행에서 호출된 pause() 함수 때문에 대기하고 있다가 사용자가 Ctrl + C 를 입력하면 sig_handler() 함수가 호출되어 수행한 후 22행으로 복귀
- **24행** pause() 함수를 호출해 사용자가 다시 Ctrl + C 를 입력하기를 기다림
- **실행 결과** 20행을 출력한 후 대기하다가 Ctrl + C 를 입력받고 sig_handler() 함수의 07~08행을 출력한 후 복귀해 22~23행을 출력
  다시 Ctrl + C 를 입력받으면 sig_handler() 함수를 호출해 처리한 후 25행을 출력

- 시그널 핸들러 지정 : sigset()

```
#include <signal.h>                                    [함수 원형]

sighandler_t sigset(int sig, sighandler_t disp);
```

  - sig : 시그널 핸들러로 처리하려는 시그널
  - disp : 시그널 핸들러의 함수명

– sigset() 함수의 특징
  - sigset() 함수의 인자 구조는 signal() 함수와 동일
  - sigset() 함수도 첫 번째 인자인 sig 에 SIGKILL과 SIGSTOP 시그널을 제외한 어떤 시그널이든 지정할 수 있음
  - 두 번째 인자인 disp에도 signal() 함수처럼 시그널 핸들러 함수의 주소나 SIG_IGN, SIG_DFL 중 하나를 지정해야 함
  - 리턴값은 시그널 핸들러 함수의 주소
  - sigset() 함수가 실패하면 SIG_ERR을 리턴
  - 리눅스에서는 sigset() 함수를 제공하지만 사용을 권하지는 않음

```c
01 #include <unistd.h>
02 #include <signal.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 void sig_handler(int signo) {
07     printf("Signal Handler Signal Number : %d₩n", signo);
08     psignal(signo, "Received Signal");
09 }
10
11 int main() {
12     void (*hand)(int);
13
14     hand = sigset(SIGINT, sig_handler);
15     if (hand == SIG_ERR) {
16         perror("signal");
17         exit(1);
18     }
19
20     printf("Wait 1st Ctrl+C... : SIGINT₩n");
21     pause();
22     printf("After 1st Signal Handler₩n");
23     printf("Wait 2nd Ctrl+C... : SIGINT₩n");
24     pause();
25     printf("After 2nd Signal Handler₩n");
26 }
```

실행

```
$ ./ch8_4.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler Signal Number : 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^CSignal Handler Signal Number : 2
Received Signal: Interrupt
After 2nd Signal Handler
```

- **14행** sigset() 함수를 사용해 시그널 핸들러를 지정
- **실행 결과** 시그널 핸들러가 한 번 호출된 후에 재지정하지 않아도 됨을 알 수 있음

알람 시그널

- 알람 시그널
  - 일정 시간이 지난 후에 자동으로 시그널이 발생하도록 함
  - 일정 시간 후에 한 번 발생시킬 수도 있고, 일정한 시간 간격을 두고 주기적으로 알람 시그널을 발생시킬 수도 있음

- alarm()
  - 일정 시간이 지난 후에 자동으로 시그널이 발생하도록 함
  - 일정 시간 후에 한 번 발생시킬 수도 있고, 일정한 시간 간격을 두고 주기적으로 알람 시그널을 발생시킬 수도 있음

```
#include <unistd.h>                                    [함수 원형]

unsigned int alarm(unsigned int seconds);
```

    - seconds : 알람을 발생시킬 때까지 남은 시간(초 단위)

  - alarm() 함수의 특징
    - 인자로 초 단위 시간을 받음
    - 인자로 지정한 시간이 지나면 SIGALRM 시그널이 생성되어 프로세스에 전달
    - 프로세스별로 알람시계가 하나밖에 없으므로 알람은 하나만 설정할 수 있음
    - 따라서 알람 시그널을 생성하기 전에 다시 alarm() 함수를 호출하면 이전 설정은 없어지고 재설정
    - 인자로 0을 지정하면 이전에 설정한 알람 요청은 모두 취소
    - alarm() 함수는 이전에 호출한 alarm() 함수의 시간이 남아 있으면 해당 시간을, 그렇지 않으면 0을 리턴

```
01 #include <unistd.h>
02 #include <signal.h>
03 #include <stdio.h>
04
05 void sig_handler(int signo) {
06     psignal(signo, "Received Signal");
07 }
08
09 int main() {
10     signal(SIGALRM, sig_handler);
11
12     alarm(2);
13     printf("Wait...₩n");
14     sleep(3);
15 }
```

실행

```
$ ch8_9.out
Wait...
Received Signal: Alarm Clock
```

- **10행** SIGALRM 시그널 처리를 위한 시그널 핸들러를 지정
- **12행** alarm() 함수의 인자로 2초를 지정해 함수를 호출
- **14행** 3초 동안 sleep하고 있는 중에 2초가 지나면 SIGALRM 시그널이 발생해 시그널 핸들러가 호출
- **실행 결과** 프로그램이 종료되기 전에 시그널 핸들러를 호출한 사실을 알 수 있음

Zombie Process

- fork()한 child process의 종료를 parent process는 무작 정기다린다
  - parent process의 process table은 child process가 정 상종료, return 되지 않으면 삭제되지 않음
  - 비정상적으로 프로세스가 계속 동작되는 경우 발생
  - 표시: zombie or defunct
  - init process에 의해 정리
  - system resource를 고갈시킬 위험
  - 강제 정리: by root
    - # kill -9 PIDs

Question?