Part 02. 리눅스 시스템 프로그래밍

# Chapter 07. 프로세스

## 진행 순서

# Chapter 07_01 프로세스 개요

**프로세스란?**
프로세스는 실행 중인 프로그램(일, task)을 의미한다.
프로세스는 메모리에 로딩중인 프로그램 이미지와 가상 메모리 인스턴스, 열린 파일 디스크립터 같은 커널 리소스,
사용자 정보, 하나 이상의 스레드 등을 포함하고 있다.

**프로세스 ID**
모든 프로세스는 프로세스 ID(pid) 라고 하는 유일한 식별자로 구분된다. (특정 시점에 unique)

**부모/자식 프로세스**
새로운 프로세스를 생성하는 프로세스를 부모 프로세스라고 하고, 새롭게 생성된 프로세스를 자식 프로세스라고 한다.

리눅스 운영체제는 부팅 시 0번 프로세스인 init 프로세스를 수행시키고
부팅 프로세스에 따라 init 프로세스가 다른 필요한 프로세스들을 생성한다. (ex. bash)

bash 프로세스의 부모 프로세스는 init(0) 프로세스가 된다.
(최신 CentOS의 경우 init(0)이 systemd 프로세스를 1번으로 생성하고
systemd 프로세스가 대부분의 어플리케이션 레이어의 프로세스 들을 실행한다.)

쉘(bash)에서 실행한 특정 프로그램(실습 바이너리 등)의 부모 프로세스는 bash 프로세스가 된다.

**# ps –ef**
현재 수행중인 프로세스들의 목록을 확인할 수 있으며, PID(프로세스 ID), PPID(부모 프로세스 ID) 등을 확인할 수 있다.

# Chapter 07_01 프로세스 개요

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

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

getpid() 시스템 콜은 호출한 프로세스의 pid를 리턴한다.
getppid() 시스템 콜은 호출한 프로세스의 부모 프로세스 pid를 리턴한다.

pid_t 자료형은 <sys/types.h> 헤더 파일에 정의되어 있으며,
리눅스에서 pid_t는 보통 C의 int 자료형에 대한 typedef 이다.

# Chapter 07_02 프로세스 생성

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

pid_t fork (void);
```

fork() 시스템 콜은 현재 실행 중인 프로세스와 동일한 프로세스(자식 프로세스)를 새롭게 생성한다.
현재 실행 중인 프로세스(부모 프로세스)와 새롭게 생성된 프로세스(자식 프로세스)는 계속 실행된다.

fork()가 정상적으로 수행될 경우
자식 프로세스에서 리턴값은 0 이며, 부모 프로세스에서 리턴값은 자식 프로세스의 pid가 된다.
실패할 경우 -1을 리턴하고 errno을 적절한 값으로 설정한다.

EAGAIN : 리소스 할당 실패
ENOMEM : 커널 메모리 부족

```
pid_t pid;
pid = fork();
if (pid > 0) { /* parent process */
        printf("Parent process – child's PID(%d)\n", pid)
} else if (pid == 0) { /* child process */
        printf("Child process\n");
} else if (pid == -1) { /* error */
        perror("fork error: ");
}
```

# Chapter 07_03 프로세스 실행

fork()는 새로운 자식 프로세스를 생성하고, 이렇게 생성된 프로세스에 새로운 바이너리를 적재하여 실행하는 과정이 필요한다. 이 기능은 exec 계열의 시스템 콜에서 제공한다.

```
#include <unistd.h>

int execl (const char *path, const char *arg, ...);
```

execl() 시스템 콜을 호출하면 현재 프로세스를 path가 가리키는 프로그램으로 대체한다.
arg 인자는 가변인자로 일반적으로 main() 의 인자로 전달되는 argv를 의미한다.
예를들어 “/home/bin/test_program”을 수행하는데 인자로 “test.file” 인자가 필요한 경우
# /home/bin/test_program test.file
상기와 같이 실행된다고 할 때
execl(“/home/bin/test_program”, “test_program”, “test_file”, NULL);
상기와 같이 execl을 통해 현재 프로세스를 test_program 으로 대체할 수 있다.
(인자의 끝을 나타내기 위한 NULL 주의)

```
int ret;

ret = execl(“/home/bin/test_program”, “test_program”, “test_file”, NULL);
if (ret == -1)
        perror(“execl error: “);
```

# Chapter 07_03 프로세스 실행

```
#include <unistd.h>

int execl (const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execle (const char *path, const char *arg, ..., char * const envp[]);

int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
```

l – 인자를 리스트로 제공
v – 인자를 벡터로 제공

p – file 인자값을 사용자의 실행 경로 환경변수에서 찾게 됨
e – 새롭게 생성될 프로세스를 위한 새로운 환경 제공

```
const char *args[] = { “test_program”, “test.file”, NULL };
int ret;

ret = execv(“/home/bin/test_program”, args);
if (ret == -1)
                perror(“execv error: “);
```

# Chapter 07_03 프로세스 실행

```
#define _XOPEN_SOURCE /* WEXITSTATUS 등을 사용할 경우 */
#include <stdlib.h>

int system (const char *command);
```

ANSI C와 POSIX는 새로운 프로세스를 생성하고 실행한 다음 종료를 기다리는 일련의 동작을 하나로 묶은 인터페이스를 정의하고 있다.
프로그램 코드 내부에서 새로운 프로그램(쉘 스크립트나 유틸리티 등)을 실행할 때 유용하다.

sysmtem() 호출은 command 인자로 주어진 명령을 실행한다.
command 인자는 /bin/sh –c <command> 와 같이 쉘에 바로 전달되어 실행된다.

호출이 성공하면 해당 명령의 상태를 리턴한다.
실행한 명령의 종료 코드는 WEXITSTATUS로 얻을 수 있다. (다음 프로세스 종료 wait 참고)

```
int ret;
ret = system(“touch empty.file”);
if (ret == -1) {
        printf(“system error”);
} else {
        printf(“system result: %d\n”, WEXITSTATUS(ret));
}
```

# Chapter 07_04 프로세스 종료

```
#include <stdlib.h>

void exit (int status);
```

POSIX와 C89 표준은 현재 프로세스를 종료하는 표준 함수를 정의하고 있다.
status 인자는 프로세스의 종료 상태를 나타내기 위한 값으로 쉘 같은 다른 프로그램에서 확인할 수 있다.

EXIT_SUCCESS – 성공
EXIT_FAILURE – 실패
리눅스에서 일반적으로 0이 성공을 나타내고, 1이나 -1처럼 0이 아닌 값을 실패로 간주한다.

정상 종료
```
exit(EXIT_SUCCESS);
```

일반적으로 main() 함수가 프로그램 끝까지 진행되어 종료되는 경우 명시적으로 exit() 코드가 존재하지 않아도 컴파일러가 묵시적으로 종료 코드 이후에 exit() 시스템 콜을 추가한다.

# Chapter 07_04 프로세스 종료

```
#include <stdlib.h>

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

atexit()는 프로그램이 정상적으로 exit()에 도달해서 종료될 때 호출할 함수를 등록한다.
exec 계열 함수를 호출하면 등록된 함수 목록을 비우므로 새로운 프로세스 주소 공간에는 존재하지 않는다.
그리고 시그널에 의해서 프로세스가 종료되는 경우 등록된 함수는 호출되지 않는다.

등록이 성공하면 0을 리턴하고, 에러 발생 시 -1을 리턴한다.

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

void final(void) {
        printf("atexit() succeeded!\n");
}

int main(void) {
        if (atexit(final))
                fprintf(stderr, "atexit() failed!\n");
        return 0;
}
```

# Chapter 07_05 자식 프로세스 종료

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

pid_t wait (int *status);
```

wait()을 이용하면 종료된 자식 프로세스의 정보를 얻을 수 있다.
wait()을 호출하면 종료된 프로세스의 pid를 리턴하며, 에러가 발생한 경우 -1을 리턴한다.

ECHILD – 자식 프로세스가 없다.
EINTR – 시그널 수신

status가 NULL이 아니라면 자식 프로세스에 대한 추가 정보가 status에 저장된다.

# Chapter 07_05 자식 프로세스 종료

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

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONFTINUED (status);

int WEXITSTATUS (status);
int WTERMSIG (status);
int WCOREDUMP (status);
```

status 정보를 해석하기 위한 여러 가지 매크로를 제공한다.

WIFEXITED - _exit()를 호출하여 정상적으로 종료된 경우 참
WIFSIGNALED – 시그널에 의해서 종료된 경우 참
WIFSTOPPED – 프로세스가 멈춘 경우 참 (ptrace 추적 중)
WIFCONFTINUED – 프로세스가 다시 시작된 경우 참 (ptrace 추적 중)

WEXITSTATUS - _exit()에 넘긴 값(하위 8비트)
WTERMSIG – 프로세스를 종료시킨 시그널 번호 리턴
WCOREDUMP – 코어 덤프 파일을 생성했을 경우 참

# Chapter 07_05 자식 프로세스 종료

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

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

    if (fork() == 0) /* child process */
        return 1;

    pid = wait(&status);
    if (pid == -1)
        perror("wait error: ");

    printf("pid = %d\n", pid);

    if (WIFEXITED(status)) {
        printf("child process terminated with exit status (%d)\n",
                WEXITSTATUS(status));
    }

    if (WIFSIGNALED(status)) {
        printf("child process killed by signal (%d)\n", WTERMSIG(status));
        if (WCOREDUMP(status)) { /* signal received SIGSEGV, etc */
            printf("child process dumped core\n");
        }
    }

    if (WIFSTOPPED(status))
        printf("child process stopped by signal (%d)\n", WSTOPSIG(status));

    if (WIFCONTINUED(status))
        printf("child process continued\n");

    return 0;
}
```

```
[root@localhost ch07]# gcc -g wait_example.c -o wait_example
[root@localhost ch07]# ./wait_example
pid = 1497
child process terminated with exit status (1)
```

# Chapter 07_05 자식 프로세스 종료

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

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

자식 프로세스가 여러개일 경우 그 중 원하는 특정한 자식 프로세스를 기다릴 수 있다.
pid 인자는 기다리기 원하는 자식 프로세스를 지정하는데 쓰인다.

**pid**

| | |
|---|---|
| < -1 | 프로세스 gid가 동일한 자식 프로세스를 기다린다. (ex. -500은 gid가 500인 프로세스) |
| -1 | 모든 자식 프로세스를 기다린다. |
| 0 | 동일한 프로세스 그룹에 속한 자식 프로세스를 기다린다. |
| > 0 | pid와 일치한 자식 프로세스를 기다린다. (ex. 500은 pid가 500인 프로세스) |

**options**

| | |
|---|---|
| WNOHANG | 종료된 자식 프로세스가 없다면 기다리지 않고 바로 리턴 |
| WUNTRACED | 반환되는 status 인자에 WIFSTOPPED 비트 설정 |
| WCONTINUED | 반환되는 status 인자에 WIFCONTINUED 비트 설정 |

# Chapter 07_06 백그라운드 실행

```
#include <unistd.h>

int daemon(int nochdir, int noclose);
```

현재 프로세스를 백그라운드로 수행할 수 있다.
nochdir 인자가 0이 아니면 현재 작업 디렉토리를 루트 디렉토리로 변경하지 않는다.
noclose 인자가 0이 아니면 열려 있는 모든 파일 디스크립터를 닫지 않는다.
일반적으로 모두 0으로 설정
성공하면 0 리턴, 실패할 경우 -1 리턴 (errno)

# Chapter 07_06 백그라운드 실행

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

int main()
{
        printf("daemon start\n");

        if (daemon(0, 0) == -1) {
                perror("daemon erro: ");
                return -1;
        }

        while (1) {
                printf("not print line!\n");
                sleep(1);
        }

        return 0;
}
```

```
[root@localhost ch07]# gcc -g daemon_example.c -o daemon_example

[root@localhost ch07]# ./daemon_example
daemon start

[root@localhost ch07]# ps -ef | grep daemon_example
root      1617     1  0 09:46 ?        00:00:00 ./daemon_example

[root@localhost ch07]# killall daemon_example
```