

제10장 프로세스 프로그래밍

- 프로세스 시작/종료
- 자식 프로세스 생성
- 프로세스 내에서 새로운 프로그램 실행
- 시그널
- 시스템 부팅

10.1 프로그램 시작 및 종료

Unix에서 프로그램은 어떻게 실행이 시작되고 종료될까?

프로그램은 10.3 절에서 살펴볼 `exec` 시스템 호출에 의해 실행된다. 이 호출은 실행될 프로그램의 시작 루틴에게 명령줄 인수(command-line arguments)와 환경 변수(environment variables)를 전달한다. C 프로그램을 컴파일 하면 실행 파일에는 C 프로그램의 코드와 더불어 C 시작 루틴(start-up routine)이 포함된다. 이 시작 루틴은 `exec` 시스템 호출로부터 전달받은 명령줄 인수, 환경 변수를 다음과 같이 `main` 함수를 호출하면서 `main` 함수에 다시 전달한다. `main` 함수에서부터 프로그램이 실행되고 실행이 끝나면 반환값을 받아 `exit` 한다.

```
exit( main( argc, argv ) );
```

`exec` 시스템 호출에서부터 C 시작 루틴, `main` 함수로 프로그램 실행이 시작되는 과정은 그림 10.1과 같다.

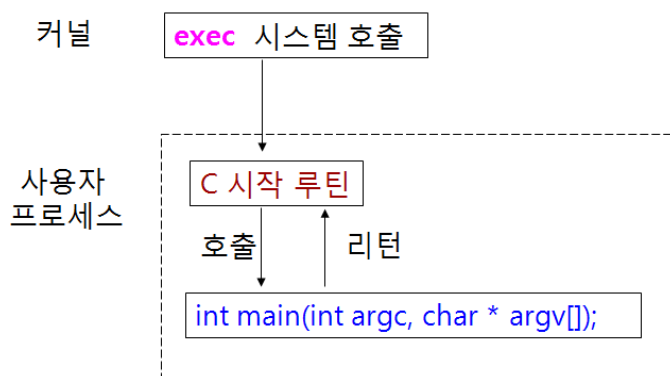


그림 10.1 프로그램 실행의 시작

`exec()` 시스템 호출은 실행되는 프로그램에게 명령줄 인수를 전달하는데 실행되는 프로그램의 `main` 함수는 `argc`와 `argv` 매개변수를 통해서 명령줄 인수의 수와 명령줄 인수에 대

한 포인터 배열을 전달받는다.

```
int main(int argc, char *argv[]);
```

argc : 명령줄 인수의 수

argv[] : 명령줄 인수 리스트를 나타내는 포인터 배열

명령줄 인수 리스트를 나타내는 포인터 배열 argv의 구성은 그림 10.2와 같다.

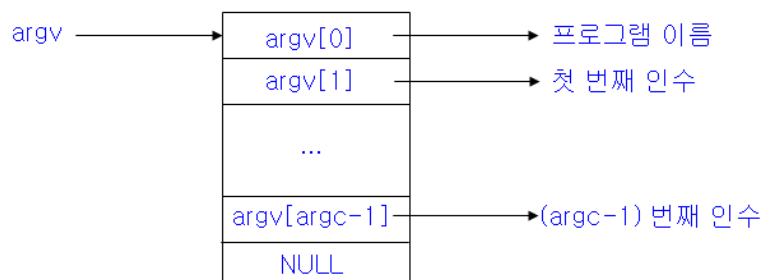


그림 10.2 명령줄 인수 리스트 argv 구성

또한 전역 변수 environ을 통해 환경 변수 리스트도 전달받는데 그 구성은 그림 10.3과 같다.

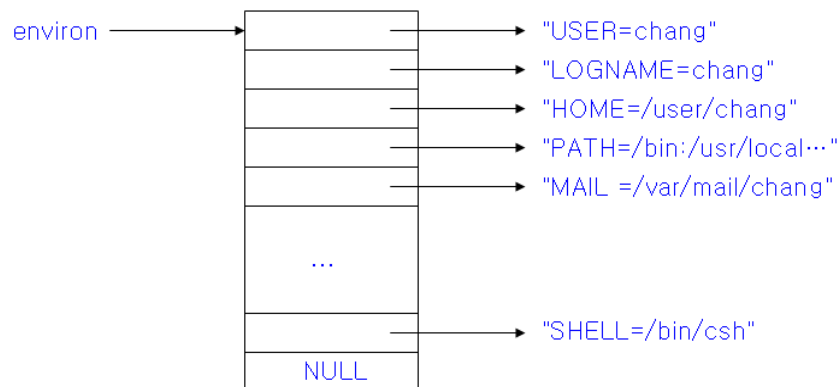


그림 10.3 환경 변수 리스트 environ의 구성

프로그램 10.1은 모든 명령줄 인수와 환경 변수를 프린트한다. 포인터 변수 ptr을 이용하여 환경 변수 리스트의 시작 위치인 environ에서부터 시작하여 1씩 증가하면서 각 환경 변수를 프린트한다.

프로그램 10.1 printall.c

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

```
int main(int argc, char *argv[])
```

```
{
```

```
    int    i
```

```
    char   **ptr
```

```
    extern char **environ;
```

```
    for (i = 0; i < argc; i++)          /* 모든 명령줄 인수 프린트 */
        printf("argv[%d]: %s\n", i, argv[i]);
```

```
    for (ptr = environ; *ptr != 0; ptr++) /* 모든 환경 변수 값 프린트*/
        printf("%s\n", *ptr);
```

```
    exit(0);
```

```
}
```

이제 프로그램이 종료하는 방법에 대해서 알아보자. 프로그램의 실행을 종료하는 방법은 정상 종료(normal termination), 비정상 종료(abnormal termination)로 크게 두 가지로 나눌 수 있다.

먼저 프로그램이 정상적으로 종료하는 방법부터 알아보도록 하자.

- main() 실행을 마치고 리턴하면 C 시작 루틴은 이 리턴값을 가지고 exit()을 호출한다.
- 프로그램 내에서 직접 exit()을 호출할 수 있다.
- 프로그램 내에서 직접 _exit()을 호출할 수 있다.

```
#include <stdlib.h>
void exit(int status);
```

exit() 시스템 호출은 뒷정리(cleanup processing)를 한 후 프로세스를 정상적으로 종료시킨다.

exit() 시스템 호출은 프로세스를 정상적으로 종료시키는데 종료 전에 모든 열려진 스트림을 닫고(fclose), 출력 버퍼의 내용을 디스크에 쓰는(fflush) 등의 뒷정리(cleanup processing)를 한다. 프로세스의 종료 상태를 알리는 종료 코드(exit code)를 부모 프로세스에게 전달한다.

```
#include <stdlib.h>
void _exit(int status);
```

`exit()` 시스템 호출은 뒷정리(cleanup processing)를 하지 않고 프로세스를 즉시 종료시킨다.

`_exit()` 시스템 호출 역시 프로세스를 정상적으로 종료시키는데 뒷정리를 하지 않고 즉시 종료된다는 점이 `exit()` 시스템 호출과 다르다.

프로그램이 비정상적으로 종료하는 방법은 2가지가 있다.

- `abort()` 시스템 호출은 프로세스에 `SIGABRT` 시그널을 보내어 프로세스를 비정상적으로 종료시킨다.
- 시그널에 의한 종료: 프로세스가 실행 중에 시그널을 받으면 갑자기 비정상적으로 종료하게 된다. 시그널에 대한 자세한 사항은 10.6 절에서 자세히 다룬다.

10.2 프로세스 구조

프로세스란 무엇인가? 프로세스에 대한 정의 혹은 설명은 여러 가지가 있지만 가장 쉬운 정의는 실행 중인 프로그램(executing program)을 프로세스라고 생각하는 것이다. 다시 말하면 프로그램이 실행되면 프로세스가 되는 것이다. 한 프로그램은 여러 번 실행될 수 있으므로 한 프로그램으로부터 여러 개의 프로세스를 만들 수 있으며 프로그램 그 자체가 프로세스는 아니라는 점을 주의하자.

프로그램을 실행 즉 프로세스를 유지하기 위해서는 무엇이 필요할지 생각해 보자. 먼저 프로세스 관리를 위한 커널 내의 프로세스에 대한 정보가 필요할 것이다. 또한 프로세스 즉 프로그램 실행을 위해서는 그림 10.4와 같이 실행 코드, 데이터, 힙, 스택 등을 메모리 내에 배치해야 한다. 이러한 메모리 배치를 프로세스 메모리 이미지라고 한다. 프로세스 메모리 이미지를 구성하는 프로그램의 실행 코드, 데이터, 스택, 힙 등의 영역의 역할은 다음과 같다.

- 코드(code)

프로세스의 실행 코드를 저장하는 영역이다.

- 데이터(data)

전역 변수(global variable) 및 정적 변수(static variable)를 위한 영역이다.

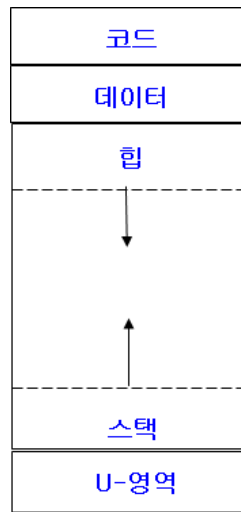


그림 10.4 프로세스 메모리 이미지

- 힙 (heap)

동적 메모리 할당을 위한 영역이다.

- 스택(stack area)

함수 호출을 구현하기 위한 (지역 변수를 포함하는) 활성 레코드(activation record)를 저장하기 위한 실행시간 스택(runtime stack)을 위한 영역이다.

- U-영역 (user-area)

열린 파일 식별자 등과 같은 시스템 관리 정보를 저장하는 영역이다.

10.3 프로세스 생성

각 프로세스는 프로세스를 구별하는 번호인 프로세스 ID를 갖고 있다. 실행 중인 프로그램 즉 프로세스가 `getpid()`를 호출하면 실행 중인 프로세스의 ID를 리턴한다. 또한 `getppid()`를 호출하면 실행 중인 프로세스의 부모 프로세스의 ID를 리턴한다.

<code>int getpid()</code>	프로세스의 ID를 리턴한다.
<code>int getppid()</code>	부모 프로세스의 ID를 리턴한다.

Unix에서는 필요에 따라 새로운 프로세스를 생성해야 하는데 `fork()` 시스템 호출이 프로세스를 생성하는 유일한 방법이다.

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

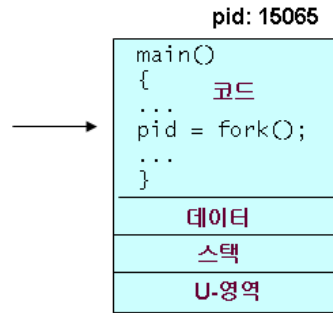
`fork()` 시스템 호출은 새로운 자식 프로세스를 생성한다. 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.

`fork()` 시스템 호출은 새로운 자식 프로세스를 생성한다. 프로세스 생성 원리를 간단히 요약하면 **자기복제(自己複製)**라고 할 수 있다. 자식 프로세스는 부모 프로세스(코드, 데이터, 스택, 힙 등)를 똑 같이 복제해 만들어진다.

`fork()` 시스템 호출을 하면 새로운 자식 프로세스가 즉시 생성되며 부모 프로세스와 자식 프로세스에게 각각 리턴한다. 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다. `fork()` 시스템 호출은 한 번 호출되지만 두 번 리턴된다는 점을 주의하자. `fork()` 호출 후에 부모 프로세스와 자식 프로세스가 병행적으로 실행을 계속한다.

다음 예제 프로그램을 통해 `fork()` 호출 후에 리턴값과 실행 흐름을 살펴보자. 실행 결과를 보면 `fork()` 뒤에 나오는 문장은 부모 프로세스와 자식 프로세스에 의해 각각 실행되며 부모 프로세스에게는 생성된 자식 프로세스의 ID(15066)이 리턴되고 자식 프로세스에게는 0이 리턴됨을 확인할 수 있다.

BEFORE fork()



AFTER fork()

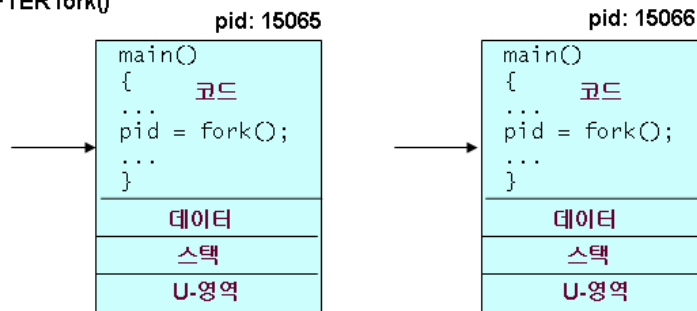


그림 10.5 프로세스 생성 전후

프로그램 10.2 fork1.c

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

```
main()
{
    int pid;
    printf("[%d] 프로세스 시작 \n", getpid());
    pid = fork();
    printf("[%d] 프로세스 : 리턴값 %d\n", getpid(), pid);
}
```

실행 결과

```
[15065] 프로세스 시작
[15065] 프로세스 : 리턴값 15066
[15066] 프로세스 : 리턴값 0
```

이 예제를 통해 `fork()` 호출 뒤에 나타나는 문장은 부모 프로세스와 자식 프로세스가 병행적으로 모두 실행한다는 것을 확인할 수 있었다.

그러면 부모 프로세스와 자식 프로세스가 서로 다른 일을 하려면 어떻게 하여야 할까?

`fork()` 호출 후에 리턴값이 다르므로 이 리턴값을 이용하면 부모 프로세스와 자식 프로세스를 구별하고 서로 다른 일을 하도록 할 수 있을 것이다. 따라서 다음과 같은 코드를 수

행하면 `fork()` 호출 후에 자식 프로세스는 자식을 위한 코드 부분을 실행하고 부모 프로세스는 부모를 위한 코드 부분을 실행한다.

```
pi d = fork();
i f ( pi d == 0 )
{ 자식 프로세스의 실행 코드 }
e l s e
{ 부모 프로세스의 실행 코드 }
```

다음 예제 프로그램은 부모 프로세스가 자식 프로세스를 생성하며 각 프로세스가 메시지와 프로세스 ID를 프린트한다. 실행 결과를 보면 자식 프로세스는 리턴값으로 0을 받았으므로 `if` 문의 `then` 부분을 실행했고 부모 프로세스는 리턴값으로 자식 프로세스 ID(15800)을 받았으므로 `if` 문의 `else` 부분을 실행했음을 확인할 수 있다.

프로그램 10.3 fork2.c

```
#i n c l u d e < s t d l i b . h >
#i n c l u d e < s t d i o . h >

mai n ()
{
    i n t pi d;

    pi d = fork();
    i f (pi d ==0){
        pri n t f (" [Chi l d] : Hel l o, worl d pi d=%d\n", getpi d());
    }
    e l s e {
        pri n t f (" [Parent] : Hel l o, worl d pi d=%d\n", getpi d());
    }
}
```

실행 결과

```
[Parent] : Hel l o, worl d pi d=15799
[Chi l d] : Hel l o, worl d pi d=15800
```

이제 하나의 부모 프로세스가 두 개의 자식 프로세스를 생성하는 다음 예제 프로그램을 살펴보자. 이 예제에서 부모 프로세스는 두 개의 자식 프로세스를 생성하며 각 자식 프로세스가 메시지와 프로세스 ID를 프린트한다.

프로그램 10.4 fork3.c

```
#i n c l u d e < s t d l i b . h >
```



```
#include <stdio.h>

main()
{
    int pid1, pid2;

    pid1 = fork();
    if (pid1 == 0) {
        printf("[Child 1] : Hello, world ! pid=%d\n", getpid());
        exit(0);
    }

    pid2 = fork();
    if (pid2 == 0) {
        printf("[Child 2] : Hello, world ! pid=%d\n", getpid());
        exit(0);
    }
}
```

실행 결과

```
[Child 1] : Hello, world ! pid=15741
[Child 2] : Hello, world ! pid=15742
```

QnA 첫 번째 자식 프로세스 부분에서 `exit(0)`를 왜 하지요 ?

```
int wait(int* status)
```

`wait()` 시스템 호출은 자식 프로세스 중에 하나가 종료될 때까지 호출한 프로세스를 기다리게 한다. 종료된 자식 프로세스의 `pid`를 리턴하고 상태 코드를 **status*에 둔다.

이제 `wait` 시스템 호출을 이용하여 부모 프로세스가 자식 프로세스가 끝나기를 기다리는 다음 예제 프로그램을 살펴보자. 부모 프로세스는 자식 프로세스가 끝나기를 기다리며 끝난 후에는 자식 프로세스 종료 메시지와 자식 프로세스로부터 받은 종료 코드 값을 프린트한다.

프로그램 10.5 forkwait.c

```
#include <stdio.h>
main() {
    int pid, child, status;
```

```
printf("[%d] 부모 프로세스 시작 \n", getpid());
pid = fork();
if (pid == 0) {
    printf("[%d] 자식 프로세스 시작 \n", getpid());
    exit(1);
}

child = wait(&status);
printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), child);
printf("\t종료 코드 %d\n", status>>8);
}
```

실행 결과

```
[15943] 부모 프로세스 시작
[15944] 자식 프로세스 시작
[15943] 자식 프로세스 15944 종료
      종료 코드 1
```

10.4 프로그램 실행

앞에서 살펴본 것처럼 부모 프로세스가 자식 프로세스를 생성하면 자식 프로세스는 부모 프로세스와 같은 코드를 실행한다. 그렇다면 자식 프로세스에게 새로운 일을 시키려면 어떻게 하여야 할까? 이를 위해서는 자식 프로세스 내에서 새로운 프로그램을 실행시킬 수 있는 방법이 있어야한다.

`exec()` 시스템 호출을 이용하여 프로세스 내에서 새로운 프로그램을 실행시킬 수 있으며 `exec()` 시스템 호출이 프로세스 내에서 새로운 프로그램을 실행시키는 유일한 방법이다. `exec()` 시스템 호출의 원리를 간단히 요약하면 **자기대치(自己代置)**라고 할 수 있다. 프로세스가 `exec()` 호출을 하면, 그 프로세스 내의 프로그램은 완전히 새로운 프로그램(코드, 데이터, 스택 등)으로 대체된다. 그리고 새 프로그램의 `main()`부터 실행이 시작한다.

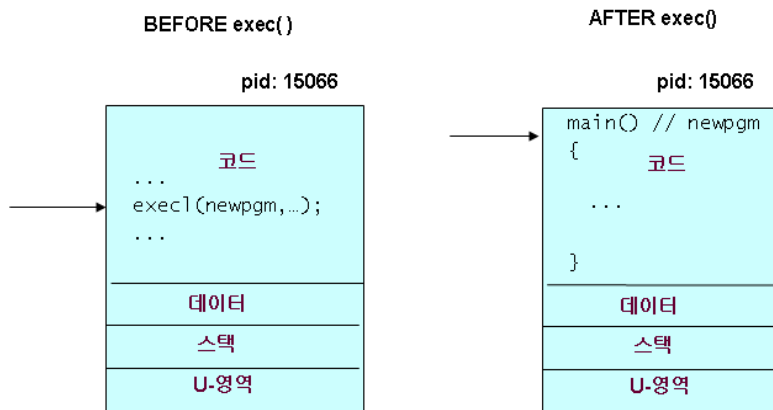


그림 10.6 프로그램 실행

`exec()` 호출이 성공하면 그 프로세스 내에 기존의 프로그램은 없어지고 새로운 프로그램으로 대체되므로 `exec()` 호출은 리턴할 곳이 없어진다. 성공한 `exec()` 호출은 절대 리턴하지 않는다는 점을 유의하자. `exec()` 호출은 실패할 경우에만 리턴한다.

```
int execl(char* path, char* arg0, char* arg1, ... , char* argn,
NULL)
int execv(char* path, char* argv[ ])
```

호출한 프로세스의 코드, 데이터, 힙, 스택 등을 `path`가 나타내는 새로운 프로그램으로 대체한 후 새 프로그램을 실행한다. 성공한 `exec()` 호출은 리턴하지 않으며 실패하면 `-1`을 리턴한다.

`exec` 시스템 호출에는 크게 `execl()`과 `execv()`이 있다. `execl()` 시스템 호출은 명령줄 인수를 하나씩 나열하고 `NULL`은 인수 끝을 나타낸다. `execv()` 시스템 호출을 할 때는

명령줄 인수를 하나씩 나열하지 않고 명령줄 인수 리스트를 포인터 배열로 만들어 이 배열의 이름을 전달한다.

보통 다음과 같이 `fork()` 시스템 호출 후에 `exec()` 시스템 호출하는 경우가 일반적이며 새로 실행할 프로그램에 대한 정보를 `arguments`로 전달한다. `exec()` 시스템 호출이 성공하면 자식 프로세스는 새로운 프로그램을 실행하게 되고 부모는 계속해서 다음 코드를 실행하게 된다. `exec()` 시스템 호출이 실패하면 자식 프로세스는 `exit(1)`를 호출하여 종료한다.

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

간단한 예제 프로그램을 살펴보자. 이 프로그램은 자식 프로세스를 생성하여 자식 프로세스로 하여금 `echo` 명령어를 실행하게 한다. 여기서는 `execl()` 시스템 호출을 사용하였으며 명령줄 인수로 `"hello"` 스트링을 주고 `NULL`은 인수 끝을 나타낸다.

```
execl ("/bin/echo", "echo", "hello", NULL);
```

자식 프로세스는 `echo` 명령어를 실행하여 명령줄 인수로 받은 `"hello"` 스트링을 그대로 프린트한다.

프로그램 10.6 execute1.c

```
#include <stdio.h>
main( ) {
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl ("/bin/echo", "echo", "hello", NULL);
        fprintf(stderr, "첫 번째 실패");
        exit(1);
    }
    printf("부모 프로세스 끝\n");
}
```

실행 결과

```
부모 프로세스 시작
hello
부모 프로세스 끝
```

이 예제 프로그램은 세 개의 자식 프로세스를 생성하여 각각에게 서로 다른 명령어를 실행 시킨다.

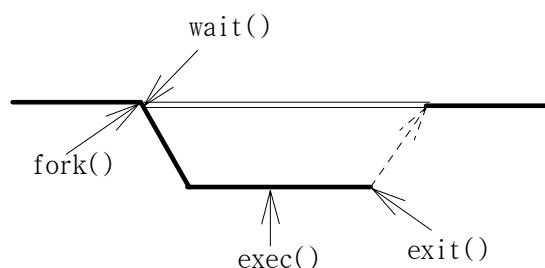
프로그램 10.7 execute2.c

```
#include <stdio.h>
main( ) {
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl ("/bin/echo", "echo", "hel l o", NULL);
        fprintf(stderr, "첫 번째 실패");
        exit(1);
    }
    if (fork( ) == 0) {
        execl ("/bin/date", "date", NULL);
        fprintf(stderr, "두 번째 실패");
        exit(2);
    }
    if (fork( ) == 0) {
        execl ("/bin/l s", "l s", "-l ", NULL);
        fprintf(stderr, "세 번째 실패");
        exit(3);
    }
    printf("부모 프로세스 끝\n");
}
```

실행 결과

위 프로그램은 정해진 명령어만 실행시킨다. 이제 명령줄 인수로 받은 임의의 명령어를 실행시키는 프로그램을 작성해보자. 다음 예제 프로그램은 명령줄 인수로 받은 명령어의 실행을 위해 자식 프로세스를 생성하고 자식 프로세스로 하여금 그 명령어를 실행하게 한다. 부모 프로세스는 자식 프로세스가 끝날 때까지 기다리며 자식 프로세스가 종료하면 자식 프로세스 종료 메시지와 자식 프로세스로부터 받은 종료 코드를 프린트한다.

이 프로그램의 실행 과정을 다음 그림과 같이 표현할 수 있다.



프로그램 10 execute3.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int child, pid, status;
    pid = fork( );
    if (pid == 0) {
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "%s: 실행 불가\n", argv[1]);
    } else {
        child = wait(&status);
        printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), pid);
        printf("\t종료 코드 %d \n", status>>8);
    }
}
```

이 프로그램을 이용하여 명령줄 인수로 받은 임의의 명령어를 실행시킬 수 있다. 예를 들어 다음 명령은 ls 명령어를 실행시킨다.

```
$ execute ls -l /tmp
```

● 기타 시스템 호출

- 현재 디렉토리 변경

각 프로세스는 현재 작업 디렉토리(current working directory)를 가지고 있다. chdir() 시스템 호출은 현재 작업 디렉토리를 매개변수가 지정한 경로 pathname으로 변경한다. 이 시스템 호출이 성공하려면 프로세스는 그 디렉토리에 대한 실행 권한이 있어야 한다. 성공하면 0 실패하면 -1을 리턴한다.

int chdir (char* pathname)	현재 작업 디렉토리를 pathname으로 변경한다.
----------------------------	------------------------------

- 프로세스의 사용자 ID와 그룹 ID

각 프로세스마다 사용자 ID가 있는데 프로세스의 사용자 ID는 로그인하여 그 프로세스를 실행시킨 사용자를 나타낸다. 다음 시스템 호출은 프로세스의 소유주의 사용자 ID와 그룹

ID를 각각 리턴한다.

<code>int getuid()</code>	프로세스의 사용자 ID를 리턴한다.
<code>int getgid()</code>	프로세스의 그룹 ID를 리턴한다.

또한 다음 시스템 호출은 프로세스의 소유주를 매개변수로 지정해 준 사용자 ID와 그룹 ID로 각각 변경한다.

<code>int setuid(uid_t uid)</code>	프로세스의 사용자 ID를 uid로 변경한다.
<code>int setgid(gid_t gid)</code>	프로세스의 그룹 ID를 gid로 변경한다.

10.5 시스템 부팅

시스템 부팅 과정을 생각해보자. 시스템이 부팅되면서 여러 개의 프로세스가 생성되는데 이 과정은 어떻게 이루어질까? 시스템 부팅 과정에서 앞에서 배운 `fork/exec`는 매우 유용하게 사용된다.

부팅이 시작되면 커널 내부에서 프로세스 ID가 0인 첫 번째 프로세스 `swapper`가 만들어진다. `swapper`는 프로세스를 스케줄링하는 기능을 한다. 이 프로세스는 `fork/exec`를 수행하여 1번 프로세스인 `init` 프로세스를 생성한다. `init` 프로세스는 역시 `fork/exec`를 반복적으로 수행하여 시스템 운영에 필요한 다양한 프로세스들(주로 서버 데몬 프로세스)을 새로 생성한다. 이 `init` 프로세스는 모든 프로세스의 조상이라고 할 수 있다. 그림 10.7은 이러한 부팅 과정을 보여주고 있는데 예를 들어 `sshd`와 같은 `ssh` 데몬 프로세스나 `getty` 프로세스를 생성한다. 이 그림에서 괄호 안의 수는 프로세스 ID를 나타낸다.

데몬 프로세스 중에는 `getty`(Linux 경우에는 `mingetty`) 프로세스가 있는데 이 프로세스로부터 로그인 과정이 시작된다. 이 프로세스는 화면에 로그인 프롬프트를 띄우고 사용자의 ID가 입력되기를 기다린다. 입력이 들어오면 `fork()` 시스템 호출은 하지 않고 바로 `exec()` 시스템 호출을 하여 `login` 프로그램(`/bin/login`)을 실행한다. 이 프로그램이 패스워드 등을 검사하고 성공하면 다시 `exec()` 시스템 호출을 하여 `shell` 프로그램(예를 들어 `/bin/sh`)을 실행한다.

그림 10.7은 이러한 로그인 과정을 보여주고 있는데 `getty` 프로세스가 `login` 프로세스, `shell` 프로세스로 변화하지만 `fork()` 시스템 호출은 하지 않고 `exec()` 시스템 호출만 하기 때문에 프로세스 ID는 모두 같다는 점을 유의하자.

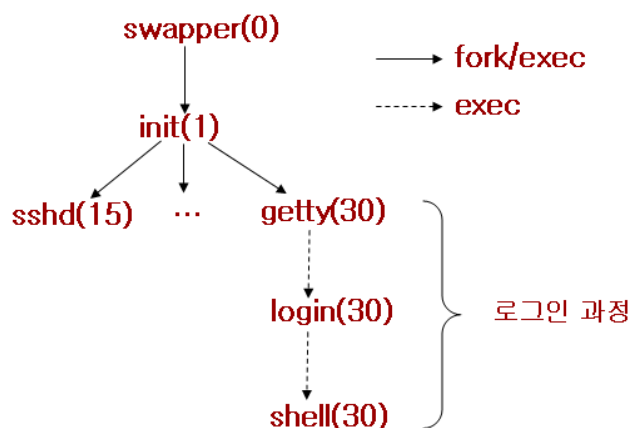


그림 10.7 부팅 및 로그인 과정

각 프로세스에 대한 보다 자세한 설명은 다음과 같다.

- `swapper` (스케줄러 프로세스)

swapper는 커널 내부에서 만들어진 프로세스로 프로세스 스케줄링을 한다. 이 프로세스는 커널 내의 코드를 실행하기 때문에 별도의 실행 파일이 존재하지 않는다.

- **init (초기화 프로세스)**

init 프로세스(/etc/init 혹은 /sbin/init)는 /etc/inittab 파일에 기술된 대로 시스템을 초기화하는데 이 파일 내에서 다시 /etc/rc* 즉 rc로 시작되는 이름의 셸 스크립트들을 실행한다. 이러한 과정을 통해서 파일 시스템 마운트, 서버 데몬 프로세스 생성, getty 프로세스 생성 등의 작업을 수행하여 시스템을 초기화한다.

- **getty 프로세스**

getty 프로세스(/etc/getty 혹은 /etc/mingetty)는 로그인 프롬프트를 내고 키보드 입력을 감지한다. 아이디, 패스워드를 입력하면 로그인 절차를 진행하기 위해 로그인 프로그램(/bin/login)을 실행한다.

- **login 프로세스**

login 프로세스는 /etc/passwd 파일을 참조하여 사용자의 로그인 아이디 및 패스워드를 검사한다. 로그인 절차가 성공하면 셸 프로그램(/bin/sh, /bin/csh 등)을 실행한다.

- **shell 프로세스**

shell 프로세스는 시작 파일을 실행한 후에 셸 프롬프트를 내고 사용자로부터 명령어를 기다린다. 명령어가 입력되면 해석하여 명령어를 실행시킨다. 명령어 실행 후에 다시 셸 프롬프트를 내고 이 과정을 반복한다.

10. 6 시그널(Signal)

프로그램 실행 도중에 예기치 않는 사건이 발생하면 이를 실행 중인 프로그램에 알려줄 수 있어야 한다. 예를 들어 연산 중에 0으로 나누는 오류가 발생하면 재빨리 이를 프로그램에 알려야 하고 프로그램에서는 이를 적절히 처리해야 한다. 이러한 의미에서 시그널은 예기치 않은 사건이 일어날 때 발생하는 소프트웨어 인터럽트라고 할 수 있다.

예를 들어 다음과 같은 경우에 시그널이 발생한다.

- 부동소수점 오류
- 정전
- 알람시계 울림
- 자식 프로세스 종료
- 키보드로부터 종료 요청 (Ctrl -C)
- 키보드로부터 정지 요청 (Ctrl -Z)

Unix에는 총 31개의 시그널이 `/usr/include/signal.h`에 정의되어 있다. 각 시그널 이름은 SIG로 시작된다.

시그널 이름	의미
SIGABRT	abort()에서 발생하는 종료 시그널
SIGALRM	알람시계 alarm() 울림
SIGINT	터미널에서 CTRL-C 할 때 발생하는 인터럽트 시그널
SIGSTP	터미널에서 CTRL-C 할 때 발생하는 중지 시그널
SIGFPE	0으로 나누기
SIGSEGV	유효하지 않은 메모리 참조
SIGPIPE	끊어진 파이프
SIGCHLD	프로세스의 종료 혹은 중지를 부모에게 알리는 시그널
SIGCONT	중지된 프로세스를 계속시키는 시그널
SIGHUP	연결 끊김
SIGILL	잘못된 하드웨어 명령어 수행
SIGIO	비동기화 I/O 이벤트 알림
SIGKILL	프로세스 종료시키는 시그널
SIGPIPE	파이프에 쓰려고 할 때 리더가 없을 때
SIGSTOP	프로세스 중지 시그널
SIGSYS	유효하지 않은 시스템 호출
SIGTERM	kill()에서 보내는 종료 시그널

`alarm()` 시스템 호출은 매개변수로 받은 초 후에 `SIGALRM` 시그널을 발생시킨다. 프로그램이 실행 중에 이 시그널을 받으면 “`Alarm clock`” 메시지를 출력하고 프로그램은 종료된다.

다음 예제 프로그램을 살펴보자. 이 프로그램에서는 5초 후에 `SIGALRM` 시그널이 발생된다. `while` 루프는 1초에 한번씩 “1초 경과”라는 메시지를 출력하다가 5초가 지나면 해당 시그널을 받아 “`Alarm clock`” 메시지를 출력하고 프로그램은 종료된다. 마지막 `printf` 문은 무한 루프 뒤에 위치해 있으며 무한 루프 실행 중에 `SIGALRM` 시그널을 받으면 프로그램이 종료되므로 절대로 실행되지 않음을 주의하자.

프로그램 10.9 `alarm.c`

```
#include <stdio.h>
main( )
{
    alarm(5);
    printf("무한 루프 \n");
    while (1) {
        sleep(1);
        printf("1초 경과 \n");
    }
    printf("실행되지 않음 \n");
}
```

시그널 처리

앞의 예에서 본 것처럼 발생한 시그널을 따로 처리하지 않으면 시그널에 따라 다르지만 많은 경우에 프로그램은 거기서 종료된다. 따라서 시그널이 발생하면 이를 잡아서 적절히 처리할 수 있어야 한다. 이를 위해서 다음과 같은 의미로 시그널에 대한 처리함수를 지정할 수 있다.

“이 시그널이 발생하면 이렇게 처리하라”

시그널에 대한 처리함수 지정은 `signal()` 시스템 호출을 통해 할 수 있는데 `signal()` 시스템 호출은 다음과 같은 형태로 각 시그널에 대한 처리 함수를 등록한다.

```
signal (int sigCode, void (*func)())
```

`sigCode`에 대한 처리 함수를 `func`으로 지정한다. `func`은 `SIG_IGN`, `SIG_DFL` 혹

은 사용자 정의 함수 이름이다. 기존의 처리함수를 리턴한다.

세 종류의 처리 함수의 의미는 다음과 같다.

- **SIG_IGN**

발생된 시그널을 무시하겠다는 의미로 SIGKILL, SIGSTOP을 제외한 시그널은 필요하면 무시할 수 있다.

- **SIG_DFL**

시그널에 대한 처리함수로 디폴트 처리함수를 사용하겠다는 의미이다. 각 시그널마다 미리 정해진 디폴트 처리함수가 있으며 따로 지정하지 않으면 이 처리함수가 수행된다.

- **사용자 정의 함수**

시그널에 대한 처리함수로 지정한 사용자 정의 함수를 사용하겠다는 의미이다.

이제 시그널 처리 함수를 이용한 예제 프로그램을 살펴보자. 다음 예제 프로그램은 명령줄 인수로 받은 임의의 명령어를 제한 시간 내에 실행시키는 프로그램이다. 이 프로그램은 프로그램 10.xx를 알람 시그널을 이용하여 확장하여 작성하였다. 명령어 실행에 제한 시간을 두기 위해서는 자식 프로세스가 명령어를 실행하는 동안 정해진 시간이 초과되면 SIGALRM 시그널이 발생하고 이때 자식 프로세스를 강제 종료하면 된다. 이를 위해서 두 가지 작업을 수행한다. 먼저 제한 시간을 입력 받아 알람시계를 동작시킨다. 두 번째로 SIGALRM 시그널에 대한 처리함수 alarmHandler()를 작성하고 이를 SIGALRM 시그널에 대한 처리함수로 지정한다. 이 처리함수는 SIGALRM 시그널이 발생되면 자동적으로 실행되어 자식 프로세스를 강제적으로 종료시킨다. 이 예제에서는 kill(pid, SIGINT) 시스템 호출을 통해 자식 프로세스에 SIGINT 시그널을 보내어 강제적으로 종료시켰다.

프로그램 10.10 limit.c

```
#include <stdio.h>
#include <signal.h>
int pid;
void alarmHandler();

main(int argc, char *argv[])
{
    int child, status, limit;

    signal(SIGALRM, alarmHandler);
    sscanf(argv[1], "%d", &limit);
    alarm(limit);
```

```

pid = fork( );
if (pid == 0) {
    execvp(argv[2], &argv[2]);
    fprintf(stderr, "%s: 실행 불가\n", argv[1]);
} else {
    child = wait(&status);
    printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), pid);
    printf("\t종료 코드 %d \n", status>>8);
}
}

void alarmHandler()
{
    printf("[알람] 자식 프로세스 %d 시간 초과\n", pid);
    kill(pid, SIGINT);
}

```

프로세스에 Signal 보내기

상황에 따라 시그널이 자동적으로 발생되기도 하지만 앞에 예에서도 본 것처럼 필요에 따라 특정 프로세스에 임의의 시그널을 강제로 보낼 필요가 있다. 이러한 기능은 한 프로세스가 다른 프로세스를 제어하는데 매우 유용하게 사용될 수 있다.

다음의 `kill()` 시스템 호출을 이용하여 특정 프로세스에 원하는 임의의 시그널을 보낼 수 있다.

```
int kill(int pid, int sigCode)
```

프로세스 *pid*에 시그널 *sigCode*를 보낸다.

성공 조건: 보내는 프로세스의 소유자가 프로세스 *pid*의 소유자와 같거나 혹은 보내는 프로세스의 소유자가 슈퍼유저이어야 한다.

시그널을 이용하여 자식 프로세스를 제어하는 예제 프로그램을 살펴보자. 프로그램 10.11은 자식 프로세스를 생성하고 실행 중인 자식 프로세스에 `SIGSTOP` 시그널을 보내어 정지시키고 다시 `SIGCONT` 시그널을 보내어 실행을 계속하게 한다. 그 후 `SIGINT` 시그널을 보내어 자식 프로세스를 강제 종료시킨다.

프로그램 10.11 control.c

```

#include <signal.h>
#include <stdio.h>

```

```
main( )
{
    int pid1, pid2;

    pid1 = fork( );
    if (pid1 == 0) {
        while(1) {
            printf("프로세스 %i d 실행\n", getpid());
            sleep(1);
        }
    }

    sleep(3);
    kill(pid1, SIGSTOP);
    sleep(3);
    kill(pid1, SIGCONT);
    sleep(3);
    kill(pid1, SIGINT);
}
```