Part

04

프로세스 간 통신 방법과 프로그램 작성

Chapter

09

프로세스 간 통신

*** 학습목표**

- 프로세스 간의 통신에서 선택할 수 있는 몇 가지 방법을 이해한다.
- 프로세스 간의 통신 기법의 차이점에 대해서 이해한다.
- 소켓에 대해서 이해한다.

Section

01 프로세스 간 통신

리눅스 시스템에서 수행되는 프로세스들은 여러 가지 필요에 의해 서로 데이터를 주고받을 필요가 있다. 데이터를 주고받는 프로세스들을 분류해보면 간단하고 짧은 메시지를 주고받거나 복잡하고 긴 메시지를 주고받을 수도 있고, 동일한 시스템의 프로세스끼리 통신을 하거나 서로 다른 시스템의 프로세스끼리 통신 할 수 있다. 그리고 통신의 주체가 되는 양쪽 끝의 프로세스가 모두 사용자 프로세스일 수도 있고 한쪽이 시스템의 커널일 수도 있다.

이 장에서는 우리가 통신 프로그램을 만들면서 선택할 수 있는 기본적인 통신 방법에 대해서 다룬다. 통신 프로그램이 어떤 유형의 메시지를 주고받는지를 따져보고 적당한 통신 방법을 선택하면 될 것이다.

프로세스 간의 동기화를 위한 것이라면 시그널을 선택하고 동일한 시스템 내에서 메시지를 주고받는 것이라면 파이프를 선택하고, 서로 다른 시스템에 프로세스끼리 메시지를 주고받는 것이라면 소켓을 선택하면 된다. 각 방법이 모두 장단점을 가지고 있기 때문에 각 방법에 대해서 이해하고 프로그램을 제작할 때 가장 적합한 방법을 선택하면 된다.

Section

2 시그널

프로그램은 가끔 기대하지 않은 이벤트를 다뤄야 할 때가 있다. 연산 오류가 발생했거나 자식 프로세스가 종료했거나 사용자의 종료 요청 등이 있을 때 프로세스는 이러한 이벤트를 다룰 수 있어야 한다. 이러한 종류의 이벤트는 인터럽트(interrupt)라고 불리기도 하는데, 정상적으로 실행되고 있는 프로세스에게 전달된다. 리눅스 시스템은 이벤트가 발생하면 관련된 프로세스에게 이벤트와 관련된 시그널을 보내게 된다.

시그널은 번호가 붙여져 있는데, 각 번호는 유일하다. 예를 들자면 사용자가 전면에서 수행중인 프로세스를 강제로 종료하기 위해 Ctrl +C 를 누르게 되면 커널은 해당 프로세스에게 SIGTERM이라는 이름의 시그널을 보내게 되는데, 이 시그널은 15번에 해당한다.

커널만 시그널을 보낼 수 있는 것은 아니다. 어떤 프로세스든지 권한만 있다면 선택된 프로세스에게 특정 시그널을 보낼 수 있다. 우리는 이전에 'kill' 명령을 사용하였는데, 이 명령은 지정한 프로세스에게 SIGTERM 시그널을 전달한다.

시그널을 수신한 프로세스는 보통 다음 중에 하나로 반응하게 된다.

① 시그널에 대해 기본적인 방법으로 대응한다. 대부분의 시그널에 대해서 프로세스는 종료하게 된다.
② 시그널을 무시한다. 단, SIGKILL과 SIGSTOP은 무시될 수 없다.
③ 프로그래머가 지정한 함수를 호출한다.

프로그래머는 프로세스가 특정 시그널을 포착했을 때 수행해야 할 별도의 함수를 지정할 수 있는데, 이런 함수를 '시그널 핸들러(Signal Handler)' 라고 부른다. 실행중인 프로세스가 시그널을 포착하게 되면 현재 진행중인 작업의 실행을 잠시 중단하고 시그널 핸들러를 실행하게 된다. 시그널 핸들러의 실행이 끝나면 아까 중단했던 작업을 이어서 실행하게 된다.

시그널의 종류는 다음 중 하나에 정의되어 있다.

```
/usr/include/sys/signal.h
/usr/include/signal.h
/usr/include/asm/signal.h
```

각 시그널은 보통 매크로로 정의한 이름으로 사용되므로 실제 값은 그렇게 중요하지 않고 그 의미가 중요하다.

시그널 목록은 다음의 [표 9-1]과 같다.

[표 9-1] 시그널 목록

매크로	설명
SIGABRT	프로세스를 중단한다. 프로세스가 abort 함수를 호출할 때 발생한다.
SIGALRM	alarm 함수로 설정한 시간이 지났을 때 발생한다.
SIGBUS	하드웨어 결함이 발생했을 때 발생한다.
SIGCHLD	자식 프로세스가 중단되거나 종료할 때 발생한다. 기본적으로 무시한다.
SIGCONT	중단된 프로세스의 실행을 계속하게 한다. SIGSTOP개 반대의 의미다.
SIGFPE	부동점 오류가 생겼을 때 발생한다.
SIGHUP	터미널의 연결이 끊어졌을 때 관련된 모든 프로세스에서 발생한다.
SIGILL	불법적인 명령어를 수행하려 할 때 발생한다.
SIGINT	사용자가 인터럽트 키를 눌렀을 때 관련된 모든 프로세스에서 발생한다.
SIGKILL	프로세스가 다른 프로세스를 종료시키기 위하여 사용한다. 커널에 의해서 많이 사용된다. 프로그램이 임의로 포착하거나 무시할 수 없다.
SIGPIPE	파이프 사용과 관련된 시그널이다.
SIGPOLL	파일 기술자를 통해서 읽기나 쓰기가 가능해지면 발생한다.
SIGQUIT	SIGINT와 비슷한 시그널로 사용자와 관련된 터미널에서 종료 키를 누르면 발생한다.
SIGSEGV	잘못된 메모리를 참조할 때 발생한다.
SIGSTOP	프로세스의 실행을 중단한다. SIGKILL처럼 임의로 포착하거나 무시할 수 없다.
SIGSYS	프로세스가 시스템 호출이 아닌 기계어 명령을 실행하려 할 때 발생한다.
SIGTERM	프로세스를 종료시킨다.
SIGSTP	사용자가 터미널에서 일시 중단 키를 누를 때 발생한다.

SIGTTIN	후면에서 실행중인 프로세스가 터미널로부터 읽기를 시도할 때 발생한다.
SIGTTOU	후면에서 실행중인 프로세스가 터미널에 쓰기를 시도할 때 발생한다.
SIGURG	소켓을 통해 대역폭을 벗어난 데이터가 수신되었을 때 발생한다.
SIGUSR1 / SIGUSR2	사용자가 임의의 목적을 위해서 사용한다.
SIGXCPU	프로세스가 자신에게 주어진 최대 CPU 사용 시간을 초과했을 때 발생한다.
SIGXVSZ	파일 크기를 초과했을 때 발생한다.

시그널을 전달하는 가장 간단한 방법은 전면에서 실행중인 프로세스에게 Ctrl + C 나 Ctrl + Z 를 눌러 SIGINT나 SIGSTP 신호를 보내는 것이다. 이 두 신호에 대한 프로세스의 기본 반응은 프로세스를 종료하는 것과 실행을 중단하는 것이다.

Section

03 파이프와 네임드 파이프

파이프는 리눅스 이전에 유닉스 환경에서부터 요긴하게 사용된 프로세스 간 통신 방법 중 하나다. 앞서 'ls –la | wc –l' 과 같이 파이프 기호(|)로 연결된 명령어 라인을 실행하는 것을 여러 차례 얘기하였다. 두 개의 명령이 파이프로 연결되면 앞에 놓인 명령이 표준 출력하는 내용을 뒤에 놓인 명령이 표준 입력으로 받아들인다고 했다. 이때 중간에서 다리 역할을 하는 것을 파이프라고 부른다.

파이프는 사용자가 파이프로 연결한 명령 라인을 실행하면 일시적으로 만들어졌다가 작업이 끝나면 삭제되는 임시 파일이다. 임시 파일도 파일이기 때문에 파이프로 연결되어 작업에 관련된 프로세스라면 읽기와 쓰기가 모두 가능하지만 대부분의 구현에서 파이프의 앞에 놓은 프로세스는 파이프로 쓰기만 수행하고 파이프의 뒤에 놓인 프로세스는 파이프로부터 읽기만 수행하게 된다.

다음은 파이를 사용한 몇 개의 예제들이다. 첫 번째 명령어 라인은 'ls' 로 루트 디렉터리에 대한 파일 항목을 표준 출력하는데, 이러한 내용이 파이프를 통해 다음 프로세스인 'wc' 의 표준 입력으로 전달된다. 명령어 라인이 실행되면서 임시로 생기는 파이프의 존재는 쉽게 확인할 수 없다. 명령어 라인이 종료되면 사라지기 때문이다. 두 번째 명령어 라인은 'ls –l' 로 루트 디렉터리의 파일 리스트를 출력하고 이를 'wc' 의 표준 입력으로 전환한다.

```
$ ls / | wc -w
     20
$ ls -l / | wc -l
     20
$
```

명령어 라인에서 'l' 를 사용해서 임시로 만들어지는 파이프는 이름을 가지지 않는다. 말 그대로 임시로 만들어졌다가 사용이 끝나면 바로 사라지기 때문이다. 이름을 가지고 있는

파이프도 있다. 이를 '네임드 파이프(Named Pipe)' 또는 'FIFO'라고 부른다. 네임드 파이프는 파일로 존재하면서 두 개 이상의 프로세스가 이를 통해 메시지를 주고받을 수 있도록 해준다. 한쪽의 프로세스가 네임드 파이프를 개방하여 쓰기를 수행하고 다른 쪽의 프로세스는 네임드 파이프를 개방하여 읽기를 수행하는 식이다.

다음은 'mkfifo' 명령을 사용하여 네임드 파이프를 생성하고 두 개의 'cat' 프로세스가 이를 사용하여 메시지를 전달하는 예제다. 먼저 실행되는 'cat' 프로세스는 네임드 파이프로부터 데이터를 가져오기 위해 입력 방향 전환을 하면서 후면에서 실행된다. 아직은 네임드 파이프에 읽을거리가 없기 때문에 'cat' 프로세스는 네임드 파이프에 읽을거리가 생길 때까지 대기하게 된다.

이어서 실행되는 두 번째의 프로세스는 표준 출력을 네임드 파이프로 전환하면서 전면에서 실행된다. 나중에 실행된 'cat' 프로세스가 사용자로부터 입력받은 내용은 출력 전환이 되었기 때문에 네임드 파이프에 저장되고 후면에서 실행되는 'cat' 프로세스는 네임드 파이프에 읽을거리가 생겼기 때문에 이를 읽어서 표준 출력하게 된다.

```
$ mkfifo fifo
$ ls -l fifo
prw-r--r--    1 usp      student         0 Nov 19 03:22 fifo
$ cat < fifo &
[1] 9919
$ cat > fifo
apple is red
apple is red
banana is yellow
banana is yellow
[1]+  Done                    cat <fifo
$
```

위의 출력 결과에서 'mkfifo'를 사용하여 'fifo'라는 이름의 네임드 파이프를 생성하였다. 'fifo'는 파이프이기 때문에 파일의 유형을 보면 'p'라고 되어 있다. 그리고 프로세스 간의 메시지를 전달하기 위한 특수 파일이므로 크기가 0이다. fifo가 크기를 가지는 것을 직접 확인하는 것은 어렵다. 'cat 〉 fifo'를 실행하여 메시지를 입력하더라도 쓰기 작업이

바로 이루어지지 않는다. 'cat < fifo' 처럼 fifo로부터 메시지를 읽으려는 프로세스가 있을 때 비로소 fifo에 대한 쓰기 작업이 수행되고 fifo로 메시지를 쓰자마자 'cat < fifo' 가 이를 읽어서 가져가버리기 때문이다. 네임드 파이프를 적절히 사용하면 두 명의 사용자가 간단하게 채팅도 할 수 있다.

다음은 두 개의 터미널로 접속하여 네임드 파이프를 사용하여 각 터미널이 간단하게 서로 채팅하는 예제다. 파이프는 단방향으로 메시지를 전송하는 방법이므로 양쪽이 서로 메시지를 주고받기 위해 두 개의 네임드 파이프를 생성해야 한다. 첫 번째 터미널은 fifo1을 메시지 전송용으로 사용하고 fifo2를 메시지 수신용으로 사용한다. 두 번째 터미널은 fifo1을 메시지 수신용으로 사용하고 fifo2를 메시지 전송용으로 사용한다.

```
$ ll
total 0
prw-r--r--    1 usp      student         0 Nov 19 03:47 fifo1
prw-r--r--    1 usp      student         0 Nov 19 03:47 fifo2
```

```
$ cat < fifo2 &
[1] 20143
$ cat > fifo1
hello
hi~
nice to meet you~
me too~
```

```
$ cat < fifo1 &
[1] 20142
$ cat > fifo2
hello
hi~
nice to meet you~
me too~
```

위의 예에서 네임드 파이프를 하나만 만들어 채팅에 사용할 수는 없다. 하나의 파이프를 만들었다고 가정하면 두 개의 터미널은 동일한 네임드 파이프에 대해 입력과 출력을 동시 수행하려고 할 것이다. 이렇게 하면 한쪽의 프로세스가 다른 쪽의 프로세스가 자신의 메시지를 가져가기 전에 자기가 되가져오는 경우가 발생한다. 전송용으로 작성한 메시지를 상대방이 확실하게 수신하도록 하려면 메시지의 흐름을 생각하여 각 흐름마다 하나의 네임드 파이프를 만들어야 한다.

Section

04 소켓

시그널과 파이프가 동일한 시스템 내에 있는 프로세스 간의 통신을 위한 방법이라면 소켓을 이용한 통신 방법은 동일한 시스템 내에 있는 프로세스 간뿐만 아니라 서로 다른 시스템에 있는 프로세스 간에도 통신이 가능하게 한다. 소켓을 이용한 통신은 커널의 도움이나 파일 시스템을 사용하여 메시지를 주고받는 것이 아니라 네트워크 장치를 통해 메시지를 주고받는다. 대부분의 소켓을 이용한 통신은 서로 다른 시스템에 존재하는 프로세스 간의 통신을 위해서 선택된다.

❶ 연결형 모델과 비연결형 모델

소켓을 사용한 통신은 크게 연결형 모델과 비연결형 모델이 있다. 연결형 모델은 두 개의 프로세스가 메시지를 주고받기 전에 연결을 설정하는 작업이 필요하며 메시지를 모두 주고받은 다음에 연결을 해제하는 작업이 필요하다.

실제로 프로그램을 작성하는 단계에서는 몇 개의 함수를 호출해서 연결과 비연결 작업을 수행할 수 있다. 연결형 모델에서 두 개의 프로세스가 가진 각각의 소켓이 서로 연결되었으면 그 다음부터 전달되는 메시지는 연결이 설정된 상대측 소켓으로 전달된다. 만약, 동일한 소켓을 사용하여 다른 프로세스로 메시지를 보내고 싶다면 현재의 연결을 해제하고 새롭게 연결을 설정해야 한다.

연결형 모델의 장점은 응용 프로그램 레벨이 아니라 선택된 프로토콜 레벨에서 확실한 메시지의 전달을 보장한다는 것이다. 물리적인 네트워크의 사정에 따라 상대방에게 보낸 메시지가 전달되지 않을 수도 있는데, 연결형 모델에서는 이러한 경우 프로토콜 레벨에서 메시지를 재전송하는 방법으로 확실한 메시지 전송을 보장한다.

비연결형 모델은 메시지를 주고받기 전에 연결을 설정할 필요가 없다. 연결형 모델과 달리 동일한 소켓을 통해 여러 프로세스들에게 메시지를 번갈아 보낼 수 있다. 메시지를 보낼 때

마다 메시지를 수신할 상대방을 지정하기만 하면 된다. 비연결형 모델은 연결형 모델에 비해서 연결이라는 작업이라는 필요하지 않기 때문에 코드가 간단하지만 프로토콜 레벨에서 확실한 메시지 전송을 보장하지 않는다는 것이 단점이다. 따라서 비연결형 모델은 메시지가 확실히 전달되지 않아도 되는 통신에서 사용하거나 또는 응용 프로그램 레벨에서 수신을 확인하는 메시지를 전달하는 등의 부가적인 기법이 필요하다.

연결형 모델과 비연결형 모델에서 사용되는 대표적인 프로토콜은 TCP(Transmission Control Protocol)과 UDP(User Datagram Protocol)이다. 이외에도 다양한 프로토콜이 있으나 오늘날 인터넷 환경에서 대부분 TCP나 UDP를 사용하며 우리가 소켓을 이용하는 통신 프로그램을 작성하면서 선택할 수 있는 프로토콜도 대부분 TCP나 UDP뿐인 경우가 많다.

❷ 인터넷 주소

인터넷 주소(Internet Address)는 다른 말로 IP 주소라고 부르기도 하는데, 네트워크에서 컴퓨터 하나를 유일하게 식별하기 위한 주소를 의미한다. IP(Internet Protocol)는 주소를 이용하여 컴퓨터를 식별하는 프로토콜이다. IP 주소는 버전 4일 경우 4바이트의 크기를 가지는 주소로 사용자 입장에서는 편하게 '222.31.200.87'과 같은 문자열 형태로 표현한다. 그러나 실제로 통신을 수행하기 위해서는 보통 문자열 형식으로 표현되는 인터넷 주소를 바이너리로 변환해야 한다.

❸ 포트 번호

인터넷 주소가 네트워크에서 컴퓨터를 유일하게 식별하기 위한 것이라면 '포트(Port)'는 인터넷 주소로 선택된 컴퓨터 내에서 실행중인 통신 프로그램 중에서 하나를 선택하는 번호다. 포트 번호는 2바이트의 크기를 가지는데, 통신을 수행하기 전에 현재 시스템에서 사용하지 않는 번호 중 하나를 선택하여 사용하면 된다.

인터넷에서 널리 사용되는 서비스는 포트 번호를 미리 약속하여 사용한다. 포트 번호를 모를 경우, 원하는 서비스를 제공하는 통신 프로그램에 연결할 수 없기 때문이다. 물론, 어떻게든 포트 번호를 알아낼 수는 있겠지만 동일한 유형의 서비스가 컴퓨터마다 서로 다른 포

트 번호를 사용한다면 이러한 것을 일일이 확인하는 것도 번거로울 것이다. 널리 사용되는 서비스를 위해 미리 약속된 포트 번호를 잘 알려진(Well-known) 포트 번호라고 하는데, 예를 들면 다음과 같다.

```
daytime : 13       ftp : 21       telnet : 23       http : 80
```

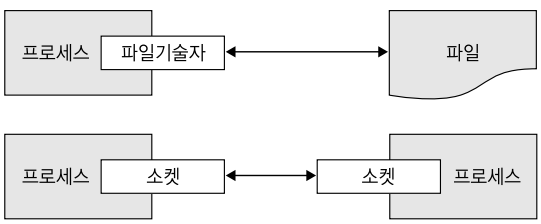
위에서 언급된 서비스가 항상 약속된 포트 번호를 사용해야 하는 것은 아니다. 현재 사용중이 아니라면 어떤 번호를 사용해도 무방하다. 하지만 이렇게 서비스별로 미리 번호를 약속해두면 통신 프로그램이 제공하는 서비스만으로도 포트 번호가 정해지므로 사용자가 통신 프로그램을 편하게 사용할 수 있게 된다.

❹ 소켓

소켓(Socket)은 파일 기술자와 같다고 볼 수 있다. 파일 기술자는 응용 프로그램이 파일을 사용하기 위해서 개방했을 때 얻는 번호로 이 번호를 통해서 어떤 파일을 어떤 상태로 개방했으며 읽기/쓰기 포인터가 어디를 가리키고 있는지 등의 정보를 알 수 있다.

소켓도 마찬가지다. 아래의 [그림 9-1]처럼 소켓이 파일 기술자와 다른 점은 파일 기술자의 너머에는 파일이 존재하지만 소켓은 자신과 마찬가지로 소켓을 가진 프로세스가 있다는 것이다. 반대쪽 프로세스 입장에서도 마찬가지로 소켓 너머에 자신처럼 소켓을 가진 프로세스가 있다. 파일 기술자로 통해서 파일로 데이터를 쓰거나 읽듯이 소켓을 통해서 상대 프로세스로 메시지를 쓰거나 읽을 수 있다.

소켓에는 상대 프로세스에 대한 주소와 포트번호 등의 정보가 담겨져 있다. 따라서 소켓이 생성되고 원하는 정보를 소켓에 연결했으면 그 다음부터는 파일로 데이터를 읽거나 쓰듯이 소켓을 통해서 메시지를 읽거나 쓰면 된다.



[그림 9-1] 파일 기술자와 소켓

연습문제

1 Ctrl + C 나 Ctrl + Z 와 같은 특정 입력들은 SIGINT 시그널이나 SIGSTP 시그널을 발생시킨다. 이와 같이 특정 시그널을 발생시키는 입력들을 조사하고 그 입력에 대응되는 시그널들을 정리하시오.

2 네임드 파이프 np를 만들고, 'ls -al > np' 명령어를 수행하고 다음을 확인하시오.

① 그 결과가 어떻게 되는가를 확인하시오.

② 그리고 다른 콘솔을 연결시켜 'cat < np' 명령어를 수행하고 그 결과가 어떻게 되는지 확인하시오.

③ 마지막으로 위의 명령어들을 수행했을 때 나타나는 현상을 보고 그 원인과 이유를 정리하시오.