네트워크 프로그래밍

20. 시그널

시그널

- 시그널이란?
  - 예상치 않은 이벤트 발생에 따른 일종의 소프트웨어 인터럽트
    - Ex) ctrl + c, ctrl + z, 자식 프로세스의 종료
  - 외부에서 프로세스에게 전달할 수 있는 유일한 통로
-
- 인터럽트와의 차이점
  - 인터럽트는 H/W에 의해 OS로 전달됨
  - 시그널은 OS에 의해 프로세스로 전달됨

    소프트웨어 인터럽트라고도 불린다
- 시그널 값의 확인
  - /usr/include/signal.h
  - /usr/include/bits/signum.h

/usr/include/bits/signum.h

```
#define         SIGHUP          1               /* hangup */
#define         SIGINT          2               /* interrupt , ctrl + c */
#define         SIGQUIT         3               /* quit */
#define         SIGILL          4               /* illegal instruction (not reset when caught) */
#define         SIGTRAP         5               /* trace trap (not reset when caught) */
#define         SIGIOT          6               /* IOT instruction */
#define         SIGABRT         6               /* used by abort, replace SIGIOT in the future */
#define         SIGEMT          7               /* EMT instruction */
#define         SIGFPE          8               /* floating point exception */
#define         SIGKILL         9               /* kill (cannot be caught or ignored) */
#define         SIGBUS          10              /* bus error */
#define         SIGSEGV         11              /* segmentation violation */
#define         SIGSYS          12              /* bad argument to system call */
#define         SIGPIPE         13              /* write on a pipe with no one to read it */
#define         SIGALRM         14              /* alarm clock */
#define         SIGTERM         15              /* software termination signal from kill */
#if defined(__rtems__)
#define         SIGURG          16              /* urgent condition on IO channel */
#define         SIGSTOP         17              /* sendable stop signal not from tty */
#define         SIGTSTP         18              /* stop signal from tty */
#define         SIGCONT         19              /* continue a stopped process */
#define         SIGCHLD         20              /* to parent on child stop or exit */
#define         SIGCLD          20              /* System V name for SIGCHLD */
#define         SIGTTIN         21              /* to readers pgrp upon background tty read */
#define         SIGTTOU         22              /* like TTIN for output if (tp->t_local&LTOSTOP) */
#define         SIGIO           23              /* input/output possible signal */
#define         SIGPOLL         SIGIO           /* System V name for SIGIO */
#define         SIGWINCH 24     /* window changed */
#define         SIGUSR1 25      /* user defined signal 1 */
#define         SIGUSR2 26      /* user defined signal 2 */
```

시그널 처리

- 프로세스에 시그널이 전달되면, 다음 네 가지 상황 중 하나가 발생
  - 시그널이 무시된다.
    - 프로세스는 시그널이 도착한 것을 알지 못한다.
  - 프로세스가 강제로 종료된다.
  - 프로세스 실행이 인터럽트되며, 이후에 프로그램이 지정한 시그널 처리 루틴(signal-handling routine)이 실행된다.
  - 시그널이 블로킹된다.
    - 프로그램이 시그널을 허용할 때까지 아무런 영향 X

보통 signal handler가 돌고있을경우
잠깐 시그널보내는걸 멈춰달라고 블럭하는것,
핸들러가 수행이 종료되면 블럭을 해제하는것이 일반적이다.

종류	이벤트	기본동작
SIGALARM	알람 타이머의 만료	프로그램 종료
SIGCHLD	자식 프로세스가 종료됨	시그널 무시
SIGINT	인터럽트 문자 (Control-C) 입력됨	프로그램 종료
SIGIO	소켓에 대해 I/O가 준비됨	시그널 무시
☆ SIGPIPE	종료된 소켓에 쓰기를 시도하려 할 때 발생	프로그램 종료

어떤 시그널은 기본 동작을 바꿀 수 없으며 시그널 자체가 블록(block)되지도 않는다. 하지만 이 사실은 위에 설명한 다섯 개의 시그널의 경우에는 해당되지 않는다.

SIGSTOP, SIGKILL

sigaction()을 이용한 특정 시그널에 대한 기본 동작의 변경

```
#include <signal.h>

int sigaction(int whichSignal, const struct sigaction * newAction,
              struct sigaction * oldAction)
```

성공 시 0, 실패 시 -1을 반환

```
struct sigaction {
          void     (*sa_handler)( int );   /* 시그널 핸들러 */
          sigset_t sa_mask;                /* 핸들러 실행 시 블록될 시그널들 */
          int      sa_flags;               /* 시그널의 설정 변경 */
}
```

여기서 sa_flags는 다루지 않고 넘어간다

```
#include <signal.h>

int sigemptyset(sigset_t *set)
int sigfillset(sigset_t *set)
int sigaddset(sigset_t *set, int whichSignal)
int sigdelset(sigset_t *set, int whichSignal)
```

위 4개의 함수는 모두 성공 시 0, 실패 시 -1을 반환

sigaction()을 이용한 특정 시그널에 대한 기본 동작의 변경

무시

```
#include <signal.h>
main() {
        struct sigaction new_act, old_act;
        new_act.sa_handler = SIG_IGN;
        sigaction(SIGINT, &new_act, &old_act);
        /* do processing */
        new_act.sa_handler = SIG_DFL;
        sigaction(SIGINT, &new_act, &old_act);
}
```

원래 시그널에 해당하는 동작으로 변경

```
#include <signal.h>

void on_intr() {
        unlink(tempfile);
        exit(1);
}

main() {
        struct sigaction new_act, old_act;
        new_act.sa_handler = SIG_IGN;
        sigaction(SIGINT, &new_act, &old_act);
        if (old_act.sa_handler != SIG_IGN) {
            new_act.sa_handler = on_intr;
            sigaction(SIGINT, &new_act, &old_act);
        }
        /* do processing */
        exit(0);
}
```

sigaction()을 이용한 특정 시그널에 대한 기본 동작의 변경

```
struct sigaction {
           void     (*sa_handler)( int );   /* 시그널 핸들러 */
           sigset_t sa_mask;                /* 핸들러 실행 시 블록될 시그널들 */
           int      sa_flags;               /* 기본 동작으로 변경할 플래그들 */
}
```

- sa_mask 필드는 whichSignal을 처리하는 중에 발생하는 시그널 중 블록되어야 할 시그널을 나타냄.
  - sa_handler가 SIG_IGN이나 SIG_DFL이 아닐 경우에만 해당.

시그널은 비트단위로 있는지 확인하도록 만들어놨기때문에
A라는 시그널 처리함수가 돌고있을때 2개의 A가 들어오면
>>늦게 들어온 2개의 A는 버려진다

대기열이 없는 시그널

특정핸들러가 돌고있다

- 다른 시그널의 처리로 인해 전달 중인 시그널이 블록 된 경우에는 어떻게 될까?
  - 시그널의 전달은 핸들러의 실행이 완료될 때까지 지연된다.
    - 펜딩(pending, 계류중인) 시그널

- 시그널은 큐(queue)에 대기 되지 않고 계류(pending) 되거나 또는 버려지거나 둘 중의 하나라는 것에 주목하자.

- 만약 시그널이 처리 되는 도중에 똑같은 종류의 시그널이 두 개 이상 도착한다면, 시그널 핸들러는 기존에 진행하던 핸들러의 실행을 마치고 나서 단 한번만 더 실행된다.

블럭과 펜딩의 차이
1.블럭:프로그래머 입장에서
핸들러 돌고있을때 다른시그널 읽지말라고 막는것

2.펜딩:시그널 입장에서 날아왔는데
본인이 막혀있어서 유보된상태인것

시그널로 인해 인터럽트된 시스템 호출

- 만약 recv()나 connect()와 같은 소켓 함수 호출로 인해 실행이 블록되어 있는 프로그램에 시그널이 전달되는 경우를 생각해보자.
  - 시그널을 인지하고 이를 처리하는 프로그램들은 블록되어 있던 시스템 함수가 반환하는 이러한 에러값(예를 들어, EINTR)에 대한 대비를 해야 한다.
  - 인터럽트된 시스템 호출은 errno를 EINTR로 설정하고 -1을 반환
    - 프로그램이 인터럽트된 함수를 재시작해야만 함.

문제가 있어서 리턴했고(-1) 에러넘버가 EINTR가 되었으니까
〉〉다시호출해야한다.

각 함수별로 man페이지 확인해서
이 상황마다 다시 호출할 수 있도록 호출!

read()의 재시작

- 시그널에 의해 인터럽트 되면 스스로 재시작

```
#include <errno.h>
#include <unistd.h>
ssize_t r_read(int fd, void *buf, size_t size) {
        ssize_t retval;
        while (retval = read(fd, buf, size), retval == -1 && errno == EINTR) ;
        return retval;
}
```

- **EINTR** The call was interrupted by a signal before any data was read
- POSIX allows a **read**() that is interrupted after reading some data to return the number of bytes already read.

write()의 재시작

- 시그널에 의해 인터럽트됐을 때 스스로 재시작하고 요청한 모든 데이터를 씀

```
#include <errno.h>
#include <unistd.h>
ssize_t r_write(int fd, void *buf, size_t size) {
        char *bufp;
        size_t bytestowrite;
        ssize_t byteswritten;
        size_t totalbytes;
        for (bufp = buf, bytestowrite = size, totalbytes = 0; bytestowrite > 0; bufp += byteswritten, bytestowrite -= byteswritten) {
                byteswritten = write(fd, bufp, bytestowrite);
                if ((byteswritten == -1) && (errno != EINTR))
                        return -1;
                if (byteswritten == -1)
                        byteswritten = 0;
                totalbytes += byteswritten;
        }
        return totalbytes;
}
```

1바이트도 안썼는데 인터룹트가 걸렸을때

- **EINTR** The call was interrupted by a signal before any data was written.
- If a **write**() is interrupted by a signal handler before any bytes are written, then the call fails with the error **EINTR**; if it is interrupted after at least one byte has been written, the call succeeds, and returns the number of bytes written.

SIGPIPE

- A write() to a socket that has received an RST generates SIGPIPE. If the process does nothing with this signal, its default action terminates the process. If the process ignores the signal, write() returns an error of EPIPE.

- 비정상 연결 종료 후
  - 서버가 첫 읽기를 시도하면 소켓은 0 (or -1 with ECONNRESET)을 반환
  - 첫 전송을 시도하면 그 순간 SIGPIPE 시그널이 프로그램으로 전달 (동기적 전송)

ECONNRESET

극단적인 예를 들면
〉〉클라가 급한종료를 했는데
서버가 클라한테 자료주려고 하다가
죽을수도 있음

〉〉
- 서버는 클라이언트가 사라졌다 하더라도 기존 자원으로 계속적인 서비스를 유지하기 위해 SIGPIPE를 항상 처리해야 한다.