# 리눅스 시스템 프로그래밍

day6 Inter Process Communication
- IPC란
- Signal


고 강 태

010-8269-3535

**james@thinkbee.kr**

**https://www.linkedin.com/in/thinkbeekr/**

**http://www.facebook.com/gangtai.goh**

# 프로세스 사이 통신

# IPC

## Inter-process communication

프로세스들은 여러 가지 필요에 의해서 서로 데이터를 주고 받는다.

- 주고받는 데이터의 크기가 다양하다.
- 프로세스가 동일한 시스템에 있거나 서로 다른 시스템에 있다.
- 통신의 주체인 양쪽 프로세스가 사용자 프로세스일 수 있고 시스템의 커널일 수도 있다.

프로세스가 수행할 통신의 유형에 따라 적절한 방법이 선택되어 데이터를 주고받는데 사용된다.

- 시그널(signal), 파이프(pipe), 소켓(socket)

# IPC 역사

80년대 멀티 태스킹을 위한 통신 기법으로 발전
SysV IPC:
- Shared memory
- Semaphore
- Message Queue
- 이들은 X/Open System Interface에 통합후 XSI IPC라 함.

POSIX는 새로운 IPC 도입:
- Posix Semaphore
- Posix Shared memory
- Posix Message Queue

# 수많은 IPC가 있다

COMMUNICATIONs:

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
- - Stream vs Datagram (vs Seq. packet)
- - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
- Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
- - File vs Anonymous
- Cross-memory attach
- - proc_vm_readv() / proc_vm_writev()

SIGNALs:

- Signals
  standard, Realtime

SYNCHRONIZATIONs:

- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

# IPC 란?

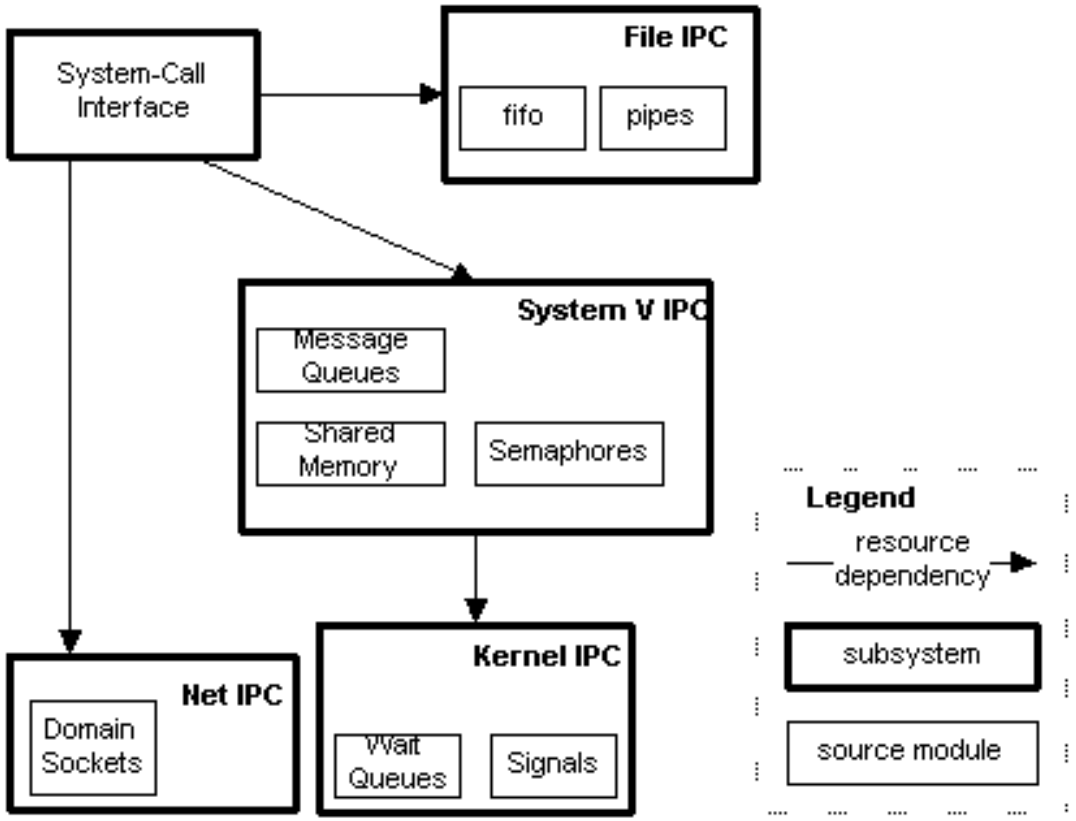
IPC(Inter Process Communication)은

pipe,socket,signal 통해서 부모 <—> 자식 프로세스는 데이터를 주고 받는다.

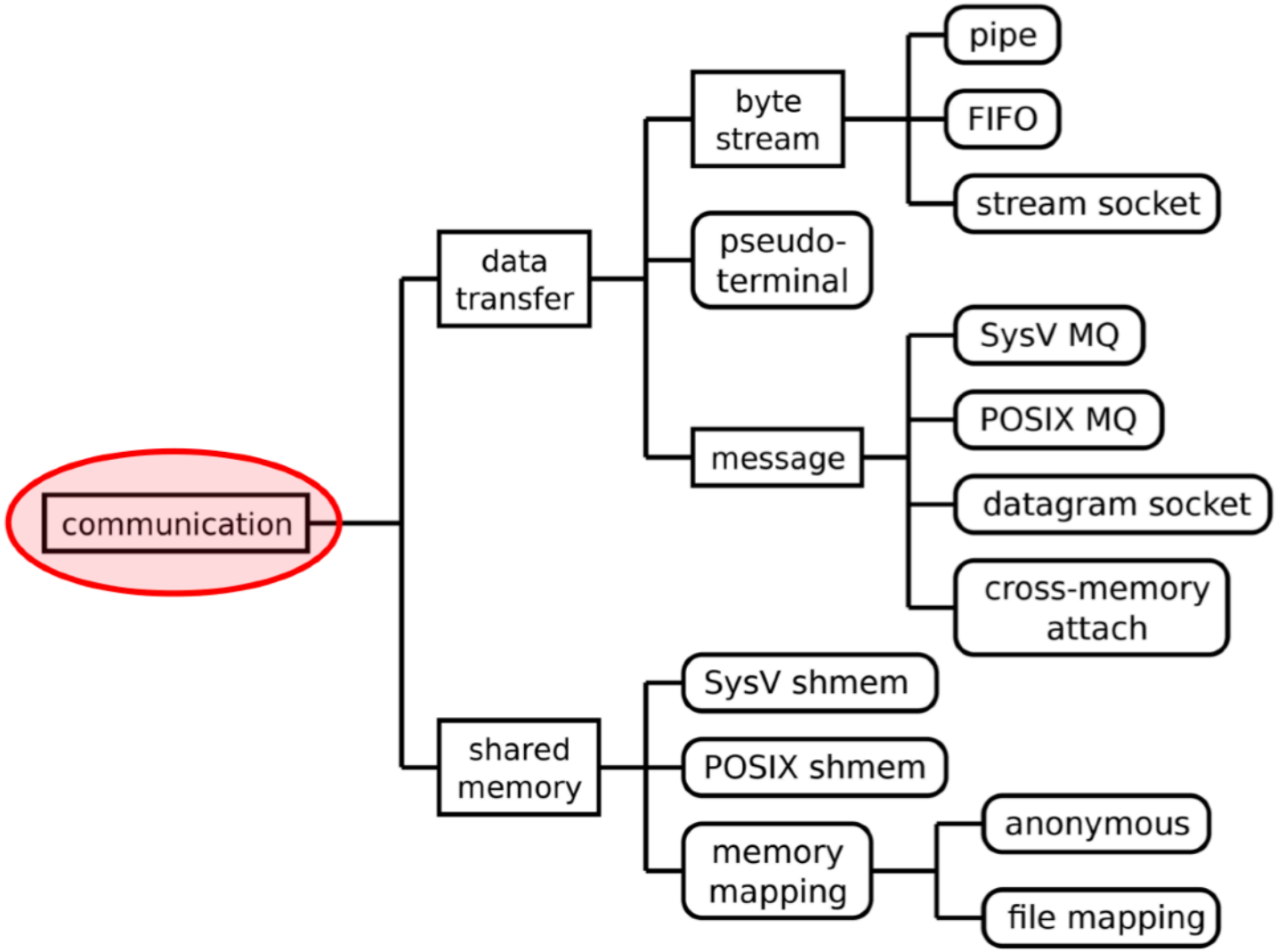
- 프로세스간 자원 공유
- 공유 자원에 대한 접근 설정

다양한 IPC가 존재

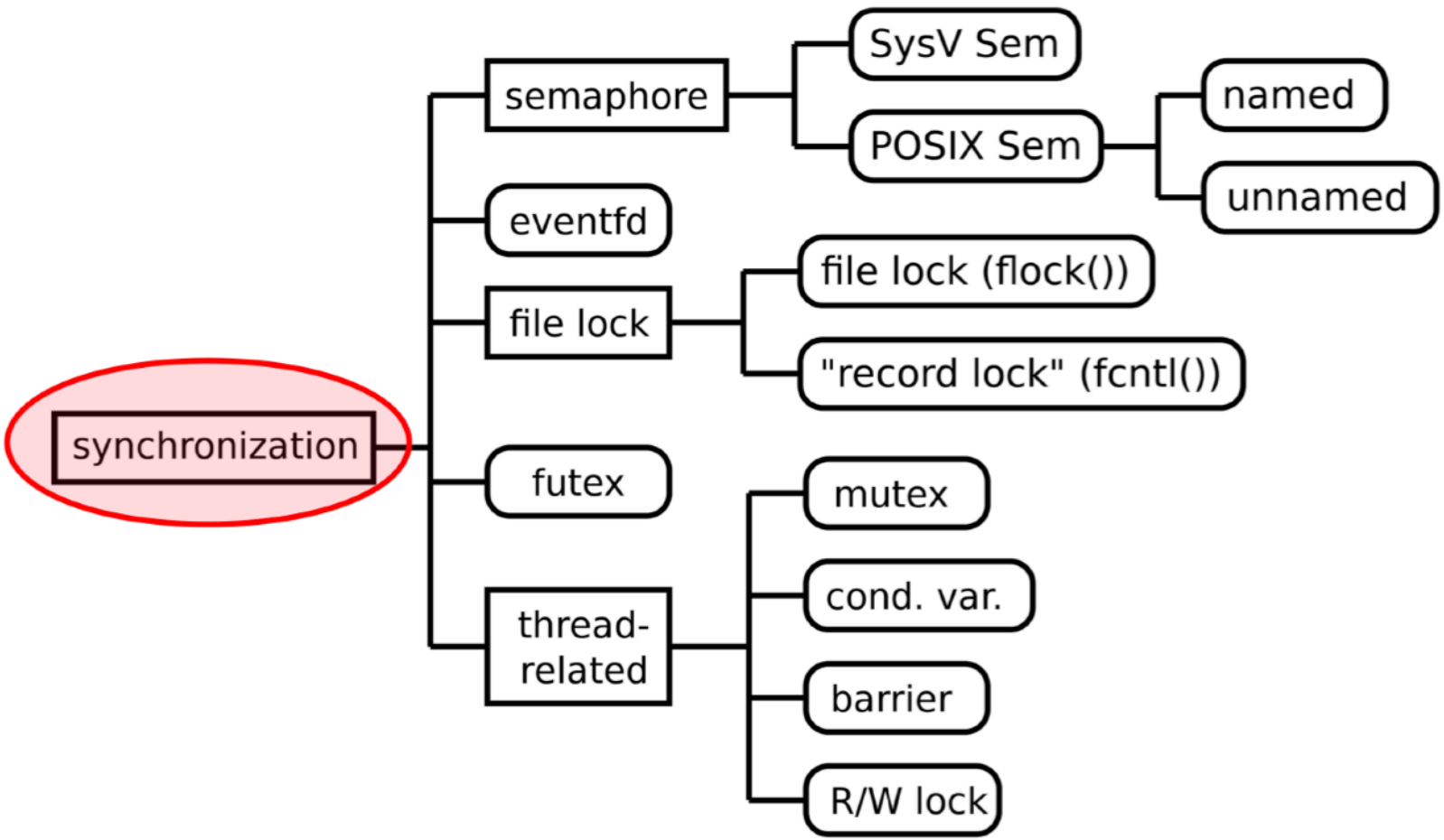
- 파이프(익명 파이프)
- 네임드 파이프
- 메시지 큐
- 공유 메모리
- 메모리 맵
- 시그널
- 세마포어
- 유닉스 도메인 소켓



# Communication



# Synchronization



# IPC - 시그널

시그널 (signal)

특정 이벤트가 발생했을 때 프로세스에게 전달하는 신호

- 연산 오류 발생, 자식 프로세스의 종료, 사용자의 종료 요청 등
- 굉장히 작은 값이다.
- 인터럽트(interrupt)라고 부르기도 한다.

시그널은 여러 종류가 있고 각각에 유일한 번호가 붙여져 있다.

- 프로그램 내에서는 매크로 상수를 사용한다.
- 예1) Ctrl + C를 누를 때 SIGTERM이 전달된다.
- 예2) kill 명령을 사용하면 해당 프로세스에게 SIGTERM이 전달된다.

# IPC - 시그널

## 시그널을 수신한 프로세스의 반응

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.

## 시그널 핸들러(signal handler)

프로세스가 특정 시그널을 포착했을 때 수행해야 할 별도의 함수

프로세스는 시그널을 포착하면 현재 작업을 일시 중단하고 시그널 핸들러를 실행한다. 시그널 핸들러의 실행이 끝나면 중단된 작업을 재개한다

- 시그널 종류

"/usr/include/asm/signal.h"  또는 교재 참고

# IPC - 시그널

실습 : 프로세스에게 시그널을 보내는 간단한 방법

- [Ctrl + C]나 [Ctrl + Z]를 눌러서 전면에서 실행 중인 프로세스에게 SIGINT나 SIGSTP 시그널을 보낸다.
- 두 신호에 대한 프로세스의 반응은 무엇인가?

# IPC - PIPE

## 파이프 (pipe)

리눅스 이전에 유닉스 환경에서부터 요긴하게 사용된 프로스스 간 통신법

파이프(|)의 앞에 놓인 프로세스가 표준 출력하는 내용을 파이프의 뒤에 놓인 프로세스가 표준 입력으로 받아들인다.

## 파이프는 파일이다.

파이프는 특수한 형태의 파일을 경유지로 하여 두 개의 프로세스가 데이터를 주고받는다.

보통 앞에 놓인 프로세스는 파이프에 대해 쓰기만 수행하고, 뒤에 놓인 프로세스는 파이프에 대해 읽기만 수행한다.

임시 파이프(anonymous pipe)와 네임드 파이프(named pipe, FIFO)로 나뉜다.

# IPC - PIPE

- 임시 파이프 (anonymous pipe)

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

명령어 라인을 수행하기 위해 각각 한 개의 임시 파일이 만들어져서 파이프로 사용된다.

명령어 라인의 수행이 끝나면 파이프 역시 사라진다.

하나의 명령어 라인을 실행하기 위해 임시로 만들어지기 때문에 이름을 가지지 않고, 그 존재를 쉽게 확인할 수 없다.

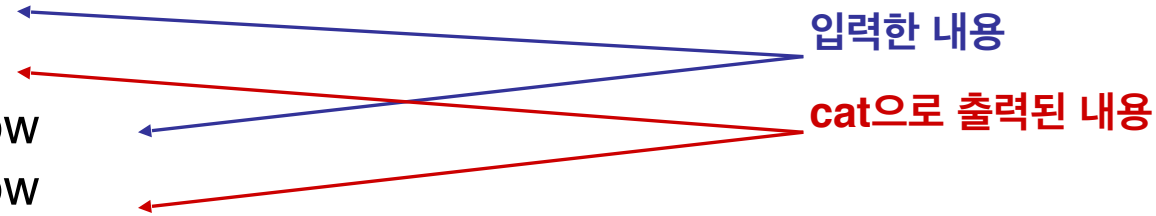
“ls” 프로세스는 파이프에 대해 쓰기 작업만 수행하고, “wc” 프로세스는 읽기 작업만 수행한다.

# IPC - PIPE

## 네임드 파이프 (named pipe, FIFO)

- “mkfifo” 명령을 사용하여 특수 파일인 네임드 파이프를 생성한다.
  - “ls” 명령으로 확인할 수 있다.
- 예제

```
$ 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
$
```



# IPC - PIPE

## 실습 : 네임드 파이프로 채팅하기

- 두 개의 네임드 파이프를 생성하여 하나는 A → B 프로세스로 나머지 하나는 B → A 프로세스로 데이터가 흐르도록 한다.
- 파이프는 흐름당 하나씩 생성하고 하나의 프로세스는 쓰기만 수행하고 나머지 하나는 읽기만 수행하도록 한다.

```
$ 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~
```

# IPC - Socket

## 네트워크 장치를 경유하는 통신 방법

서로 다른 시스템의 프로세스끼리 데이터를 주고 받을 수 있다.

시그널, 파이프는 동일 시스템 내에서만 가능하다.

## 연결형 모델과 비연결형 모델

| 연결형 모델 | 비연결형 모델 |
| --- | --- |
| 통신을 위해서 연결을 설정하는 작업이 필요하다. | 연결을 설정하는 작업이 필요 없다. |
| 통신이 끝난 뒤 연결을 해제하는 작업이 필요하다. | 연결 설정이 없으므로 연결 해제 역시 필요없다. |
| 통신 수행 시 확실한 메시지 전달이 프로토콜 수준에서 보장된다. | 메시지 전달이 확실하지 않다. 실패할 수도 있다. |
| TCP (transmission control protocol) | UDP (user datagram protocol) |

# IPC - Socket

- IP 주소
  - 네트워크 상에서 컴퓨터 하나를 유일하게 식별하기 위한 주소
  - 4바이트 바이너리로 표현된다. (IP version4)
  - 사용자 입장에서는 편하게 “222.31.200.87”과 같은 문자열 형태로 표현한다.
  - 프로세스는 바이너리로 된 IP 주소를 사용한다. (변환이 필요하다.)
  - IP (internet protocol)는 주소를 이용하여 컴퓨터를 식별하는 프로토콜이다.
  - 인터넷 주소 (internet address)

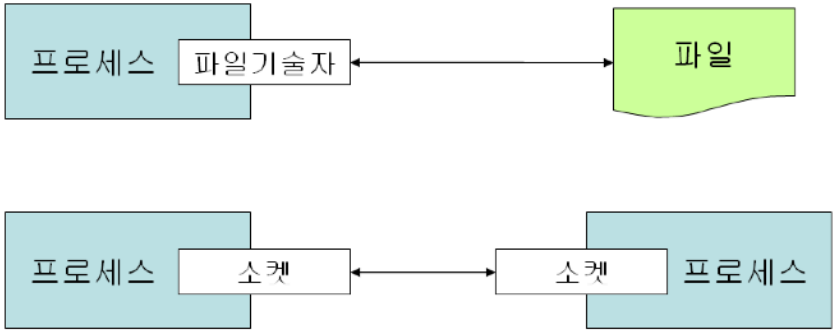
# IPC - Socket

- 포트 번호 (port)
  - 인터넷 주소로 선택된 컴퓨터 내에서 실행 중인 통신 프로그램 중 하나를 선택하기 위한 번호
  - 2바이트의 크기를 가진다.
  - 하나의 시스템 내에서 중복될 수 없다. (사실 중복될 수 있으나 이는 논외로 한다.)
  - 사용하지 않는 번호 중에 하나를 가져와 사용한다.
  - 잘 알려진 포트 (well-known port)
  - 잘 알려진 인터넷 서비스를 위해 미리 약속한 포트

| 서비스 | 포트 번호 |
|---|---|
| daytime | 13 |
| ftp | 21 |
| ssh | 22 |
| http | 80 |

# IPC - Socket

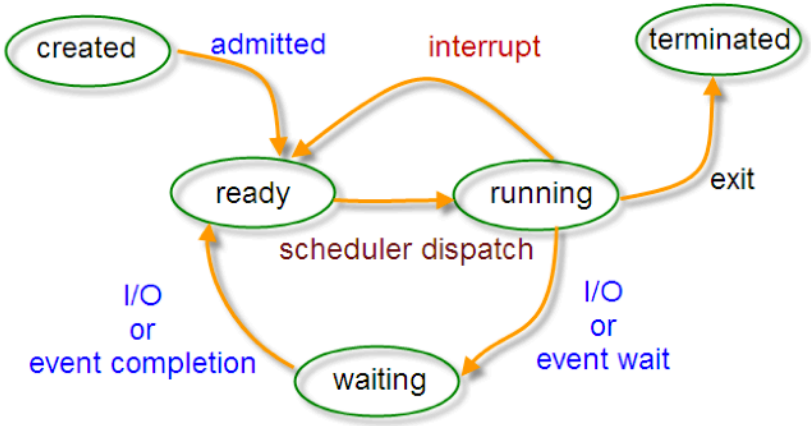
- 소켓은 파일 기술자와 같다.
  - 프로세스가 다른 프로세스와 데이터를 주고 받기 위해 필요하다.
  - 소켓과 파일 기술자가 다른 점은…
  - 파일 기술자 너머에는 파일이 있지만, 소켓 너머에는 자신처럼 소켓을 가진 프로세스가 있다.
  - 프로세스는 소켓에 데이터를 쓰거나, 소켓에서 데이터를 읽어서 다른 프로세스와 통신한다.
  - 소켓에는 상태 프로세스에 대한 인터넷 주소, 포트 번호가 등의 정보가 담겨져 있다.
  - 소켓 (socket)



# Signal 이란?

# Signal

특정 이벤트가 발생했을 때 프로세스에게 전달하는 신호
- 연산 오류 발생, 자식 프로세스의 종료, 종료 요청 등
- 인터럽트(interrupt)라고 부르기도 한다.



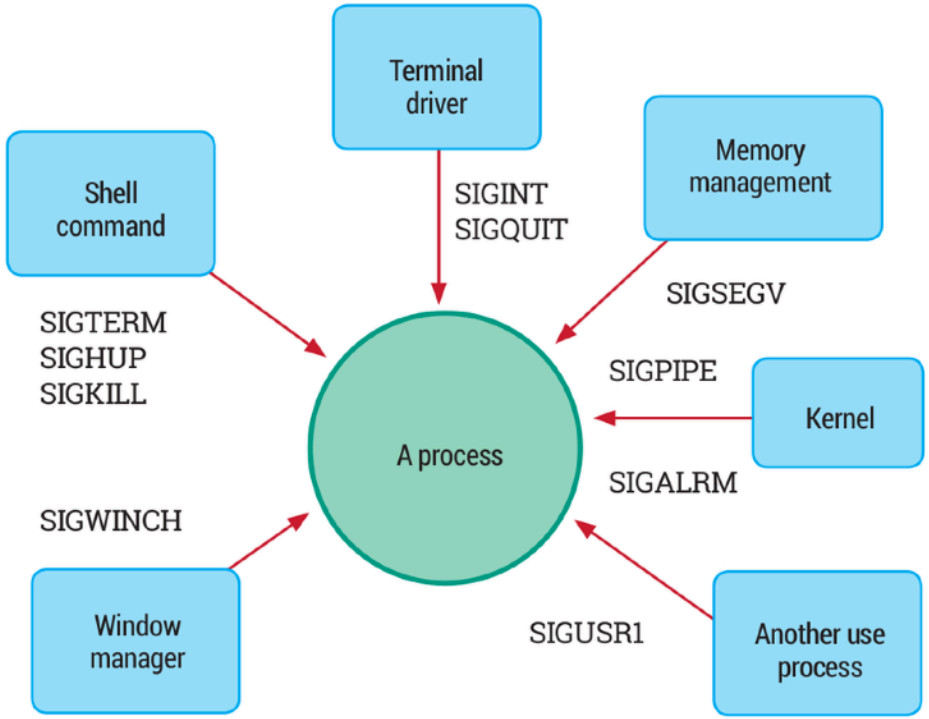

예1) [Ctrl + C]나 [Ctrl + Z]를 눌러서 전면에서 실행 중인 프로세스에게 SIGINT나 SIGSTP 시그널을 보낸다.

예2) kill 명령을 사용하면 해당 프로세스에게 SIGTERM이 전달된다.

```
$ kill -9 2352
```

# Signal

시그널은 여러 종류가 있고 각각에 유일한 번호가 붙여져 있다.




$ man signal

```
$ kill -l
 1) SIGHUP  2) SIGINT  3) SIGQUIT      4) SIGILL  5) SIGTRAP
 ...
```

# 명령으로 Signal 다루기

프로세스에 인터럽트(interrupt)를 발생시키는 명령:

1. 실행중 프로세스의 표준 입력에서 [Ctrl + C]나 [Ctrl + Z]
   - SIGINT, SIGSTP 시그널을 보낸다.

$ man signal

2. kill 명령으로 시그널을 해당 프로세스에게 인터럽트할 수 있다.

```
$ ps -ef
....
$ kill -9 2352
```

```
$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 ...
$ kill -SIGINT 2345
```

# 대표적 시그널

리눅스 시그널은 SIG 로 signal.h 에 정의

| **SIGKILL** | **프로세스를 강제 종료** |
|---|---|
| SIGALARM | 알람을 발생한다 |
| SIGSTP | 프로세스를 멈춰라 |
| SIGCONT | 멈춰진 프로세스를 움직이게 하라 |
| SIGINT | 프로세스에 인터럽트하라. Ctrl+C |
| SIGSEGV | 프로세스가 다른 메모리영역을 침범했다. |
| SIGPIPE | 파이프가 깨졌을 때 발생 |

# signal

시그널 발생 함수:

```
raise(3)        :스레드 호출 시그널 전송
kill(2)         :프로세스, 그룹, 모든 프로세스에 시그널 전송
killpg(3)       : 특정 프로세스 그룹 멤버들에 시그널 전송
pthread_kill(3): 특정 POSIX 스레드에 시그널 전송
tgkill(2)       : 특정 프로세스 안의 스레드에 시그널 전송
sigqueue(3)     : 큐에 특정 시그널에 실시간 시그널 전송
```

시그널 대기 함수:

```
pause(2)       : 시그널을 받을 때 까지 대기한다.
sigsuspend(2) : 시그널 매스크를 변경하고 마스크 없는 시그널 받을 때까지 대기
```

# Old 시그널 핸들러 함수

전달된 시그널을 처리하는 시그널 핸들러 함수로 전통적
- signal()

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
 - signum : 시그널 번호. SIGINT,  SIG_IGN, SIG_DFL 등등
 - handler: 사용자 선언 핸들러 함수
```

호환성 문제: SVR4, BSD, Linux 등 다른 작동 의미
**sigaction() 함수 권장**

# 시그널 처리 기본동작

## 시그널처리

- 각 시그널은 처리할 기본동작이 있다.(대부분 시그널 처리는 프로세스 종료)
- 어떤 시그널 처리 기본 동작은 시그널을 “무시(discard)”하는 것.
- 시그널 처리의 기본동작을 프로그래머가 변경 할 수 있음
  - 미리 정해진 함수를 수행하도록 할 수 있음
  - 시그널을 블록시킬 수 있음

## 시그널 수신시 처리 할 수 있는 동작

시그널수신시호출할함수(시그널핸들러)를등록

- 정해진 기본 동작 수행: 프로세스 종료 또는 시그널 무시
- 사용자가 지정한 작업 수행
  - **시그널 핸들러** 수행
  - 시그널 무시: 시그널을 받지 않은것 처럼 아무 영향을 받지 않음
  - 시그널 블록
- SIGKILL, SIGSTOP은 프로그래머가 변경 할 수 없음

## 시그널 대기 큐

- 시그널을 블록하면 해당 시그널이 프로세스에 도착했을때 시그널 큐에 들어감
- 이때 프로세스는 하던 동작을 계속하고 나중에 시그널을 처리 할 수 있음
  - 블록시킬 시그널은 bitmask 형태의 **시그널 블록 마스크**에 추가
  - 블록된시그널은누적되지않음

# signal.c (1)

day6/signal.c

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

static void sig_usr(int); /* one handler for both signals */

int main(void) {

    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
        perror("can't catch SIGUSR1");
        return -1;
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
        perror("can't catch SIGUSR2");
        return -1;
    }
    for (;;)
        pause();

    return 0;
}
```

# signal.c (2)

day6/signal.c

```
static void sig_usr(int signo) /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        fprintf(stderr, "received signal %d\n", signo);
    return;
}
```

```
$ ./signal
```

# pause

시그널이 전달될 때까지 대기한다

```
#include <unistd.h>

int pause(void);
```

| 반환값 | 항상 -1을 반환한다. |
|---|---|

- pause를 실행하면 프로세스는 임의의 시그널이 수신할 때까지 대기 상태가 된다.
- 아무 시그널이나 상관없다.
- 수신된 시그널이 프로세스를 종료시키는 것이라면 프로세스는 pause 상태에서 벗어나자마자 종료된다.
- 무시하도록 설정된 시그널에 대해서는 반응하지 않는다.
- 시그널 핸들러가 등록된 시그널이라면 시그널 핸들러를 실행하고 나서 pause 상태를 벗어난다.

# signal8.c

day6/signal8.c

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

main()
{
    printf("pause return %d\n", pause());
}
```

# signal8.c

day6/signal8.c

```c
#include <unistd.h>
#include <signal.h>

main() {
    struct sigaction act;

    sigfillset(&(act.sa_mask));
    act.sa_handler = handler;

    sigaction(SIGINT, &act, NULL);

    printf("pause return %d\n", pause());
}

void handler(int signum) {
    printf("\nSIGINT catched\n\n");
}
```

```
$./signal8

SIGINT cought

pause return -1
$
```

Ctrl+C를 입력한다.

# signal.c 실행

ps 명령으로 signal의 PID를 찾아 SIGUSR1, SIGUSR2 를 전송

```
5$ ps -ef |grep signal
root         968       1  0 Jun02 ?        00:00:00 /usr/bin/python3 /usr/share/
unattended-upgrades/unattended-upgrade-shutdown --wait-for-signal
qkboo      20301   19324  0 10:02 pts/0    00:00:00 ./signal
qkboo      20304   17630  0 10:03 pts/1    00:00:00 grep --color=auto signal
$ kill -SIGUSR1 20301
$ kill -SIGUSR2 20301
```

kill 명령으로 시그널 전송

```
$ ./signal
received SIGUSR1
received SIGUSR2
```

# Alarm

지정한 시간이 경과한 후에 자신에게 SIGALRM 시그널을 보낸다.

```
#include <unistd.h>

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

| | |
|---|---|
| *seconds* | 초 단위의 시간이다. |
| 반환값 | 지정한 시간 중 남은 시간을 반환한다. 0 이상의 값이다. |

- alarm 설정은 한번에 하나만 등록할 수 있다.
- 여러 개를 누적해서 등록할 수 없다.
- 마지막에 등록한 하나의 alarm만 유효하다.
- 이전에 등록된 alarm은 취소된다.
- 지금까지 사용했던 sleep 함수는 alarm을 사용하여 구현되었다.
- sleep과 alarm은 함께 사용하지 않는 것이 좋다.

# alarm()

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

static void sig_handler(int);

int counter = 0;
int main(void) {
    if (signal(SIGALRM, sig_handler) == SIG_ERR) {
        perror("can't catch SIGALRM");
        return -1;
    }
    alarm(1);
    while(1)
        ;
    // for (;;)        pause();
    return 0;
}

void sig_handler(int signo) {
    printf("Alarm count: %d !\n", counter++);
    alarm(1);
}
```

kill 명령?

# signal7.c

day6/signal7.c

```c
#include <unistd.h>
#include <signal.h>

void timeover(int signum)
{
    printf("\n\ntime over!!\n\n");
    exit(0);
}

main()
{
    char buf[1024];
    char *alpha = "abcdefghijklmnopqrstuvwxyz";

    int timelimit;
    struct sigaction act;

    act.sa_handler = timeover;
    sigaction(SIGALRM, &act, NULL);
```

sigaction 으로 구현

# signal7.c

day6/signal7.c

```
    printf("input timelimit (sec)..\n");
    scanf("%d", &timelimit);

    alarm(timelimit);

    printf("START!!\n > ");
    scanf("%s", buf);

    if(!strcmp(buf, alpha))
        printf("well done.. you succeed!\n");
    else
        printf("sorry.. you fail!\n");
}
```

# signal7.c

day6/signal7.c

```
$./signal7
input timelimit (sec)..
100
START!!
> abcdefghijklmnopqrstuvwxyz
well done.. you succeed!

$./signal7
input timelimit (sec)..
3
START!!
> abcdefghijk

time over!!

$
```

# Sigaction

# sigaction()

Posix의 시그널 핸들러 구현체

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
 - signum : 시그널 번호. SIGINT,  SIG_IGN, SIG_DFL 등등
 - act:  설치할 시그널 핸들러
 - oldact: 새 핸들러 설치전 시그널 핸들러 백업. 불필요시 NULL
 - return:
```

```
struct sigaction {
   void     (*sa_handler)(int);
   void     (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t   sa_mask;
   int        sa_flags;
   void     (*sa_restorer)(void);
};
```

http://man7.org/linux/man-pages/man2/signal.2.html

# struct sigaction

Posix의 시그널 핸들러 구현체

```
struct sigaction {
    void      (*sa_handler)(int)
     - 시그널 핸들러가 호출할 함수. 혹은 매크로 지정 SIG_DFL, SIG_IGN
       기본 행동 명시(동작:SIG_DFL, 무시:SIG_IGN)
    void      (*sa_sigaction)(int, siginfo_t *, void *);
     - 확장된 시그널 핸들러 사용시 호출할 함수
    sigset_t   sa_mask;
     - 시그널 블로킹 마스크가 저장되는 시그널 세트
    int        sa_flags;
     - 옵션 플래그
    void      (*sa_restorer)(void);
}
```

블로킹 마스크: sigfillset, sigemptyset, sigaddset, sigdelset 함수로 조작.

예) sa_mask가 1로 채워지만 SIGKILL, SIGSTOP 제외하고 모두 블록

# struct sigaction

- void (*sa_sigaction)(int, siginfo_t *, void *);

sa_handler 대신에 사용할 수 있다.
sa_handler에 비해 추가 정보를 알 수 있다.
sa_sigaction과 sa_handler 중에 하나만 사용한다.
sa_sigaction을 사용하려면 sa_flags를 SA_SIGINFO로 지정한다.

- sigset_t  sa_mask;

시그널 마스크 : 시그널 집합을 의미한다.
sa_mask에 등록된 시그널은 시그널 핸들러가 실행되는 동안 봉쇄된다.
봉쇄 (blocking)
무시가 아니라 시그널 핸들러 실행이 완료될 때까지 처리가 미뤄진다.
현재 처리 중인 시그널도 봉쇄된다.

# struct sigaction

int sa_flags;

시그널 처리 절차를 수정하는데 사용된다.

| 값 | 의미 |
| --- | --- |
| SA_SIGINFO | sa_handler 대신에 sa_sigaction을 선택한다. |
| SA_NOCLDSTOP | signum이 SIGCHLD일 때 자식 프로세스가 종료되거나 중단되더라도 부모 프로세스는 이를 알려고 하지 않는다. |
| SA_ONESHOT | signum에 대해서 시그널 핸들러를 최초에 한번만 실행하고 그 다음부터는 동일한 시그널에 대해서 SIG_DFL에 해당하는 기본적인 동작만 수행하게 된다. |
| SA_RESETHAND | SA_ONESHOT과 같다. |

# signal3.c

day6/signal3.c

```c
int num = 0;

main() {
    static struct sigaction act;

    void int_handle(int);

    act.sa_handler = int_handle;
    sigfillset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    while(1) {
        printf("i'm sleepy..\n");
        sleep(1);
        if(num >= 3)
            exit(0);
    }
}
```

```c
void int_handle(int signum)  {
    printf("SIGINT:%d\n", signum);
    printf("int_handle called %d times\n", ++num);
}
```

# signal3.c

day6/signal3.c

```
$ ./signal3
i'm sleepy..
SIGINT:2
int_handle called 1 times
i'm sleepy..
i'm sleepy..
SIGINT:2
int_handle called 2 times
i'm sleepy..
SIGINT:2
int_handle called 3 times
$
```

# signal4.c (1)

day6/signal4.c

sa_handler 를 사용한 예제

```c
#include <signal.h>
#include <unistd.h>

int num = 0;

main()
{
    static struct sigaction act;

    void int_handle(int);

    act.sa_handler = int_handle;
    sigfillset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);
```

# signal4.c (2)

day6/signal4.c

```
    while(1) {
        printf("i'm sleepy..\n");
        sleep(1);
        if(num >= 2) {
            act.sa_handler = SIG_DFL;
            sigaction(SIGINT, &act, NULL);
        }
    }
}
void int_handle(int signum) {
    printf("SIGINT:%d\n", signum);
    printf("int_handle called %d times\n", ++num);
}
```

sa_handler = SIG_IGN ???

```
$./signal4
i'm sleepy..
SIGINT:2
int_handle called 1 times
i'm sleepy..
SIGINT:2
int_handle called 2 times
i'm sleepy..
```

# Signal sets

# Signal Sets

시그널 세트 (signal set)

- 복수 개의 시그널(multiple signals)을 표현하는 데이터 형
- sigprocmask 함수 등에서 사용(블록될 시그널을 지정)

시그널 세트 데이터 형 (data type)

- 변수의 각 비트가 하나의 시그널을 지칭
- 31개까지의시그널은정수형변수로표현 (4.3+BSD)
- 시그널 개수가 많은 경우 sigset_t 형의 변수를 사 용

# 시그널 함수와 Signal Sets

시그널을 다루기 위해 필요한 시스템 호출/표준 라이브러리 함수

| 함수 | 의미 |
|---|---|
| sigemptyset | 시그널 집합을 시그널이 없는 비어 있는 상태로 초기화한다. |
| sigfillset | 시그널 집합을 모든 시그널이 포함된 상태로 초기화한다. |
| sigaddset | 시그널 집합에 특정 시그널을 추가한다. |
| sigdelset | 시그널 집합에서 특정 시그널을 삭제한다. |
| sigaction | 특정 시그널에 대한 프로세스의 행동을 설정한다. |
| sigprocmask | 봉쇄할 시그널의 목록을 변경한다. |
| kill | 특정 프로세스에게 특정 시그널을 전달한다. |
| raise | 자기 자신에게 특정 시그널을 전달한다. |
| alarm | 설정된 시간이 경과한 후에 자기 자신에게 시그널을 전달한다. |
| pause | 시그널이 도착할 때까지 대기 상태가 된다. |

# Signal Sets

시그널을 다루려면 시그널 집합을 만들어야 한다. sigemptyset, sigfillset, sigaddset, sigdelset, sigismember 는 시그널 집합을 생성하거나 조작한다.

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

| | |
|---|---|
| *set* | sigset_t형의 시그널 집합이다. |
| *signum* | 시그널 번호이다. |
| 반환값 | 호출이 성공하면 0을 반환하고, 실패하면 -1을 반환한다. 단 sigisnumber는 호출이 성공하면 1이나 0을 반환하고 실패하면 -1을 반환한다. |

# Signal Sets

시그널 집합에 지정한 시그널이 포함되어 있는지를 검사한다.

- sigemptyset(set)

  set으로 주어진 시그널 집합을 아무런 시그널도 포함되어 있지 않은 비어 있는 상태로 초기화한다.

- sigfillset(set)

  sigemptyset와는 반대로 모든 시그널이 포함된 상태로 시그널 집합을 초기화한다.

- sigaddset, sigdelset

  시그널 집합에서 지정한 시그널을 추가하거나 제거한다.

- sigismember

# Signal Sets

sigset_t 조작 함수

```
int sigemptyset(sigset_t *set);
 - 모두 비우는 함수. set를 모두 0으로 만든다.
int sigaddset(sigset_t *set, int signo);
 - 시그널 세트 특정 시그널 번호에 1개 비트 채운다.
int sigdelset(sigset_t *set, int signo);
 - 시그널 세트 특정 시그널 번호에 1개 비트 지운다.
int sigfillset(sigset_t *set);
 - 모든 시그널 채운다.
int sigismember(const sigset_t *set, int signo);
```

보통  채울 시그널이 많으면 sigfillset로 채우고 sigemptyset으로 채우지 않을 시그널을 뺀다.

채울 시그널이 적으면 sigemptyset으로 모두 비우고 sigaddset으로 채울 시그널만 지정한다.

# 실습: Signal Sets

day6/posix_sig_basic.c

시그널 핸들러 설치

```c
#include <signal.h>

void sa_handler_usr(int signum);

int main() {
    struct sigaction sa_usr1;
    struct sigaction sa_usr2;

    memset(&sa_usr1, 0, sizeof(struct sigaction));
    sa_usr1.sa_handler = sa_handler_usr;
    sigfillset(&sa_usr1.sa_mask);

    memset(&sa_usr2, 0, sizeof(struct sigaction));
    sa_usr2.sa_handler = sa_handler_usr;
    sigemptyset(&sa_usr2.sa_mask);

    sigaction(SIGUSR1, &sa_usr1, NULL); // filled sa_mask
    sigaction(SIGUSR2, &sa_usr2, NULL); // empty sa_mask
```

sa_usr1핸들러가 실행중 일 때는, 블로킹되어 핸들러 종료된 다음 전달

sa_usr2핸들러가 실행중 일 때는, 즉시 전달되어 처리.

# 실습: Signal Sets

day6/posix_sig_basic.c

시그널 핸들러 설치

```
    printf("[MAIN] SIGNAL-Handler installed, pid(%d)\n",
getpid());
    for (;;)
    {
        pause();
        printf("[MAIN] Recv SIGNAL...\n");
    }
    return EXIT_SUCCESS;
}
```

```
void sa_handler_usr(int signum) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("\tSignal(%s):%d sec.\n",
               signum == SIGUSR1 ? "USR1" : "USR2", i);
        sleep(1);
    }
}
```

# 실습: Signal Sets

day6/posix_sig_basic.c

실행

```
$ ./posix_sig_basic
[MAIN] SIGNAL-Handler installed, pid(8154)
[MAIN] Recv SIGNAL...
        Signal(USR2):0 sec.
        Signal(USR2):1 sec.
        Signal(USR2):2 sec.
        Signal(USR1):0 sec.
        Signal(USR1):1 sec.
        Signal(USR1):2 sec.
```

sa_usr2는 블로킹이 풀려 즉시 핸들러 전달
- 중복되어 보내도 처리를 완료한다.
sa_usr1은 블로킹 되어 핸들러 처리후 전달

```
$ kill -SIGUSR2 8154
$ kill -SIGUSR1 8154
```

sa_usr2는 블로킹이 풀려서 여러번 시그널을 보내도 인터럽트가 안된다.

# 실습: Signal Sets

day6/posix_sig_basic.c

SIGUSR1, SIGUSR2 이외 다른 시그널?

```
C$ ./posix_sig_basic
[MAIN] SIGNAL-Handler installed, pid(8268)
        Signal(USR1):0 sec.
        Signal(USR1):1 sec.
        Signal(USR1):2 sec.
        Signal(USR1):3 sec.
        Signal(USR1):4 sec.
        Signal(USR1):5 sec.
        Signal(USR1):6 sec.
        Signal(USR1):7 sec.
        Signal(USR1):8 sec.
        Signal(USR1):9 sec.
Illegal instruction: 4
```

sa_usr1은 블로킹 되어 핸들러 처리한 후에야 시그널을 받는다.

```
$ kill -SIGUSR1 8268
$ kill -SIGILL 8154
```

처음 SIGUSR1 시그널 이후 다른 시그널은 인터럽트 된다.

# 실습: Signal Sets

day6/posix_sig_basic.c

NODEEP 플래그

```c
#include <signal.h>

void sa_handler_usr(int signum);

int main() {
    struct sigaction sa_usr1;
    struct sigaction sa_usr2;

    memset(&sa_usr1, 0, sizeof(struct sigaction));
    sa_usr1.sa_handler = sa_handler_usr;
    sigfillset(&sa_usr1.sa_mask);

    memset(&sa_usr2, 0, sizeof(struct sigaction));
    sa_usr2.sa_handler = sa_handler_usr;
    sa_usr2.sa_flags = SA_NODEFER;
    sigemptyset(&sa_usr2.sa_mask);

    sigaction(SIGUSR1, &sa_usr1, NULL); // filled sa_mask
    sigaction(SIGUSR2, &sa_usr2, NULL); // empty sa_mask
}
```

SA_NODEFER은 시그널이 중복된다

# 실습: Signal Sets

day6/posix_sig_basic.c

SIGUSR1, SIGUSR2 이외 다른 시그널?

```
$ ./posix_sig_basic
[MAIN] SIGNAL-Handler installed, pid(8345)
        Signal(USR2):0 sec.
        Signal(USR2):1 sec.
        Signal(USR2):2 sec.
        Signal(USR2):0 sec.
        Signal(USR2):1 sec.
        Signal(USR2):2 sec.
        Signal(USR2):0 sec.
        Signal(USR2):0 sec.
        Signal(USR2):0 sec.
        Signal(USR2):0 sec.
        Signal(USR2):1 sec.
```

sa_usr1은 블로킹 되어 핸들러 처리한 후에야 시그널을 받는다.

```
$ kill -SIGUSR1 8268
$ kill -SIGILL 8154
```

처음 SIGUSR2 시그널 이후 다른 시그널은 인터럽트 된다.

# kill, raise

kill은 특정 프로세스나 프로세스 그룹에게 지정한 시그널을 전달하고, raise는 자기 자신에게 지정한 시그널을 전달한다.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);

#include <signal.h>
int raise(int sig);
```

| | |
|---|---|
| *pid* | 프로세스의 식별 번호이다. |
| *sig* | 시그널 번호이다. |
| 반환값 | 호출이 성공하면 0을 반환하고, 실패하면 -1을 반환한다. |

# kill, raise

## kill의 pid의 값에 따른 의미

| pid | 의미 |
|---|---|
| pid > 0 | 프로세스의 식별 번호이다. 해당 프로세스에게만 시그널을 보낸다 |
| pid = 0 | 자신과 같은 그룹에 있는 모든 프로세스에게 시그널을 보낸다. |
| pid = -1 | 자신과 같은 그룹에 있는 모든 프로세스에게 시그널을 보낸다. 단, 프로세스 식별 번호가 1인 프로세스를 제외한다. |
| pid < -1 | 프로세스 그룹 식별 번호가 -pid인 모든 프로세스에게 시그널을 보낸다. pid가 -100이라면 그룹 식별 번호가 100인 모든 프로세스를 의미한다. |

## kill과 raise

raise(sig);
는 아래와 같다.
kill(getpid(), sig);

# signal6.c

day6/signal6.c

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

main() {
    pid_t pid;
    int count = 5;

    if((pid = fork()) > 0) {
        sleep(2);
        kill(pid, SIGINT);
        raise(SIGINT);
        printf("[parent] bye!\n");
    }
    else if(pid == 0) {
        while(count)
        {
            printf("[child] count is %d\n", count--);
            sleep(1);
        }
    }
    else
        printf("fail to fork\n");
}
```

```
$./signal6
[child] count is 5
[child] count is 4
$
```

# signal1.c (1)

day6/signal1.c

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

void handler(int signum);
int flag = 5;

main() {
        struct sigaction act;
        sigset_t set;

        sigemptyset(&(act.sa_mask));
        sigaddset(&(act.sa_mask), SIGALRM);
        sigaddset(&(act.sa_mask), SIGINT);
        sigaddset(&(act.sa_mask), SIGUSR1);

        act.sa_handler = handler;
        sigaction(SIGALRM, &act, NULL);
        sigaction(SIGINT, &act, NULL);
        sigaction(SIGUSR1, &act, NULL);

        printf("call raise(SIGUSR1) before blocking\n");
        raise(SIGUSR1);
```

# signal1.c (2)

day6/signal1.c

```
        sigemptyset(&set);
        sigaddset(&set, SIGUSR1);
        sigprocmask(SIG_SETMASK, &set, NULL);

        while(flag) {
                printf("input SIGINT [%d]\n", flag);
                sleep(1);
        }

        printf("call kill(getpid(), SIGUSR1) after blocking\n");
        kill(getpid(), SIGUSR1);

        printf("sleep by pause.. zzZZ\n");
        printf("pause return %d\n", pause());

        printf("2 seconds sleeping..zzZ\n");
        alarm(2);
        pause();
}
```

# signal1.c (3)

day6/signal1.c

```
void handler(int signum)
{
        flag--;

        switch(signum) {
        case SIGINT:
                printf("SIGINT(%d)\n", signum);
                break;
        case SIGALRM:
                printf("SIGALRM(%d)\n", signum);
                break;
        case SIGUSR1:
                printf("SIGUSR1(%d)\n", signum);
                break;
        default:
                printf("signal(%d)\n", signum);
        }
}
```

# signal1.c (4)

```
$ ./signal1
call raise(SIGUSR1) before blocking
SIGUSR1(10)
input SIGINT [4]
input SIGINT [4]
SIGINT(2)
input SIGINT [3]
SIGINT(2)
input SIGINT [2]
input SIGINT [2]
SIGINT(2)
input SIGINT [1]
SIGINT(2)
call kill(getpid(), SIGUSR1) after blocking
sleep by pause.. zzZZ
SIGINT(2)
pause return -1
2 seconds sleeping..zzZ
SIGALRM(14)
$
```

# signal2.c

day6/signal2.c

```c
#include <signal.h> #include <unistd.h>

main() {
    sigset_t set;
    int result;

    sigemptyset(&set);
    result = sigismember(&set, SIGALRM);
    printf("SIGALRM is %s a member\n", result ? "" : "not");
    sigaddset(&set, SIGALRM);
    result = sigismember(&set, SIGALRM);
    printf("SIGALRM is %s a member\n", result ? "" : "not");

    sigfillset(&set);
    result = sigismember(&set, SIGCHLD);
    printf("SIGCHLD is %s a member\n", result ? "" : "not");
    sigdelset(&set, SIGCHLD);
    result = sigismember(&set, SIGCHLD);
    printf("SIGCHLD is %s a member\n", result ? "" : "not");
}
```

# signal2.c

day6/signal2.c

```
$ ./signal2
SIGALRM is not a member
SIGALRM is  a member
SIGCHLD is  a member
SIGCHLD is not a member
$
```

# Signal Mask

# sigpromask

특정 시그널을 선택적으로 블록시키기 위한 함수

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

| | |
|---|---|
| *how* | sigprocmask 함수의 행동 방식을 결정한다. |
| *set* | 새롭게 적용하는 시그널 마스크이다. |
| *oldset* | 이전에 적용되어 있는 시그널 마스크를 저장한다. |
| 반환값 | 호출이 성공할 경우 0을 반환하고, 실패하면 -1을 반환한다. |

프로세스가 대단히 중요한 코드를 실행 중일 때 작업에 방해를 받지 않기 위해 시그널을 무시할 수도 있다.

시그널 봉쇄 → 중요한 작업 수행 → 시그널 봉쇄 해제

# sigpromask

특정 시그널을 선택적으로 블록시키기 위한 함수

int how

| 값 | 의미 |
| --- | --- |
| SIG_BLOCK | 현재 봉쇄 설정된 시그널의 목록에 두 번째 인자 set에 포함된 시그널을 추가한다. |
| SIG_UNBLOCK | 현재 봉쇄 설정된 시그널의 목록에서 두 번째 인자 set에 포함된 시그널을 제외한다. |
| SIG_SETMASK | 현재 봉쇄 설정된 시그널의 목록을 두 번째 인자 set가 가진 목록으로 대체한다. |

# signal5.c (1)

day6/signal5.c

```c
#include <signal.h> #include <unistd.h>

main() {
    sigset_t set;
    int count = 3;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);

    sigprocmask(SIG_BLOCK, &set, NULL);

    while(count) {
        printf("don't disturb me (%d)\n", count--);
        sleep(1);
    }

    sigprocmask(SIG_UNBLOCK, &set, NULL);
    printf("you did not disturb me!!\n");
}
```

# signal5.c (1)

day6/signal5.c

```
$./signal5

don't disturb me (3)
don't disturb me (2)
don't disturb me (1)
you did not disturb me!!
$ ex10-05
don't disturb me (3)
don't disturb me (2)
don't disturb me (1)

$
```

← 이쯤에서 Ctrl+C를 입력한다.