12. IPC: Signal and Shared Memory

Hyunchan, Park

http://oslab.jbnu.ac.kr

Division of Computer Science and Engineering

Jeonbuk National University

학습내용

- IPC: Inter-process communication
- Signal
- Shared Memory

IPC:
Inter-process communication

IPC 의 필요성

- Many processes are co-operating
  - 서로 협업하는 프로세스: 한 가지 목적을 위해 여러 프로세스가 함께 동작
    - 큰 프로그램의 모듈화, 병렬 작업을 통한 성능 향상, 사용자 편의성 향상 등
    - 예) Game: video and audio processing, background data loading, chatting, etc.
  - 이를 위해서는 상호 간의 정보 교환이 필수
- 그러나 프로세스들은..가상 메모리 공간 (Virtual Memory Space) 에 갇혀 있음
  - 각 프로세스는 자기 자신만의 독립되고, 고립된 (isolated) 메모리 공간을 가짐
    - 프로세스들은 서로 다른 프로세스의 space 를 침범할 수 없음! 존재조차 모름.
  - 함께 일을 해야 하는데, 그럼 어떻게 정보를 주고 받지? → IPC!!

IPC: Inter-Process Communication

- IPC: 프로세스 간 통신
  - 모든 프로세스들의 공간에 접근할 수 있는 OS의 도움을 받아,
    프로세스들 간에 서로 정보를 교환하는 것
  - 예) Message queue, shared memory, signal, pipe, socket 등
  - (더 쉬운 예. CTRL+C, V)
- 어떤 medium 을 어떤 방식으로 사용해서 정보를 전달하느냐에 따라
  서로 다른 통신 특성을 갖게 됨
  - 커널 및 유저 메모리 공간, 네트워크 등이 medium 이 될 수 있음
  - 예) Signal: 커널 메모리 공간을 통해 아주 단순한 정보만 단방향으로 전달
  - 예) Shared memory: 유저 메모리 공간을 통해 양방향으로 정보 전달 가능
    - 그러나 양쪽이 동시에 데이터를 수정할 수 있기 때문에 동기화 문제 발생 가능
  - 예) Socket: 네트워크를 통해 원격 시스템에서 동작하는 프로세스와 통신 가능
- 협업 방식 및 상황에 따라 적절한 IPC 를 선택할 수 있어야 함

Signal

시그널의 개념

- 시그널
  - 프로세스에 무슨 일이 발생했음을 알리는 간단한 메시지를 비동기적으로 보내는 것
  - 예) 메모: “전화왔음” (누구한테?? 거기까진 알 수 없음. 메모 종류도 한정적)
- 발생사유
  - 0으로 나누기처럼 프로그램에서 예외적인 상황이 일어나는 경우
  - Kill(2) 처럼 시그널을 보내는 시스템콜을 호출해 다른 프로세스에 시그널을 보내는 경우
  - 사용자가 CTRL+C 와 같이 키를 입력해 인터럽트를 발생시킨 경우
- 시그널 처리방법
  - 각 시그널에 지정된 기본 동작 수행. 대부분의 기본 동작은 프로세스 종료
  - 무시: 발생한 시그널을 무시하고 진행
  - 핸들러 호출: 특정 시그널 처리를 위한 함수(핸들러)를 지정해두고, 시그널을 받으면 해당 함수를 호출
  - 블록: 발생한 시그널을 처리하지 않고 그대로 둠. 블록을 해제하면 그때 전달되어 처리

시그널 종류

시그널	번호	기본 처리	발생 요건
SIGHUP	1	종료	행업으로 터미널과 연결이 끊어졌을 때 발생
SIGINT	2	종료	인터럽트로 사용자가 Ctrl+C를 입력하면 발생
SIGQUIT	3	코어 덤프	종료 신호로 사용자가 Ctrl+\를 입력하면 발생
SIGILL	4	코어 덤프	잘못된 명령 사용
SIGTRAP	5	코어 덤프	추적(trace)이나 중단점(breakpoint)에서 트랩 발생
SIGABRT	6	코어 덤프	abort 함수에 의해 발생
SIGEMT	7	코어 덤프	에뮬레이터 트랩으로 하드웨어에 문제가 있을 경우 발생
SIGFPE	8	코어 덤프	산술 연산 오류로 발생
SIGKILL	9	종료	강제 종료로 발생
SIGBUS	10	코어 덤프	버스 오류로 발생
SIGSEGV	11	코어 덤프	세그먼테이션 폴트로 발생
SIGSYS	12	코어 덤프	잘못된 시스템 호출로 발생
SIGPIPE	13	종료	잘못된 파이프 처리로 발생
SIGALRM	14	종료	알람에 의해 발생
SIGTERM	15	종료	소프트웨어적 종료로 발생
SIGUSR1	16	종료	사용자 정의 시그널 1

시그널 종류

- os에 따라 번호와 종류의 차이가 있을 수 있음
- 사용할 시그널의 자세한 내용은 항상 os에 따라 확인하고 사용

```
ubuntu@41983:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

시그널 보내기: kill(2)

- (이미 배운) kill 명령어
  - 프로세스에 시그널을 보내는 명령 (예. PID가 3255인 프로세스에 9번 SIGKILL 전달)

```
# kill -9 3255
```

- 시그널 보내기: kill(2)

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

int kill(pid_t pid, int sig);
```

  - pid가 0보단 큰 수 : pid로 지정한 프로세스에 시그널 발송
  - pid가 -1이 아닌 음수 : 프로세스 그룹ID가 pid의 절대값인 프로세스 그룹에 속하고, 시그널을 보낼 권한을 가지고 있는 모든 프로세스에 시그널 발송
  - pid가 0 : 특별한 프로세스를 제외하고 프로세스 그룹ID가 시그널을 보내는 프로세스의 프로세스 그룹ID와 같은 모든 프로세스에게 시그널 발송
  - pid가 -1 : 시그널을 보내는 프로세스의 유효 사용자ID가 root가 아니면, 특별한 프로세스를 제외하고 프로세스의 실제 사용자ID가 시그널을 보내는 프로세스의 유효 사용자ID와 같은 모든 프로세스에 시그널 발송

시그널 보내기: raise(2) and abort(3)

- 시그널 보내기: raise(2)

```
#include <signal.h>

int raise(int sig);
```

  - 함수를 호출한 프로세스 자기 자신에게 시그널 발송

- 시그널 보내기: abort(3)

```
#include <stdlib.h>

void abort(void);
```

  - 함수를 호출한 프로세스에 자기 자신에게 SIGABRT시그널 발송
  - SIGABRT 시그널은 프로세스를 비정상적으로 종료시키고 코어덤프 생성

[예제 1] kill()

hw11 > C s1.c

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

int main(int argc, char* argv[]) {
    int signo;
    pid_t pid;

    if(argc != 3) {
        printf("< Usage: ./s1 signo pid>\n");
        return 1;
    }

    signo=atoi(argv[1]);
    pid = (pid_t) atoi(argv[2]);

    printf("Sending signal %d to %d\n", signo, pid);

    kill(pid, signo);

    return 0;
}
```

```
ubuntu@41983:~/hw11$ gcc -o s1 s1.c
ubuntu@41983:~/hw11$ sleep 50 &
[1] 2629357
ubuntu@41983:~/hw11$ ps
    PID TTY          TIME CMD
2336792 pts/1    00:00:00 bash
2629357 pts/1    00:00:00 sleep
2629418 pts/1    00:00:00 ps
ubuntu@41983:~/hw11$ ./s1 9 2629357
Sending signal 9 to 2629357
[1]+  Killed                    sleep 50
```

시그널 핸들러: signal(3)

- 시그널 핸들러
  - 시그널을 받았을 때, 기본 동작을 대체하여 이를 처리하기 위해 지정된 함수
  - 시그널을 받으면, 받은 시점에 수행 중이던 코드에서 마치 핸들러를 호출한 것 처럼 동작
  - 프로세스를 종료하기 전에 처리할 것이 있거나, 특정 시그널에 대해 종료하고 싶지 않을 경우 지정
- 시그널 핸들러 지정: signal(3)

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);
```

  - disp : sig로 지정한 시그널을 받았을 때 처리할 방법
    - 시그널 핸들러 함수명
    - SIG_IGN : 시그널을 무시하도록 지정
    - SIG_DFL : 기본 처리 방법으로 처리하도록 지정
  - (어떤 Unix 시스템에서는) 한 번 핸들러가 수행되고 나면, 핸들러 지정이 취소됨

DESCRIPTION

The behavior of **signal()** varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use: use sigaction(2) instead.** See Portability below.

시그널 핸들러: sigset(3)

- 시그널 핸들러 지정: sigset(3)

```
#include <signal.h>

void (*sigset(int sig, void (*disp)(int)))(int);
```

  - disp : sig로 지정한 시그널을 받았을 때 처리할 방법
    - 시그널 핸들러 함수명
    - SIG_IGN : 시그널을 무시하도록 지정
    - SIG_DFL : 기본 처리 방법으로 처리하도록 지정

- sigset함수는 signal함수와 달리 시그널 핸들러가 한 번 호출된 후에 기본동작으로 재설정하지 않고, 해당 시그널 핸들러를 자동 재지정한다.

These functions are provided in glibc as a compatibility interface for programs that make use of the historical System V signal API. This API is obsolete: new applications should use the POSIX signal API (sigaction(2), sigprocmask(2), etc.)

[예제 2] signal()

hw11 > C s2.c

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

void handler(int signo) {
    psignal(signo, "Received signal:");
}

int main(int argc, char* argv[]) {
    int i, signo;

    if(argc != 2) {
        printf("Usage: ./s2 signal_to_wait\n");
        return 1;
    }

    signo = atoi(argv[1]);

    if(signal(signo, handler) == SIG_ERR) {
        perror("signal");
        return 3;
    }

    for(i=0; i<10; i++) {
        printf("** Waiting for signal %d : %02d seconds\n", signo, i);
        sleep(1);
    }

    printf("Finished!\n");

    return 0;
}
```

```
PSIGNAL(3)

NAME
       psignal, psiginfo - print signal message

SYNOPSIS
       #include <signal.h>

       void psignal(int sig, const char *s);
```

```
ubuntu@41983:~/hw11$ gcc -o s2 s2.c
ubuntu@41983:~/hw11$ ./s2 2
** Waiting for signal 2 : 00 seconds
** Waiting for signal 2 : 01 seconds
** Waiting for signal 2 : 02 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 03 seconds
** Waiting for signal 2 : 04 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 05 seconds
** Waiting for signal 2 : 06 seconds
** Waiting for signal 2 : 07 seconds
** Waiting for signal 2 : 08 seconds
** Waiting for signal 2 : 09 seconds
Finished!
```

시그널 핸들러: sigaction()

- sigaction 함수
  - signal이나 sigset 함수처럼 시그널을 받았을 때 이를 처리하는 함수 지정
  - signal, sigset 함수보다 다양하게 시그널 제어 가능

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

  - signum : 처리할 시그널 번호
  - act : 시그널을 처리할 방법을 지정한 구조체 주소
  - oldact : 기존에 시그널을 처리하던 방법을 저장할 구조체 주소

  - 첫번째 인자로 SIGKILL과 SIGSTOP을 제외한 어떤 시그널도 사용 가능

struct sigaction

- 핸들러의 동작을 보다 세밀하고 정확하게 제어하기 위한 정보를 전달

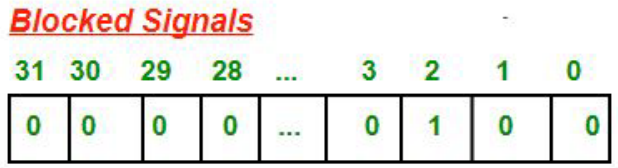
```
struct sigaction {
    int sa_flags;
    union {
        void (*sa_handler)();
        void (*sa_sigaction)(int, siginfo_t *, void *);
    } _funcptr;
    sigset_t sa_mask;
};
```

  - sa_flags : 시그널 전달 방법을 수정할 플래그
    - 예) SA_RESETHAND: 시그널 핸들러가 한 번 호출되면 지정이 취소됨
  - sa_handler/sa_sigaction : 시그널 처리를 위한 동작 지정
    - sa_flags에 SA_SIGINFO가 설정되어 있지 않으면 sa_handler에 시그널 처리동작 지정
    - sa_flags에 SA_SIGINFO가 설정되어 있으면 sa_sigaction 멤버 사용
  - sa_mask : 시그널 핸들러가 수행되는 동안 블록될 시그널을 지정한 시그널 집합

시그널 집합: sigset_t

- 복수의 시그널을 한 번에 지정, 처리하기 위해 도입한 개념

```
typedef struct {
    unsigned int __sigbits[4];
} sigset_t;
```



  - Bit masking: 각 비트가 각각 시그널의 지정 여부를 표시
  - 프로세스 수행 상태에 따라 전체 시그널의 상태가 혼동스러울 수 있으므로, 항상 전체 시그널 모두에 대해 어떤 처리를 지정하도록 하기 위함

- 시그널 집합 비우기 : sigemptyset(3)

```
int sigemptyset(sigset_t *set);
```

  - 시그널 집합에서 모든 시그널을 0으로 설정

- 시그널 집합에 모든 시그널 설정: sigfillset(3)

```
int sigfillset(sigset_t *set);
```

  - 시그널 집합에서 모든 시그널을 1로 설정

시그널 집합: sigset_t

- 시그널 집합에 시그널 설정 추가: sigaddset(3)

```
int sigaddset(sigset_t *set, int signo);
```

  - signo로 지정한 시그널을 시그널 집합에 추가

- 시그널 집합에서 시그널 설정 삭제: sigdelset(3)

```
int sigdelset(sigset_t *set, int signo);
```

  - signo로 지정한 시그널을 시그널 집합에서 삭제

- 시그널 집합에 설정된 시그널 확인: sigismember(3)

```
int sigismember(sigset_t *set, int signo);
```

  - signo로 지정한 시그널이 시그널 집합에 포함되어 있는지 확인

[예제 3] sigaction() with sigset_t

hw11 > C s3.c

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

void handler(int signo) {
    psignal(signo, "Received signal:");
}

int main(int argc, char* argv[]) {
    int i;
    struct sigaction act;

    sigemptyset(&act.sa_mask);          //SIGINT 에 대한 핸들러가 수행되는 중에
    sigaddset(&act.sa_mask, SIGINT);    //다시 SIGINT 가 중첩 수행되지 않도록 함
    act.sa_flags = 0;
    act.sa_handler = handler;

    if(sigaction(SIGINT, &act,(struct sigaction *) NULL) < 0) {
        perror("sigaction");
        return 3;
    }

    for(i=0; i<10; i++) {
        printf("** Waiting for signal %d : %02d seconds\n", SIGINT, i);
        sleep(1);
    }

    printf("Finished!\n");

    return 0;
}
```

```
ubuntu@41983:~/hw11$ gcc -o s3 s3.c
ubuntu@41983:~/hw11$ ./s3
** Waiting for signal 2 : 00 seconds
** Waiting for signal 2 : 01 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 02 seconds
** Waiting for signal 2 : 03 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 04 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 05 seconds
^CReceived signal:: Interrupt
** Waiting for signal 2 : 06 seconds
** Waiting for signal 2 : 07 seconds
** Waiting for signal 2 : 08 seconds
** Waiting for signal 2 : 09 seconds
Finished!
```

[예제 4] sigaction() with SIG_IGN

hw11 > C s4.c

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

void handler(int signo) {
    psignal(signo, "Received signal:");
}

int main(int argc, char* argv[]) {
    int i;
    struct sigaction act;

    sigemptyset(&act.sa_mask);          //SIGINT 에 대한 핸들러가 수행되는 중에
    sigaddset(&act.sa_mask, SIGINT);    //다시 SIGINT 가 중첩 수행되지 않도록 함
    act.sa_flags = 0;

    //act.sa_handler = handler;
    act.sa_handler = SIG_IGN;           //handler 대신 SIG_IGN 을 지정하여 시그널 무시

    if(sigaction(SIGINT, &act,(struct sigaction *) NULL) < 0) {
        perror("sigaction");
        return 3;
    }

    for(i=0; i<10; i++) {
        printf("** Waiting for signal %d : %02d seconds\n", SIGINT, i);
        sleep(1);
    }

    printf("Finished!\n");

    return 0;
}
```

```
ubuntu@41983:~/hw11$ gcc -o s4 s4.c
ubuntu@41983:~/hw11$ ./s4
** Waiting for signal 2 : 00 seconds
** Waiting for signal 2 : 01 seconds
^C** Waiting for signal 2 : 02 seconds
^C^C^C** Waiting for signal 2 : 03 seconds
^C^C^C^C^C** Waiting for signal 2 : 04 seconds
^C^C^C^C^C** Waiting for signal 2 : 05 seconds
^C^C^C^C^C** Waiting for signal 2 : 06 seconds
** Waiting for signal 2 : 07 seconds
** Waiting for signal 2 : 08 seconds
^C^C** Waiting for signal 2 : 09 seconds
^C^C^C^C^CFinished!
```

시그널의 블록: sigprocmask(2)

```
#include <signal.h>

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

- how : 시그널을 블록할 것인지, 해제할 것인지 여부
  - SIG_BLOCK : set에 지정한 시그널 집합을 시그널 마스크에 추가
  - SIG_UNBLOCK : set에 지정한 시그널 집합을 시그널 마스크에서 제거
  - SIG_SETMASK : set에 지정한 시그널 집합으로 현재 시그널 마스크 대체
- set : 블록하거나 해제할 시그널 집합 주소
- oldset : NULL 또는 이전 설정값을 저장한 시그널 집합 주소

- Obsoleted services: sighold(2) and sigrelse(2)

```
#include <signal.h>

int sighold(int sig);
int sigrelse(int sig);
```

[예제 5] sigprocmask()

hw11 > C s5.c

```c

void handler(int signo) {
    psignal(signo, "Received signal:");
}

int main(int argc, char* argv[]) {
    int i;
    struct sigaction act;
    sigset_t set_to_block;

    sigemptyset(&act.sa_mask);          //SIGINT 에 대한 핸들러가 수행되는 중에
    sigaddset(&act.sa_mask, SIGINT);    //다시 SIGINT 가 중첩 수행되지 않도록 함
    act.sa_flags = 0;

    act.sa_handler = handler;

    sigaction(SIGINT, &act, NULL);  //handler 설정

    sigemptyset(&set_to_block);
    sigaddset(&set_to_block, SIGINT);   //빈 시그널 집합에 SIGINT 만 추가
    sigprocmask(SIG_BLOCK, &set_to_block, NULL);    //위 시그널 집합에 대해 블록

    for(i=0; i<10; i++) {
        printf("** Waiting for signal %d : %02d seconds\n", SIGINT, i);

        if (i == 5) sigprocmask(SIG_UNBLOCK, &set_to_block, NULL);  //5초 후, 블록 해제

        sleep(1);
    }

    printf("Finished!\n");

    return 0;
}
```

```
ubuntu@41983:~/hw11$ ./s5
** Waiting for signal 2 : 00 seconds
** Waiting for signal 2 : 01 seconds
^C** Waiting for signal 2 : 02 second
** Waiting for signal 2 : 03 seconds
** Waiting for signal 2 : 04 seconds
** Waiting for signal 2 : 05 seconds
Received signal:: Interrupt
** Waiting for signal 2 : 06 seconds
** Waiting for signal 2 : 07 seconds
** Waiting for signal 2 : 08 seconds
** Waiting for signal 2 : 09 seconds
Finished!
```

시그널 대기: sigsuspend(2)

- 지정한 시그널(들)이 도착할 때까지 대기

```
#include <signal.h>

int sigsuspend(const sigset_t *set);
```

  - set : 기다리려는 시그널들을 제외한 다른 모든 시그널들을 지정한 시그널 집합
  - 정확한 동작: 현재 동작 중인 프로세스의 시그널 마스크를 set 으로 대치하여, 기다리려는 시그널들을 제외한 다른 시그널들은 블록 처리됨. 따라서 기다리려는 시그널만 프로세스로 전달될 수 있게 설정되는 것

- Obsoleted service: sigpause (2)

```
#include <signal.h>

int sigpause(int sig);
```

[예제 6] sigsuspend(2)

hw11 > C s6.c

```c
void handler(int signo) {
    psignal(signo, "Received signal:");
}

int main(int argc, char* argv[]) {
    int i;
    struct sigaction act;
    sigset_t set_to_wait;

    sigemptyset(&act.sa_mask);          //SIGINT 에 대한 핸들러가 수행되는 중에
    sigaddset(&act.sa_mask, SIGINT);    //다시 SIGINT 가 중첩 수행되지 않도록 함
    act.sa_flags = 0;

    act.sa_handler = handler;

    sigaction(SIGINT, &act, NULL);  //handler 설정

    sigfillset(&set_to_wait);
    sigdelset(&set_to_wait, SIGINT);   //SIGINT 만 제외된 집합 설정

    printf("** Waiting for signal %d ...\n", SIGINT);

    sigsuspend(&set_to_wait);    //집합에서 제외된 시그널을 대기

    printf("** We got it!\n");

    return 0;
}
```

```
ubuntu@41983:~/hw11$ gcc -o s6 s6.c
ubuntu@41983:~/hw11$ ./s6
** Waiting for signal 2 ...
^CReceived signal:: Interrupt
** We got it!
```

알람 시그널: alarm(2)

- 알람 시그널
  - 일정한 시간이 지난 후에 자동으로 시그널이 발생하도록 하는 시그널
  - 일정 시간 후에 한 번 발생시키거나, 일정 간격을 두고 주기적으로 발송 가능

- 알람 시그널 생성: alarm(2)

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

  - sec : 알람이 발생시킬 때까지 남은 시간(초 단위)
  - 일정 시간이 지나면 SIGALRM 시그널 발생
  - 프로세스별로 알람 시계가 하나 밖에 없으므로 알람은 하나만 설정 가능

[예제 7] alarm(2)

hw11 > C s7.c

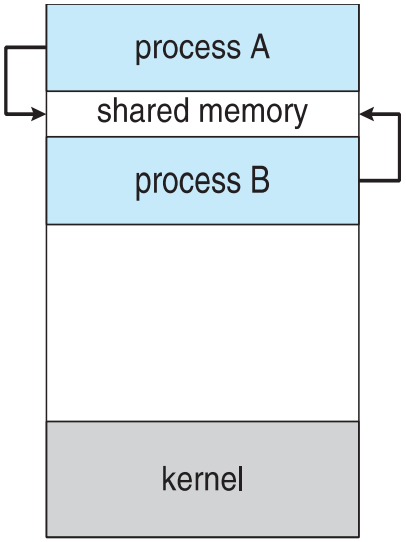
```c
void handler(int signo) {
    psignal(signo, "Received signal:");
    alarm(2);   //핸들러 호출 2초 후, 다시 알람 설정
}

int main(int argc, char* argv[]) {
    int i, signo;

    if(signal(SIGALRM, handler) == SIG_ERR) {
        perror("signal");
        return 3;
    }

    alarm(3);   //시작하고 3초 후 알람 울리기

    for(i=0; i<10; i++) {
        printf("** Waiting for signal %d : %02d seconds\n", signo, i);
        sleep(1);
    }

    printf("Finished!\n");

    return 0;
}
```

```
ubuntu@41983:~/hw11$ gcc -o s7 s7.c
ubuntu@41983:~/hw11$ ./s7
** Waiting for signal 0 : 00 seconds
** Waiting for signal 0 : 01 seconds
** Waiting for signal 0 : 02 seconds
Received signal:: Alarm clock
** Waiting for signal 0 : 03 seconds
** Waiting for signal 0 : 04 seconds
Received signal:: Alarm clock
** Waiting for signal 0 : 05 seconds
** Waiting for signal 0 : 06 seconds
Received signal:: Alarm clock
** Waiting for signal 0 : 07 seconds
** Waiting for signal 0 : 08 seconds
Received signal:: Alarm clock
** Waiting for signal 0 : 09 seconds
Finished!
```

Shared Memory

Shared memory

- 공유 메모리
  - 같은 메모리 공간을 두 개 이상의 프로세스가 공유하는 것
    - 물론 OS의 허가를 받고 공유함. 그런데 처음 허가 한 번 받고 나면 끝. OS는 관여하지 않음
  - 같은 메모리 공간을 사용하므로 이를 통해 데이터를 주고 받을 수 있음
  - 문제점: 동기화
    - A가 데이터를 기록하는 동안, 같은 위치에 B가 데이터를 쓰면?
    - A가 데이터를 아직 다 쓰지 않았는데, B는 다 쓴 줄 알고 데이터를 읽어가면?



(b)

* Rolling paper

공유 메모리 생성: shmget(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

- key : IPC_PRIVATE 또는 ftok()로 생성한 키 값
  - 상호 약속된 키 값을 기입해야 공유 메모리를 접근할 권한을 얻을 수 있음
  - IPC_PRIVATE: 부모-자식 간의 공유 메모리 공유를 위해 사용
  - key_t ftok(const char *pathname, int proj_id);
    - 주어진 파일명(절대 경로)과 proj_id 를 이용해 키 값을 생성함
    - 절대 경로, proj_id 가 동일해야 같은 키 값이 생성됨
- size : 공유할 메모리 크기. 페이지 크기의 배수로 지정.
  - 페이지: 메모리를 관리하는 기본 단위. 보통 4KB
- shmflg : 공유 메모리의 속성을 지정하는 플래그
  - IPC_CREAT: 공유 메모리 공간을 새로 생성함
  - IPC_EXCL: IPC_CREAT 와 함께 사용해서, 만약 기존 공간이 있다면 fail 하도록 설정
- 공유 메모리 식별자를 리턴
  - OS가 공유 메모리 공간을 관리하는 자료 구조를 가리킴. 파일 기술자와 유사.

공유 메모리 연결 및 해제

- 공유 메모리 연결: shmat(2)

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

  - shmid : 공유 메모리 식별자
  - shmaddr : 공유 메모리를 연결할 주소
  - shmflg : 공유 메모리에 대한 읽기/쓰기 권한
    - 0(읽기/쓰기 가능), SHM_RDONLY(읽기 전용)

- 공유 메모리 연결 해제: shmdt(2)

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

  - shmaddr: 연결을 해제할 공유 메모리 주소

공유 메모리 제어

- 공유 메모리 제어: shmctl(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

  - cmd : 수행할 제어기능
    - IPC_RMID : 공유 메모리 제거 (할당 해제)
    - IPC_SET : 공유 메모리 정보 내용 중 shm_perm.uid, shm_perm.gid, shm_perm.mode 값을 세번째 인자로 지정한 값으로 변경
    - IPC_STAT : 현재 공유 메모리의 정보를 buf에 지정한 메모리에 저장
    - SHM_LOCK : 공유 메모리를 잠근다.
    - SHM_UNLOCK : 공유 메모리의 잠금을 해제한다.

[예제 8] 공유 메모리: Server

hw11 > C server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/shm.h>

#define BUFSIZE (80)

int main(int argc, char* argv[]) {
    key_t key;
    int shmid, size;
    char buf[BUFSIZE];
    void *shmaddr;

    key = ftok("shmfile", 1);           //shmfile 을 이용해 키 생성. 파일이 존재해야 함
    size = sysconf(_SC_PAGESIZE) * 4;   //페이지 사이즈 * 4 = 16 KB
    shmid = shmget(key, size, IPC_CREAT|0666);  //공유 메모리 접근 권한 설정

    shmaddr = shmat(shmid, NULL, 0);    //할당받은 공유 메모리 공간의 주소를 얻음

    printf("Shared memory info: key=%d shmid=%d shmaddr=%p\n", key, shmid, shmaddr);

    memset(shmaddr,0,size);             //공유 메모리 공간 전체를 0으로 초기화

    memcpy(buf, shmaddr, BUFSIZE);      //해당 공간의 처음 80 B 만큼의 데이터 복사하여 출력
    printf("Initialized: %s\n", buf);   //0으로 초기화하였으니, 아무런 내용 없음

    printf("Waiting for client...\n");

    sleep(3);   //3초간 대기. 다른 프로세스가 해당 공간을 수정하도록

    memcpy(buf, shmaddr, BUFSIZE);      //해당 공간의 변경된 내용을 확인
    printf("Changed: %s\n", buf);

    shmctl(shmid, IPC_RMID, NULL);      //공유 메모리 공간 할당 해제

    return 0;
}
```

```
ubuntu@41983:~/hw11$ ./server
Shared memory info: key=16907456 shmid=2 shmaddr=0x7fae64913000
Initialized:
Waiting for client...
Changed:
ubuntu@41983:~/hw11$
```

[예제 8] 공유 메모리: Client

hw11 > C client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/shm.h>

#define BUFSIZE (80)

int main(int argc, char* argv[]) {
    key_t key;
    int shmid, size;
    char buf[BUFSIZE];
    void *shmaddr;

    key = ftok("shmfile", 1);           //shmfile 을 이용해 키 생성. 파일이 존재해야 함
    size = sysconf(_SC_PAGESIZE) * 4;   //페이지 사이즈 * 4 = 16 KB
    shmid = shmget(key, size, IPC_CREAT|0666);  //공유 메모리 접근 권한 설정

    shmaddr = shmat(shmid, NULL, 0);    //할당받은 공유 메모리 공간의 주소를 얻음

    printf("Shared memory info: key=%d shmid=%d shmaddr=%p\n", key, shmid, shmaddr);

    memcpy(buf, shmaddr, BUFSIZE);      //현재 메모리 공간의 처음 80 B 내용 확인.
    printf("Client: %s\n", buf);        //0으로 초기화하였으니, 아무런 내용 없음

    strcpy(buf, "Hello Server! I am Client.");  //buf 문자열 수정
    memcpy(shmaddr, buf, BUFSIZE);      //buf 의 문자열을 공유 메모리 공간으로 복사

    shmdt(shmaddr);         //공유 메모리 공간 연결 해제

    strcpy(buf, "Message is deleted");  //buf 문자열 수정
    memcpy(shmaddr, buf, BUFSIZE);      //buf 의 문자열을 다시 기존 공유 메모리 공간으로 복사 시도
                                        //그러나 연결 해제 되었으므로, 할당되지 않은 공간에 접근한 것 -> Seg. fault
    return 0;
}
```

```
ubuntu@41983:~/hw11$ ./server
Shared memory info: key=16907456 shmid=4 shmaddr=0x7f558bcfa000
Initialized:
Waiting for client...
Changed: Hello Server! I am Client.
ubuntu@41983:~/hw11$
```

```
ubuntu@41983:~/hw11$ ./client
Shared memory info: key=16907456 shmid=4 shmaddr=0x7f36d6603000
Client:
Segmentation fault (core dumped)
ubuntu@41983:~/hw11$
```

개인 과제 10: signal + shared memory

- 내용: 프로그램 1개 작성
  - fork()를 수행하여 자식 프로세스를 생성하는 프로그램
  - SIGUSR1, SIGUSR2 에 대해 핸들러 등록. 간단히 시그널 수신 결과를 출력함
  - 키로 사용할 파일은 임의의 문자열로 작성
  - Parent process
    - 4KB 의 공유 메모리 공간을 할당하고, 0으로 초기화
    - Child 에게 SIGUSR1 을 보내 공간이 할당되었음을 알림
    - SIGUSR2 를 대기. 다른 모든 시그널은 블록
    - 공유 메모리 공간에서 초반 80 B 영역을 문자열로 출력
  - Child process
    - SIGUSR1 을 대기 . 다른 모든 시그널은 블록
    - 공유 메모리 공간을 연결하고,
    - 공유 메모리를 할당할 때 사용한 키 파일의 내용을 공유 메모리 공간에 복사 (80 B)
    - Parent 에게 SIGUSR2 를 보내어 수정이 완료되었음을 알림

> **getppid**() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using **fork**(), or, if that process has already terminated, the ID of the process to which this process has been reparented (either init(1) or a "subreaper" process defined via the prctl(2) PR_SET_CHILD_SUBREAPER operation).

- 제출 기한
  - 11/30 (월) 23:59 (지각 감점: 5%p / 12H, 1주 이후 제출 불가)
  - 동작 설명과 결과가 포함된 간단한 보고서와 소스 파일을 압축하여 LMS 제출