POSIX Signals

Inter Process Communication(IPC)의 형태 Signal로 통신을 구현한다.

Multitasking / Concurrency

- fork: 자식프로세스 생성
- exit: 현재 프로세스 종료
- wait / waitpid: 자식 프로세스의 종료를 기다림
- execve: 현재 프로세스를 종료하고 새로운 프로그램을 실행한다

Signal

- signal은 시스템에서 일어난 이벤트를 프로세스에게 전달하는 것이다. 작은 메세지 형태이다.
  pipe - data delivery
  signal - data delivery 아니고 그냥 신호..

- asynchronous 메세지이다. (비동기)
  언제든지 도착할 수 있는 메세지

- each signal has a number, samll integer (다른 데이터 아님)

Asynchronous Reception

- 시그널은 블락되거나 무시될 수 있다.
- 두개의 프로세스에서 시그널을 받을 수 있다.
- signal handler를 사용자가 정의할 수 있다.

Signal Concepts

Sending a signal

- Kernel이 destination process에 시그널을 보낸다
- kill(dst pid, signal 종류)

Receiving a signal

- 목적 프로세스가 시그널을 받으면,
    1. ignore
    2. default action
    3. catch signal by signal handler

Default actions

- Abort
- Dump
- Ignore
- Stop
- Continue

Signal semantics

- Signal은 pending된다.
  -> 동일한 Type Signal은 최대 하나만 Pending된다.

> A -> B -> A -> B
> A를 처리하는 도중에 B pending, A pending, B는 이미 pending된 signal 존재하기 때문에 discard

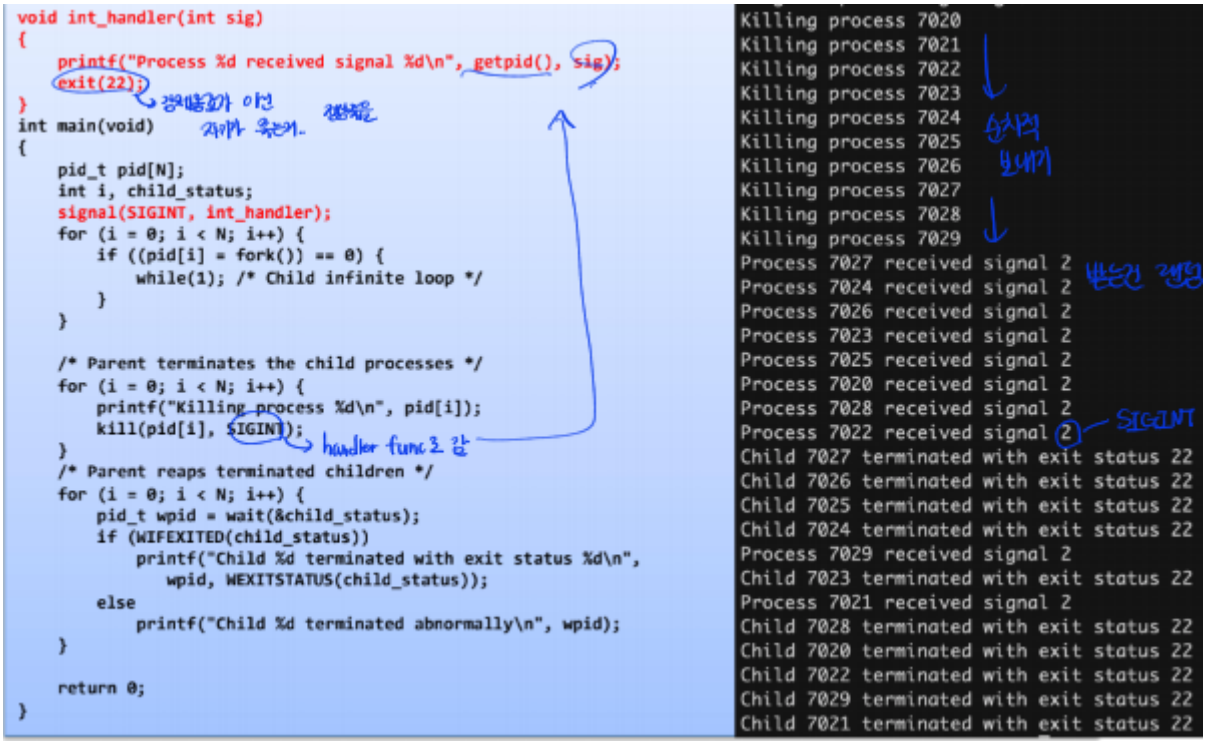
- process는 signal을 블락할 수 있다.
  -> 블락된 시그널은 전달될 수 있으나, 언블락 되기 전에는 전달/도달하지 못한다.
  -> SIGKILL, SIGSTOP은 블락되지 않고 바로 전달된다. 또한, catch, handle이 불가하다.

- signal block?
  signal handling 중에 발생된 signal을 handling 끝날 때까지 block하는 것.

Sending Signals

```
int kill(pit_t pid, int sig)
---
kill(getppid(), SIGUSR1)
//부모 프로세스에게 SIGUSR1 보내기
```

signal 보내는 시간과 받는 시간은 같지 않다.



Handling Signals

- 펜딩된 시그널은 큐의 형태가 아니다 (동일 타입이면 씹힘)
- 시그널 핸들러 중간에 도착한 시그널은 블락됨
  쌓이는 게 아니라 각 시그널 공간의 비트로 해당 타입 펜딩 여부를 확인하는 것이다.
  펜딩된 시그널 처리는 OS별로 상이하지만, 보통은 순차적으로 처리한다.

▶ 실습 01: Ex2.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>

#define N (10)
pid_t pid[N];
int ccount = 0;

void handler (int sig) {
    pid_t id;
    //wait: 블로킹 함수, 자식프로세스 죽을 때까지 기다림
    //waitpid: 논블로킹 함수

    //id = wait(NULL); //-> 무한루프에 빠짐

    //wait((id=wait(NULL))>0)과 동일하게 동작
    // 차이점 -> wait: 올때까지 기다림 / waitpid: 종료된 자식 없으면 원래 하던 동작 계속

    //waitpid(terminate된 모든 child, NULL, terminate된 child없으면 0 리턴)
    while((id=waitpid(-1, NULL, WNOHANG))>0) {
        ccount--;
        printf ("Received signal %d from pid %d\n", sig, id);
    }
}

int main(void) {
    int i;
    ccount = N;
    signal (SIGCHLD, handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            exit(0); /* child */
        }
    }
    while (ccount > 0)
        sleep (5);
    return 0;
}
```


wait

blocking 함수
자식이 죽을 때까지 그 자리에서 기다린다.
--> 무한루프 이유? 자식 1 죽어서 핸들링하는 와중에 자식 2, 3이 죽어도 동일 type signal은 pending되지 않기 때문이다..

```
wait(&status)
```

10번 signal 보내도 10번 다 처리하지 못한다.

- 보완

```
while(wait(&status)>0)
```

이렇게 하면 시그널 전부 처리 가능하지만 그 자리에 멈춰서 올때까지 기다림.

waitpid

non-blocking 함수
기다리지 않고 없으면 그냥 다른 거 실행..
WNOHANG: terminate된 child 없으면 0 리턴

```
while(waitpid(-1, &status, WNOHANG)>0)
```

Signal Portability

시스템별 상이함

Signal Reception

The Kernel may deliver a signal at any time

1. 시그널 발생
2. PC값 스택에 넣기
3. Signal handler로 jump
4. 핸들러 종료
5. PC값으로 넘어감

Shared Data

Signa은 언제 발생할지 모르니까 공유 자원을 사용하지는 않는게 안전함

Signal Concurrency

```
void prepend (struct ListNode *node){
  //1
  node -> next = list;
  //2
  list = node;
  //3
}
```

```
void handler(int sig){
  prepend (new_listnode());
}

int main(int srgc, char * argv[]){
  signal(SIGINT, &handler);
  prepend(new_listnode());

  return 0;
}
```

1번에서 signal 발생

1. H->list
2. M->H->list

2번에서 signal 발생

1. M->list
2. H->list
3. M->list

3번에서 signal 발생

1. M->list
2. H->M->list

H 노드가 사라지면 (2) 메모리가 누수된다.
--> 시그널에서 공유자원 다루는 건 좀 자제..

Summary

- 시그널은 IPC이다.
- 각 시그널은 작은 메세지이다 (pipe같은 데이터 전달 아님)
- 시그널은 함수에 의해 핸들링 될 수 있다.
- 핸들러는 동시성을 발생시킨다. (공유자원 접근시 Concurrency)
- 공유자원을 시그널에서 사용할 때는 주의해야한다.

Code

wait

▶ wait.c

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

int count = 0;

```

```c
//하나의 자식만 처리한다
//foo 처리 도중 자식이 죽어도 처리하지 않음 -> signal 씹힘
void foo(int signum) {
    int status;
    wait(&status);
    printf("status: %d\n", status);
    count++;
    //count!=30
}


//여러 자식을 처리한다
//foo 처리 도중 자식이 죽으면 while문을 통해 다 처리

// void foo(int signum) {
//     int status;
//     while (wait(&status) > 0) {
//         printf("status: %d\n", status);
//         count++;
//         count==30
//     }
// }


int main() {
    //자식 프로세스가 죽으면
    signal(SIGCHLD, foo);
    int i;
    pid_t pid;

    for (int j = 0 ; j < 30; ++j) {
        pid = fork();
        if (pid == 0) {
            for (i = 0; i < 3; ++i) {
                printf("child process..\n");
                if (j > 5) {
                    sleep(1);
                }
            }
            exit(j);
        }
    }

    for (i = 0; i < 20; ++i) {
        printf("parent process..%d\n", i);
        sleep(1);
    }
    printf("%d",count);
    exit(0);
}


                                                    Colored by Color Scripter
```

wait() -> 해당 자식만 처리함 처리 도중에 다른 자식 죽어도 처리하지 않는다

while(wait()>0) -> 다중 자식 처리 처리 도중에 다른 자식 죽으면 while문을 통해 그것까지 처리

waitpid

▶ waitpid.c

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

int count=0;
void foo(int signum) {
  int status;
  //==while(wait(&status)>0)
  //waitpid(모든 자식 프로세스에 대해, 상태값, 죽은자식 없으면 0 리턴)
  while (waitpid(-1, &status, WNOHANG) > 0) {
     printf("status: %d\n", status);
     count++;
     //count==30
   }
}

int main() {
  signal(SIGCHLD, foo);
  int i;
  pid_t pid;

  for (int j = 0 ; j < 30; ++j) {
     pid = fork();
     if (pid == 0) { //자식 프로세스
       for (i = 0; i < 3; ++i) {
         printf("child process..\n");
         if (j > 5) {
           sleep(1);
         }
       }
       exit(0);
     }
   }
  // 부모 프로세스
  for (i = 0; i < 30; ++i) {
     printf("parent process..%d\n",i);
     sleep(1);
  }
  printf("%d",count);
  exit(0);

}


Colored by Color Scripter cs
```

wait	waitpid
blocking 실행 될때까지 기다림	non-blocking 없으면 기다리지 않고 자기 할 일 함

wait	waitpid
wait(&status) signal 씹힘	
while(wait(&status)>0)	while(waitpid(-1,&status,WNOHANG)>0)