Shell Lab: Writing Your Own Unix Shell

Report

20220127 임유진

1. 개요

이번 Lab에서는 job control을 수행할 수 있는 간단한 Unix shell 프로그램을 작성해 봄으로써 프로세스 컨트롤과 시그널에 대한 이해도를 높이는 것을 목적으로 한다.

2. 이론적 배경

- Shell

Shell 은 운영체제 커널과 사용자를 연결하는 인터페이스를 제공하며, 사용자를 대신하여 다른 프로그램을 실행해주는 응용 프로그램이다. Shell 은 prompt 를 출력하고, stdin 으로 들어오는 command line 을 기다리며, command line 의 내용에 따라 작업을 수행하는 것을 반복한다. 이때 command line 은 공백으로 구분되는 일련의 ASCII 단어들로 이루어지며, command line 의 첫 번째 단어는 built-in command 또는 실행 가능한 파일의 경로, 나머지 단어는 command-line 인자들이다.

- Reaping

커널은 종료된 프로세스를 시스템에서 즉시 제거하지 않으며, 종료된 자식 프로세스에 대해서는 부모 프로세스가 reaping 을 해주어야 프로세스가 사라지게 된다. 이때 종료되었지만 부모 프로세스에 의해 reaping 되지 않아 종료된 상태로 남아있는 프로세스들을 좀비 프로세스라고 한다. 부모 프로세스는 waitpid 함수를 통해 자식 프로세스를 reaping 할 수 있으며, 만약 부모 프로세스가 자식 프로세스보다 먼저 종료되거나 좀비가 된 자식 프로세스들을 소거하지 않고 종료된다면 커널은 모든 프로세스의 조상인 init 프로세스가 이 프로세스들을 reaping 하도록 한다. 하지만 좀비 프로세스는 시스템의 메모리 자원을 잡아먹기 때문에 Shell 과 같이 오랫동안 수행되는 프로그램에서는 명시적으로 reaping 을 통해 좀비 프로세스들을 소거해주어야 한다.

- Signal

시그널은 프로세스에게 시스템 내에서 어떤 이벤트가 발생했음을 알리는 메시지이며, 커널이 시스템 이벤트를 감지하거나 어떤 프로세스가 kill 함수를 호출해 시그널을 특정 프로세스에 보내는 것을 커널에 요청하는 경우 발생할 수 있다.

- Signal Handler

시그널을 받은 프로세스는 받은 시그널을 bit vector 에 보관하며, context switch 로 자신의 수행 순서가 되었을 때 bit vector 을 확인해 시그널이 있으면 동작하게 된다. 시그널의 타입에 따라 프로세스의 종료, 프로세스의 종료 후 코어 덤프, SIGCONT 시그널에 의해 재시작될 때까지 정지, 시그널 무시 중 하나의 기본 동작을 수행하게 된다. 이때, signal 함수를 통해 특정 시그널 타입에 대해 SIG_IGN, SIG_DFL, 사용자 정의 함수로 동작을 변경할 수 있다.

- Signal Blocking & Unblocking

리눅스는 시그널을 block 하는 방법으로 묵시적인 방법과 명시적인 방법을 제공한다. 먼저, 묵시적 blocking 이란 커널이 핸들러에 의해 처리되고 있는 타입의 모든 pending 시그널을 block 하는 것을 말한다. 명시적 blocking 방법은 block 되는 시그널의 집합을 변경하는 sigprocmask 함수를 사용하는 것으로, how 값이 SIG_BLOCK 일 경우 set 의 시그널들을 blocked 에 추가, SIG_UNBLOCK 일 경우 set 의 시그널들을 blocked 에서 제거, SIG_SETMASK 일 경우 blocked 를 set 으로 설정하도록 동작한다.

3. 풀이 과정

함수 별 구현 방법은 아래와 같다.

- void eval(char *cmdline)

eval 함수는 사용자가 입력한 command line 을 evaluate 하여 built-in command 인 경우 즉시 수행하고, 아닌 경우 새로운 자식 프로세스를 생성하여 자식 프로세스의 컨텍스트 내에서 인자로 받은 실행 가능 목적파일을 통해 새로운 프로그램을 로드하고 실행한다. eval 함수 내에서 사용한 변수는 아래와 같다.

char *argv[MAXARGS]	cmdline 을 parsing 한 문자열들의 주소를 값으로 가지는 char* 형 배열
int bg	parseline()의 반환값, 1 일 경우 프로세스가 background 에서 실행되어야 함을 나타낸다.
int state	job 의 state
pid_t pid	fork()의 반환값, 부모 프로세스에서는 자식 프로세스의 pid 값을, 자식 프로세스에서는 0 의 값을 가진다.
sigset_t mask	SIGCHLD 만 가지는 시그널 집합
sigset_t prev_mask	기존에 블록되던 시그널 집합
sigset_t all_mask	모든 시그널을 가지는 시그널 집합

우선 parseline 함수의 호출을 통해 사용자로부터 입력받은 cmdline 을 parsing 하여 argv 에 저장하며, bg 에 반환값을 저장한다.

만약 Shell 에 빈 입력이 들어오는 경우 argv[0]이 NULL 값을 가지게 되며, 빈 입력을 무시하기 위해 먼저 argv[0]이 NULL 인 경우 즉시 리턴하도록 하였다.

argv[0]이 NULL 이 아닌 경우 builtin_cmd 함수를 호출하여 command line 의 첫번째 단어가 built-in command 인지 확인하며, built-in command 의 경우 builtin_cmd 함수 내에서 즉시 실행이 이루어지며 1 이 반환된다. 따라서 built-in command 가 아니어서 0 이 반환된 경우가 아니라면 builtin_cmd 함수의 실행 후 eval 에서 리턴되도록 하였다.

built-in command 가 아니어서 0 이 반환된 경우에는 새로운 자식 프로세스를 생성하고 프로그램을 로드하여 실행시켜야 한다. fork() 호출을 통해 새로운 자식 프로세스가 생성되었을 때, 자식 프로세스가 커널에 의해 스케줄링되고 부모 프로세스가 다시 스케줄링되기 전에 종료되어 부모 프로세스가 다시 스케줄링되었을 때 SIGCHLD 펜딩 시그널에 의해 SIGCHLD 핸들러 루틴에서 delete job 을 수행하고 부모 프로세스의 메인 루틴으로 돌아가 addjob 이 수행되는 일이 발생할 수 있다. 이처럼 부모 프로세스에서 addjob 이 수행되기 전 delete job 이 수행되는 가능성을 방지하기 위해 부모 프로세스에서 자식 프로세스를 생성하기 전 SIGCHLD 시그널을 block 하는 것이 필요하다. 따라서 sigprocmask 를 이용해 SIGCHLD 시그널을 block 한 이후 fork()를 통해 자식 프로세스를 생성하도록 하였다.

fork 의 반환값이 0 인 경우, 즉 자식 프로세스인 경우에는 각 자식 프로세스들이 고유한 프로세스 그룹 ID 를 가질 수 있도록 setpgid(0, 0)의 호출을 통해 자식 프로세스를 그룹 ID 가 자식 프로세스의 프로세스 ID 와 동일한 그룹에 넣어주었다. 또한, 자식 프로세스는 부모 프로세스와 동일한 blocked 시그널 벡터를 가지게 되며, 자식 프로세스의 생성 전 부모 프로세스에서 SIGCHLD 시그널을 block 하였었기 때문에 자식 프로세스에서 sigprocmask 를 통해 SIGCHLD 를 unblock 해주었다. 이후 자식 프로세스에의 컨텍스트 내에서 새로운 프로그램을 로드하고 실행하기 위해 execve() 함수를 이용하였으며, execve()함수는 인자로 받은 파일 이름을 찾을 수 없는 경우에만 -1 을 리턴하기 때문에 해당 경우에 Command 를 찾을 수 없다는 에러메시지를 출력한 후 생성된 자식 프로세스를 종료시켜주었다.

fork 의 반환값이 0 이 아닌 자식 프로세스의 프로세스 ID 를 가지는 경우, 즉 부모 프로세스인 경우에는 새롭게 생성한 자식 프로세스를 job list 에 추가해주어야 한다. 이때 해당 job 이 background 에서 실행되어야 하는지 여부를 나타내는 bg 의 값을 통해 job 의 state 를 BG 또는 FG 로 설정해주었다. 또한 자식 프로세스가 생성되었지만, addjob 이 수행되기 전 시그널에 의해 addjob 이 수행되지 않는 것을 방지하기 위해 addjob 함수 호출 직전의 all_mask 를 이용해 모든 시그널을 block 해준 후 addjob 이 수행된 후 모든 시그널을

unblock 해주었다. 이후 해당 job 이 background job 이라면 job 의 정보를 출력하고, foreground job 이라면 waitfg 함수의 호출을 통해 foreground job 이 완료될 때까지 기다리도록 구현하였다.

eval 함수에서 사용하는 fork(), setpgid(), execve(), sigprocmask() 함수의 경우 시스템 호출(system call)로, 모두 에러로 인해 실패한 경우 -1 을 반환하는 함수이다. 따라서 함수의 호출과 함께 함수의 반환값이 0 보다 작은지 확인하여 0 보다 작은 경우 unix_error 함수나 printf 를 활용해 적절한 에러 메시지를 출력하도록 하였다.

- int builtin_cmd(char **argv)

builtin_cmd 함수에서는 cmdline 을 parsing 한 문자열들의 주소가 저장된 argv 배열을 인자로 받는다. 이때 cmdline 의 첫번째 단어가 built-in command 혹은 실행 가능한 파일의 경로이므로, argv[0]이 built-in command 인 경우 즉시 실행하고 그렇지 않은 경우 0 을 반환해주면 된다.

문자열 비교를 위해 〈string.h〉에 포함되어 있는 strcmp 함수를 사용했다. tsh shell 에서 지원하는 built-in command 인 jobs, quit, bg, fg 에 대해, argv[0]과 built-in command 가 같아 strcmp 의 반환값이 0 이 되는 경우 해당 command 를 즉시 실행시켰다.

bg 와 fg 는 모두 do_bgfg 함수를 통해 실행되므로, or 연산을 통해 처리해주고 매개변수를 argv 로 하여 do_bgfg 를 실행시켜주었다. 실행 중이거나 정지되어 있는 background job 들을 나열하는 jobs 명령어는 기존에 존재하는 listjobs 함수의 호출을 통해 실행시켜주었다. shell 을 종료시키는 quit 명령어는 exit 함수를 통해 shell 프로세스를 종료시켜주었다. 이때 quit 명령어를 제외하고는 eval 함수로 돌아가게 되므로, built-in command 가 실행되었음을 확인할 수 있게 하기 위해 1 을 반환시켰다. 조건문에 해당하지 않은 경우, 즉 argv[0]이 built-in command 가 아니었던 경우에 대해서는 0 을 반환시켜주었다.

builtin_cmd 함수에서 사용된 exit 함수는 시스템 호출(system call)이지만 함수를 호출하면 프로세스가 종료되어 버리므로 반환값을 확인하는 작업을 해주지 않았다.}

- void do_bgfg(char **argv)

do_bgfg 는 built-in command 인 bg 와 fg 를 실행하는 함수이며, do_bgfg 함수에서 사용한 변수는 아래와 같다.

char *id	bg, fg command 의 〈job〉 argument
char buf[MAXJID]	JID 로 주어진 〈job〉 argument 를 숫자로 변경하기 위해 사용하는 버퍼
struct job_t *job	〈job〉 argument 에 해당하는 job 의 포인터
pid_t pid	〈job〉 argument 에 해당하는 job 의 PID
int job_id	〈job〉 argument 에 해당하는 job 의 JID
int i	for loop 에서 사용되는 index variable

우선 〈job〉 argument 를 나타내는 argv[0]의 값을 id 변수에 복사하였고, id 가 NULL 이 아닌 경우에 대해 〈job〉 argument 로부터 PID 와 JID 를 구하였다. 우선 〈job〉 argument 의 첫번째 문자가 %인 경우, 즉 〈job〉 argument 가 JID 로 주어졌을 때는 두번째 문자부터 buf 로 옮긴 후 atoi() 함수를 통해 숫자로 변경해주고 getjobjid() 함수를 통해 해당 job 의 포인터를 구하였다. atoi() 함수는 인자로 받은 문자열을 숫자로 변경할 수 없는 경우 0 을 반환하며, JID 를 구한 경우에는 해당 JID 를 가지는 job 이 없는 경우 getjobjid() 함수가 NULL 을 반환하기 때문에, !job || !job_id 조건을 통해 해당 job 이 존재하지 않는 경우와 JID 가 숫자가 아닌 경우에 대해 해당 job 이 없음을 알리는 문구를 출력하도록 하였다. 정상적으로 JID 와 해당하는 job 을 구한 경우에는 구조체의 pid 변수에 접근하여 pid 값을 설정해주었다.

〈job〉 argument 가 PID 로 주어진 경우에는 바로 atoi() 함수를 통해 숫자로 변경해주었으며, getjobpid() 함수를 통해 해당 job 의 포인터를 구하였다. atoi() 함수는 인자로 받은 문자열을 숫자로 변경할 수 없으면 0 을 반환하므로, 해당 경우에 대해 argument 가 PID 또는 %jobid 가 되어야 한다는 문구를 출력해주었다. 만약 PID 는 구하였지만 해당 PID 에 해당하는 job 이 없어 getjobpid() 함수의 반환값이 NULL 이 되는 경우에는 해당 process 가 없다는 문구가 출력되도록 하였다.

이후, command 가 fg 일 때와 bg 일 때를 나누어 command 가 실행되도록 하였다. 우선 각각의 경우에서 〈job〉 argument 가 주어지지 않은 경우, 즉 id 가 NULL 인 경우에는 PID 또는 %jobid 가 필요하다는 메시지와 함께 즉시 리턴하도록 하였다. 〈job〉 argument 가 주어진 경우에 대해서는 위에서 구했던 JID 와 PID 를 통해 command 를 실행하였다.

command 가 fg 인 경우 kill() 함수를 사용했으며, -pid 를 인자로 사용하여 해당 PID 를 가지는 프로세스의 그룹의 모든 프로세스에게 SIGCONT 시그널을 보내주고, job 의 state 를 FG 로 수정하며 waitfg 함수를 호출하여 해당 job 이 종료되는 것을 기다리도록 하였다.

command 가 bg 인 경우 kill() 함수를 사용했으며, -pid 를 인자로 사용하여 해당 PID 를 가지는 프로세스의 그룹의 모든 프로세스에게 SIGCONT 시그널을 보내주고, job 의 state 를 BG 로 바꾸며 해당 job 의 내용을 출력하고 리턴하도록 하였다.

do_bgfg 함수에서 사용된 kill() 함수는 시스템 호출(system call)로, 에러 발생 시 -1 을 리턴하기 때문에 kill() 함수의 반환값이 0 보다 작은 경우 unix_error 함수 호출을 통해 적절한 error 메시지가 출력되도록 하였다.

- void waitfg(pid_t pid)

waitfg 함수에서는 인자로 받은 pid 를 프로세스 ID 로 갖는 foreground 프로세스가 종료될 때까지 기다린다. waitfg 함수에서 사용한 변수는 아래와 같다.

struct job_t *fg_job	foreground job 의 포인터

현재 foreground job 이 존재한다면 PID 를, 존재하지 않는다면 0 을 반환하는 fgpid 함수를 이용하였으며, foreground job 이 종료되면 pid 와 fgpid 의 반환값이 달라지게 되므로 인자로 받은 pid 와 fgpid 의 반환값이 같은 동안 반복되는 busy loop 를 사용하였다. 또한 verbose flag 의 설정 여부에 따라 인자로 받은 pid 에 해당하는 process 가 더 이상 foreground process 가 아님을 출력하도록 하였다.

- **void sigchld_handler(int sig)**

sigchld_handler 에서는 child job 이 종료되거나 정지하는 경우 커널이 보내는 SIGCHLD 을 받아 핸들링한다. sigchld_handler 함수에서 사용한 변수는 아래와 같다.

pid_t pid	waitpid 의 반환값
int job_id	종료되거나 정지된 job 의 JID
int wstatus	종료되거나 정지된 자식 프로세스의 상태
struct job_t *job	종료되거나 정지된 job 의 포인터
int olderrno	errno 의 복원을 위해 기존의 errno 저장
sigset_t all_mask, prev_mask	모든 시그널을 가지는 시그널 집합, 이전에 block 되던 시그널을 가지는 시그널 집합

waitpid 의 첫번째 인자인 pid 의 설정에 따라 waitpid 의 대기 집합이 결정되며, 이는 아래와 같다.

pid	action
-1	임의의 자식 프로세스
> 0	프로세스 ID 가 pid 와 동일한 자식 프로세스
0	프로세스 그룹 ID 가 waitpid 를 호출한 프로세스의 프로세스 ID 와 동일한 자식 프로세스
<-1	프로세스 그룹 ID 가 pid 인자의 절댓값과 동일한 자식 프로세스

묵시적 시그널 block 으로 인해 핸들러에 의해 처리되고 있는 유형의 시그널은 block 되며, 시그널은 큐에 들어가지 않기 때문에 시그널 핸들러가 시그널을 처리하는 동안 도착한 시그널은 이후 시그널 핸들러에 의해 처리되지 않을 수 있다. 따라서 한 번 SIGCHLD 시그널로 인해 시그널 핸들러가 호출되었을 때 종료되거나 중지되는 자식 프로세스들을 모두 처리하는 것이 필요하며, 이를 위해 while 문을 사용하였고 임의의 자식 프로세스에 대하여 waitpid 를 실행하기 위해 첫번째 인자 pid 를 -1 로 설정하였다.

또한, waitpid 의 세번째 인자인 options 을 아래와 같은 상수들로 설정하여 waitpid 의 기본 동작을 수정할 수 있다.

options	action
WNOHANG	자식 프로세스가 종료되기를 기다리지 않고, 자식 프로세스가 종료되어 있지 않다면 즉시 0 을 반환한다.
WUNTRACED	waitpid 의 기본 동작은 대기 집합의 프로세스가 종료되었을 때만 반환하는 것이지만, WUNTRACED 를 사용하면 대기 집합의

	프로세스가 종료되거나 정지될 때까지 호출한 프로세스의 실행을 정지하며 해당 프로세스의 PID 를 반환하게 된다.
WCONTINUED	대기 집합의 실행중인 프로세스가 종료될 때까지, 혹은 정지되어 있는 프로세스가 SIGCONT 를 받아 다시 실행될 때까지 호출한 프로세스의 실행을 정지한다.

sigchld_handler 에서는 대기 집합의 프로세스가 종료되기를 기다리지는 않으며, 종료되거나 중지된 프로세스들에 핸들링을 해야 하므로 WNOHANG | WUNTRACED 의 옵션을 사용하면 만약 대기 집합의 자식 프로세스 중 종료되거나 중지된 것이 없다면 즉시 0 이 반환되며, 종료되거나 중지된 자식 프로세스가 존재한다면 해당 자식의 PID 가 반환되도록 하였다. 따라서 while 문이 waitpid 의 반환값이 0 보다 큰 경우동안 반복되도록 구현하였다.

waitpid 함수 호출 시 두번째 인자가 NULL 이 아니면 두번째 인자가 가리키는 주소에 waitpid 가 리턴하게 만든 자식 프로세스의 상태정보가 status 로 인코딩되어 저장된다. 인코딩된 자식 프로세스의 상태 정보 status 를 해석하기 위해 아래와 같은 매크로가 존재한다.

WIFEXITED(status)	exit, return 을 통해 자식 프로세스가 정상적으로 종료한 경우 true 반환
WEXITSTATUS(status)	WIFEXITED()가 true 를 반환하는 경우에 정의되며, 정상적으로 종료된 자식 프로세스의 exit status 반환
WIFSIGNALED(status)	자식 프로세스가 시그널로 인해 종료한 경우 true 반환
WTERMSIG(status)	WIFSIGNALED()가 true 를 반환하는 경우에 정의되며, 자식 프로세스를 종료시킨 시그널의 번호 반환
WIFSTOPPED(status)	자식 프로세스가 현재 중지되어 있는 경우 true 반환
WSTOPSIG(status)	WIFSTOPPED()가 true 를 반환하는 경우에 정의되며, 자식 프로세스를 중지시킨 시그널의 번호 반환
WIFCONTINUED(status)	자식 프로세스가 SIGCONT 시그널에 의해 다시 시작된 경우 true 반환

위의 매크로 중 WIFEXITED, WIFSIGNALED, WIFSTOPPED 를 사용하여 SIGCHLD 시그널을 받았을 때, 해당 시그널이 자식 프로세스가 정상적으로 종료되어 발생한 것인지, 시그널로 인해 종료되어 발생한 것인지, 자식 프로세스가 중지되어 발생한 것인지 확인한다. 각각의 경우에 대해 WEXITSTATUS, WTERMSIG, WSTOPSIG 의 매크로를 이용하여 각 상황에 맞는 메시지가 출력될 수 있도록 하였다. 또한, 자식 프로세스가 종료된 경우에는 deletejob 함수 호출을 통해 해당 job 을 joblist 에서 제거하고, 자식 프로세스가 중지된 경우에는 job 의 state 를 ST 로 변경해주도록 하였다. 또한 deletejob 함수가 실행되는 동안에는 모든 시그널을 block 하도록 sigprocmask 를 통해 설정하였다.

sigchld_handler 함수에서 사용하는 sigprocmask() 함수는 시스템 호출(system call)로, 에러로 인해 실패한 경우 -1 을 반환하는 함수이다. 따라서 함수의 호출과 함께 함수의 반환값이 0 보다 작은지 확인하여 0 보다 작은 경우 unix_error 함수를 통해 적절한 에러 메시지를 출력하도록 하였다.

verbose flag 가 설정되어 있는 경우 핸들링의 과정이 출력될 수 있도록 하였고, errno 를 저장하였다가 복원하도록 구현하였다.

- void sigint_handler(int sig)

sigint_handler 함수는 사용자가 입력하는 SIGINT(ctrl-c) 시그널을 잡아 핸들링하는 사용자 정의 핸들링 함수이며, sigint_handler 함수에서 사용한 변수는 아래와 같다.

pid_t pid	foreground job 의 프로세스 ID
int olderrno	errno 의 복원을 위해 기존의 errno 저장

sigint_handler 는 사용자가 ctrl-c 를 입력하였을 때 foreground job 에게 SIGINT 시그널을 보내주어야 한다. 따라서 우선 fgpid 함수를 호출하여 현재 foreground job 의 프로세스 ID 를 반환받아 pid 변수에 저장하였고, foreground job 이 존재하여 pid 가 0 이 아닌 경우 kill 함수를 통해 foreground 프로세스 그룹에 속하는 프로세스에 SIGINT 시그널을 보내주었다.

kill 함수의 경우 시스템 호출(system call)이며, 오류 발생시 -1 을 반환한다. 따라서 kill 함수의 반환값이 0 보다 작을 시 unix_error 함수의 호출을 통해 적절한 에러메시지를 출력하도록 했다. 또한 sigint_handler 함수에서 errno 를 저장하고 복원하도록 구현하였다.

또한 함수의 호출 과정에서 verbose flag 가 설정되어 있다면 과정이 출력되도록 구현하였다.

- void sigtstp_handler(int sig)

sigtstp_handler 함수는 사용자가 입력하는 SIGTSTP(ctrl-z) 시그널을 잡아 핸들링하는 사용자 정의 핸들링이며, sigtstp_handler 함수에서 사용한 변수는 아래와 같다.

pid_t pid	foreground job 의 프로세스 ID
int olderrno	errno 의 복원을 위해 기존의 errno 저장

sigtstp_handler 는 사용자가 ctrl-z 를 입력하였을 때 foreground job 에게 SIGTSTP 시그널을 보내주어야 한다. 따라서 우선 fgpid 함수를 호출하여 현재 foreground job 의 프로세스 ID 를 반환받아 pid 변수에 저장하였다. foreground job 이 존재하여 pid 에 0 이 아닌 값이 저장된

경우 kill 함수를 통해 foreground 프로세스 그룹에 속하는 프로세스에 SIGTSTP 시그널을 보내주었다.

kill 함수의 경우 시스템 호출(system call)이며 오류 발생 시 -1 을 반환하므로, kill 함수의 반환값이 0 보다 작을 경우 unix_error 함수 호출을 통해 적절한 에러 메시지가 출력되도록 하였다. 또한 sigtstp_handler 함수가 errno 를 저장했다가 복원하도록 구현하였다.

또한 verbose flag 가 설정되어 있다면 핸들링 과정이 출력될 수 있도록 구현하였다.

4. 결론

이번 Lab 을 수행하면서 Shell 의 main routine 을 작성해 봄으로써 fork, execve 함수를 통해 새로운 프로세스를 생성하고, 프로그램을 로드하고 실행하는 것을 실습해볼 수 있었다. 또한, foreground job 과 background job 에 따른 job control 과 부모 프로세스가 종료된 자식 프로세스를 모두 reaping 할 수 있도록 하는 방안에 대해 생각해볼 수 있었다. 그 과정에서 kill 을 통해 특정 프로세스로 시그널을 보내거나, sigprocmask 함수를 호출해 특정 시그널을 블록/블록 해제하는 등 시그널과 관련된 다양한 함수를 직접 사용해볼 수 있었다.