2018 시스템 프로그래밍
- Lab 08 -

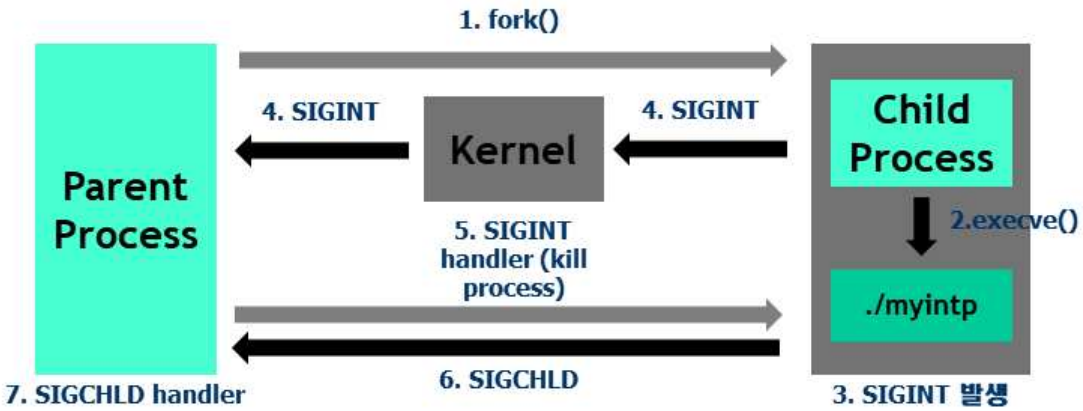
제출일자	2018.11.25
분 반	02
이 름	박상현
학 번	201702012

trace08 [화면 캡쳐] - 결과 화면

```
c201702012@2018-sp:~/shlab-handout$ ./tsh
eslab_tsh> ./myintp
Job [1] (26610) terminated by signal 2
eslab_tsh>
```

```
c201702012@2018-sp:~/shlab-handout$ ./sdriver -t 08 -s ./tsh -V
Running trace08.txt...
Success: The test and reference outputs for trace08.txt matched!
Test output:
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
tsh> ./myintp
Job [1] (25612) terminated by signal 2
tsh> quit

Reference output:
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
tsh> ./myintp
Job [1] (25620) terminated by signal 2
tsh> quit

c201702012@2018-sp:~/shlab-handout$
```

trace08 [과정 설명] - 설명 & flowchart 진행 과정



다음은 trace08의 진행과정이다. 먼저, 이전 실습들을 통해 fork()하여 자식을 생성하고, 자식이 execve()를 통해 프로세스를 실행하도록 하였다. 이 trace에서는 myintp를 실행시킨다. 우선 myintp가 어떤 파일인지 알아보자.

```
18 int main()
19 {
20     signal(SIGALRM, sigalrm_handler);
21     alarm(JOB_TIMEOUT);
22
23     if (kill(getppid(), SIGINT) < 0) {
24     perror("kill");
25     exit(1);
26     }
27
28     while(1);
29     exit(0);
30 }
```

23줄에서 보듯이, myintp는 getppid()를 이용하여 부모의 pid를 얻어, 이를 인자로 부모에게 kill함수를 통해 명령을 전달한다. 이 때 전달하는 명령은 SIGINT이며, 이는 커널을 통해 부모에게 전달된다. 부모가 SIGINT를 전달받으면, main()함수에 의해 sigint_handler를 호출하여 자식프로세스를 죽이도록 명령한다. 죽은 자식프로세스는 SIGCHLD를 부모에게 보내게 되는데, 부모는 이를 받아 또다른 작업을 처리할 수 있다. 여기에서는 자식을 제거하고, 화면에 제거한 자식을 출력해주었다.

```
void sigchld_handler(int sig)
{
    int child_status = 0;
    pid_t pid;

    while((pid = waitpid(-1, &child_status, WNOHANG | WUNTRACED ) ) > 0 ){  // 자식프로세스 종료를 대기
        if(WIFSIGNALED(child_status) != 0){ // 자식프로세스 종료를 체크
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
            deletejob(jobs, pid);   // 자식프로세스를 목록에서 제거
        }
        else if(WIFSTOPPED(child_status) != 0){
            getjobpid(jobs, pid)->state = ST;
            //jobs[pid2jid(pid)-1].state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
        }
        else if(WIFEXITED(child_status) != 0){
            deletejob(jobs, pid);
        }
        else{
            unix_error("waitpid error\n");
        }
    }

    return;
}
```

이는 최종 sigchld_handler인데, 빨간색 네모가 trace08의 SIGINT 처리 관련 내용이다. 이 handler는 자식의 종료를 대기한다. 자식이 종료하였을 때, child_status를 확인하여 만약 signal에 의해 종료되었을 경우 제거할 자식을 출력하고, 자식을 jobs[]에서 제거해준다.

이제 mask와 race condition에 대한 설명이다.

밑의 eval()함수를 보자. 이전의 실습과는 달리, sigset_t mask라는 sigset_t 타입의 mask 변수를 선언해준다. 이 mask를 이용하여 특정 시그널들을 블록하고 해제하는 등의 작업을 수행할 수 있다.

여기서 signal을 왜 block해야 하는지가 중요하다. 결론부터 말하면, Race condition을 없애기 위함이다. 구체적으로는, 부모프로세스가 자식의 작업을 add하기도 전에 자식의 작업이 끝나는 것인데, 만일 이렇게되면 끝난 작업을 작업리스트에 추가하게 되는 꼴이 된다. 실행과정을 살펴보면, 자식의 프로세스가 부모가 add하기 전에 끝난다. 이 때 자식의 프로세스는 부모에게 SIGCHLD를 보내고, 부모는 main()에 의해 sigchld_handler로 이동해 SIGCHLD를 처리하게 된다. 만에하나 sigchld_handler에 작업리스트에서 delete하는 작업이 있다면, 있지도 않은 작업을 delete하는 황당한 경우가 생길 수 있는 여지가 생긴다. 따라서 이를 방지해야한다. 이를 방지하기 위해서는 trace08에서는 SIGINT와 SIGCHLD를 block한 mask를 만든다. 그 후에 fork 이전에 sigprocmask(SIG_BLOCK, &mask, NULL)를 사용하여 mask에 add된 signal들을 블록한다. 그다음 fork를 수행하는데, 이 때 자식은 부모의 block된 mask를 상속받기 때문에 unblock을 해주어야한다. signal을 unblock해준 자식이 execve()를 통해 myintp를 실행하여 SIGINT를 부모에게 보낸다. 하지만 부모는 지금 SIGINT와 SIGCHLD가 block된 mask를 쓰고 있다. 따라서 부모는 race condition이 끝나는 지점인 addjob 다음에 unblock을 하여 자식의 SIGINT를 받고 자식을 죽이고 자식에게

SIGCHLD를 받아 작업을 수행한다.

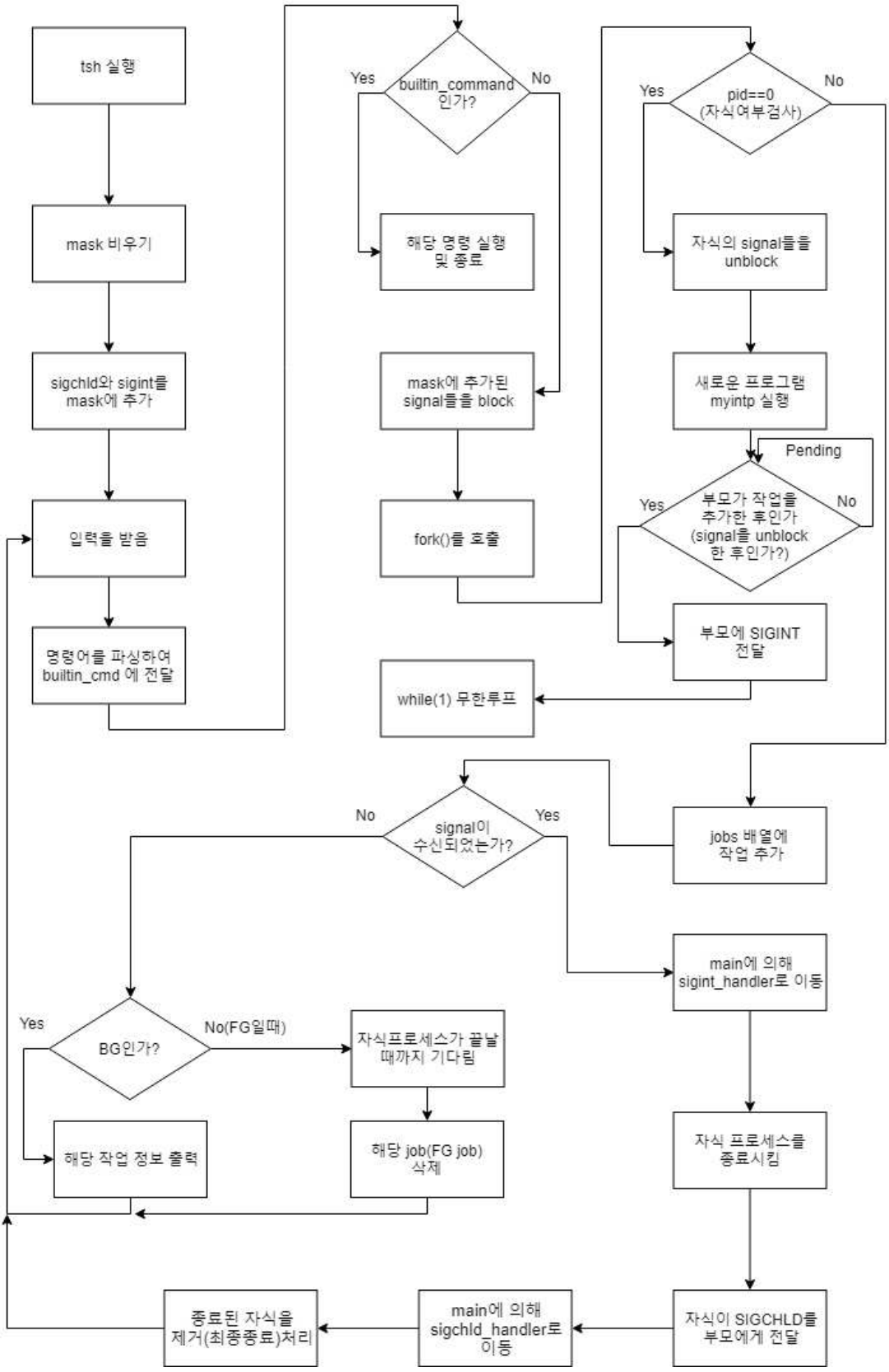
```
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
 */
void eval(char *cmdline) //     ///////////////////////////////////////////////////////eval/////////
{

    char *argv[MAXARGS];    //  command 저장
    pid_t pid;
    int bg;
    sigset_t mask;

    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGTSTP);

    bg = parseline(cmdline, argv);

    if(!builtin_cmd(argv)){

        sigprocmask(SIG_BLOCK, &mask, NULL);    //  시그널 블록

        if( ( pid = fork() ) == 0 ){    //  Child Process 인 경우 , execve() 수행
            setpgid(0, 0);
            sigprocmask(SIG_UNBLOCK, &mask, NULL);//    시그널 언블록

                if((execve(argv[0], argv, environ) < 0 )){
                    printf("%s : command not found\n", argv);
                    exit(0);
                }
        }

        addjob(jobs, pid, (bg == 1 ? BG :  FG), cmdline);   //  add a job to array jobs[]
        sigprocmask(SIG_UNBLOCK, &mask, NULL);  //    시그널 언블록

        if(!bg){ /*foreground job*/

            waitfg(pid, 1);

        }
        else{    /*background job*/

            //  백그라운드 작업 수행시 작업 내용 출력

            //      jid  pid, cmdline
            printf("([%d]) (%d) %s", pid2jid(pid), pid, cmdline);
        }

    }

    return;
}
```

위 eval()함수는 최종 eval함수이다.

또다른 race condition이 foreground에서 생긴다. 프로세서는 한순간 하나의 foreground를 수행하도록 되어있는데, foreground 프로세스가 실행되는 도중에 fork가 진행되면 두 개의 프로세스가 foreground로 실행될 가능성이 있다. 이를 방지하기 위해 이전에 작성한 waitfg()를 수정하여 실행되고 있는 foreground 작업이 종료될 때까지 wait해주도록 코드를 짠다.

```
void waitfg(pid_t pid, int output_fd)
{

    struct job_t *j = getjobpid(jobs, pid);
    char buf[MAXLINE];

    //  The FG job has already completed and been reaped by the handler
    if(!j)
        return;

    /*
       Wait for process pid to longer be the foreground process.
       Note : using pause() instead of sleep() would introduce a race
       that could cause us to miss the signal
     */
    while(j->pid == pid && j->state == FG)
        sleep(1);
    if(verbose){
        memset(buf, '\0', MAXLINE);
        sprintf(buf, "waitfg: Process (%d) no longer the fg process:q\n", pid);
        if(write(output_fd, buf, strlen(buf)) < 0){
            fprintf(stderr, "Error writing to file\n");
            exit(1);
        }
    }
    return;

    /*pid = waitpid(pid, NULL, output_fd);
      deletejob(jobs, pid);
      return;
     */
}
```

다음은 trace08의 flowchart이다. 기존의 trace에서 추가하여 작성해보았다.



trace09 [화면 캡쳐] - 결과 화면

```
c201702012@2018-sp:~/shlab-handout$ ./tsh
eslab_tsh> ./mytstpp
Job [1] (26674) stopped by signal 20
eslab_tsh>
```

```
c201702012@2018-sp:~/shlab-handout$ ./sdriver -t 09 -s ./tsh -V
Running trace09.txt...
Success: The test and reference outputs for trace09.txt matched!
Test output:
#
# trace09.txt - Send SIGTSTP to foreground job.
#
tsh> ./mytstpp
Job [1] (25800) stopped by signal 20
tsh> jobs
(1) (25800) Stopped    ./mytstpp

Reference output:
#
# trace09.txt - Send SIGTSTP to foreground job.
#
tsh> ./mytstpp
Job [1] (25808) stopped by signal 20
tsh> jobs
(1) (25808) Stopped ./mytstpp

c201702012@2018-sp:~/shlab-handout$
```

trace09 [과정 설명] - 설명 & flowchart 진행 과정

trace08에서 SIGINT를 발생하여 FG job(./myintp)을 종료시켰다면, trace09는 SIGTSTP을 발생하여 FG job(./mytstpp)을 stop시킨다.

signal을 전송하고 수신하는 과정은 trace08과 동일하다. 따라서 구체적인 설명은 하지 않겠다. 한가지 고려사항은 eval()함수에 SIGTSOP signal을 mask에 추가하여준다는 것이다.

```
int main()
{
    signal(SIGALRM, sigalrm_handler);
    alarm(JOB_TIMEOUT);

    if (kill(getppid(), SIGTSTP) < 0) {
    perror("kill");
    exit(1);
    }

    while(1);
    exit(0);
}
```

위는 mytstpp.c의 코드인데, myintp와 비슷하게 작동하지만, 부모에게 보내는 signal이 SIGTSTP으로 바뀐 것을 알 수 있다.

다음은 sigchld_handler의 SIGINT가 아닌 SIGTSTP의 처리 코드이다.
SIGTSTP이 오면, 정지된 자식을 기다려야 하므로 waitpid의 option에 WUNTRACED를 넣어준다.

```
void sigchld_handler(int sig)
{
    int child_status = 0;
    pid_t pid;

    while((pid = waitpid(-1, &child_status, WNOHANG | WUNTRACED ) ) > 0 ){  //  자 식 프 로 세 스  종 료 를  대 기
        if(WIFSIGNALED(child_status) != 0){ //  자 식 프 로 세 스  종 료 를  체 크
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
            deletejob(jobs, pid);   //  자 식 프 로 세 스 를  목 록 에 서  제 거

        else if(WIFSTOPPED(child_status) != 0){
            getjobpid(jobs, pid)->state = ST;
            //jobs[pid2jid(pid)-1].state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
        }
        else if(WIFEXITED(child_status) != 0){
            deletejob(jobs, pid);
        }
        else{
            unix_error("waitpid error\n");
        }
    }

    return;
}
```

위의 코드에서 주석으로 단 부분은 jobs배열을 통해 해당 jid에 접근 후 state에 접근하여 상태정보를 변경한 코드이고, 그 윗줄의 코드는 getjobpid(jobs, pid)함수를 이용하여 해당 jid에 접근 후 state에 접근하여 상태정보를 변경한 코드이다. 두가지 모두 사용 가능하다.

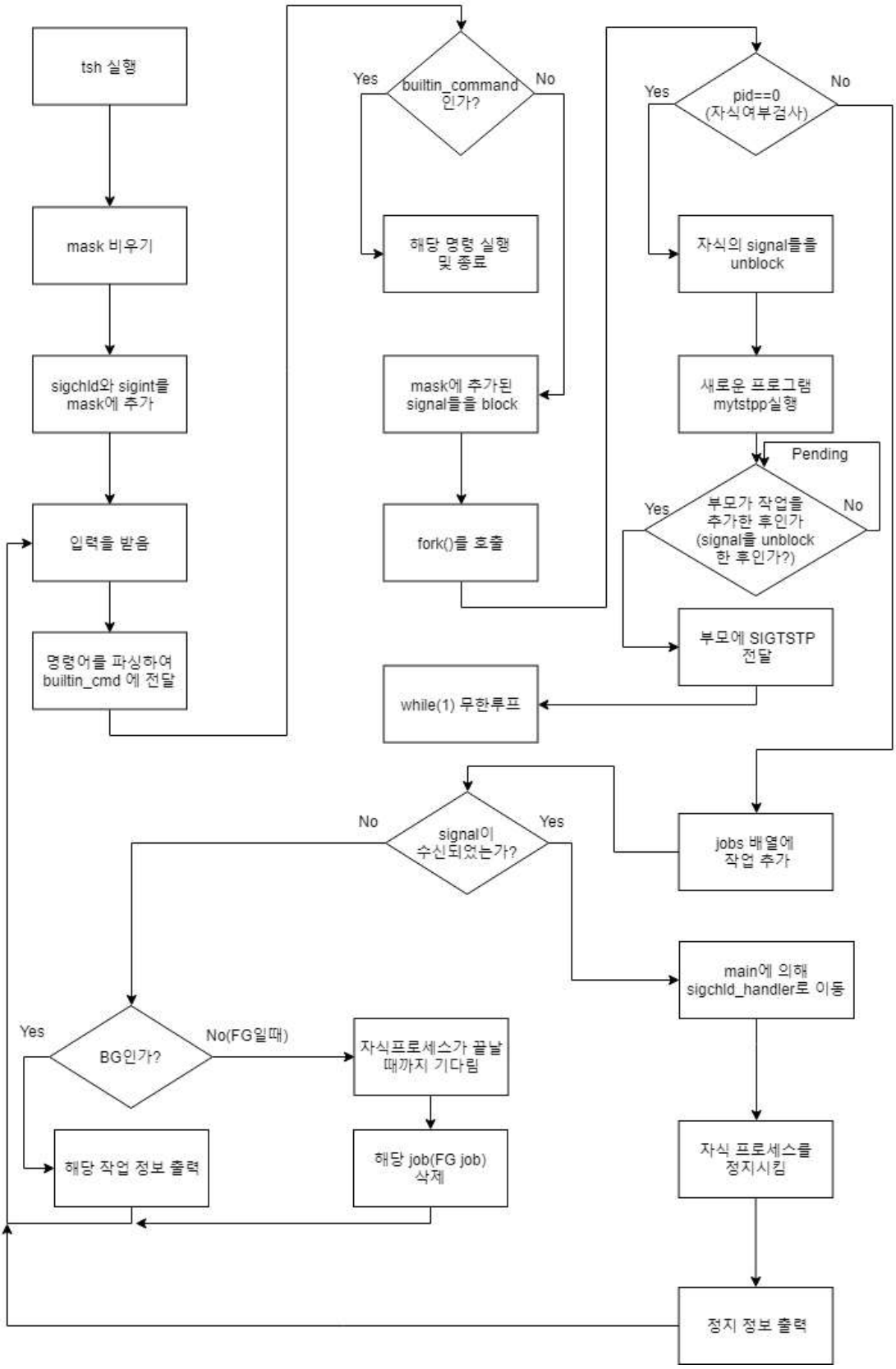
```
/* getjobpid  - Find a job (by PID) on the job list */
struct job_t *getjobpid(struct job_t *jobs, pid_t pid) {
    int i;

    if (pid < 1)
        return NULL;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid)
            return &jobs[i];
    return NULL;
}
```

race condition들(addjob, waitfg)의 경우도 역시 trace08과 같은 관계로 설명은 생략한다.

다음은 sigtstp_handler()이다. sigint_handler와 동일하다.

```
/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if(pid != 0){
        kill(pid, SIGTSTP/*sig*/);
    }
    return;
}
```

다음은 trace09의 flowchart이다. trace08과 거의 유사하다.



trace10 [화면 캡쳐] - 결과 화면

```
c201702012@2018-sp:~/shlab-handout$ ./tsh
eslab_tsh> ./myspin1 5 &
(1) (27020) ./myspin1 5 &
eslab_tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
eslab_tsh>
```

```
c201702012@2018-sp:~/shlab-handout$ ./sdriver -t 10 -s ./tsh -V
Running trace10.txt...
Success: The test and reference outputs for trace10.txt matched!
Test output:
#
# trace10.txt - Send fatal SIGTERM (15) to a background job.
#
tsh> ./myspin1 5 &
(1) (25328) ./myspin1 5 &
tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
tsh> quit

Reference output:
#
# trace10.txt - Send fatal SIGTERM (15) to a background job.
#
tsh> ./myspin1 5 &
(1) (25338) ./myspin1 5 &
tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
tsh> quit

c201702012@2018-sp:~/shlab-handout$
```

trace10 [과정 설명] - 설명 & flowchart 진행 과정

trace10에서는 background작업이 정상 종료될 경우 SIGCHLD가 발생하는데, 이 signal을 sigchld_handler가 처리하여 BG job을 종료(제거)처리해주는 trace이다.

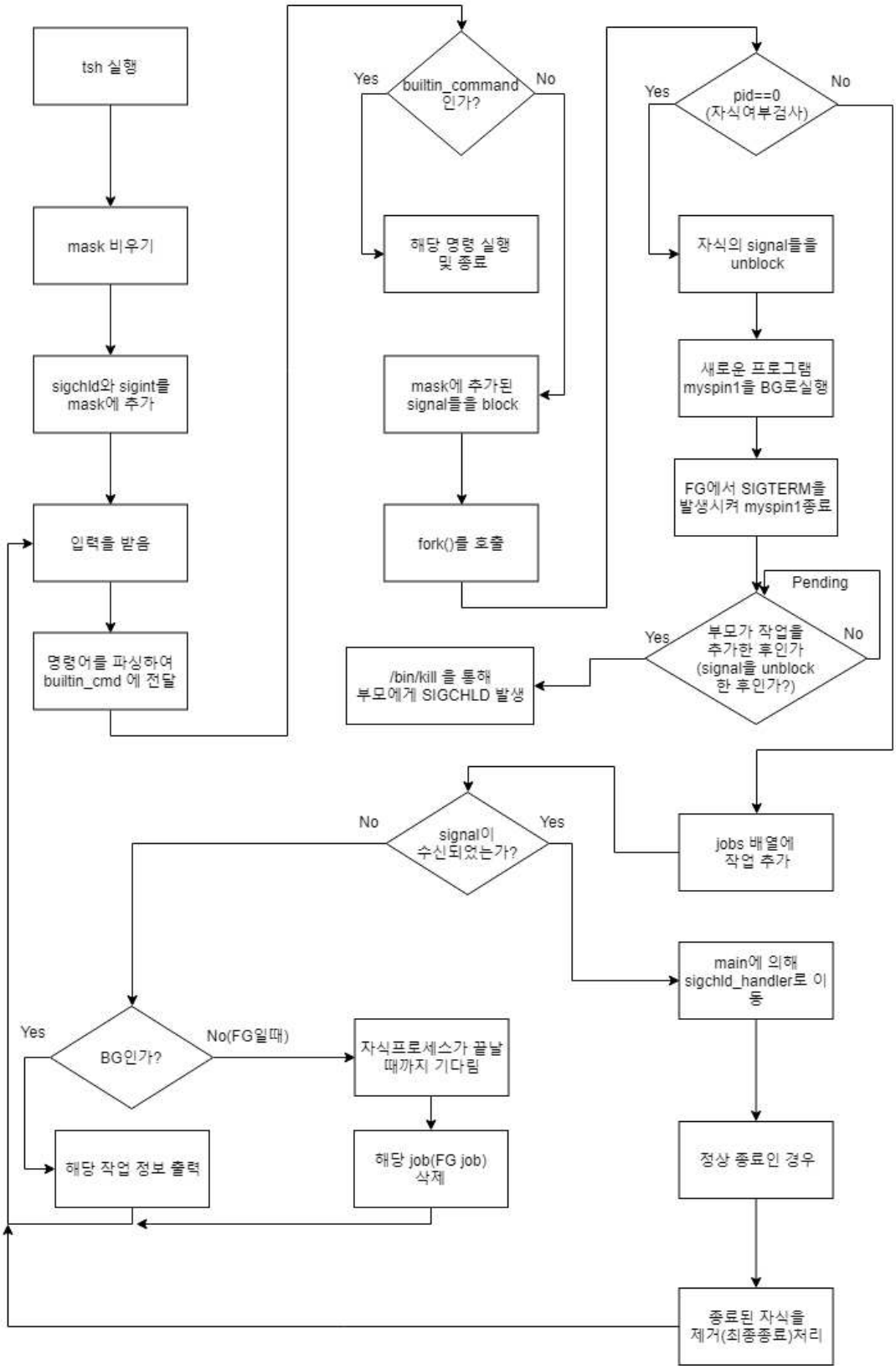
```
void sigchld_handler(int sig)
{
    int child_status = 0;
    pid_t pid;

    while((pid = waitpid(-1, &child_status, WNOHANG | WUNTRACED ) ) > 0 ){  // 자식프로세스 종료를 대기
        if(WIFSIGNALED(child_status) != 0){ //  자식프로세스 종료를 체크
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
            deletejob(jobs, pid);   //  자식프로세스를 목록에서 제거
        }
        else if(WIFSTOPPED(child_status) != 0){
            getjobpid(jobs, pid)->state = ST;
            //jobs[pid2jid(pid)-1].state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
        }
        else if(WIFEXITED(child_status) != 0){
            deletejob(jobs, pid);
        }
        else{
            unix_error("waitpid error\n");
        }
    }

    return;
}
```

위 사진은 sigchld_handler에서 작업이 정상종료될 경우 발생하는 SIGCHLD를 선택적으로 골라서 해당 원하는 작업을 처리하는 모습이다.
코드를 자세히 보면, WIFEXITED(child_status)가 true일 경우, 즉 정상종료된 child의 경우 저 else-if문에 진입하게 되고, 종료된 자식프로세스를 제거(최종종료)해준다.
정상종료시의 waitpid의 option은 0 또는 2일것같다. 0은 종료된 자식을 기다리는 것이고, 2는 정지되거나 종료된 자식을 기다리는 것이기 때문이다. 다만, 0을 넣었을때의 trace10의 동작 여부는 코드상의 실험은 해보지 않았다.

race condition들(addjob, waitfg)의 경우도 역시 trace08과 같은 관계로 설명은 생략한다.

SIGCHLD는 이미 trace08에서 모두 처리해주었기 때문에, sigchld_handler() 수정 외에는 따로 추가할 작업이 없다.

다음은 trace10의 flowchart이다.



trace11 [화면 캡쳐] - 결과 화면

```
#
# trace11.txt - Child sends SIGINT to itself
#
/bin/echo -e tsh\076 ./myints
NEXT
./myints
NEXT

/bin/echo -e tsh\076 quit
NEXT
quit
```

```
c201702012@2018-sp:~/shlab-handout$ ./tsh
eslab_tsh> myints
Job [1] (3918) terminated by signal 2
eslab_tsh>
```

```
c201702012@2018-sp:~/shlab-handout$ ./sdriver -t 11 -s ./tsh -V
Running trace11.txt...
Success: The test and reference outputs for trace11.txt matched!
Test output:
#
# trace11.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (28243) terminated by signal 2
tsh> quit

Reference output:
#
# trace11.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (28251) terminated by signal 2
tsh> quit

c201702012@2018-sp:~/shlab-handout$
```

trace11 [과정 설명] - 설명 & **자동으로 통과되는 이유** & flowchart(생략)

trace11은 자식프로세스가 자신에게 스스로 SIGINT를 전송하였을 때 정상적으로 SIGINT 처리가 되도록 구현하는 trace이다.

trace08에서는 myintp를 실행한 반면, trace11에서는 myints를 실행하는 것을 확인할 수 있다.

먼저, myints를 분석해보자.

```
int main()
{
    signal(SIGALRM, sigalrm_handler);
    alarm(JOB_TIMEOUT);

    if (kill(getpid(), SIGINT) < 0) {
    perror("kill");
    exit(1);
    }

    while(1);
    exit(0);
}
```

이전의 trace08의 kill()함수에서는 getppid()를 통해 부모에게 SIGINT signal을 전달했던것에 반해 trace11은 kill()함수에서 자기 자신의 pid인 getpid()를 통해 자신에게 SIGINT signal을 전달하는 것이다. 따라서, 누구에게 보내느냐만 달라질 뿐, 그 이후의 일은 sigint_handler와 sigchld_handler가 각각 알아서 동일하게 작동하는 것이다. 따라서 자동으로 통과되는 것이였다.

trace11의 flowchart는 trace08의 것과 일치한다.
따라서 추가하지 않겠다.

trace12 [화면 캡쳐] - 결과 화면

```
 1 #
 2 # trace12.txt - Child sends SIGTSTP to itself
 3 #
 4 /bin/echo -e tsh\076 ./mytstps
 5 NEXT
 6 ./mytstps
 7 NEXT
 8
 9 /bin/echo -e tsh\076 jobs
10 NEXT
11 jobs
12 NEXT
13
14 quit
```

```
c201702012@2018-sp:~/shlab-handout$ ./tsh
eslab_tsh> ./mytstps
Job [1] (3950) stopped by signal 20
eslab_tsh>
```

```
c201702012@2018-sp:~/shlab-handout$ ./sdriver -t 12 -s ./tsh -V
Running trace12.txt...
Success: The test and reference outputs for trace12.txt matched!
Test output:
#
# trace12.txt - Child sends SIGTSTP to itself
#
tsh> ./mytstps
Job [1] (28354) stopped by signal 20
tsh> jobs
(1) (28354) Stopped    ./mytstps

Reference output:
#
# trace12.txt - Child sends SIGTSTP to itself
#
tsh> ./mytstps
Job [1] (28362) stopped by signal 20
tsh> jobs
(1) (28362) Stopped ./mytstps

c201702012@2018-sp:~/shlab-handout$
```

trace12 [과정 설명] - 설명 & **자동으로 통과되는 이유** & flowchart(생략)

trace12은 자식프로세스가 자신에게 스스로 SIGTSTP를 전송하였을 때 정상적으로 SIGTSTP 처리가 되도록 구현하는 trace이다.

trace09에서는 mytstpp를 실행한 반면, trace12에서는 mytstps를 실행하는 것을 확인할 수 있다.

먼저, mytstps를 분석해보자.

```
int main()
{
    if (kill(getpid(), SIGTSTP) < 0) {
    perror("kill");
    exit(1);
    }
    exit(0);
}
```

이것 또한 trace11과 굉장히 유사하다.
이전의 trace09의 kill()함수에서는 getppid()를 통해 부모에게 SIGINT signal을 전달했던것에 반해 trace12은 kill()함수에서 자기 자신의 pid인 getpid()를 통해 자신에게 SIGTSTP signal을 전달하는 것이다. 따라서, 누구에게 보내느냐만 달라질 뿐, 그 이후의 일은 sigtstp_handler가 각각 알아서 동일하게 작동하는 것이다. 따라서 자동으로 통과되는 것이였다.

trace12의 flowchart는 trace09의 것과 일치한다.
따라서 추가하지 않겠다.

trace06 자동으로 통과되는 이유

저번 보고서에서 당연하게 여겨 작성하지 않았던 trace06의 작동 이유입니다.
저번 보고서에서도 드러나긴 하지만,
"직접 명시"하지 않았기에, 이번보고서에서라도 추가합니다.

기본적으로 trace05의 목표는 background로 작업을 실행하는 것입니다.
foreground로의 작업실행은 이전의 trace들에서 다뤘던 것들입니다.(execve()함수 사용)

trace06은 myspin1을 BG에서, myspin2 를 FG에서 실행하여 동시에 FG와 BG작업이 실행되도록 하는 trace입니다.
먼저 trace05에서 BG실행을 구현하였기에 "myspin1 &" 은 정상작동 될것이며, 그 작업의 jid, pid, cmdline을 출력할 것입니다.
다음으로 실행할(execve()할) "myspin2 1"은 FG로 실행되어 별 문제없이 작동하는 것입니다.
따라서 flowchart도 새롭게 추가할 필요가 없을것입니다.