Lab Assignment 2: Shell Lab

공과대학 컴퓨터공학부
2020-14378
윤교준 / Gyojun Youn
youngyojun@snu.ac.kr

구현

eval

`void eval(char *cmdline)`은 shell input을 parsing한 후 적절한 처리를 수행해야 하는, shell의 main과도 같은 함수다.

먼저, `parseline` 함수를 이용하여, input string의 기계적인 parse 작업을 수행하고, 빈 줄과 built-in commands에 대하여 먼저 처리하였다.

```
void eval(char *cmdline)
{
  char *argv[MAXARGS];
  pid_t pid;    /* process id */
  int bg;       /* background job? */

  /* Parse the command line. */
  bg = parseline(cmdline, argv);

  /* Empty line is given, nothing to do. */
  if (NULL == argv[0]) {
    return;
  }

  /* For built-in commands, we are done. */
  if (builtin_cmd(argv)) {
    return;
  }

  // 후략
}
```

이제, 일반적인 (실행 파일을 수행하는 형태의) 명령어를 처리해야 한다. 이는 fork한 자식 프로세스에서 `execve`를 수행함으로써, command 처리를 달성할 수 있다. child process와 parent process 간의 race condition을 막기 위하여, `fork` 이전에 `SIGCHLD` signal을 block해야 한다.

```
  /* get full sigset */
  sigset_t sig_full;
```

```
  if (sigfillset(&sig_full) < 0) {
    unix_error("sigfillset error");
  }

  /* get sigset with only SIGCHLD */
  sigset_t sig_sigchld;
  if (sigemptyset(&sig_sigchld) < 0) {
    unix_error("sigemptyset error");
  }
  if (sigaddset(&sig_sigchld, SIGCHLD) < 0) {
    unix_error("sigaddset error");
  }

  /* Block SIGCHLD to fork. */
  sigset_t sig_prev;
  if (sigprocmask(SIG_BLOCK, &sig_sigchld, &sig_prev) < 0) {
    unix_error("sigprocmask error");
  }

  /* fork */
  if ((pid = fork()) < 0) {
    unix_error("fork error");
  }
```

Child process는 blocked된 `SIGCHLD` signal을 다시 허용하고, `execve`로 command를 수행해야 한다. 이때, 이 자식 프로세스는 shell (parent) process와는 다른 process group에 속해야 하므로, `setpgid`를 반드시 호출해야 한다.

```
    /* Restore sigset. */
    if (sigprocmask(SIG_SETMASK, &sig_prev, NULL) < 0) {
      unix_error("sigprocmask error");
    }

    /* Make child process run its own process group. */
    if (setpgid(0, 0) < 0) {
      unix_error("setpgid error");
    }

    /* run the user's job given */
    if (execve(argv[0], argv, environ) < 0) { /* never return if succeed */
      /* if it does, then the command given is unknown */
      app_error_printf("%s: Command not found\n", argv[0]);
    }
```

Parent process 또한 `SIGCHLD` signal을 다시 허용해야 한다. 그 전에, child process를 수행했다는 job을 기록해두어야 한다. 이때에도 race condition을 막기 위하여, 모든 signal을 block해야 한다.

```
void eval(char *cmdline)
```

```
{
  // 전략

  /* parent process */

  /* Block all signals before adding jobs. */
  if (sigprocmask(SIG_BLOCK, &sig_full, NULL) < 0) {
    unix_error("sigprocmask error");
  }

  struct job_t *job = addjob(jobs, pid, bg ? BG : FG, cmdline);

  /* Restore sigset. */
  if (sigprocmask(SIG_SETMASK, &sig_prev, NULL) < 0) {
    unix_error("sigprocmask error");
  }

  if (bg) { /* background job */
    /* print appropriate message */
    printjob(job);
  } else { /* foreground job */
    /* wait to be terminated */
    waitfg(pid);
  }

  return;
}
```

Child process에서 `execve`의 error는 `sigchld_handler`가 처리해주기 때문에, 여기서는 특별한 예외처리가 필요하지 않다.

builtin_cmd

이 shell은 네 개의 내장 명령어 `quit`, `fg`, `bg`, `jobs`를 지원한다. Command가 이 네 명령어 중 하나에 해당하는지 확인하고, 그에 대응되는 작업을 수행하면 된다. 가장 직관적인 함수 중 하나다.

```
int builtin_cmd(char **argv)
{
  /* quit */
  if (!strcmp(argv[0], "quit")) {
    exit(0);
  }

  /* fg and bg */
  if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg")) {
    do_bgfg(argv);
    return 1;
  }

  /* jobs */
  if (!strcmp(argv[0], "jobs")) {
```

```
    listjobs(jobs);
    return 1;
  }

  return 0; /* not a builtin command */
}
```

do_bgfg

`bg`와 `fg` 명령어를 처리하는 함수다. 이 두 명령어는 하나의 부가적인 인자를 필요로 한다. 이 인자가 주어지지 않은 경우를 예외 처리하자.

```
void do_bgfg(char **argv)
{
  /* fg and bg need job argument */
  if (NULL == argv[1]) {
    print_bgfg_noarg(argv[0]);
    return;
  }

  // 후략
}
```

먼저, 문자 `%`를 prefix로 가지는 job id가 주어진 경우를 처리한다. `./tshref`를 여러 번 테스트해본 결과, `%` 이후의 문자열을 `atoi`로 읽어들여 job id를 받아오며, 에러 메세지에는 이 인자 string을 그대로 보여주는 것 같다.

```
  if ('%' == argv[1][0]) {  /* job id is given */
    int jid = atoi(argv[1] + 1);  /* job id */

    /* find the job */
    if (NULL == (job = getjobjid(jobs, jid))) { /* no such jobs */
      print_bgfg_nojob(argv[1]);
      return;
    }

    pid = job->pid;
  }
```

위에 해당하지 않는다면, process id가 주어진 경우다. 여기서는 process group id를 따로 다루지는 않는다. 이 경우에 `./tshref`는, 위와는 다르게, 첫 글자가 숫자인지를 검사하는 듯 하다. 또한, 문자열 전체가 올바른 수인지는 따로 확인하지 않고, `atoi`로 process id를 읽어들이는 듯 하다. 대응되는 에러 메세지를 모두 구하고, 사려 깊은 케이스 분석으로 구현하였다.

```
  else {  /* process id is given */
    /* pid must be numeric */
```

```
    if (!isdigit(argv[1][0])) {
      print_bgfg_notint(argv[0]);
      return;
    }

    pid = atoi(argv[1]);

    /* find the job */
    if (NULL == (job = getjobpid(jobs, pid))) { /* no such jobs */
      print_bgfg_noproc(pid);
      return;
    }
  }
```

위 과정을 통하여, 대응되는 job과 process id를 얻었다.

```
  struct job_t *job;  /* job entry */
  pid_t pid;          /* process id */
```

이전과 마찬가지로, corresponding child process와 race condition이 발생하지 않도록, `SIGCONT` signal을 주기 전에 모든 signal을 block해야 한다.

```
  /* get full sigset */
  sigset_t sig_full;
  if (sigfillset(&sig_full) < 0) {
    unix_error("sigfillset error");
  }

  /* Block all signals. */
  sigset_t sig_prev;
  if (sigprocmask(SIG_BLOCK, &sig_full, &sig_prev) < 0) {
    unix_error("sigprocmask error");
  }

  /* Send SIGCONT signal to every process of the group. */
  if (kill(-pid, SIGCONT) < 0) {
    unix_error("kill error");
  }
```

Global variables에 race condition이 발생하기 이전에, jobs 처리 또한 수행하자. 위에서 구한 `job`의 `state` 정보만 다뤄주면 충분하다.

```
  /* Note that argv[0] is either "bg" or "fg". */
  int bg = 'b' == argv[0][0]; /* bg command? */

  // 중략

  /* update the state of the job */
  if (bg) { /* bg command */
    job->state = BG;
    printjob(job);
  } else {  /* fg command */
    job->state = FG;
  }
```

이제, blocked signal 정보를 다시 복구한다. `fg` command의 경우, background process가 foreground로 올라왔으므로, 기존과 같이 `waitfg`를 해야 한다.

```
void do_bgfg(char **argv)
{
  // 전략

  /* Restore sigset. */
  if (sigprocmask(SIG_SETMASK, &sig_prev, NULL) < 0) {
    unix_error("sigprocmask error");
  }

  /* for fg command, wait the process */
  if (!bg) {
    waitfg(pid);
  }
}
```

waitfg

Foreground process가 종료될 때까지 계속 기다리는 함수다. System call의 `sleep`을 통해 context switch를 계속 유발해야 한다. 인자로 주어진 `pid`를 `fgpid`로 검사해도 충분하지만, 100%의 확신이 들지 않았다. 경험과 직관이 말해주는 가장 안전한 방법으로, 원하는 process가 종료되었는지 여부를 판별하였다.

```
void waitfg(pid_t pid)
{
  struct job_t *job;

  while (NULL != (job = getjobpid(jobs, pid)) && FG == job->state) {
    if (sleep(1) < 0) {
      unix_error("sleep error");
    }
  }
}
```

sigchld_handler

종료된 child process를 적절한 후처리로 reaping해야 한다. Signal handler는 `errno`를 기록했다가 복원해야 한다는 reference 문서를 보았다. 이에 따라, `errno`를 기록해둔다.

```
void sigchld_handler(int sig)
{
  int prev_errno = errno; /* store errno */

  // 후략
}
```

이제, stop된 child process를 모두 가져와서 reaping하자. Global variables로 작업하기 전에 모든 signal을 block해야 함을 기억해야 한다.

```
  /* get stopped child process but do not wait for */
  while (0 < (pid = waitpid(-1, &wstatus, WUNTRACED | WNOHANG))) {
    /* Block all signals. */
    if (sigprocmask(SIG_BLOCK, &sig_full, &sig_prev) < 0) {
      safe_unix_error("sigprocmask error");
    }

    /* reap the process with appropriate message */
    reap_zombie(pid, wstatus);

    /* Restore sigset. */
    if (sigprocmask(SIG_SETMASK, &sig_prev, NULL) < 0) {
      safe_unix_error("sigprocmask error");
    }
  }
```

`void reap_zombie(pid, wstatus)` 함수는 child process `pid`의 종료 status `wstatus` 정보를 바탕으로, 이 프로세스가 어떻게 종료되었는지를 판단하고, 적절한 signal message를 출력하는 함수다.

```
void reap_zombie(pid_t pid, int wstatus)
{
  /* process ended normally */
  if (WIFEXITED(wstatus)) {
    deletejob(jobs, pid);
    return;
  }

  /* process ended by an uncaught signal */
  if (WIFSIGNALED(wstatus)) {
    safe_print_sigend(pid, pid2jid(pid), WTERMSIG(wstatus), 1);
    deletejob(jobs, pid);
```

```
    return;
  }

  /* process is stopped */
  if (WIFSTOPPED(wstatus)) {
    /* check job is in the job list */
    struct job_t *job = getjobpid(jobs, pid);
    if (NULL != job) {
      safe_print_sigend(job->pid, job->jid, WSTOPSIG(wstatus), 0);

      /* set the state stopped */
      job->state = ST;
    }
    return;
  }
}
```

Signal handler는 async-safe 해야 하므로, 모든 message 출력 또한 async-safe 해야 한다. `printf`를 사용하지 않도록 유의한다.

System call인 `waitpid`의 에러를 확인하고, `errno`를 복구한다.

```
void sigchld_handler(int sig)
{
  // 전략

  /* detect waitpid error */
  if (pid < 0 && errno && ECHILD != errno) {
    safe_unix_error("waitpid error");
  }

  /* restore errno */
  errno = prev_errno;
}
```

sigint_handler

Foreground process에게 `SIGINT` signal을 전달하면 된다. 직관적이다.

```
void sigint_handler(int sig)
{
  int prev_errno = errno; /* store errno */

  pid_t pid = fgpid(jobs);  /* get foreground pid */

  if (0 < pid) {  /* must check pid is valid */
    /* send SIGINT signal */
    if (kill(-pid, SIGINT) < 0) {
```

```
      safe_unix_error("kill error");
    }
  }

  /* restore errno */
  errno = prev_errno;
}
```

sigtstp_handler

이번에는 foreground process에게 `SIGTSTP` signal을 전달하면 된다.

```
void sigtstp_handler(int sig)
{
  int prev_errno = errno; /* store errno */

  pid_t pid = fgpid(jobs);  /* get foreground pid */

  if (0 < pid) {  /* must check pid is valid */
    /* send SIGTSTP signal */
    if (kill(-pid, SIGTSTP) < 0) {
      safe_unix_error("kill error");
    }
  }

  /* restore errno */
  errno = prev_errno;
}
```

addjob (modified)

Skeleton에 이미 주어진 `struct job_t *addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)` 함수는 추가한 `job`의 포인터를 반환하도록 수정하였다.

Signal-safe helper routines

Signal handler에서 출력 등을 위하여 async-safe한 함수가 필요하다. 이에 필요한 함수들을 구현하였다.

`size_t safe_strlen(const char *str)` 함수는 null-terminated string `str`의 길이를 반환한다.

`void safe_reverse(char *str)` 함수는 string `str`을 reverse한다.

`void safe_ltoa(long v, char *str)` 함수는 정수 `v`를 10진법으로 나타내어 `str`에 기록한다.

`ssize_t safe_write(const char *str)`는 문자열을, `ssize_t safe_printlong(const long n)`는 정수 `n`을 출력한다.

```
ssize_t safe_write(const char *str)
{
  return write(STDOUT_FILENO, str, safe_strlen(str));
}

ssize_t safe_printlong(const long n)
{
  char str[32];
  safe_ltoa(n, str);
  return safe_write(str);
}
```

`void safe_error(const char *str)` 함수는 문자열 `str`을 출력하고 비정상 종료한다.

```
void safe_error(const char *str)
{
  safe_write(str);
  _exit(1);
}
```

`unix_error` 함수 또한 async-safe하게 바꾸어야 한다. `errno`에 대응되는 error message를 async-safe한 `strerror_r` 함수로 구한 후, `safe_write`로 출력해야 한다.

```
void safe_unix_error(char *msg)
{
  char str_err[1024];
  strerror_r(errno, str_err, 1024);

  safe_write(msg);
  safe_write(": ");
  safe_write(str_err);
  safe_write("\n");

  _exit(1);
}
```

`sigchld_handler`의 `reap_zombie` 함수에서 terminated/stoppted child process 정보를 출력할 때 사용할 async-safe 함수도 구현해야 한다.

```
void safe_print_sigend(pid_t pid, int jid, int sig, int termed) {
  safe_write("Job [");
  safe_printlong((long)jid);
  safe_write("] (");
  safe_printlong((long)pid);
  safe_write(") ");
  safe_write(termed ? "terminated" : "stopped");
  safe_write(" by signal ");
  safe_printlong((long)sig);
  safe_write("\n");
}
```

어려웠던 점

Signal control과 block 등을 정확하게 하지 않으면, race condition이 일어나서 global variables가 망가지거나, 정상적인 실행 과정으로는 예상할 수 없는 결과 등이 발생하였다. 특히, 후자의 경우는 원인을 알아내지 못하였다.

Signal이 queue의 형태로 쌓이지 않고, 동시 다발적으로 (단일 bit에) stacking 되는 성질 때문에, 버그를 규명하는 작업부터 오류의 발단점을 찾고 이를 디버깅하는 과정 전체가 매우 힘들었다. 주어진 자료에 충분히 좋은 힌트가 잘 적혀있었기에, 이 shell을 완성할 수 있었다.

디버깅 과정에서는 `write` 함수를 사용해야 했는데, 이를 잘 다루기 까다로웠다. `printf`와 같은 형태의 wrapper function을 작성하여 해결하였다.

다양한 환경을 구성하여 테스트해보기 까다로웠다. System call의 exception 처리가 제대로 되었는지를 확인하고 싶었지만, 특정한 하나의 system call을 수행했을 때 `errno`가 flagged 되도록 하는 방법을 끝내 찾지 못하였다. 또한, 하나의 프로세스가 다른 프로세스를 찾아 kill 하는 상황 등을 테스트하기 위해, 직접 그러한 코드를 하나하나 작성해야 함이 참 힘들었다.

`./tshref`의 에러 처리 과정과 대응되는 에러 메세지 형식을 구하는 과정이 어려우면서도, 하나의 퍼즐을 푸는 것 같아 즐거웠다. 이 과정은 나의 shell `./tsh`의 완성도와 안정도를 높이는 데에도 큰 도움이 되었다.

새롭게 배운 점

Signal을 어떻게 허용하고 block 하는지, 그리고 이를 통해 kernel과의 context switching 과정에서 어떻게 signal handler의 호출을 막을 수 있는지 그 과정을 명확하게 깨닫게 되었다.

Interrupt를 어떻게 다루며, async-safe의 의미란 무엇인지 알게 되었다. 수업에서 배운 많은 지식을 십분 활용해야 shell을 완성할 수 있었다.

`man` page를 능숙하게 다룰 수 있게 되었다. 인터넷 검색보다 편하고, 신뢰도도 높다.

Kernel 코드를 뜯어보았고, 많은 시간을 투자했지만, 이해하지 못하였다.