lab2_2019-11730

Implement

`void eval(char *cmdline)`

`eval` 함수는 사용자가 입력한 명령어를 받아 파싱하여 실행하는 함수입니다.

```
bg = parseline(cmdline, argv);
```

1. `cmdline`에서 명령어와 인자를 파싱합니다.

```
if (!builtin_cmd(argv)) {
    ...
}
```

2. 파싱한 결과가 내장 명령어인지 확인합니다.

```
if (sigemptyset(&action.sa_mask) < 0) {
    unix_error("sigemptyset error\n");
}
if (sigaddset(&action.sa_mask, SIGCHLD) < 0) {
    unix_error("sigaddset(SIGCHLD) error\n");
}
if (sigaddset(&action.sa_mask, SIGINT) < 0) {
    unix_error("sigaddset(SIGINT) error\n");
}
if (sigaddset(&action.sa_mask, SIGTSTP) < 0) {
    unix_error("sigaddset(SIGTSTP) error\n");
}

if (sigprocmask(SIG_BLOCK, &action.sa_mask, NULL) < 0) {
    unix_error("sigprocmask(SIG_BLOCK) error\n");
}
```

3. SIGCHLD, SIGINT, SIGTSTP 시그널을 block합니다.

```
if ((pid = fork()) == 0) {
    ...
}
```

4. `fork()` 함수를 호출해 자식 프로세스를 생성합니다.

```
if (sigprocmask(SIG_UNBLOCK, &action.sa_mask, NULL) < 0) {
    unix_error("sigprocmask(SIG_UNBLOCK) error\n");
}
```

5. 자식 프로세스에서 block된 시그널을 unblock합니다.

```
if (setpgid(0, 0) < 0) {
    unix_error("setpgid error\n");
}
```

6. 자식 프로세스의 그룹 ID를 자식 프로세스의 PID로 변경합니다.

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

7. execve() 함수를 호출해 명령어를 실행합니다.

```
addjob(jobs, pid, FG, cmdline);
addjob(jobs, pid, BG, cmdline);
```

8. 부모 프로세스에서는 생성된 작업을 job list에 추가합니다.

```
if (sigprocmask(SIG_UNBLOCK, &action.sa_mask, NULL) < 0) {
    unix_error("sigprocmask(SIG_UNBLOCK) error\n");
}
```

9. 부모 프로세스에서 block된 시그널을 unblock합니다.

```
waitfg(pid);
```

10. 작업이 foreground인 경우, 작업이 완료될 때까지 대기합니다.

```
printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
```

11. 작업이 background인 경우, 작업이 생성되었음을 출력합니다.

`int builtin_cmd(char **argv)`

`builtin_cmd` 함수는 사용자가 내장 명령어를 입력하면 즉시 실행하고, 내장 명령어가 아닌 경우 0을 반환하는 함수입니다.

```
char* cmd = argv[0];
```

1. 인자로 전달된 문자열 배열 `argv`에서 명령어(cmd)를 가져옵니다.

```
if (strcmp(cmd, "quit") == 0) {
    exit(0);
}
```

2. 만약 입력한 명령어가 `"quit"`이면 shell을 종료합니다.

```
else if (strcmp(cmd, "jobs") == 0) {
    listjobs(jobs);
    return 1;
}
```

3. 만약 입력한 명령어가 `"jobs"`이면 `listjobs()` 함수를 이용해 실행 중인 작업들의 리스트를 출력합니다.

```
else if (strcmp(cmd, "bg") == 0 || strcmp(cmd, "fg") == 0) {
    do_bgfg(argv);
    return 1;
}
```

4. 만약 입력한 명령어가 `"bg"` 또는 `"fg"`이면, `do_bgfg()` 함수를 이용해 stopped된 background 작업을 restart하거나 job을 foreground로 이동시킵니다.

```
return 0;
```

5. 만약 입력한 명령어가 내장 명령어가 아니면 0을 반환합니다.

`void do_bgfg(char **argv)`

`do_bgfg` 함수는 내부 명령어인 bg와 fg를 처리하는 함수입니다.

```
char *cmd = argv[0];
char *id = argv[1];
```

1. 인자로 전달된 문자열 배열 `argv`에서 명령어(cmd)와 ID(id)를 각각 가져옵니다.

```
if (id == NULL) {
    printf("%s command requires PID or %%jobid argument\n", cmd);
    return;
}
```

2. ID가 제공되지 않은 경우, 오류 메시지를 출력하고 함수를 종료합니다.

```
if (id[0] == '%') {
    int jid = atoi(id + 1);
    job = getjobjid(jobs, jid);

    if (job == NULL) {
        printf("%s: No such job\n", id);
        return;
    }
}
```

3. ID가 "%"로 시작하는 경우, 해당하는 작업 ID를 가져와서 작업이 존재하는지 확인합니다. 존재하지 않는 경우, 오류 메시지를 출력하고 함수를 종료합니다.

```
else if (isdigit(id[0])) {
    pid_t pid = atoi(id);
    job = getjobpid(jobs, pid);

    if (job == NULL) {
        printf("(%s): No such process\n", id);
        return;
    }
}
```

4. ID가 숫자로 시작하는 경우, 해당하는 프로세스 ID를 가져와서 작업이 존재하는지 확인합니다. 존재하지 않는 경우, 오류 메시지를 출력하고 함수를 종료합니다.

```
else {
    printf("%s: argument must be a PID or %%jobid\n", cmd);
    return;
}
```

5. 그렇지 않은 경우, 오류 메시지를 출력하고 함수를 종료합니다.

```
if (strcmp(argv[0], "bg") == 0) {
    if (kill(-(job->pid), SIGCONT) < 0) {
        unix_error("kill error\n");
    }
    job->state = BG;
    printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
}
```

6. 명령어가 "bg"인 경우, 해당 작업의 상태를 BG로 변경하고 SIGCONT 시그널을 보내서 background에서 작업을 재개시킵니다. 작업의 정보(작업 ID, 프로세스 ID, 명령어)를 출력합니다.

```
else if (strcmp(argv[0], "fg") == 0) {
    if (kill(-(job->pid), SIGCONT) < 0) {
        unix_error("kill error\n");
    }
    job->state = FG;
    waitfg(job->pid);
}
```

7. 명령어가 "fg"인 경우, 해당 작업의 상태를 FG로 변경하고 SIGCONT 시그널을 보내서 foreground에서 작업을 재개시킵니다. 작업이 완료될 때까지 대기합니다.

void waitfg(pid_t pid)

waitfg 함수는 인자로 받은 pid가 더 이상 foreground 프로세스가 아닐 때까지 기다리는 함수입니다.

```
while (pid == fgpid(jobs)) {
    ...
```

```
}
```

1. 만약 현재 pid가 foreground job의 pid와 다르면, 즉 foreground job이 끝나면 반복문을 종료합니다.

```
sleep(1);
```

2. pid가 foreground job인 동안에는 `sleep` 함수를 사용하여 1초마다 체크합니다.

`void sigchld_handler(int sig)`

`sigchld_handler` 함수는 자식 프로세스가 종료되거나 SIGSTOP 또는 SIGTSTP 시그널을 받아 일시 중지된 경우 호출되는 시그널 핸들러입니다.

```
while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
    ...
}
```

1. `waitpid()` 함수를 사용하여 대기중인 자식 프로세스가 있는지 확인합니다. WNOHANG와 WUNTRACED 플래그를 사용하여 대기 중인 자식 프로세스를 즉시 반환하고, 현재 실행중인 자식 프로세스를 기다리지 않습니다.

```
if (WIFSTOPPED(status)) {
    struct job_t *job = getjobpid(jobs, pid);
    job->state = ST;
    printf("Job [%d] (%d) stopped by signal %d\n", job->jid, pid, WSTOPSIG(status));
}
```

2. 자식 프로세스가 SIGSTOP 또는 SIGTSTP 신호를 수신하여 중지된 경우 작업의 상태를 ST로 설정하고, 해당 작업이 중지된 이유를 출력합니다.

```
else if (WIFSIGNALED(status)) {
    printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
    deletejob(jobs, pid);
}
```

3. 자식 프로세스가 시그널에 의해 종료된 경우 해당 작업이 중지된 이유를 출력하고, 작업 목록에서 해당 작업을 삭제합니다.

```
else if (WIFEXITED(status)) {
    deletejob(jobs, pid);
}
```

4. 자식 프로세스가 정상적으로 종료된 경우 해당 작업을 작업 목록에서 삭제합니다.

`void sigint_handler(int sig)`

`sigint_handler` 함수는 SIGINT 시그널을 처리하기 위한 시그널 핸들러입니다.

```
pid_t pid = fgpid(jobs);
```

1. 현재 foreground job의 PID를 가져옵니다.

```
if (pid > 0) {
    if (kill(-pid, SIGINT) < 0) {
        unix_error("kill error\n");
    }
}
```

2. foreground job이 존재하면 해당 PID에 SIGINT를 보냅니다.

`void sigstp_handler(int sig)`

`sigint_handler` 함수는 SIGTSTP 시그널을 처리하기 위한 시그널 핸들러입니다.

```
pid_t pid = fgpid(jobs);
```

1. 현재 foreground job의 PID를 가져옵니다.

```
if (pid > 0) {
    if (kill(-pid, SIGTSTP) < 0) {
        unix_error("kill error\n");
    }
}
```

2. foreground job이 존재하면 해당 PID에 SIGTSTP를 보냅니다.

What was difficult

1. `sigprocmask` 함수를 사용해 시그널을 block 하고 unblock 하는 코드를 작성하는 부분이 어려웠습니다. 처음 코드를 작성할 때 시그널을 왜 block와 unblock을 하는 과정이 필요한지 이해하지 못해서, 적절한 시점에 `sigprocmask` 함수를 사용하는 것이 어려웠습니다.
2. 다양한 에러 케이스를 핸들링하는 부분이 어려웠습니다. 예를 들어, 사용자가 bg나 fg 명령어를 입력할 때 인자로 job ID나 프로세스 ID를 올바르게 입력하지 않았을 경우에 대한 처리가 필요했습니다. 이러한 에러 케이스를 고려하면서 코드를 작성하는 것이 어려웠습니다.
3. 시그널을 사용하는 방법과 그 흐름을 파악하는 것이 어려웠습니다.

Something new and surprising

1. 시그널과 시그널 핸들러에 대해 자세히 알게 되었습니다. 부모 프로세스와 자식 프로세스 사이에서 시그널을 주고 받는 과정에서 시그널이 어떻게 전달되고 처리되는지 이해할 수 있었습니다. 또한 시그널이 shell에서는 어떻게 활용되는지에 대해 알게 되었습니다.
2. system call의 return value를 체크하는 것의 중요성과 그것이 프로그램의 안정성에 미치는 영향을 알게 되었습니다.

3. setpgid, sigemptyset, sigaddset, sigprocmask, execve, kill과 같은 system 함수의 동작과 사용법을 알게 되었습니다.