**System Programming Project 2**

담당 교수 : 김영재 교수님

이름 : 이 상 연

학번 : 20201617

1. **개발 목표**

본 프로젝트는 Unix 운영체제 환경에서의 process의 실행과 pipe, signal 을 활용한 process 간의 통신에 대한 이해를 기반으로, system-level process control 을 동작하도록 하는 쉘 프로그램을 구현해 보는 것을 개발 목표로 한다. Unix 환경에서의 시스템 컨트롤을 위해서 본 프로젝트에서는 C 언어를 활용하여 개발한다. 추가적으로 본 프로젝트에서 개발하는 쉘 프로그램은 리눅스의 기본 쉘인 bash 쉘의 동작을 기반으로 삼아서 진행한다.

본 프로젝트는 3개의 phase로 구성되어 있고, 각 phase에서의 개발 목표는 아래와 같다.

* Phase Ⅰ : Building and Testing Your Shell

명세서에서 제시하는 shell builtin 명령어들이 실행 가능하도록 가장 간단한 형태의 쉘을 구현한다. ‘cd’, ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’, ‘cat’, ‘echo’, ‘exit’ 명령어가 정상적으로 수행되어야 한다. fork(), exec() system call 을 활용하도록 한다. 쉘에서 실행하는 명령 중에서 쉘의 builtin command 인 것과 그렇지 않은 프로그램이 존재한다. 예를 들어 ‘ls’, ‘mkdir’, ‘rmdir’ 등은 /bin 디렉토리에 있는 프로그램이기 때문에 이 프로그램을 실행하면 된다. 그렇기 때문에 이런 명령어들을 처리할 때에는 fork() 를 호출해서 생성한 자식 프로세스에서 exec()를 호출해서 프로그램을 실행한다. 또한 ‘cd’, ‘exit’ 와 같은 shell builtin 명령어들은 그 동작을 코드에서 직접 구현하도록 한다.

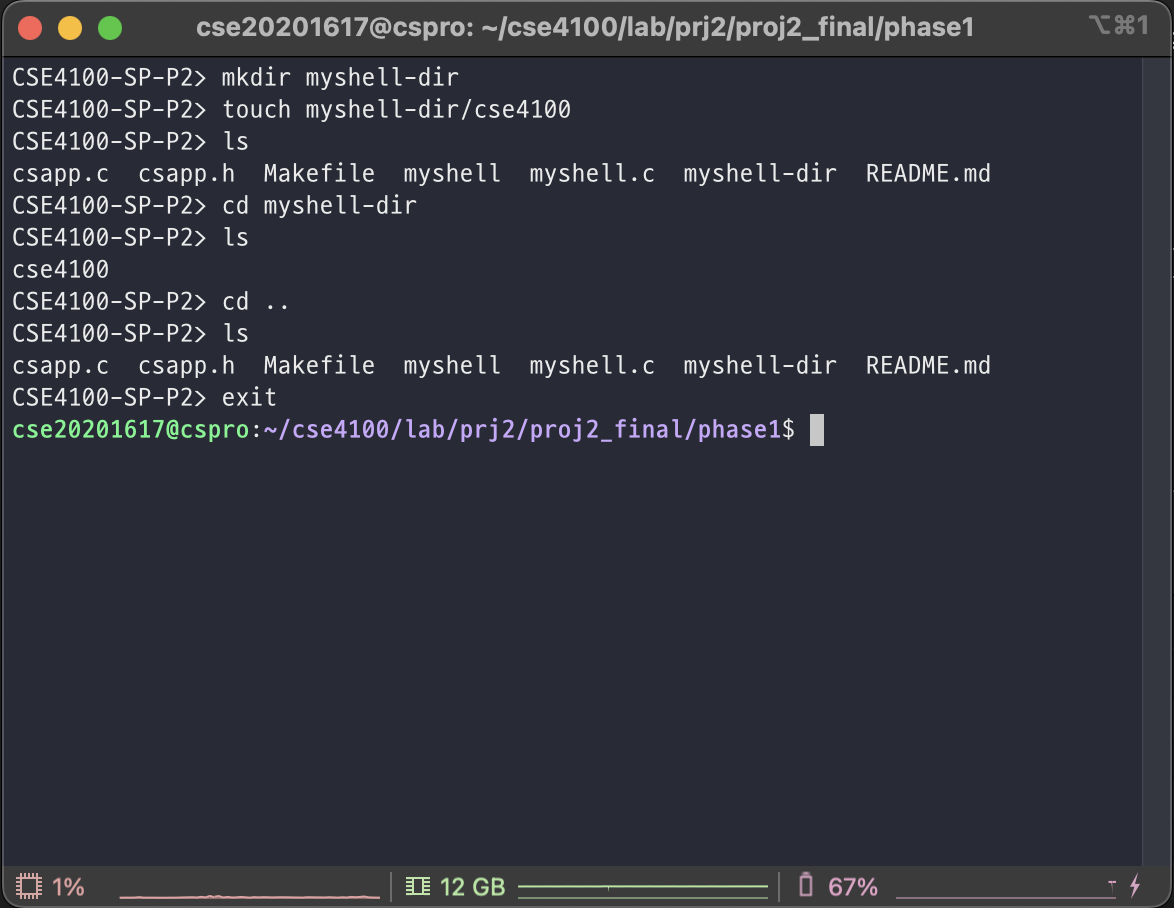
* Phase Ⅱ : Redirection and Pipe

Unix shell 에서는 pipe 라는 Inter process communication 을 위한 기능을 제공한다. Pipe는 ‘|’ 기호로 여러 개의 명령어를 이어주는 것을 통해 사용할 수 있다. 더 자세히는, 프로세스 혹은 실행된 프로그램의 결과를 다른 프로세스로 넘겨줄 때에 사용된다. 각 프로세스의 메모리는 독립적으로 존재하기 때문에 이러한 프로세스 간 통신에 사용되는 기법이 IPC 이다. Pipe 의 구현을 위해서는 stdin 과 stdout 으로 연결되어 있던 프로세스의 입출력을 pipe로 변경해야 한다. 이를 위해서 dup(), dup2() system call 을 호출하는 함수를 활용한다. 실제 쉘에서 pipe 로 연결할 수 있는 명령어 수에 대한 제한은 없지만 본 프로젝트에서는 그 개수를 32개로 제한하여 구현하도록 한다. 그리고 마지막 명령어의 출력은 stdout으로 터미널에 출력되도록 한다.

* Phase Ⅲ : Run Process in Background

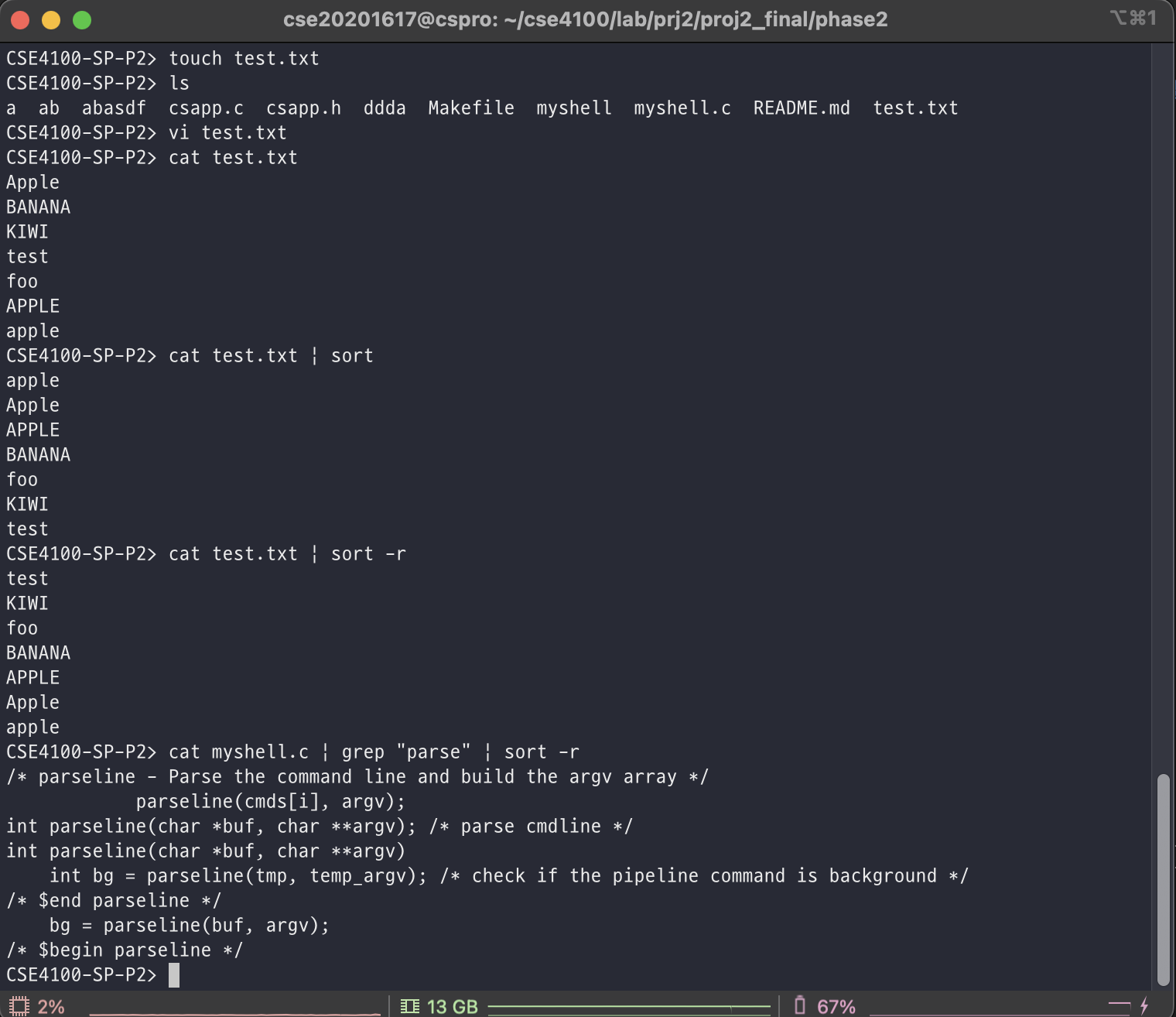
마지막은 쉘 프로그램에서 제공하는 핵심적인 기능인 job control 기능을 구현한다. 수행하고자 하는 작업을 foreground 혹은 background 에서 수행하고, background에서 수행 중이라면 동시에 새로운 커맨드들을 입력받을 수 있도록 해야 한다. Foreground job은 수행되는 동안 터미널에 새로운 커맨드 입력이 불가능하지만 ctrl-c, ctrl-z 와 같은 키보드 입력을 수행할 수 있다. ctrl-z 입력을 받은 job은 stopped 상태로 되고, ctrl-c 입력을 받은 job은 즉시 종료된다. 실행 중인 쉘 프로그램은 키보드 입력으로 인해 종료되거나 멈추지 않도록 추가적인 설정이 필요한다. ‘jobs’ 명령어는 job 리스트에 존재하는 job 들을 출력해준다. ‘fg’ 명령어는 stopped 되어 있거나 background 에서 실행 중인 job 을 foreground 로 실행되도록 한다. ‘bg’ 명령어는 stopped 되어 있는 background job을 running background job으로 만든다. ‘kill’ 명령어는 job 리스트에 존재하는 job에 SIGTERM signal 을 보내서 종료되도록 한다. 본 프로젝트에서 구현하는 ‘bg, ‘fg’, ‘kill’ 명령어의 인자로 올 수 있는 형식은 ‘%JID’로 제한하도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1



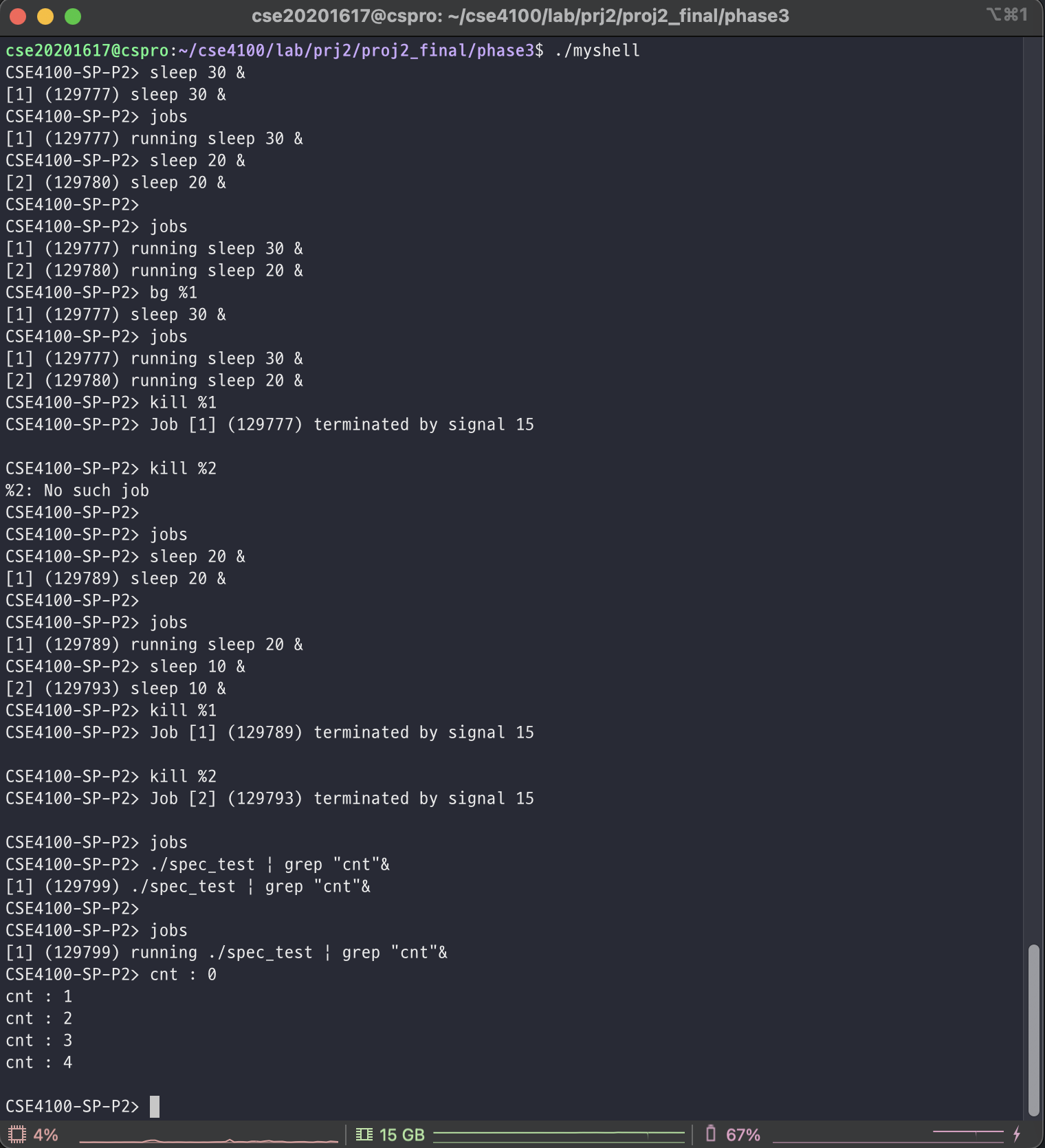
CSPRO 서버에서 Phase1 구현 프로그램을 테스트한 결과이다. 명세서에서 정상적으로 동작해야 한다고 언급한 명령어들에 대해 테스트를 했고, 정확하게 동작하는 것을 확인할 수 있었다. ‘cd’, ‘exit’ 명령어는 쉘의 빌트이 명령어이기 때문에 동작 로직을 프로그램 함수 내애서 구현했고, 다른 명령어들은 fork() 로 자식 프로세스를 생성한 후에 execvp() 함수를 호출해서 실행되도록 구현했다.

1. Phase 2



CSPRO 서버에서 Phase2 구현 프로그램을 실행해서 여러 명령어들을 입력해보며 테스트한 결과이다. 여러 개의 pipeline 명령이 이어진 입력에 대해 잘 처리되고 있다. 따옴표(“”)로 감싸서 전달한 인자에 관련된 처리도 잘 이루어지고 있는 것을 확인할 수 있다.

1. Phase 3



CSPRO 서버에서 Phase3 구현 프로그램을 실행해서 여러 명령어들을 통해 테스트한 결과이다. 우선 위에서 보는 것과 같이 백그라운드로 명령을 실행할 때 job 컨트롤이 잘 이루어지고 있는 것을 확인할 수 있다. kill 명령을 통해 실행중인 job을 종료시키는 동작도 잘 수행되고 있다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 함수를 호출하면, 부모 프로세스의 메모리 영역을 복사한 자식 프로세스가 생성된다. 이는 독립적인 메모리에 할당된다. Phase1 에서의 ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’ 등의 명령어는 이 자식 프로세스에서 수행된다. 위와 같은 명령어들은 시스템에 구현되어 있는 프로그램이다. ‘type command’ 를 통해 쉘 빌트인 명령인지 아닌지를 확인할 수 있다. CSPRO 서버의 /usr/bin 디렉토리에 위 프로그램들이 저장되어 있는 것을 아래의 사진을 통해 알 수 있다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

쉘에 명령어를 입력했을 때 해당하는 프로그램을 실행하기 위해서는 exec() 함수를 호출해야 한다. exec() 를 호출하면 현재 프로세스의 메모리에, 파라미터로 전달되는 경로의 프로그램을 로드하고, 새로운 프로세스로 덮어씌운다. 본 프로젝트에서는 execvp() 함수를 사용하서 파라미터로 전달되는 프로그램의 이름을 전달하면 직접 환경변수를 통해 경로를 찾아서 실행한다.

만약 쉘 빌트인 명령어인 경우 myshell.c 의 builtin\_command() 함수 내에서 해당하는 빌트인 명령어를 처리한다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

자식 프로세스에서 프로그램 실행이 종료되고 나면, connection 을 종료해야 한다. 예를 들어 ‘ls’ 명령어의 동작을 수행하는 경우를 생각해보자. 자식 프로세스에서 execvp() 함수를 호출해서 ‘ls’ 프로그램이 위치한 경로를 찾아서 수행한다. ‘ls’ 프로그램이 실행되면 터미널에 현재 경로의 파일 정보에 대해 출력하고 프로그램이 종료된다. 이렇게 종료된 프로그램은, 부모 혹은 init 프로세스에 의해 reaping 되기 전에는 좀비 프로세스 상태가 되며, 남아있다. 그렇기 때문에 기본적으로 부모 프로세스에서 좀비 프로세스가 된 자식 프로세스를 reaping 해줘야 한다. 부모 프로세스에서 reaping 하는 것은 wait() 함수를 통해서 수행된다.

프로세스에서 Fork() 함수를 실행하면, 이를 호출한 프로세스(부모 프로세스)의 메모레가 복사되어서 새로운 자식 프로세스가 생성된다. 부모 프로세스오 ㅏ자식 프로세스는 Fork()가 실행된 이후부터는 각자 그 이후부터의 코드를 독립적으로 실행한다. 이 때 부모 프로세스에서는 Fork() 함수에 대한 리턴값으로 생성된 자식 프로세스의 PID 를 받고, 자식 프로세스는 0을 받는다. 이를 이용해서 부모 프로세스와 자식 프로세스를 각각 다른 동작을 하도록 하는 코드를 작성할 수 있다.

자식 프로세스가 stopped 상태가 되거나 종료되는 경우 자신을 생성한 부모 프로세스에 SIGCHLD 시그널을 전송한다. 그러나 부모 프로세스에서는 여러 이유들로 인해 이 SIGCHLD 시그널들을 즉각적으로 처리하지 못할 수 있다. 그렇기 때문에 wait(), waitpid() 에 적절하게 인자를 전달해서 자식 프로세스를 기다리는동안 부모 프로세스를 suspend 상태로 만들거나, 종료된 자식 프로세스가 없는 경우 계속 실행시키도록 할 수 있다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - Design
* 명령 분할: 사용자 입력 문자열에서 ‘|’ 기호로 파이프라인 조각을 나누어서, 최대 MAX\_PIPE\_CMDS 개까지 토큰화
* 프로세스 그룹화: 파이프라인에 속한 모든 자식 프로세스를 하나의 프로세스 그룹(pgid)로 묶어서, 단일 터미널 제어 단위로 관리
* 입출력 연결: 순차적으로 생성한 파이프의 read / write 파일 디스크립터를 dup2() 로 자식 프로세스의 stdin / stdout 에 연결
  + - Implementation
* 토큰화

|  |
| --- |
| /\* seperate command \*/  char \*token = strtok(buf, "|");  while (token && num\_cmds < MAX\_PIPE\_CMDS) {  while (\*token == ' ') token++;  cmds[num\_cmds++] = token;  token = strtok(NULL, "|");  } |

사용자의 입력을 ‘|’을 기준으로 잘라서 파이프라인에 연결된 개별 명령어 문자열을 추출한다. strtok() 함수를 통해 buf 내부에서 | 전까지의 영역을 한 토큰으로 잘라내고, 그 시작 주소를 token 에 봔한한다. 초기 호출 이후 buf 내용은 첫 구분자 위치에 널 문자가 들어가 토큰이 분리된다.

* 파이프 생성 & fork 반복

|  |
| --- |
| for (int i = 0; i < num\_cmds; ++i) {  int pipefd[2];  int is\_last = (i == num\_cmds - 1);  if (!is\_last && pipe(pipefd) < 0) { /\* create a pipe \*/  printf("pipe error\n");  exit(1);  }  pid\_t pid = fork();  if (pid < 0) { /\* fork error \*/  printf("fork error\n");  exit(1);  }  if (pid == 0) { /\* child process \*/  Sigprocmask(SIG\_UNBLOCK, &mask, NULL); /\* unblock SIGCHLD in child process \*/  if (pgid == 0)  pgid = getpid(); /\* set process group id to the current child process \*/  setpgid(0, pgid); /\* set the child process group id to pgid \*/  /\* Read from the previous command \*/  if (prev\_fd != -1) {  dup2(prev\_fd, STDIN\_FILENO);  close(prev\_fd);  }  /\* Connect output to the next command \*/  if (!is\_last) {  close(pipefd[0]); /\* close the read end of the pipe \*/  dup2(pipefd[1], STDOUT\_FILENO); /\* redirect stdout to the write end of the pipe \*/  close(pipefd[1]); /\* close the write end of the pipe \*/  }  char \*argv[MAXARGS];  parseline(cmds[i], argv);  if (!builtin\_command(argv)) {  if (execvp(argv[0], argv) < 0) {  printf("%s: command not found\n", argv[0]);  exit(1);  }  }  }  // parent process  if(pgid == 0) { /\* if the process is the first in the pipeline \*/  pgid = pid;  add\_job(jobs, pgid, bg ? BG : FG, cmdline);  struct job\_t \*job = get\_job\_pid(jobs, pgid);  job->nprocs = num\_cmds;  job->exited = 0;  }  setpgid(pid, pgid); /\* set the child process group id to pgid \*/  /\* find the job with pgid \*/  struct job\_t \*job = get\_job\_pid(jobs, pgid);  if (job) { /\* add the pid to the job \*/  job->pids[job->idx++] = pid;  }  if (prev\_fd != -1)  close(prev\_fd); /\* close the read end of the previous pipe \*/  if (!is\_last) {  close(pipefd[1]); /\* close the write end of the pipe \*/  prev\_fd = pipefd[0]; // pass the read end to the next command  }  } |

파이프라인에 속한 각 명령을 순서대로 실행하면서, 프로세스 그룹과 입출력 리다이렉션을 설정하는 루프이다. 파이프를 생성하고 is\_last 가 거짓인 경우에만 pipe() 로 새로운 파이프를 생성하고, 읽기 / 쓰기 끝을 pipefd[0], pipefd[1] 에 넣는다. fork() 로 자식 프로세스를 생성한 후 자식 프로세스에서는 SIGCHLD를 unblock 한다. 그 후 첫번째 자식 프로세스인 경우엔 해당 프로세스의 pid 를 pgid 로 설정한다. 이전 파이프가 있는 경우에는 dup2(prev\_fd, STDIN\_FILENO)로 표준 입력을 연결한다. 마지막 명령이 아닌 경우엔 dup2(pipefd[1], STDOUT\_FILENO)로 표준 출력을 다음 파이프로 연결한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

토큰화 단계에서 num\_cmds 를 구하고, for 루프에서 파이프 생성 / 연결 / 실행 동작을 수행한다. 첫 프로세스인 경우에는 input 리다이렉션 없이 실행하고, 중간에 있는 프로세스들의 경우에는 이전 파이프를 stdin으로, 새로운 파이프 write end 를 stdout 으로 연결한다. 마지막 프로세스의 경우에는 output 리다이렉션 없이 실행하고, 부모 프로세스는 이 시점에서 waitfg 혹은 백그라운드 프로세스인 경우에 이에 관한 정보를 출력한다. 이러한 구조 덕분에 파이프라인 길이에 상관없이 동일한 알고리즘으로 처리할 수 있다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

백그라운드 명령인지 식별하기 위해서 입력받은 커맨드 라인의 마지막 인자로 ‘&’ 가 있으면 bg = 1 로 표시하고, 토큰화 전에 & 를 잘라낸다. 백그라운드 job 인 경우에는 waitfg()를 호출하지 않고, bash 쉘에서의 동작과 유사하게 “[jid] (pid) cmdline” 형식으로 출력한 후에, 다른 입력을 받을 수 있도록 프롬프트를 출력한다.

|  |
| --- |
| // eval, eval\_pipe 공통  char tmp[MAXLINE];  strcpy(tmp, last\_cmd);  int bg = parseline(tmp, argv); // argv에 '&' 제거, bg=1 반환 |
| if (bg) {  // 백그라운드: job 추가 후 즉시 프롬프트로 복귀  printf("[%d] (%d) %s", pid\_to\_jid(...), pid, cmdline);  } else {  // 포그라운드: 터미널 제어권 넘기고 waitfg 호출  tcsetpgrp(..., pgid);  waitfg(pgid);  tcsetpgrp(..., shell\_pgid);  } |

* 1. **개발 방법**
* 공통

|  |
| --- |
| typedef enum {UNDEF, FG, BG, ST} job\_state;  struct job\_t {  pid\_t pid; /\* process id \*/  int jid; /\* job id, start from 1\*/  job\_state state; /\* 0: UNDEF | 1: FG | 2: BG | 3: ST \*/  char cmdline[MAXLINE]; /\* command \*/  int nprocs; /\* total number of processes in the job \*/  int exited; /\* number of processes that have exited \*/  int idx; /\* index variable to store the pid in the pids \*/  pid\_t pids[MAX\_PIPE\_CMDS]; /\* each pid of the child processes in the same process group \*/  };  struct job\_t jobs[MAXJOBS]; /\* Global array for storing the jobs \*/ |

job 의 state 를 UNDEF, FG, BG, ST 로 정의했고, 이를 위한 enum type인 job\_state 를 정의했다. 각 job 들을 관리하기 위한 job\_t 구조체를 정의했다. jobs 는 현재 쉘에서 관리해야 하는 모든 job 들을 저장하고 있는 전역배열이다

* Phase2
  + tokenization & temp buffer

eval\_pipe() 함수 내에서 strtok(buf, ‘|’)로 각 커맨드들을 토큰화하여 cmds[] 에 저장한다. 마지막 명령어에 대해 백그라운드 명령인지 검사하기 위해 strcpy(tmp, cmds[num\_cmds – 1]) 로 복사하여 parseline() 에 인자로 넘겨서 리턴값을 보고 백그라운드로 처리해야 하는 job 인지 아닌지 판별한다.

* + process group id

첫 번째 자식에서 pgid = getpid(); setpgid(0, pgid); 로 이 자식 프로세스의 process id 를 process group id 로 설정한다. 이 후의 모든 파이프라인 커맨드 자식 프로세스에 대해서도 setpgid(pid, pgid) 를 호출해서 process groupd id 를 설정한다.

* + add job

|  |
| --- |
| // parent process  if(pgid == 0) { /\* if the process is the first in the pipeline \*/  pgid = pid;  add\_job(jobs, pgid, bg ? BG : FG, cmdline);  struct job\_t \*job = get\_job\_pid(jobs, pgid);  job->nprocs = num\_cmds;  job->exited = 0;  }  ------------------------------------------------------------------------------  /\* $begin add\_job \*/  /\* add\_job - Add a job to the job list \*/  int add\_job(struct job\_t \*jobs, pid\_t pid, job\_state state, char \*cmdline)  {  if (pid < 1)  return 0;  for (int i = 0; i < MAXJOBS; i++) {  if (jobs[i].pid == 0) {  jobs[i].pid = pid;  jobs[i].state = state;  jobs[i].jid = next\_jid++;  if (next\_jid > MAXJOBS)  next\_jid = 1;  strcpy(jobs[i].cmdline, cmdline);  return 1;  }  }    /\* if jobs is full \*/  printf("Too many jobs!!!\n");  return 0;  }  /\* $end add\_job \*/ |

위는 add\_job() 함수의 구현과, eval\_pipe() 함수에서 job을 추가하는 부분의 코드이다. 여러 파이프라인으로 연결된 명령어를 처리할 때의 첫 번째 명령어만 jobs 에 새로운 job으로 추가하고, 이후의 명령어에 대해서는 jobs 에서 해당하는 job의 pids 배열에 pid만 추가하도록 구현했다.

* Phase3
  + Identify Background process

|  |
| --- |
| /\* Should the job run in the background? \*/  if ((bg = ((\*argv[argc-1] == '&')) != 0)) {  argv[--argc] = NULL;  } else if ((bg = (argv[argc - 1][strlen(argv[argc-1]) - 1] == '&'))) { /\* case where '&' is attached without space. ex) ls -al&\*/  argv[argc - 1][strlen(argv[argc - 1]) - 1] = '\0';  }  return bg; |

이는 parseline() 함수의 일부이다. parseline() 함수에서 위와 같이 “command&”, “command &” 경우에 모두 백그라운드 로 처리할 수 있도록 작성했다.

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

**도표이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 2 (pipeline)**

**도표이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 3 (background)**

**도표이(가) 표시된 사진

자동 생성된 설명**