**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 이진용

학번 : 20191630

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**
* 개발 환경 : cspro2.sogang.ac.kr (Ubuntu 20.04.2 LTS)
* 개발 언어 : C
* 컴파일 버전 : gcc 9.4.0

command 를 입력받아 실행하는 linux shell을 구현한다. 내장되어 있는 bash shell을 reference로 한다. 구현한 myShell은 기본적으로 프로세스를 생성하여 명령어를 수행하고 결과를 stdout으로 출력해준다. 추가적으로 입력받은 명령어의 history를 저장하고 이를 출력 및 재실행 가능하게 한다.

phase2 에서는 파이프를 이용한 프로세스간 IPC 통신을 통하여 pipe 명령어를 구현한다.

ex) ls | grep my

마지막으로 phase3에선 stop, running, terminated, done 등 다양한 프로세스의 상태를 제어하여 background process를 구현한다. & 키워드를 사용하여 command를 background로 running 시키거나 control z 입력을 통해 foreground process를 stop 시키고 background로 돌려준다. 이외에 프로세스 terminated, terminal interrupt 등 특정 signal에 대한 handler를 구현하여 프로세스를 제어한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

* 명령어 실행 시 process를 fork하고 명령어를 실행
* /bin 폴더에 있는 기본 binary 실행 파일 실행가능 (ls, mkdir, rmdir …)
* built-in command 실행 가능 (cd , exit, history)
  + cd : 현재 위치 pwd를 이동한다.
  + exit : myShell 종료
  + history : 이전에 입력했던 명령어의 목록을 출력한다.
* 이전 입력 명령어 재실행 ( !!, !#)

1. Phase 2

* 파이프를 이용한 다중 명령어 처리 ex) ls | grep my

1. Phase 3

* & 명령어를 통해 command를 background running 상태로 실행
* foreground process를 control-z 입력을 통해 background stop 상태로 변경
* foreground process를 control-c 입력을 통해 종료
* jobs 명령어 입력시 현재 모든 background process의 상태를 출력
* fg 명령어를 통해 target process를 foreground running 상태로 변경
* bg 명령어를 통해 target process를 background running 상태로 변경
* kill 명령어를 통해 target process를 terminate 상태로 변경
  1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 함수를 통하여 현재 parent process의 리소스를 복사하여 child process를 생성한다. 이 때 child process는 command에서 parse한 argvs을 똑같이 들고 있게 된다. child process는 argvs에 있는 명령어가 built-in 명령어인지 확인 후 아니라면 /bin 에 있는 binary 파일을 찾아 실행한다.

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

child process에서 명령어를 실행하고 종료될때까지 parent process에서는 wait() 함수를 호출하여 child process가 종료될때까지 block 시켜준다. 이후 child process가 종료되면 SIGCHLD signal이 발생하게 되고 parent process는 이를 확인 후 해당 프로세스가 종료되었음을 확인 후 wait을 탈출하여 command 입력창으로 돌아오게 된다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

pipeline을 통해 파이프 앞의 명령어의 출력을 다음에 실행할 명령어의 입력을 넣어 구현한다. 예를 들어 ls | grep my 명령어는 ls 명령어의 출력을 stdout으로 나오게 하지 않고 다음 grep 명령어의 input으로 넣어 grep 명령어를 실행하게 한다.

해당 내용을 구현하기 위해선 파이프를 통하여 출력 결과를 stdout 대신 파이프에 저장하고 다음에 실행할 process에게 stdin에 출력이 저장된 파이프를 넘겨줘 명령어를 실행하게 한다. 파이프 입력, 출력 descriptor 을 저장할 fd[2] 배열을 선언한뒤 pipe() 함수를 통해 해당 fd[2]에 파이프 descriptor를 저장한다. 이후 dup2() 함수를 통해 stdin, stdout에 pipe에 담긴 내용을 연결한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

command를 parse할 때 command 내 pipeline 개수를 세고 저장한다. 저장된 파이프 개수를 통해 loop을 돌며 pipe\_count-1만큼 fork를 통하여 위의 서술한 과정을 진행한다. 마지막 명령어는 파이프의 저장된 출력 결과를 stdin에 복사하고 stdout을 되돌린다. 이후 명령어를 실행하여 최종 결과를 출력한다.

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

background 프로세스를 실행하는 주된 logic은 fork 하여 생성된 child process를 wait하지 않고 command 입력창으로 돌아오는 것이다. 다만 해당 process의 상태를 저장하고 관리하기 위해 Job 구조체를 선언하여 linked list형태로 구현하여 새로운 background process가 생성될 때마다 Job list에 추가하여 parent process에서 확인할 수 있도록 한다.

background process를 parent process에서 wait해주지 않기 때문에 해당 프로세스는 종료되면 defunt 상태가 된다. 이를 방지하여 해당 프로세스를 reaping해주기 위하여 프로세스의 상태가 변화하면 발생하는 signal인 SIGCHLD의 handler를 구현한다. 이 handler에서 waitpid WNOHANG 옵션을 사용하여 control flow를 block하지 않고 해당 프로세스를 reaping 시켜준다. 추가로 제대로 reaping 되었다면 background job node에서 삭제한다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

**Phase1 (fork & signal)**

1. **parse command**

| **char \*\*parse\_command(char command[]) {  char \*\*result;  char \*temp;  int count = 0;  char temp\_command[MAX\_COMMAND\_LENGTH];  strcpy(temp\_command, command);   // allocate result to store argvs  result = (char \*\*)malloc(SIZE\_OF\_CHAR\_POINTER);     temp = strtok(temp\_command, " ");  result[count] = temp; //store first command ex)ls, cd   count++;  while (temp != NULL)  {  //if next argv is null, break  result = realloc(result, SIZE\_OF\_CHAR\_POINTER \* (count + 1));  temp = strtok(NULL, " ");  result[count] = temp;   count++;  }   return result; }** |
| --- |
|  |
| command를 입력받아 공백마다 잘라 argvs에 저장한다 ex) ls -al -> “ls”, “-al” 이 때 strtok함수를 이용하여 command의 공백을 \0로 치환한다. 이후 temp에 자른 문자열을 나타내는 주소를 저장하고 이 주소값을 argvs에 저장한다. 이 과정을 strtok이 null을 반환할 때까지 반복한다. 최종적으로 모든 argvs가 담긴 result를 반환한다.   1. **execute command** |
| **void execute\_command(char \*\*args) {  int status; // store child process status  \_\_pid\_t pid;   if (execute\_excp\_command(args))  {  return;  }  else  {  if (fork() == 0)  {  // child process  if (execvp(args[0], args) < 0)  {  // if excution failed, print error  printf("%s: Command not found.\n", args[0]);  exit(0);  }  }  else  {  // parent process   wait(&status);  }  } }**  parsing 한 args을 통해서 명령어를 실행하는 함수이다. 가장먼저 command가 built-in command인지 확인 한뒤 아니라면 fork를 진행한다. 이 때 child process는 parent process의 args을 그대로 복사하기 때문에 복사된 args으로 명령어를 실행한다. /bin 에 있는 바이너리 파일을 실행하고 실패 시 에러 메세지를 출력한 뒤 종료한다. 해당 flow동안 parent process는 child process가 종료될 때까지 wait하고 종료 시 command 입력창으로 돌아오게 된다.   1. **command history** |
| **int read\_bash\_history() {  // open bash\_history  getcwd(project\_path, 300);  strcat(project\_path, "/.bash\_history");  fp = fopen(project\_path, "a");  fclose(fp);  fp = fopen(project\_path, "r");  char command[MAX\_COMMAND\_LENGTH] = {  0,  };  char \*temp;   // read all command history  while (!feof(fp))  {  fgets(command, MAX\_COMMAND\_LENGTH, fp);  if (command[0] == '\n' || command[0] == 0) // if command is blank , pass  continue;  command[strcspn(command, "\n")] = 0;  add\_command\_history(command, 0);  }  fclose(fp); }**  myShell 실행 시 가장먼저 working directory 에 있는 .bash\_history 파일을 열어 기존의 history들을 읽어온다. 이 때 줄넘김과 비어있는 명령어를 제외하고 파일에 모든 명령어들을 추가해준다. |
| **void add\_command\_history(char command[], int write\_file) {  // command is same with previous command, pass  if (history\_count)  if (!strcmp(command, command\_history[history\_count - 1]))  return;   history\_count++;  command\_history = (char \*\*)realloc(command\_history, sizeof(char \*) \* history\_count);  command\_history[history\_count - 1] = (char \*)malloc(sizeof(char) \* MAX\_COMMAND\_LENGTH);  strcpy(command\_history[history\_count - 1], command);   // if write flag is set, write it to file  if (write\_file)  {   fp = fopen(project\_path, "a");   fprintf(fp, "\n%s", command);  fclose(fp);  } }** |

history를 추가해주는 함수이다. history는 char double pointer로 선언하여 추가해줄 때마다 동적할당을 통해 size를 늘려준다. 추가될 때마다 index를 1씩 증가시켜 stack 자료구조로 구현하였다. 또한 파일에서 읽어올 때와 구분하기 위해 write\_file flag를 확인하여 command 입력 시에만 파일에 write하게 한다.

| **int replace\_history\_command(char command[]) {  int cur;  int index;  int len;  char temp[MAX\_COMMAND\_LENGTH];  char atoi\_str[100];  len = strlen(command);  int flag = 0; // flag store success replacing history command  for (cur = 0; cur < len - 1; cur++)  {  if (command[cur] == '!')  {  if (command[cur + 1] == '!') // if !! command match, replace !! to target command  {  if (history\_count == 0)  {  printf("-bash: !!: event not found\n");  return 0;  }   strcpy(temp, command\_history[history\_count - 1]);  len += strlen(temp) - 2;  if (strlen(command + cur + 2))  strcat(temp, command + cur + 2);  strcpy(command + cur, temp);  flag = 1;  }  else if (atoi(command + cur + 1)) // if !# command match, replace !# to target command  {  index = atoi(command + cur + 1);  sprintf(atoi\_str, "%d", index);  if (index > history\_count || index < 0)  {  printf("-bash: !%s: event not found\n", atoi\_str);  return 0;  }   strcpy(temp, command\_history[index - 1]);  len += strlen(temp) - strlen(atoi\_str) + 1;  strcat(temp, command + cur + strlen(atoi\_str) + 1);  strcpy(command + cur, temp);  flag = 1;  }  else if (command[cur + 1] == '0')  {  printf("-bash: !0: event not found\n");  return 0;  }  }  }  if (flag)  printf("%s\n", command);  return 1; }** |
| --- |

!!, !# command를 search하여 해당 명령어를 replace 시켜주는 함수이다. bash shell history에서는 !!와 !# 입력 시 해당 history 를 찾아 그대로 replace 한 뒤에 실행 시키기에 command를 parsing 하기전에 command에서 !!, !#을 감지하여 history에서 해당 명령어를 찾아 replace하게 된다. 이 때 replace에 실패하면 command를 실행하지 않게 하기위해 return 값을 다르게 한다.

**Phase2 (pipelining)**

1. **parsing pipeline**

| **// parse\_command()**  **// ...**  **do  {   next\_command = strstr(pipe\_command, "|"); //check command has pipe character  if (next\_command != NULL)  {  next\_command[0] = 0;  }   count = 0;  char \*\*result = (char \*\*)malloc(SIZE\_OF\_CHAR\_POINTER); // allocate result to store argvs  temp = strtok(pipe\_command, " ");  //store first command ex)ls, cd  result[count] = temp;   count++;  while (temp != NULL)  {  //if next argv is null, break  result = realloc(result, SIZE\_OF\_CHAR\_POINTER \* (count + 1));     temp = strtok(NULL, " \'\"");   result[count] = temp;   count++;  }    // store all argvs  args[pipe\_count++] = result;    if (next\_command != NULL)  pipe\_command = next\_command + sizeof(char);   } while (next\_command != NULL);  // store pipe counts'  \*pipe\_commands\_count = pipe\_count;** |
| --- |

기존의 parse\_command 함수에서 pipeline으로 구분하여 여러개의 command를 저장하게 구현했다. 기존의 double pointer였던 args를 triple pointer로 변경하여 둘 이상의 command의 argv를 저장할 수 있게 한다. 추가적으로 command를 execute할 때 pipe의 개수를 확인해야 하기에 추가적으로 pipe count도 저장하여 준다.

1. **execute pipe command**

| **// execute\_command()**  **// ...**  **for (i = 0; i < pipe\_count - 1; ++i) // excute pipe command  {  pipe(fd); // open pipe to connect excute output with next input of excute   create\_sub\_process(in, fd[1], args[i]);  close(fd[1]); // close unused pipe  in = fd[0]; // connect current output to next input  }  if (in != STDIN\_FILENO) // if current input is not stdin  {  dup2(in, 0); // connect current input to stdin  }  if (i == 0)  {  if (execute\_excp\_command(args[i]))  return;  }   if (pid = fork() == 0)  {   //child process  if (execvp(args[i][0], args[i]) < 0)  {  // if excution failed, print error  printf("%s: Command not found.\n", args[i][0]);  exit(1); // return abort  }  close(0); // close stdin  exit(0);  }  else  {   waitpid(pid, &status, 0);  dup2(saved\_stdin, 0); // recover stdin  }** |
| --- |

| **int create\_sub\_process(int in, int out, char \*\*args) {  pid\_t pid;  int status;  if ((pid = fork()) == 0)  {  if (in != STDIN\_FILENO)  {  // if input is not stdin, connect input to stdin, close input  dup2(in, STDIN\_FILENO);  close(in);  }  if (out != STDOUT\_FILENO)  {  // if output is not stdout, connect output to stdout, close output  dup2(out, STDOUT\_FILENO);  close(out);  }   if (!execute\_excp\_command(args)) // check this is built in command  {   if (execvp(args[0], args) < 0)  {  printf("%s: Command not found.\n", args[0]);  exit(1);  }  }   exit(0);  }  else  waitpid(pid, &status, 0); // wait child process terminated   return pid; }** |
| --- |

pipe command를 실행하기 위해 기존의 execute command를 변형하였다. execute\_command() 함수에서 최종적으로 실행하는 코드는 동일하게 하되, 실행 이전에 pipe count가 2 이상이라면 기존의 명령어를 먼저 실행시키고 이에 대한 출력을 stdout 대신 파이프에 저장하여 다음에 실행시킬 명령어의 stdin으로 넣어주게 된다. pipe count만큼 for문을 돌아 create\_sub\_process() 함수를 호출하여 입력과 출력을 위의 과정을 처리해준다. 이 때 필요없어진 pipe는 close하고 파이프와 stdin, stdout 을 연결시켜준 뒤 execution을 진행한다. for문을 탈출하고 execute\_command()로 돌아와 stdin에 출력값들을 연결해주고 마지막 명령어를 실행한다.

**Phase3 (background process)**

1. **& background execution**

| **// execute\_command() // ... is\_bg\_process = parse\_bg\_command(args[i]); // check this command is background execution  if (execute\_excp\_command(args[i])) // if this command is built in, execute and return  return;   if ((pid = fork()) == 0)  {  //child process  if (execvp(args[i][0], args[i]) < 0)  { // if execution failed, print error  printf("%s: Command not found.\n", args[i][0]);  exit(1); // return abort  }   close(0); // close stdin  exit(0);  }  else  {  current\_pid = pid; // store current pid   // recover stdin  dup2(saved\_stdin, STDIN\_FILENO);   if (is\_bg\_process)  {  //if this process is background, create job  create\_bg\_process(pid, command, 0);  }   if (!is\_bg\_process)  {  //wait untill process stopped, or terminated  waitpid(pid, &status, WUNTRACED);  }  }** |
| --- |

background process 실행의 main logic은 해당 process를 wait해주지 않는 것이다. wait하지 않고 해당 process의 정보만을 저장하여 main process에서 해당 프로세스를 제어하게 구현한다.

| struct JOB{  Job \* prev;  Job \* next;  int id;  int status;  pid\_t pid;  char command[MAX\_COMMAND\_LENGTH];  }; typedef struct JOB Job; Job \* first\_job; Job \* last\_job;  void create\_bg\_process(pid\_t pid, char command[], int is\_stop) {  // allocate new job, store info  Job \*new\_job = (Job \*)malloc(sizeof(Job));  setpgid(pid, pid);  new\_job->pid = pid;  new\_job->prev = NULL;  new\_job->next = NULL;  new\_job->id = curruent\_job\_id++;  if (is\_stop)  new\_job->status = PROCESS\_STOP;  else  new\_job->status = PROCESS\_RUNNING;  strcpy(new\_job->command, command);  remove\_char(new\_job->command, '&');   // add to job list  if (first\_job == NULL)  first\_job = new\_job;   if (last\_job != NULL)  {  last\_job->next = new\_job;  new\_job->prev = last\_job;  }  last\_job = new\_job;   // print process state  if (new\_job->status == PROCESS\_RUNNING)  print\_bg\_process\_create(new\_job->pid, new\_job->id);  else  print\_bg\_process\_state(new\_job, PROCESS\_STOP); } |
| --- |

jobs 명령어를 통해 background process들을 확인해야 하기에 Job 구조체를 선언하여 double linked list 자료구조로 모든 Job들에 접근 가능하게 한다. 새로운 job을 동적할당 한 뒤에 last\_job 뒤에 추가해주는 형태로 구현한다. 추가적으로 foreground에서 control-z 입력 시 process 가 stopped 된 상태로 background로 생성될 수 있으니 이부분도 고려하여 구현한다.



| void terminate\_process\_handler(int sig) {  // when process terminate itself or signal, this handler called  int status;  pid\_t pid;  Job \*cur = first\_job;  int flag = 0;   while ((pid = waitpid(-1, &status, WNOHANG)) > 0)  {  // wait terminated process, get it's status  while (cur != NULL)  {  //find target process in background job list  if (cur->pid == pid)  {  if (WIFEXITED(status))  {  // if process terminated normally  Sio\_puts("\n");  print\_bg\_process\_state(cur, PROCESS\_DONE);  remove\_job\_node(cur);  }   else if (WTERMSIG(status) == SIGKILL)  {  // if process terminated by signal  Sio\_puts("\n");  print\_bg\_process\_state(cur, PROCESS\_TERMINATE);  remove\_job\_node(cur);  }    break;  }  cur = cur->next;  }  } }  void remove\_job\_node(Job \*job) {  sigset\_t mask, prev;  sigaddset(&mask, SIGCHLD);  sigprocmask(SIG\_BLOCK, &mask, &prev);    // remove job in job list  if (job == first\_job)  {  first\_job = job->next;  }  if (job == last\_job)  {  last\_job = job->prev;    if(last\_job == NULL)  curruent\_job\_id = 1;  else  curruent\_job\_id = last\_job->id + 1;  }  if (job->next != NULL)  job->next->prev = job->prev;  if (job->prev != NULL)  job->prev->next = job->next;   free(job);  sigprocmask(SIG\_UNBLOCK, &mask, &prev); } |
| --- |

background process는 main process에서 wait해주지 않기 때문에 해당 프로세스가 종료되면 defunt 상태로 남아있게 된다. 때문에 해당 프로세스가 종료될 때 발생하는 SIGCHLD siganl handler를 구현한다. 이 handler는 waitpid() 함수를 통해 종료된 process를 reaping해주는데 이 때 WNOHANG 을 사용하여 프로세스가 stopped된 상태여도 control flow를 block하지 않는다. reaping 된 프로세스는 Job list에서 삭제하며 해당 프로세스가 정상종료를 통해 terminate 되었는지, signal을 통해 종료되었는지를 확인한 후 해당 상태를 출력한다.

1. **terminal signal handler ( control-z, control-c)**

| void suspend\_process\_handler(int sig) {  // when get terminal stop signal, stop process  if (current\_pid == 0)  {  // when it is root process, pass  return;  }   Job \*job = first\_job;  while (job != NULL)  {  //find current fg process and stop  if (job->pid == current\_pid)  {  job->status = PROCESS\_STOP;  remove\_char(job->command, '&');  kill(current\_pid, SIGSTOP);  return;  }   job = job->next;  }  // if current process is not in job list, add it to job  create\_bg\_process(current\_pid, current\_command, PROCESS\_STOP); } |
| --- |

해당 handler는 control-z 입력 시 foreground에서 running하고 있는 process를 stop 상태로 background로 넘겨주는 기능을 한다. current\_pid에 현재 foreground로 돌아가는 process pid를 통해 Job list에 있는 process라면 해당 프로세스의 상태를 변경하고 만약 Job list에 없는 foreground process라면 stop된 상태의 background process를 추가해준다.

| **void terminate\_current\_process\_handler(int sig) {  // when get interrupt signal, terminate current process  pid\_t pid = current\_pid;  Job \*job;  job = first\_job;  if (pid != 0)  {  //find target job  while (job != NULL)  {  if (job->pid == pid)  {  // terminate target process  kill(pid, SIGKILL);  }   job = job->next;  }    } }** |
| --- |

해당 handler는 foreground에서 running 중인 process를 control-c 입력시 (SIGINT 시그널 발생) 해당 process에 SIGKILL signal을 보내준다. SIGKILL signal을 보내게 되면 해당 process는 SIGCHLD siganl을 발생시키며 terminated 되기에 terminate\_process\_handler() 가 해당 프로세스를 reaping 시켜주고 Job list에서 삭제해준다.

1. **background process control command (jobs, fg, bg, kill)**

| **if (strcmp(args[0], "jobs") == 0) // if command is jobs  {  Job \*job = first\_job;  while (job != NULL) // print all jobs  {  if (job->status == PROCESS\_RUNNING)  print\_bg\_process\_state(job, PROCESS\_RUNNING);  else  print\_bg\_process\_state(job, PROCESS\_STOP);  job = job->next;  }  return 1;  }** |
| --- |

built-in command로 구현한 jobs command는 first\_job부터 last\_job까지 모든 job들을 출력해준다.

| else if (strcmp(args[0], "fg") == 0)  {  Job \*job = first\_job;  pid\_t pid;  int status;  while (job != NULL)  {  if (job->id == atoi(args[1] + 1))  {  //reset sigmask  sigaddset(&mask, SIGTSTP);  sigprocmask(SIG\_UNBLOCK, &mask, &prev);  Signal(SIGCONT, resume\_process\_handler);   Sio\_puts(job->command);  Sio\_puts("\n");  current\_pid = job->pid;  kill(job->pid, SIGCONT); // resume target process   waitpid(job->pid, &status, WUNTRACED); // catch process state terminated or stopped  if (WIFSTOPPED(status)) // if process stopped  {  print\_bg\_process\_state(job, PROCESS\_STOP);  return 1;  }  remove\_job\_node(job); // when process terminated, remove current job  return 1;  }   job = job->next;  }  return 1;  } |
| --- |

fg command를 실행하게 되면 먼저 id를 통해 Job list에서 해당 process를 찾는다. 해당 프로세를 찾게되면 SIGCONT signal을 보내주어 stopped 된 process를 running 상태로 바꾸고 해당 process를 wait해준다. 해당 process가 stop되거나 terminated 되면 wait를 빠져나가게 되고 이때의 상태를 확인하여 해당 프로세스를 상태를 Job에 저장하거나 삭제시켜준다.

| else if (strcmp(args[0], "bg") == 0)  {    Job \*job = first\_job;  int status;  char output[MAX\_COMMAND\_LENGTH \* 2];  while (job != NULL)  {  if (job->id == atoi(args[1] + 1))  {  //print job state  strcat(job->command, " &");  sprintf(output, "[%d] %s\n", job->id, job->command);  current\_pid = job->pid;  Sio\_puts(output);   // change process state and continue  job->status = PROCESS\_RUNNING;  kill(job->pid, SIGCONT); // send signal to continue process   return 1;  }   job = job->next;  }  Sio\_puts("bg: no such job\n");  return 1;  } |
| --- |

bg command도 마찬가지로 해당 process에 SIGCONT signal을 보내주고 프로세스 상태를 변경해준다. fg와 차이로 해당 프로세스는 wait해주지 않고 command 입력창으로 바로 돌아오게 된다.

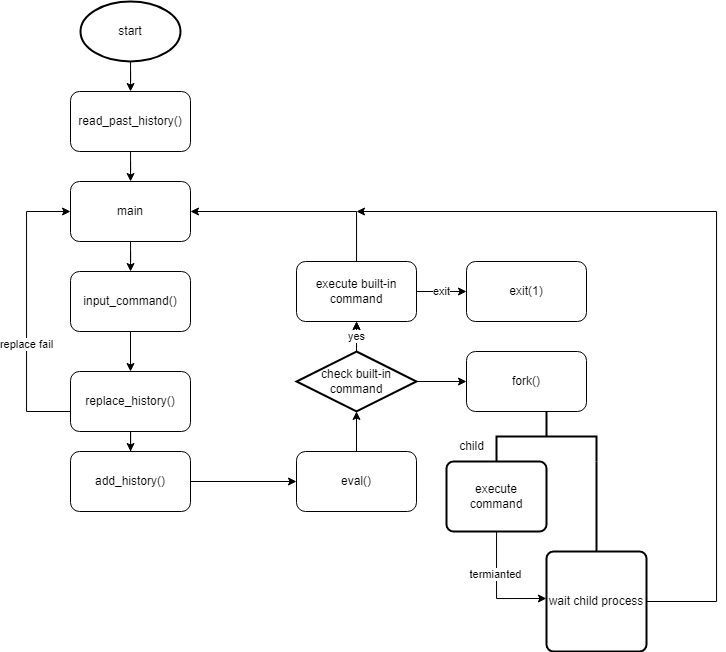
| else if (strcmp(args[0], "kill") == 0)  {  Job \*job = first\_job;   while (job != NULL)  {  if (job->id == atoi(args[1] + 1)) // if find job with it's id, send kill signal  {  kill(job->pid, SIGKILL);  return 1;  }   job = job->next;  }  Sio\_puts("kill: no such job\n"); // if can't find job  return 1;  } |
| --- |

마지막으로 kill 명령어이다. kill 명령어는 SIGINT handler와 마찬가지로 job id로 background process를 찾아 해당 프로세스에 SIGKILL signal을 보내준다.

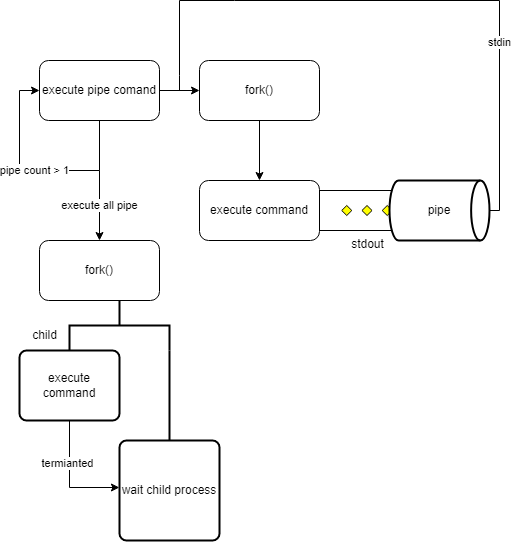
1. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

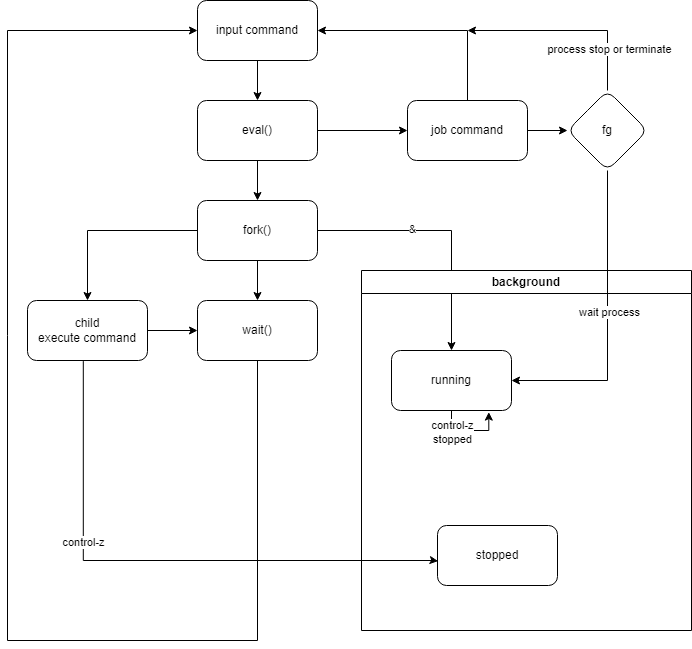
**Phase 1 (fork)**

****

**Phase 2 (pipeline)**

****

**Phase 3 (background)**

****