**System Programming Project 2**

담당 교수 : 김영재

이름 : 차수환

학번 : 20222089

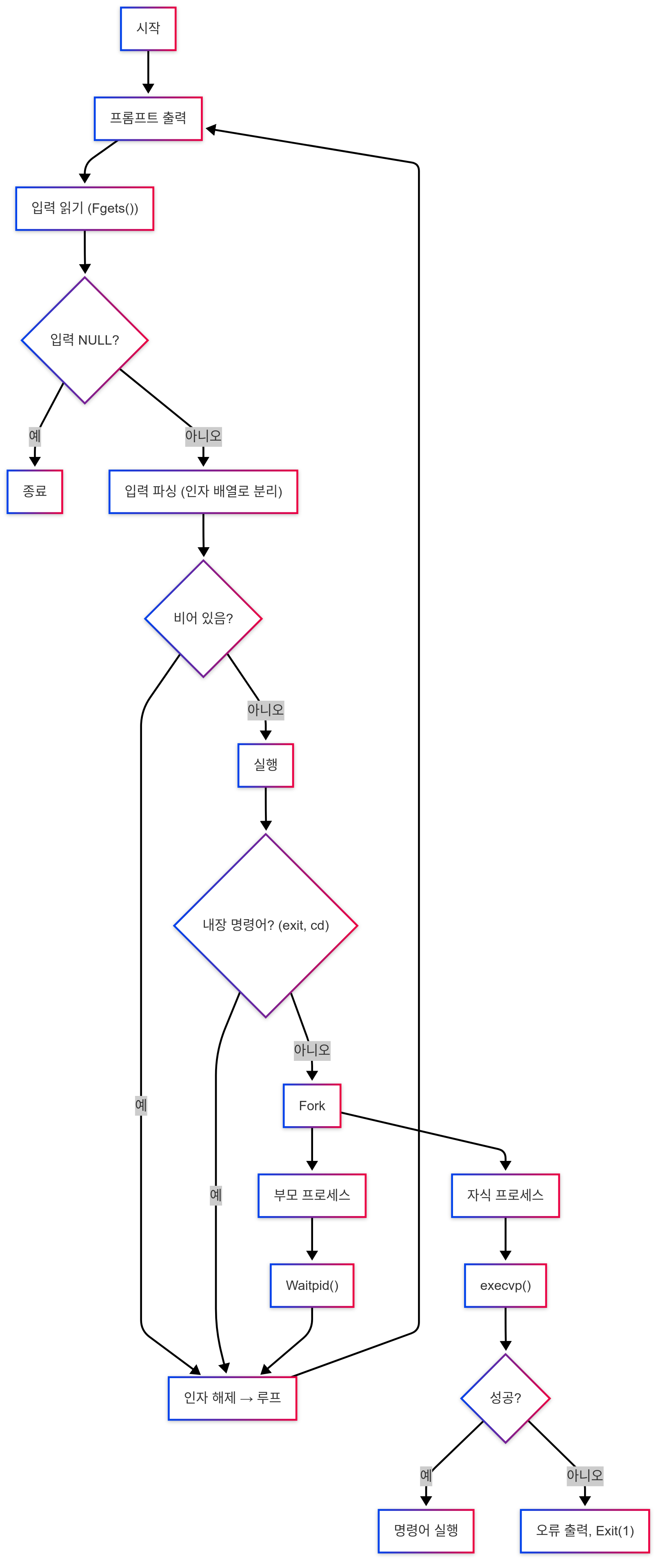
1. **개발 목표**

* 이 프로젝트는 MyShell이라는 Linux shell을 구현하는 것을 목표로 한다.
* MyShell은 사용자의 명령어들을 처리하고, Pipeline, Background 작업, Signal Handler을 지원하는 기본 UNIX shell을 모방한다.
* 내장 명령어와 외부 명령어를 실행하고, Phase 1을 시작으로 Phase 3까지 3단계를 통해 점진적으로 기능을 확장한다.

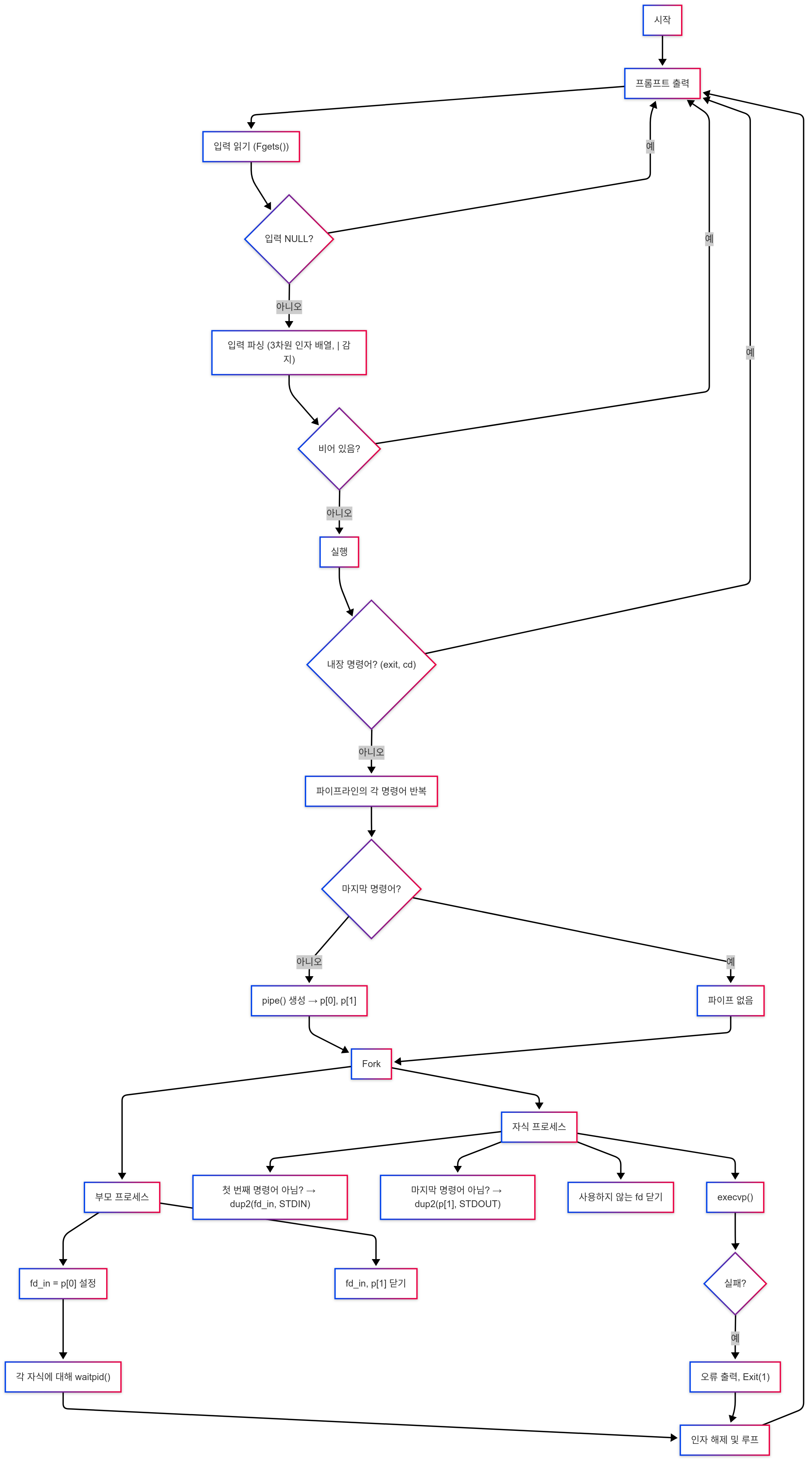
1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1
   * MyShell은 기본 명령어의 실행을 지원한다.
   * 사용자의 입력을 읽고 parsing하여, exit이나 cd같은 내장 명령어나, 외부 명령어를 fork()와 execvp()를 사용하여 실행한다.
   * 자식 프로세스가 실행되면, 완료될 때까지 대기한 후, 다시 프롬프트를 표시한다.
3. Phase 2
   * MyShell은 명령어 Pipeline과, 따옴표나 공백과 같은 특수 문자 처리를 지원하도록 확장된다.
   * pipe()를 사용해 여러 명령어를 연결하고, 따옴표 안의 입력을 올바르게 parsing하도록 한다.
4. Phase 3
   * 기존 MyShell에 백그라운드 작업(e.g. sleep 10 &)과, 작업 관리(jobs, bg, fg, kill), Signal Handler(SIGINT, SIGTSTP, SIGCHLD)를 추가한다.
   * 작업 상태를 관리하고, tcsetpgrp()로 터미널의 제어를 보장한다.
   1. **개발 내용**

* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
    - 외부 명령어를 실행할 때, Fork()를 호출해 child process를 생성하여 명령어를 실행한다.
    - child process는 입력된 명령어와 인자를 실행하고, parent process는 child의 종료를 기다린다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - child process는 기본 시그널 동작을 상속받고, SIGINT 등으로 종료된다.
    - parent process는 child process의 종료를 기다리고, 종료 상태를 확인해 오류를 처리한다.
    - 시스템 호출 오류가 발생하면 오류 메세지를 출력한다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - Pipeline을 처리하기 위해 명령어 수만큼 루프를 돌게 한다.
    - 입력을 받으면 |를 기준으로 인자 배열로 나눈다.
    - 각 명령어는 별도의 child process에서 실행되고, 파이프는 pipe()로 생성되며, 각 파이프는 읽기(p[0])와 쓰기(p[1]) 파일 디스크립터를 제공한다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - 최대 16개의 명령어를 지원하며, 입력을 parsing할 때, |를 만나면 새로운 명령어 배열을 시작하고, 인자를 저장한다.
    - 명령어의 수를 기반으로 루프를 돌며, 각 명령어에 대해 파이프를 생성하고 파일 디스크립터를 설정하도록 한다.
    - 파이프라인의 각 단계에서, Dup2()로 입력과 출력을 연결한다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - 입력 끝에 &가 있으면 백그라운드 플래그르 설정하여, 명령어를 백그라운드에서 실행하도록 한다.
    - 작업은 작업 목록에 추가되고, 포그라운드 작업의 경우에는 터미널의 제어를 child로 넘기고, 종료 또는 중지를 기다리며, 백그라운드 작업의 경우 즉지 다음 입력을 받도록 한다.
    - 각각의 명령어들은 Fork()로 child proess를 생성하고, 동일한 프로세스 그룹에 속하도록 설정한다.
    - 시그널을 통해 child process의 상태 변화를 감지하고, job 상태를 업데이트한다.
  1. **개발 방법**
* Phase 1:
  + myshell\_readinput(): Fgets()로 사용자 입력을 읽고, 최대 1024자의 문자열을 동적 할당한다. 입력이 없으면 메모리를 해제하고 NULL을 반환한다.
  + myshell\_parseinput(): 입력을 공백으로 분리해 인자 배열(char \*\*)을 생성한다. strchr()로 공백을 처리하고, 메모리는 Malloc()으로 할당된다.
  + myshell\_execute(): 내장 명령어(exit, cd)는 builtin\_command()로 처리한다. 외부 명령어는 Fork()로 child process를 생성하고, child는 execvp()로 명령어를 실행한다. execvp()는 PATH를 검색해 실행 파일을 찾으며, 실패 시 메세지를 출력하고 exit(1)로 종료한다. 부모는 Waitpid()로 자식의 종료를 기다리고, 오류 발생 시 unix\_error()로 메시지를 출력한다. 메모리는 Free()로 정리된다.
  + builtin\_command(): exit는 프로그램을 종료하고, cd는 chdir()로 디렉토리를 변경한다. getenv("HOME")를 사용해 인자가 없는 cd를 처리한다.
* Phase 2:
  + myshell\_parseinput(): 입력을 |와 따옴표를 기준으로 3차원 인자 배열(char \*\*\*)로 나눈다. strpbrk()로 파이프, 공백, 따옴표가 있는지 감지하고, 따옴표 안의 문자열은 공백을 포함하여 하나의 인자로 처리한다. 최대 16개의 명령어와 128개의 인자를 지원하며, 메모리는 Malloc()으로 할당된다.
  + myshell\_execute(): Pipeline을 처리하기 위해 명령어의 수만큼 루프를 돈다. 각 명령어는 Fork()로 child process를 생성한다. pipe()로 파이프를 생성해 읽기(p[0])와 쓰기(p[1]) 파일 디스크립터를 만든다. child는 Dup2()로 이전 명령어의 출력을 STDIN\_FILENO로, 자신의 출력을 p[1]로 리디렉션한다. 사용하지 않는 파일 디스크립터는 Close()로 닫는다. execvp()로 명령어를 실행하며, 실패 시 오류 메시지를 출력하고 exit(1)한다. 부모는 Waitpid()로 모든 자식을 기다리고, 파일 디스크립터를 정리한다.
  + free\_args(): 3차원 인자 배열을 해제하고, 각 명령어의 인자와 배열을 Free()를 사용하여 정리한다.
* Phase 3:
  + myshell.h: Job 구조체를 추가하여 pgid, state(FG, BG, STOP, DONE, TERM), jid, cmdline, next, prev를 정의하고, 컴파일러가 레지스터로 저장하지 않게 하기 위해서 volatile로 선언한다.
  + myshell\_readinput(): tcsetpgrp()로 터미널이 어느 프로세스 그룹의 제어를 받고 있는지를 설정하고 Fgets()로 입력을 읽는다.
  + myshell\_parseinput(): |, &, 따옴표를 처리해 3차원 인자 배열과 백그라운드 플래그를 반환한다. &는 백그라운드 작업을 나타내고, 따옴표 안의 문자열은 하나의 인자로 처리된다.
  + myshell\_execute(): 내장 명령어는 builtin\_command()로 처리하며, 외부 명령어는 파이프라인의 처리와 백그라운드를 지원한다. 각 명령어는 Fork()로 child process를 생성하고, Setpgid()로 동일한 프로세스 그룹(pgid)으로 설정한다. 파이프는 pipe()와 Dup2()로 연결된다. 백그라운드 작업은 add\_job()로 BG 상태로 추가되며, 포그라운드 작업은 FG로 추가된다. 포그라운드는 tcsetpgrp(pgid)로 터미널 제어를 해당 프로세스 그룹으로 넘기고, waitpid(-pgid, WUNTRACED)로 대기한다. 백그라운드는 waitpid(-pgid, WNOHANG)로 즉시 종료된 작업을 확인한다.
  + builtin\_command(): jobs는 print\_jobs()로 작업 목록을 출력하고, bg와 fg는 find\_job\_by\_jid()로 작업을 찾아 SIGCONT로 실행한다. fg는 tcsetpgrp()로 터미널의 제어를 넘긴다. kill %jid는 SIGTERM으로 작업을 종료한다. exit는 모든 작업을 종료한다.
  + 작업 관리 함수: init\_jobs()는 작업 목록을 초기화하고, add\_job()은 새 작업을 추가하며, delete\_job()은 완료된 작업을 제거한다. update\_job\_state()는 작업 상태를 변경하고, print\_jobs()는 BG 혹은 STOP 상태의 작업을 출력한다. clear\_done\_jobs()는 DONE 상태이거나 TERM인 작업을 정리한다. 모든 함수는 Sigprocmask()로 SIGCHLD를 차단해 동시 접근을 보호한다.
  + 시그널 핸들러: sigchld\_handler()는 waitpid(-1, WNOHANG | WUNTRACED | WCONTINUED)로 자식의 상태를 확인하고, WIFEXITED, WIFSIGNALED, WIFSTOPPED, WIFCONTINUED로 상태를 업데이트한다. sigint\_handler()와 sigtstp\_handler()는 FG 작업에 SIGINT, SIGTSTP를 보내 종료 또는 중지한다.

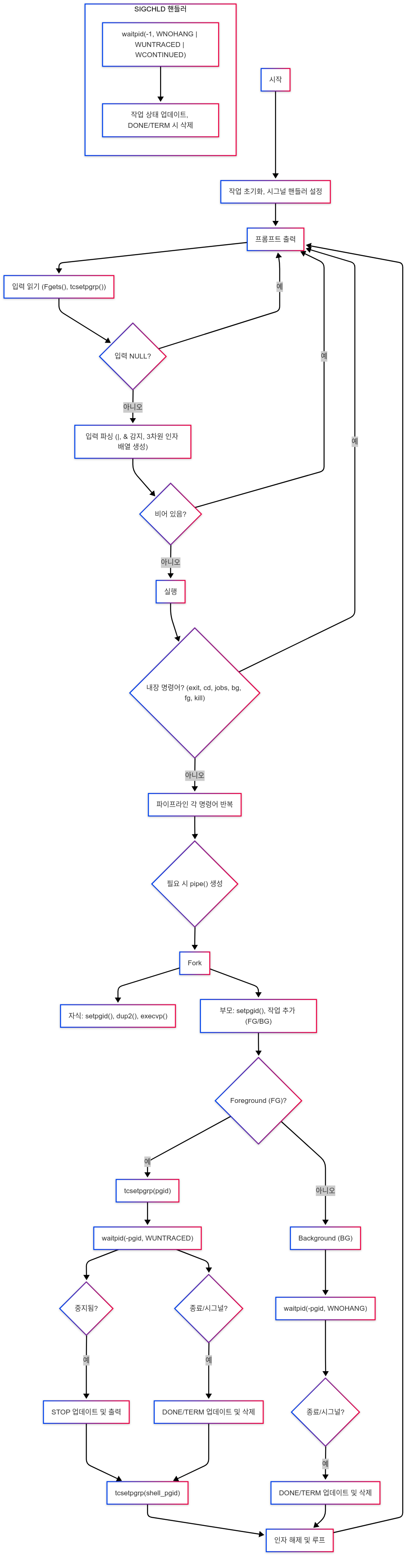
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****