System Programming Project 1

1. 개발 목표

본 프로젝트는 실제 Linux Shell을 제한적인 범위 내에서 개발해봄으로써 지금까지 강의에서 다룬 'System-Level Process Control', 'Signal Handling', 'Inter Process Communication' 등의 개념을 익히는 것을 목표로 한다.

프로젝트는 총 3가지 Phase로 이루어져 있다. 첫 번째 단계는 Shell의 기본적인 'Read – Evaluation' 과정을 구현한다. 두 번째 단계는 IPC를 가능케 하는 파이프라인을 구현하는 과정이며, 마지막 세 번째 단계는 Shell이 Background Process를 문제 없이 다룰 수 있게 만드는 과정이다.

즉, 단계를 거듭해 나가면서 안정적인 'MyShell' 프로그램을 점진적으로 완성해 나가는 것이 핵심이다. 그 과정에서 상기한 개념들을 직접 적용해보고, 각 개념의 적용에서 발생하는 문제점들에 대해 Troubleshooting, Debugging을 해가며 구현 측면에서의 주요 포인트가 무엇인지를 이해할 수 있게 된다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

Linux Shell의 기본적인 'Read – Evaluation Cycle'을 구현하는 단계이다. 'ls', 'ps', 'mkdir', 'touch', 'cat', 'echo' 등과 같은 기본적인 명령들부터, 'cd'나 'exit'과 같은 Built-In-Command의 정상적 동작을 보장해야 한다. 현재 본인의 'MyShell' 프로그램에서는 상기한 명령들을 포함해, 'execvp' System Call에서 실행할 수 있는 모든 명령을 지원하고 있다.

2. Phase 2

Linux Shell의 IPC(Inter Process Communication)를 위한 Pipeline 기능을 구현하는 단계이다. 파이프 개수에 제한 없이, 파이프 사이의 공백 여부에 상관없이, Linux Shell에서 돌아가는 것과 같이 자유로운 파이프라인 기능을 구현하는 것이 핵심으로, 현재 본인의 'MyShell' 프로그램에서는 상기한 내용이 모두 정상적으로 동작하고 있다.

3. Phase 3

본 Phase의 핵심은 강의에서 다룬 다양한 Signal 개념을 토대로 프로세스를 Background에서 실행시키는 것이다. 이때 단순히 프로세스를 Background에서 돌리는 것 뿐만 아니라, 'Ctrl+Z' / 'Ctrl+C' 타이핑, 그리고 'fg', 'bg', 'kill' 등의 Built-In–Command로 프로세스가 Foreground Running, Background Running, Suspended, Terminated 상태를 자유롭게 오갈 수 있게 통제하는 것이 주요하다. 단, 프로세스가 네 가지 상태를 자유롭게 오가되, 현재 상태에 따라 해당 상태의 특성을 유지해야 한다.

또한, 'jobs' Built-In-Command를 구현하여 상기한 과정을 명시적으로 확인할 수 있도록 지원한다. 본인의 'MyShell' 프로그램은 이러한 일련의 과정을 모두 정상적으로 지원하고 있다.

B. 개발 내용

- **Phase1 (fork & signal)**

✓ fork를 통해서 child process를 생성하는 부분

우선, fork에 앞서, 'eval' 함수에서 입력 명령 'cmdline'을 바탕으로 일련의 과정을 거쳐 Argument List 'argv'를 만든다. Argument가 Built-In-Command인지 확인해, Built-In일 경우, fork를 호출하지 않고 별도의 처리 루틴을 호출해 Shell 내에서 해결하고 다시 'main'으로 돌아간다.

Built-In이 아닐 경우, fork를 호출하여 새 프로세스를 생성한다. 이때, 새 프로세스에서 SIGINT, SIGTSTP 시그널에 반응하지 않도록 Signal Masking을 진행한다. 'Ctrl+C', 'Ctrl+Z'에 대한 Child 프로세스의 반응을 Child가 직접 하지 않고, Parent가 Signal Handler를 통해 대신 처리하게 하기 위함이다.

Masking이 끝나면, 'Execvp' 함수를 통해 새 프로세스에 Target Command File을 입힌다. 'Execvp' 함수를 사용하는 이유는, 우리가 다루는 Command의 파일들이 특정 Directory에만 있는 것이 아니기 때문에, Path를 함께 전해야 하는 'Execve' 대신, 파일명만으로도 파일을 찾는 'Execvp'를 사용하는 것이다.

✓ connection을 종료할 때 parent process에게 signal을 보내는 signal handling 하는 방법 & flow

Parent에서는 우선, 'Ctrl+C' 처리를 하기 위한 Job Queue Push 작업을 한다. 앞서 말한 SIGINT Signal Handler의 동작을 위한 작업으로, 생성한 프로세스가 수행되는 도중 Shell에서 'Ctrl+C'가 눌렸을 때, Handler에서 SIGINT를 캐치한 후, Job Queue를 순회하여 Push된 Foreground Process를 찾고, 해당 프로세스를 죽이는 시그널을 보내려는 것이다. 이로 인해 종료된 Child 프로세스는 후술할 SIGCHLD Signal Handler가 Reaping할 것이다.

이어서, 'Ctrl+C'가 눌리지 않는 상황을 처리하자. Parent는 'Sigsuspend'함수를 이용한 Explicit Waiting을 수행한다. Child의 종료로 인해 발생하는 SIGCHLD를 Catch하는 Signal Handler를 프로그램에 설치하고, 해당 Handler에서 잔존하는 '종료 프로세스'들에 대해 반복적으로 'Waitpid' 함수를 호출해 Reaping한다. 이러한 과정으로 Child Process를 정확하게 처리할 수 있다.

즉, Parent에서는 'Ctrl+C 대비'에서 'Wait'으로 이어지는 Flow를 진행한다고 보면 되는데, 이러한 일련의 과정 시작과 끝에서, 'Sigprocmask'로 'Possible Interrupting Signal'을 잠시 Blocking하고 있음을 강조한다. 또한, Deadlock의 발생을 방지하기 위해 각 Handler에서는 모두 'Async-Signal-Safety'가 보장되는 함수들만 사용한다는 점도 강조한다. 한편, Push된 Queue Element도 각 Handler에서 Dequeue하기 때문에 문제가 발생하지 않는다.

- **Phase2 (pipelining)**

  ✓ Pipeline('|')을 구현한 부분 (design & implementation)

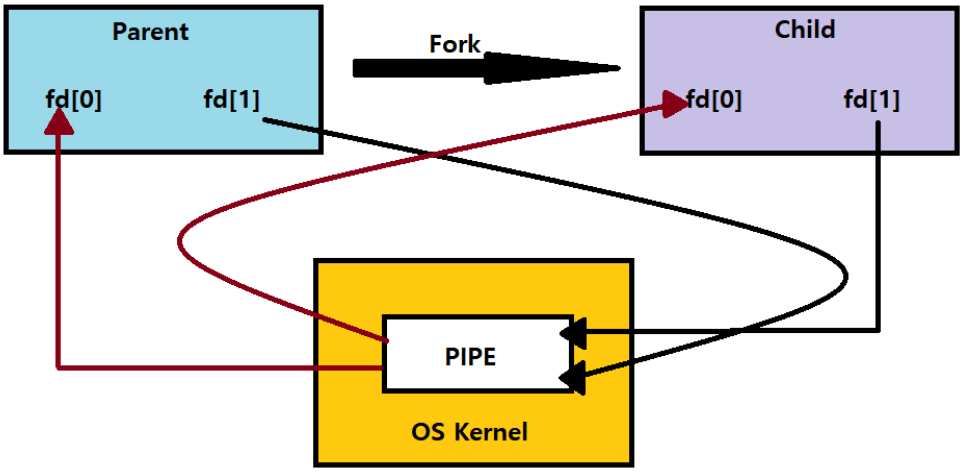
  우선, 파이프라인을 처리하기 위해, 입력 명령 'cmdline'에 Pipe(|)가 존재하는지를 'Fgets' 수행 이후 바로 확인한다. 전역 플래그를 통해 이를 제어하며, 만약 파이프가 존재한다면, 이어지는 'eval' 함수에서 File Descriptor 배열 'fd'에 'Pipe' 함수를 통해 파이프라인을 형성한다.

  'eval' 함수에선, Phase1에서처럼 'cmdline'을 'buffering'과 'parseline' 함수를 통해 변형한다. 이때의 'buffering' 함수가 중요한데, 이전 Phase1에서와 같이 Buffering을 진행하되, '|' 문자를 만나면 더 이상의 Buffering을 하지 않는다. 이는 후술할 'Recursion'을 구현하기 위함이다. 파이프를 만나기 이전까지의 명령만 취급해 처리하고, 파이프 이후의 명령은 다음 'eval' 함수 호출에서 처리하려는 의도이다. (cmdline의 시작 주소값도 이에 맞춰 변화)

따라서, 만약 'buffering' 함수의 Buffering 과정에서 파이프 문자를 만나지 않는다면, 그것은 '애초에 파이프가 없는 명령'이거나, '다중 파이프 상황에서 마지막 재귀 호출의 "cmdline" 명령 조각'에 해당할 것이다. 이는 'last_flag'라는 플래그 변수를 통해 기록한다. (이 플래그의 이용 방법은 후술한다.)

'cmdline' 처리('buf' 문자열이 만들어진다.)가 완료되면, Phase1에서처럼 Built-In-Command 확인 과정을 거친 후, 본격적인 프로세스 생성에 돌입한다. 본인은 'Recursion'을 통해 파이프라인을 처리하기 때문에, 파이프 문자 이전 'cmdline' 변형 결과 리스트 'argv'에 대해서만 Phase1과 같이 'fork -> execvp' 한다. 파이프 문자 이후의 'cmdline'은 재귀 호출로 넘기는데, 이때의 핵심은, 'Recursion' 관계에서, 'Caller인 Command Process'의 출력 파일을 가리키는 Descriptor 'fd[0]'을 'Callee인 Command Process'에 넘긴 후, 'Callee'에서 이를 입력 파일의 File Descriptor로 활용한다는 것이다.

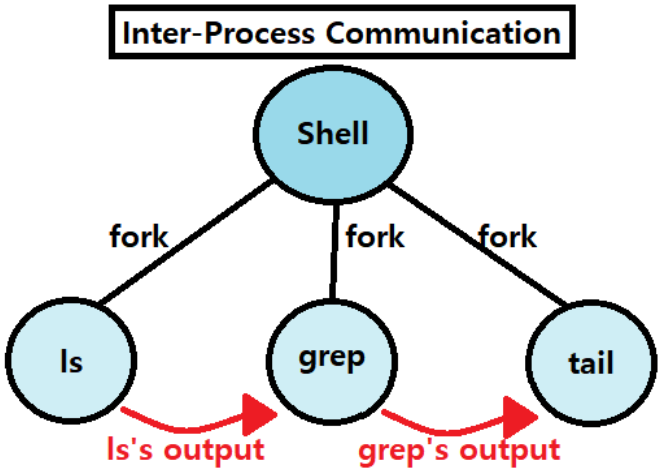
배열 'fd(Size는 2)'에서 'fd[0]'을 다음 재귀 호출로 넘기는 이유는, 아래의 그림을 통해서 설명할 수 있다. 본인은 아래와 같이 'fd' 관계를 정립했다.



fork로 생성된 Command Process는 fd[1]에 그 수행 결과를 출력한다. 그리고 데이터는 fd[1]에서 Pipeline을 거쳐 fd[0]으로 이어진다. 따라서 'fd[0]'을 넘기는 것이다. 보다 더 자세한 내용은 아래 항목에서 서술한다.

✓ Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

설명에 앞서, 다중 파이프 라인 상황을 상정함을 밝힌다. 해당 가정이 깔리면, 단일 파이프 상황도 자연스럽게 Cover된다.



최초 'main'에서 'cmdline' 입력 시 파이프가 구체적으로 3개 존재했다고 해보자. 예시는 "ls | grep 'my' | tail -2"라는 명령이다.

1) 첫 번째 호출 : ls만 추출된다.

- fork를 하여 새 프로세스가 생성된다. last_flag는 FALSE이다.

- 이때, Child에서는 Dup2 함수를 이용해 'eval' 함수의 Parameter 변수인 'passed_fd'로 입력으로 받을 파일을 대체한다. 최초 호출에서는 이 값이 0, 즉, stdin을 가리키기 때문에 변화가 없다.

- 이어서, Dup2 함수를 다시 호출해 'fd[1]'으로 출력 파일을 대체한다.

- 따라서, Child(ls)의 수행 결과는 fd[1]으로 출력된다. (정확히는, fd[1]이 가리키는 File이지만, 표현의 편의 상, 그냥 Descriptor를 명시한다.)

- Pipeline을 통해 결과 데이터는 fd[0]으로 흐르고, fd[0]을 passed_fd로 넘기며 다음 'eval' 함수를 호출한다.

2) 두 번째 호출 : grep 'my'가 추출된다.

- fork를 하여 새 프로세스가 생성된다. last_flag는 여전히 FALSE이다.

- 이때, Child에서는 Dup2 함수를 이용해 'eval' 함수의 Parameter 변수인 'passed_fd'로 입력 파일을 대체한다.

- 따라서, 'grep'의 입력 데이터로, 이전 'eval' 호출에서의 Command Process 출력 데이터가 입력된다.

- 이어서, Dup2 함수를 다시 호출해 'fd[1]'으로 출력 파일을 대체한다.

- 따라서, Child(grep)의 수행 결과는 fd[1]으로 출력된다.

- Pipeline을 통해 결과 데이터는 fd[0]으로 흐르고, fd[0]을 passed_fd로 넘기며 다음 'eval' 함수를 호출한다

3) 세 번째(마지막) 호출 : tail -2가 추출된다.

- fork를 하여 새 프로세스가 생성된다. last_flag는 TRUE이다..

- last_flag TRUE 상황에서는, Child가 Dup2 함수를 한 번만 호출한다. 'eval' 함수의 Parameter 변수인 'passed_fd'로 입력 파일을 대체한다.

- 따라서, 'tail'의 입력 데이터로, 이전 'eval' 호출에서의 Command Process 출력 데이터가 입력된다.

- last_flag가 TRUE이기 때문에, Dup2를 한 번 더 호출하지 않고, stdout으로 Command Process의 출력 데이터를 내보낸다. 만약, 파이프가 없는 상황에서는, Command 프로세스의 수행에 있어서 특정 입력 데이터가 필요치 않으므로, 앞선 Dup2는 상관이 없다.

이러한 재귀 흐름을 거쳐서 본 'MyShell'에서 IPC 기능이 실현된다. 이때, 각 'eval' 호출에서는, fork 이후 Parent에서 File Descriptor를 조정해주는 작업이 필요하다. 그래야만 상기한 흐름이 정상적으로 동작하기 때문이다.

last_flag가 FALSE일 때는, 즉, 중간 과정 호출일 경우, 현재 호출에서의 fd[1]과 넘겨받은 passed_fd를 Close한다. 이들은 더 이상 필요하지 않기 때문이다.

last_flag가 TRUE일 때는, File Descriptor Table을 최초 상태로 돌려주어야 한다. Parent Process는 다시 'main'으로 돌아가 Shell 동작을 이어가야 하기 때문이다. 이를 위해 temp_fd라는 변수가 도입된다. temp_fd는, 'Non-Pipe 상황'일 경우 STDOUT_FILENO를, 'Pipe 상황'일 경우 STDIN_FILENO를 가리킨다. 이유는 간단하다. 'Non-Pipe'이면, 그냥 원래대로 두는 것이고, 'Pipe' 상황이면, STDOUT_FILENO를 temp_fd(0번)에 위치시키고, STDIN_FILENO를 passed_fd 인덱스에 위치시킨 다음, Pipeline을 흐르게 하여 뒤집는 것이다. 이렇게만 해도, Shell 프로그램은 Pipe 포함 command를 처리하고 나서 계속해서 정상

동작할 수 있다.

- **Phase3 (background process)**

  ✓ Background ('&') process를 구현한 부분

  Background Process를 생성하기 위해선, Background의 의미를 떠올려봐야 한다. Background란, Shell이 Process를 생성한 후, 해당 Process의 종료를 기다리지 않고, 다시 자신의 일을 하러 가는 것이다. 이런 의미를 고려해보면, Background Process를 생성하는 것은 어렵지 않다. 프로세스 생성 후, Phase 1과 2에서 Foreground Process 종료를 기다리던 것과 다르게, 기다리지 않으면 된다. 이는, 'cmdline'에 Ampersand 기호가 있는지를 'eval' 함수에서 확인 후, 이를 'bg'라는 플래그 변수에 기록한 다음, 'bg' 값이 TRUE인 경우, Wait Routine을 수행하지 않고, FALSE인 경우, Wait Routine을 수행하면 된다. (Pipe와 Background 상황이 함께 중첩된 경우, 각 Recursive Call에서 상기한 'Ampersand' 찾기를 반복하면 된다. Pipe 처리 시 'cmdline'을 쪼갤 때, 파이프 문자의 뒷부분은 그대로 유지하기 때문에, 'cmdline'의 가장 마지막에 붙는 '&'를 각 호출에서 계속 감지할 수 있다.)

  본 Phase에서의 핵심은 Background Process를 단순히 생성하는 일이 아니다. 본 Phase에서 구현해야 하는 Built-In-Commands인 'fg', 'bg', 'kill' 명령과, 키보드 상에서 'Ctrl+C', 'Ctrl+Z'가 눌리는 상황이 무작위 순서로 연속 수행될 때, Foreground Process는 Foreground Job의 속성을, Background Process는 Background Job의 속성을, Suspended Process는 Suspended Job의 속성을 그대로 유지하도록 제어하는 것이 본 Phase의 핵심이다. 각자 State에 따라 State의 속성을 유지하고, State가 상호 전환될 경우, 전환된 State의 속성을 가지도록 해야 하는 것이다.

  Background Process 구현은 내용이 많기에, 목록을 나누어 설명하겠다.

  1) Background Process Reaping (SIGCHLD Signal Handler)

  Background Process가 종료되는 경우는 크게 2가지가 있다. kill 명령을 통해 죽이거나, 아니면, Process가 정상적으로 수행되어 종료되는 경우이다. 두 경우 모두 Phase1 설명에서 소개했던 SIGCHLD Signal Handler로 처리할 수 있다. 차이점은, 정상 종료일 경우, Signal Handler 내에서 Dequeue를 수행하

고(Foreground Process도 마찬가지), Background Process kill 상황인 경우, kill Command 구현 함수 내에서 Dequeue를 수행하면 되는 것인데, 이때 Job Queue와 관련된 디테일은 바로 아래 항목에서 서술하겠다.

2) Job Queue Management

Phase1 설명에서 Job Queue를 소개한 바 있다. 수행 중인 Foreground Process가 정상적으로 종료되기 이전에 키보드에서 'Ctrl+C'가 눌려 강제로 종료된 상황을 SIGINT Signal Handler로 처리할 때, Job Queue가 사용된다고 설명하였다.

Phase3에서는 Job Queue가 위의 상황을 포함해서 프로그램 전반에 걸쳐 매우 중요한 역할을 담당한다. 'eval' 함수에서 Foreground Process 뿐만 아니라, Background Process가 생성되었을 때도 해당 프로세스를 Job Queue에 Push한다. 이는 기본적으로 본 Phase에서 요구하는 'jobs' Built-In-Command를 구현하기 위함이다. 수행 중인 Background Process와 Suspended Process를 화면에 출력할 때, Job Queue를 순회한다. 또한, SIGINT와 SIGTSTP Signal Handler에서 Foreground와 Background Process 속성을 구현할 때에도 Job Queue가 주요한 역할을 담당한다.

Job Queue는 Job이라는 구조체를 Type으로 하는 배열이다. Job 구조체에는 pid, index, State 등의 정보가 담긴다. Queue를 배열로 구현한 이유는, 연결리스트나 다른 자료구조로 구현할 경우, 동적 할당이 필수적인데, 동적 할당을 위한 'malloc' 함수가 'Async-Signal-Safety'가 보장되지 않기 때문에, 의도치 않은 Deadlock을 피하고자 배열을 선택하였다. (배열이 구현 측면에서 더 쉽기도 하며, 배열 크기를 크게 잡으면, 사실상 모든 상황을 Cover할 수 있기 때문에 문제가 없다.)

Job Queue 구현에 대한 상세한 코드 레벨 설명은 C 항목에서 소개하겠다.

3) Foreground Process 중지 (SIGTSTP Signal Handler)

Foreground Process는 수행 도중 키보드에서 'Ctrl+Z'가 눌리면 중단된다. 'Ctrl+Z' 타이핑은 SIGTSTP 시그널을 Shell 프로세스에게 보낸다. 따라서,

SIGTSTP Signal Handler를 설치한다. Handler에서는 SIGTSTP 시그널을 캐치한 후, Job Queue를 순회하여 Queue 내의 Valid한 Element들 중, State가 Foreground인 Element에 대해, Enqueue 시 기록된 pid를 조회하여, 해당 프로세스에게 SIGSTOP Signal을 보내고, 해당 Element의 State를 Stopped로 변경한다. 이를 통해, 오로지 Foreground Process만 'Ctrl+Z'에 반응하게 만드는 것이다.

한편, 프로그램의 안정성을 높이기 위해 Signal Handler의 시작과 끝에 'Sigprocmask'를 두어 Interrupting Signal을 Blocking한다. SIGTSTP Handler 뿐만 아니라 모든 Handler에서 이 작업이 필요하다.

4) Foreground Process 강제 종료 (SIGINT Signal Handler)

이 상황에 대해선, 앞서 Phase1에 대한 설명에서 간단히 소개한 바 있다. 키보드에서 'Ctrl+C'가 눌리면 SIGINT Signal이 Shell Process에 도달하고, 설치된 Handler가 이를 캐치한 다음, Job Queue를 순회해, Foreground State인 Element들에 대해서만 SIGKILL을 보낸다. 그 다음, 해당 Element를 Dequeue 한다. 이를 통해 오로지 Foreground Process만 'Ctrl+C'에 반응하게 만든다. 3)과 4) 항목을 통해, Background / Suspended Process는 SIGINT와 SIGTSTP에 대해 반응하지 않는 것이다.

5) 'eval' 함수에서 주의할 점

상기한 SIGTSTP Handler와 본 SIGINT Handler가 정확히 동작하기 위해선, 'eval' 함수에서 Child 프로세스 생성 시 SIGINT와 SIGTSTP를 Masking한 Signal Mask Set을 상속시키는 과정이 필수적이다. Handler를 통해 SIGINT와 SIGTSTP를 처리하므로, 이 두 시그널이 Child Process 자체에서는 처리하지 않도록 설정하는 것이다. 이는 'Sigprocmask'를 통해 수행할 수 있다.

이와 별개로, Background Process 생성 시엔 Race 문제를 피하기 위한 추가적인 'Sigprocmask' 함수 사용이 필요하다. 사실, 'eval' 함수 뿐만 아니라, 프로세스 State를 다루는 모든 코드 부분에는 위와 같은 Masking 과정이 필요하다. 자세한 코드-레벨 설명은 C 항목에서 이어간다.

6) fg, bg, kill Built-In-Command 구현

'fg'와 'bg' Command 구현은, 'eval' 함수의 프로세스 생성 루틴과 유사하게 구현할 수 있다. 우선, 입력 받은 pid에 해당하는 프로세스가 현재 Queue에서 Valid한지를 확인한다. Valid 프로세스인 경우, 해당 프로세스에 SIGCONT Signal을 보낸다. 차이점은, 'fg'의 경우, SIGCONT에 의해 다시 실행된 프로세스의 종료를 'Sigsuspend Wait Routine'을 이용해 기다리는 것이고, 'bg'의 경우 기다리지 않는 것이다. 한편, 두 명령 모두 Job Queue에서의 해당 프로세스 State를 'Foreground'나 'Background'로 Update한다.

kill 명령은, 단순히 SIGKILL을 해당 프로세스에 전송하고, Dequeue하기만 하면 된다. 강제 종료이기 때문이다. 'fg', 'bg', 'kill' Command 수행 과정에서의 '오류 입력' 처리와 'Possible Interrupting Signal Masking' 처리를 추가하면 본 Commands의 구현은 마무리된다.

이러한 내용들을 통해 Phase3의 Background Process 처리를 구현할 수 있다. 보다 더 자세한 디테일은 아래의 C 항목에서 이어간다.

C. 개발 방법

위에서 설명한 개발 내용을 코드 레벨에서 분석해보자. Phase3는 Phase1과 2를 모두 Cover하고 있으므로 Phase3 제출 소스 코드를 기준으로 분석하겠다. 단, 미리 제공된 <csapp.h>, <csapp.c>와 관련된 부분은 설명에서 제외한다.

(1) myshell.h - Job Queue 구현 부분

```
typedef enum { Invalid, Foreground, Background, Stopped }State;

typedef struct {                /* structure for Job Queue */
    int idx;                    // index of an element in the job queue
    pid_t pid;                  // pid of an element in the job queue
    State state;                // state of an element in the job queue
    char cmdline[MAXLINE];      // command of an element in the job queue
    int last_idx;               // is an element the last element of queue?
}Job;

Job queue[MAXJOBS];             // array based Job Queue
int queue_size;
int queue_last;                 // the index of the last element in the Job queue
```

- State라는 enum 자료형은 Job Queue의 Element Process State를 가독성있게 표현하기 위한 수단이다.

- Job 구조체 안에는 Queue에서의 Index, Process ID, Process State, Command Line, last_idx 정보가 담긴다.

- 이때, last_idx는 'Element가 Queue에서 가장 마지막에 위치한 Valid Element'인지를 나타낸다. Background나 Suspended Process가 Queue에 새로 추가될 때, Linux Shell과 같은 순서로 인덱싱을 하기 위해 이러한 플래그 변수가 도입되었다.

- queue_size는 Job Queue에서 Valid Element가 현재 몇 개 존재하는지를 추적하는 변수이며, queue_last는 가장 마지막 Valid Element의 Index를 추적하는 변수이다. 이들 모두 역시나 Linux Shell의 Job Queue Push 논리를 구현하기 위해 존재한다.

```
void init_queue(void) {                     /* Job queue initialization function */
    for (int i = 1; i < MAXJOBS; i++) {
        queue[i].idx = 0;
        queue[i].pid = 0;
        queue[i].state = Invalid;
        queue[i].cmdline[0] = '\0';
        queue[i].last_idx = FALSE;
    }
}

void Enqueue(pid_t pid, State state, char *cmdline) {   /* Job queue push function */
    int i, k = 1;

    if (queue_size == 0) {              // if job queue is empty, then push to the front
        i = 1;
        queue_last = i;
    }
    else {                              // if queue isn't empty, then push to next position
        i = ++queue_last;               // of the current last element
        if (i >= MAXJOBS)
            unix_error("Enqueue error");    // if queue is full, then abort with error msg

        queue[i - 1].last_idx = FALSE;      // update the former last element as non-last
    }

    queue[i].idx = i;
    queue[i].pid = pid;
    queue[i].state = state;

    queue[i].cmdline[0] = cmdline[0];
    for (int j = 1; cmdline[j]; j++) {                      // special string copy routine
        if ((cmdline[j] == ' ' && cmdline[j - 1] == ' ')    // ignore the consequtive spaces
            || cmdline[j] == '&' || cmdline[j] == '\n') continue;   // and ampersand symbol

        queue[i].cmdline[k++] = cmdline[j];
    }
    queue[i].cmdline[k] = '\0';

    queue[i].last_idx = TRUE;           // pushed element is now the last element of queue
    queue_size++;
}
```

- init_queue : Job Queue를 초기화한다.

- Enqueue : Job Queue에 새 Element를 추가한다. 'eval' 함수에서 새 프로세스 생성 시 사용한다. Queue가 비어있을 땐 1번 인덱스부터 부여하고, Queue가 비어있지 않을 땐, queue_last 값에 해당하는 인덱스에 새 Element를 추가한다. Job Queue에 남아있는 가장 마지막 Valid Process 다음의 위치에 삽입한다. 이때, 새 Element 삽입 시, 기존의 Last Element가 더 이상 Last가 아님을 last_idx 플래그 조정으로 처리하고 있다.

한편, 공지사항 조건을 만족하기 위해 cmdline 복사 시 연속된 띄어쓰기와 '&' 문자는 제외하고 있음도 주목할만하다.

```
void Dequeue(pid_t pid) {                   /* Job queue pop function */
    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].state != Invalid && queue[i].pid == pid) {
            queue[i].idx = 0;
            queue[i].pid = 0;
            queue[i].state = Invalid;       // make the element as Invalid
            strcpy(queue[i].cmdline, "");

            if (queue[i].last_idx)          // if deleted element was the last element,
                queue_last--;               // then decrement the position of last element

            queue_size--;
            return;
        }
    }
}

void Update_queue(pid_t pid, State state) { /* Job queue update function */
    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].pid == pid) {
            queue[i].state = state;         // change the state of element
            return;
        }
    }
    unix_error("Update_queue error");
}

Job *Search_queue(pid_t pid) {              /* Job queue search function */
    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].pid == pid)
            return &(queue[i]);             // return the address of element
    }
    return NULL;
}
```

- Dequeue : 입력 받은 pid에 해당하는 프로세스를 Job Queue에서 제거한다. 만약 Target Job이 Last Element인 경우, queue_last 값을 조정한다.

- Update_queue : 입력 받은 pid에 해당하는 프로세스의 State를 변경한다.

- Search_queue : 입력 받은 pid에 해당하는 Element 그 자체를 반환한다.

(2) myshell.c – 전역 변수

```
/* Global Variables */
volatile sig_atomic_t PID;                  /* global variable for reaping routine */
int pipe_flag;                              /* is cmdline has at least one pipe? */
int read_flag;                              /* flag for sigint_handler */
```

- PID : Sigsuspend를 이용한 Explicit Reaping 시에 사용하는 변수이다. Corruption을 막기 위해 volatile 선언과, sig_atomic_t 선언이 필요하다.

- pipe_flag : 현재 입력 상황이 파이프라인 상황인지를 나타내는 플래그 변수이다.

- read_flag : SIGINT Signal Handler에서, 'Ctrl+C'가 cmdline 입력 이후에 타이핑되었는지를 나타내는 플래그 변수이다. 이 여부에 따라 출력이 다르다.

(3) myshell.c – 'Read-Evaluation Cycle'

```
int main(void) {
    char cmdline[MAXLINE];                  // command line

    init_queue();
    Signal(SIGTSTP, sigtstp_handler);       // Ctrl+Z Handling
    Signal(SIGCHLD, sigchld_handler);       // SIGCHLD Handling
    Signal(SIGINT, sigint_handler);         // Ctrl+C Handling

    while (1) {                             // Read-Eval Cycle
        read_flag = FALSE;

        Sio_puts("CSE4100-SP-P#1> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin)) exit(0);
        read_flag = TRUE;                   // flag for Ctrl+C Handling

        for (int i = 0; cmdline[i]; i++) {
            if (cmdline[i] == '&') {        // this routine is for identifying any input forms
                cmdline[i] = '\0';          // of ampersand. for example,   1) ls &    2) ls&
                strcat(cmdline, " &\n");    // if ampersand is attached, then space it!
                break;
            }
        }

        pipe_check(cmdline);                // is cmdline has at least one pipe?
        eval(cmdline, 0, 0);                // evaluate the command
    }

    return 0;
}
```

- 'MyShell' 프로그램에 SIGINT, SIGTSTP, SIGCHLD Handler를 설치한다.

- cmdline 입력 후에, '&' 문자가 있으면 이를 ' &'로 변형하는 루틴이 있다. 이는 'ls&'와 같이, Ampersand가 Argument에 붙어 입력되는 경우를 처리하기 위함이다.

- 'eval' 함수 호출에 앞서 Pipe 상황을 체크한다. 이는 'eval' 내에서 pipe System Call을 호출할지 말지를 결정한다.

```
void eval(
    char *cmdline,                          // command line read in the main procedure
    int passed_fd,                          // file descriptor passed from parent to child (for pipe)
    int cnt                                 // how many times eval func has been called (for pipe)
) {
    char *argv[MAXARGS];                        // arguments list for Execvp() func
    char buf[MAXLINE];                          // buffer holds the modified command line
    pid_t pid;                                  // process id of new generated process
    int bg, idx, fd[2], last_flag, temp_fd;     // these are important variables
    Job *temp_job;                              // for the implementation of eval function
    sigset_t mask_all, mask_one, mask_two, prev_one;

    if (pipe_flag) Pipe(fd);                    // if pipe situation, let's construct the pipeline
    last_flag = buffering(cmdline, buf, &idx);  // buff the cmdline until meeting | or \0 (for pipe)
    bg = parseline(buf, argv);                  // parse buffer and insert into argv List
    bg = bg_check(cmdline);                     // check if it's background situation one more time
    if (argv[0] == NULL) return;                // ignore empty lines

    Sigset_setting(&mask_all, &mask_one, &mask_two);    // setting for an explicit signal masking

    if (!builtin_command(argv)) {               // check if argv is the built-in-command such as cd
        if (bg) Sigprocmask(SIG_BLOCK, &mask_one, &prev_one);   // prevent the race problem when bg

        /* Child Process */
        if ((pid = Fork()) == 0) {
            Sigprocmask(SIG_SETMASK, &prev_one, NULL);
            Sigprocmask(SIG_BLOCK, &mask_two, &prev_one);   // block SIGINT/TSTP for child process
                                                            // these will be handled via handlers!
            if (!last_flag) {
                Close(fd[0]);
                Dup2(passed_fd, STDIN_FILENO);          // file descriptor passing routine for when
                Dup2(fd[1], STDOUT_FILENO);             // there are multiple pipes in the command
            }
            else Dup2(passed_fd, STDIN_FILENO);         // if 'the last cmd of pipe' or 'non-pipe cmd'

            Execvp(argv[0], argv);
        }
        /* Parent Process */
        if (!bg) {  /* Foreground Process */
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_one);
            Enqueue(pid, Foreground, cmdline);      // push process into the Job queue, preparing for
                                                    // the situation that fg process has been stopped
            PID = 0;
            while (!PID) {                          // reaping chlid process with sigsuspend routine
                if (foreground_check())             // only reaping the foreground process, not stopped
                    Sigsuspend(&prev_one);
                else break;
            }
        }
        else {      /* Background Process */
            Sigprocmask(SIG_BLOCK, &mask_all, NULL);
            Enqueue(pid, Background, cmdline);      // push process into the Job queue
            temp_job = Search_queue(pid);

            if (cnt == 0) Sio_printBGjob(temp_job->idx, pid);   // print the background process
        }
```

```
        /* Common routine for Foreground and Background */
        if (!last_flag) {                           // file descriptor closing routine for when
            Close(fd[1]);                           // there are multiple pipes in the command
            Close(passed_fd);
        }
        else {                                      // if 'the last cmd of pipe' or 'non-pipe cmd'
            if (cnt == 0) temp_fd = 1;                  // if 'non-pipe cmd', then temp_fd is stdout
            else temp_fd = 0;                           // if 'the last cmd of pipe', then stdin

            Dup2(STDIN_FILENO, passed_fd);          // restore the input file descriptor
            Dup2(STDOUT_FILENO, temp_fd);           // restore the output file descriptor
        }
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);

        if (!last_flag)
            eval(cmdline + idx + 1, fd[0], cnt + 1);    // recursively call eval for pipelining
    }
    return;
}
```

- pipe_flag가 TRUE이면, 파이프라인을 형성한다.

- cmdline을 Buffering한 후, parseline 함수를 통해 Argument List를 만든다.

- bg_check는 Pipe와 Background가 함께 중첩된 상황에서, 파이프 사이의 각 명령을 모두 백그라운드로 수행하기 위해 매 Recursive Call마다 호출된다.

- Sigset_setting은 myshell.h에 정의한 함수로, mask_all은 모든 시그널을, mask_one은 SIGCHLD, SIGINT, SIGTSTP를, mask_two는 SIGINT와 SIGTSTP를 포함한다. Masking에 쓰이는 Set들을 구성하는 함수이다.

- bg가 TRUE일 때만, 즉, 백그라운드 상황에서만 mask_one을 이용한 Masking을 진행한다. 이는 Race를 방지하기 위한 명령이다.

- Fork 이후, mask_two로 Child Process를 Masking함으로써, 생성 프로세스가 SIGINT, SIGTSTP에 직접 영향을 받지 않도록 만들어준다. (Handler로 처리해야 하므로!)

- 파이프라인 상황인 경우, last_flag TRUE, 즉, 마지막 명령 조각 상황과, 그 외 중간 상황을 구분해 위의 코드와 같이 Dup2를 진행한다. 사용하지 않는 File Descriptor는 닫는다.

- 경로에 상관없이 명령 파일을 수행하기 위해 Execvp 함수를 사용한다.

- Parent로 넘어와서, bg가 FALSE, 즉, Foreground 상황인 경우, Enqueue 이후, 명시적인 Reaping Routine을 수행한다. 이때, foreground_check 함수를 통해, Running Foreground Process가 남아있을 때만 while문이 동작하도록 만들어준다.

- Background 상황인 경우, Enqueue 이후, 백그라운드 표시 출력만 한다.

- 이어서, B 항목에서 설명한 Parent Process File Descriptor 처리 루틴을 수행한다.

- last_flag가 FALSE, 즉, 파이프라인 상황에서, 마지막 명령이 아닌 경우, Recursive Call을 수행한다.

```
int buffering(char *cmdline, char *buf, int *idx) {
    int i = *idx, last_flag = TRUE, j;

    for (i = 0; cmdline[i]; i++) {
        if (cmdline[i] == '\'') {       // if meet ' or ", replace all spaces with temp value -1
            cmdline[i] = ' ';                                       // until re-meeting ' or "
            for (j = i + 1; cmdline[j] && cmdline[j] != '\''; j++) {
                if (cmdline[j] == ' ')
                    cmdline[j] = -1;
            }
            cmdline[j] = ' ';
        }                               // this routine is for handling some inputs like 'ab  c'
        if (cmdline[i] == '\"') {
            cmdline[i] = ' ';
            for (j = i + 1; cmdline[j] && cmdline[j] != '\"'; j++) {
                if (cmdline[j] == ' ')
                    cmdline[j] = -1;    // value -1 will be replaced with 'space' in parseline func
            }
            cmdline[j] = ' ';
        }
        if (cmdline[i] == '|') {                // if meet |(pipe), cut the cmdline! (for recursion)
            last_flag = FALSE;                  // meeting pipe means 'it's not the last cmd
            break;
        }
        buf[i] = cmdline[i];                    // buffering
    }
    buf[i] = '\0'; *idx = i;                // Record idx variable for Recursive call in eval func

    return last_flag;
}
```

- buffering 함수의 핵심은, 파이프 문자를 만나면 Recursion을 위해 Buffering을 멈춘다는 점과, ' a'나 "a b c"와 같이, 띄어쓰기를 포함한 Argument를 다루는 부분이다. 따옴표 내의 띄어쓰기를 -1이라는 비-ASCII 코드 Value로 임시 대체한다. 띄어쓰기 정보를 잃지 않기 위함이다.

```
int parseline(char *buf, char **argv) {
    char *delim;                            // points to first space delimiter
    int argc, bg;                           // number of args and background flag

    if (buf[strlen(buf) - 1] == ' ' ||
        buf[strlen(buf) - 1] == '\n')
        buf[strlen(buf) - 1] = ' ';         // replace trailing '\n' or ' ' with space
    else buf[strlen(buf)] = ' ';

    while (*buf && (*buf == ' '))           // ignore leading spaces
        buf++;

    argc = 0;
    while ((delim = strchr(buf, ' '))) {    // build the argv list
        argv[argc] = buf;
        *delim = '\0';
```

```
        for (int i = 0; argv[argc][i]; i++) {   // replace temp value -1
            if (argv[argc][i] == -1)                // with space! (see 'buffering' func)
                argv[argc][i] = ' ';
        }
        argc++;

        buf = delim + 1;
        while (*buf && (*buf == ' '))       // ignore spaces
            buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)                          // ignore blank line
        return 1;

    if ((bg = (*argv[argc - 1] == '&')) != 0) // should the job run in the background?
        argv[--argc] = NULL;

    return bg;
}
```

- parseline 함수는 cmdline에서 띄어쓰기를 기준으로 Argument List를 구성한다. 앞서 Buffering에서 대체한 임시 Value -1에 대해 다시 띄어쓰기로 복구하는 부분을 주목하자.

- 그 외의 parseline 구현 부분은 기본적인 문자열 처리 루틴이다.

```
/* Is cmdline has at least one pipe? */
void pipe_check(char *cmdline) {
    pipe_flag = FALSE;

    for (int i = 0; cmdline[i]; i++) {
        if (cmdline[i] == '|') {        // pipe found!
            pipe_flag = TRUE;
            break;
        }
    }
}

/* Is cmdline has ampersand symbol? */
int bg_check(char *cmdline) {
    for (int i = 0; cmdline[i]; i++) {
        if (cmdline[i] == '&')          // this routine is added just for implementing
            return TRUE;                    // phase 3! (not essential for phase 1 and 2)
    }
    return FALSE;
}

/* Check if any remaining foreground processes*/
int foreground_check(void) {
    for (int i = 1; i < MAXJOBS; i++) {     // in the job queue
        if (queue[i].state == Foreground)   // this func is for Wait routine of fg processes
            return TRUE;
    }
    return FALSE;
}
```

- 이미 언급한 바 있는, 간단한 체크 함수들이다.

(4) myshell.c – 'Built-In-Commands 구현 부분'

```c
int builtin_command(char **argv) {
    if (!strcmp(argv[0], "cd"))          /* command for moving location (cd) */
        return cd_command(argv);
    if (!strcmp(argv[0], "&"))           /* ignore singleton & */
        return TRUE;
    if (!strcmp(argv[0], "jobs"))        /* command for printing the current job queue */
        return jobs_command();
    if (!strcmp(argv[0], "bg"))          /* bg command */
        return fgbgkill_command(argv, 0);
    if (!strcmp(argv[0], "fg"))          /* fg command */
        return fgbgkill_command(argv, 1);
    if (!strcmp(argv[0], "kill"))        /* kill command */
        return fgbgkill_command(argv, 2);
    if (!strcmp(argv[0], "exit"))        /* shell termination command */
        exit(0);

    return FALSE;                        // not a builtin command
}
```

- 문자열 비교를 통해 Built-In-Command인지 확인한다. 아니면 FALSE를 반환한다.

```c
int fgbgkill_command(char **argv, int option) {
    int idx, pid;
    Job *temp_job;
    sigset_t mask_one, prev_one;

    if (argv[1] == NULL || argv[1][0] != '%') {     // incorrect usage handling: No param or jobspec
        if (option != 2) printf("No Such Job\n");
        else printf("Incorrect Kill Usage\n");
        return TRUE;
    }

    argv[1] = &(argv[1][0]) + 1;
    idx = atoi(argv[1]);
    if (idx < 1 || idx >= MAXJOBS) {    // incorrect usage handling: with jobspec, but incorrect idx
        printf("No Such Job\n");
        return TRUE;
    }
    pid = queue[idx].pid;
    if (queue[idx].state == Invalid) {  // incorrect usage handling: with jobspec, but invalid elem
        printf("No Such Job\n");
        return TRUE;
    }

    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);          // prevent the possible interruption by SIGCHLD signal
    Sigprocmask(SIG_BLOCK, &mask_one, &prev_one);

    if (option == 0) { /* bg command */
        Sio_printjob(idx, queue[idx].cmdline);      // print which job is switched to background
        Kill(pid, SIGCONT);                         // continue the process
        Update_queue(pid, Background);
    }
```

```
    else if (option == 1) { /* fg command */
        Sio_printjob(idx, queue[idx].cmdline);      // print which job is switched to foreground
        Kill(pid, SIGCONT);
        Update_queue(pid, Foreground);

        PID = 0;
        while (!PID) {                      // wait for the termination of foregrounded job
            if (foreground_check())         // only SIGINT or SIGTSTP can interrupt this situation
                Sigsuspend(&prev_one);
            else break;
        }
    }
    else { /* kill command */
        Kill(pid, SIGKILL);             // just kill the process and dequeue it
        Dequeue(pid);
    }
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);
    return TRUE;
}
```

- jobspec이 없는 등의 잘못된 Argument 입력에 대해 예외처리를 수행한다.

- B 항목에서 설명한 부분을 모두 그대로 구현하였다.

```
int jobs_command(void) {
    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].pid != 0) {
            printf("[%d] ", i);
            switch (queue[i].state) {       // print all background & suspended elems
            case Background:
                printf("running %s\n", queue[i].cmdline); break;
            case Stopped:
                printf("suspended %s\n", queue[i].cmdline); break;
            default: break;
            }
        }
    }
    return TRUE;
}

/* Procedure for 'cd' command */
int cd_command(char **argv) {
    if (argv[1] == NULL || !strcmp(argv[1], "~"))   // go to the home dir
        return chdir(getenv("HOME")) + 1;
    if (chdir(argv[1]) < 0)                         // go to the location that argv is pointing
        printf("No Such File or Directory\n");

    return TRUE;
}
```

- jobs Command의 경우, Background와 Stopped Process만 출력한다.

- cd Command의 경우, 'cd ~'이나, 'cd' 명령 입력 시 HOME으로 이동해야 한다.

(5) myshell.c – Signal Handling 부분

```c
void sigchld_handler(int sig) {
    int olderrno = errno, status;
    sigset_t mask_all, prev_all;

    Sigfillset(&mask_all);                  // mask any possible interrupting signals
    while ((PID = waitpid(-1, &status, WNOHANG)) > 0) {     // catch all terminated processes
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
                                            // if a process terminated normally, or SIGPIPE
        if (WIFEXITED(status) || status == SIGPIPE) // situation, then dequeue the job info!
            Dequeue(PID);                   // if not, don't dequeue because it's suspended!

        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }

    if (!((PID == 0) || (PID == -1 && errno == ECHILD)))
        Sio_error("waitpid error");
    errno = olderrno;
}

/* SIGINT Handler for 'Ctrl+C' situation */
void sigint_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev_one;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_one);

    if (!read_flag) Sio_puts("\nCSE4100-SP-P#1> "); // when just enter key pressed
    else Sio_puts("\n");    // read_flag means 'cmdline read' or 'during the evaluation'

    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].state == Foreground) {     // send SIGKILL "only" to the foreground,
            Kill(queue[i].pid, SIGKILL);            // not to the background or suspended
            Dequeue(queue[i].pid);
        }
    }
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);

    errno = olderrno;
}
```

- SIGCHLD Handler에서는, waitpid 함수를 while문의 조건 부분에서 호출함으로써, 잔존 종료 프로세스를 모두 캐치하고자 한다.

- WNOHANG 매크로 문자는 '모든 종료 프로세스'를 받아들이도록 만든다.

- WIFEXITED(status)와 status == SIGPIPE Check를 통해, 정상 종료 상황, 또는 백그라운드 파이프라인 상황에서의 비정상 출력 시에만 Dequeue가 이뤄지도록 한다. 그 외의 kill이나 'Ctrl+C' 상황에선, 각 구현 부분에서 따로 Dequeue를 진행한다.

- SIGINT Handler에서는 read_flag 값에 따라 출력을 달리하여 Linux Shell 느낌을 더 살리고자 하였다.

- Foreground Process만 SIGINT에 반응하도록 하면서 Job Queue를 순회하고 있다.

```
void sigtstp_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev_one;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_one);

    Sio_puts("\n");
    for (int i = 1; i < MAXJOBS; i++) {
        if (queue[i].state == Foreground) {     // send SIGSTOP "only" to the foreground,
            Kill(queue[i].pid, SIGSTOP);            // not to the background or suspended
            queue[i].state = Stopped;
        }
    }
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);

    errno = olderrno;
}
```

- 마찬가지로, Foreground Process만 SIGTSTP에 반응하도록 Job Queue를 순회한다.

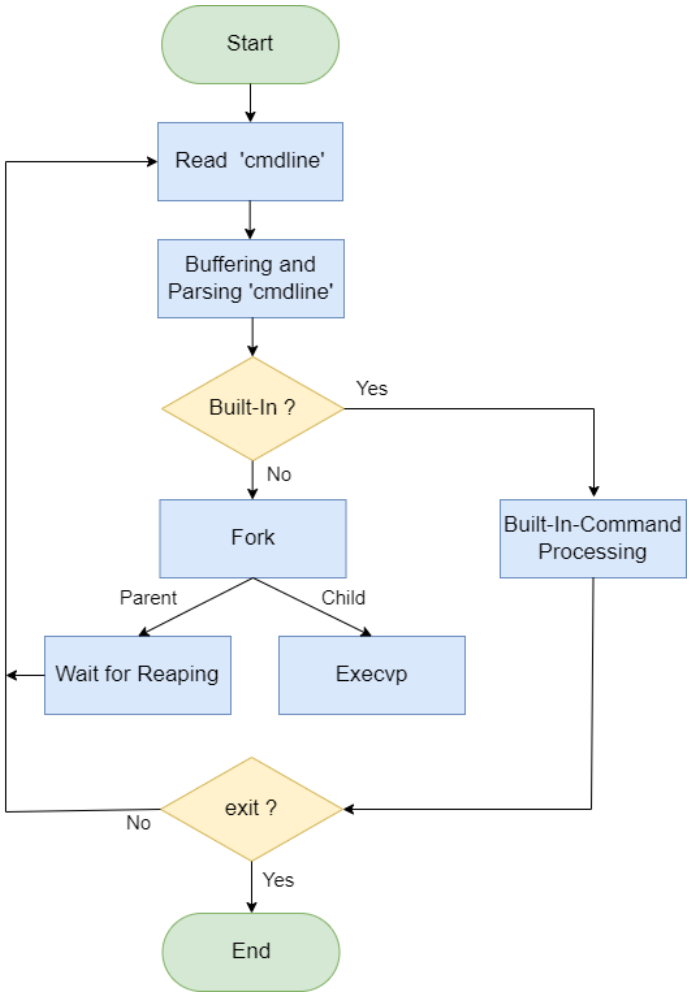
- 세 Handler 모두에서, Async-Signal-Safety 보장 함수만 사용한다. 또한, Interrupting Signal을 모두 임시 Blocking한다.

3. 구현 결과

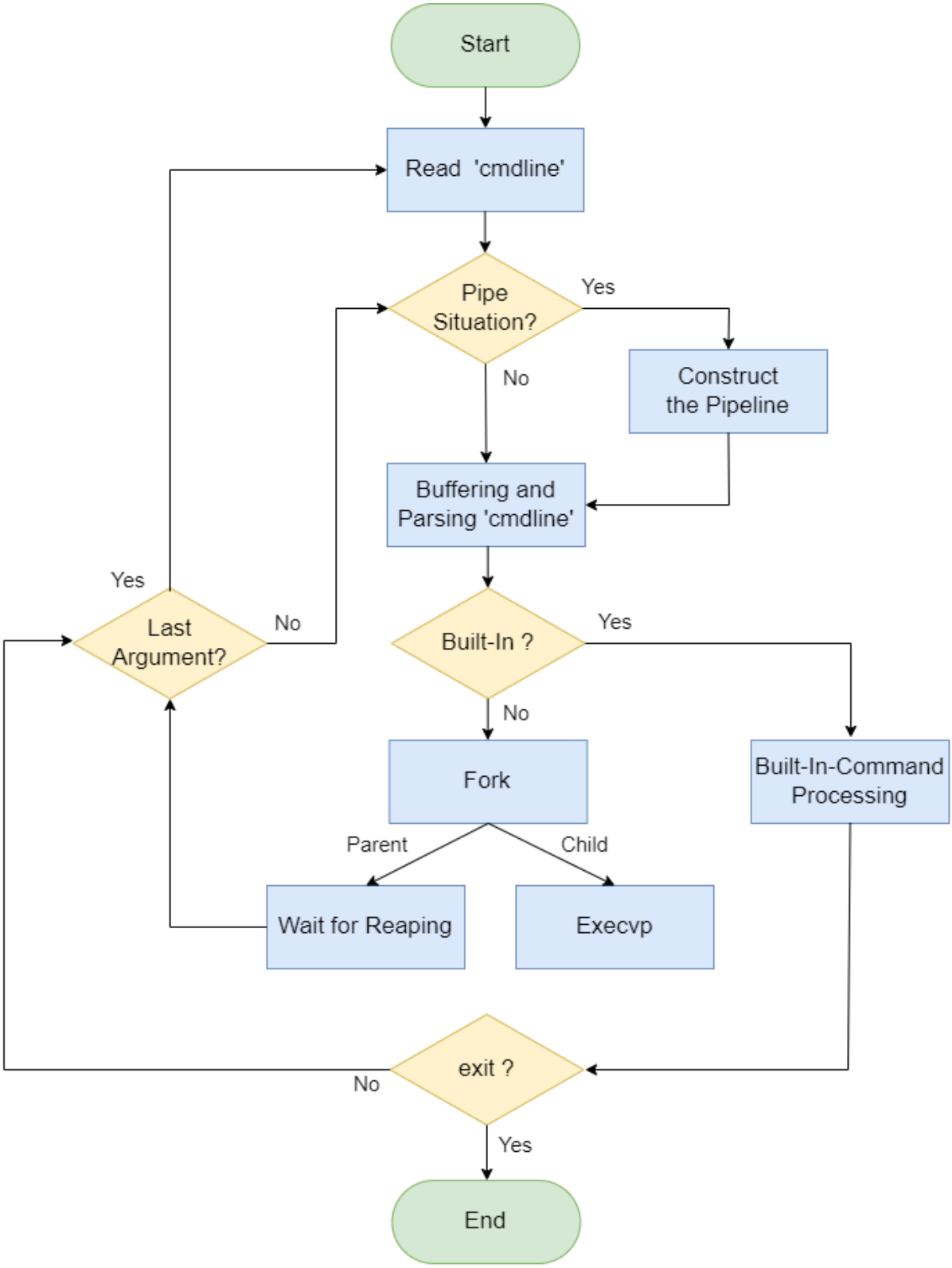
A. Flow Chart

1. Phase 1 (fork)

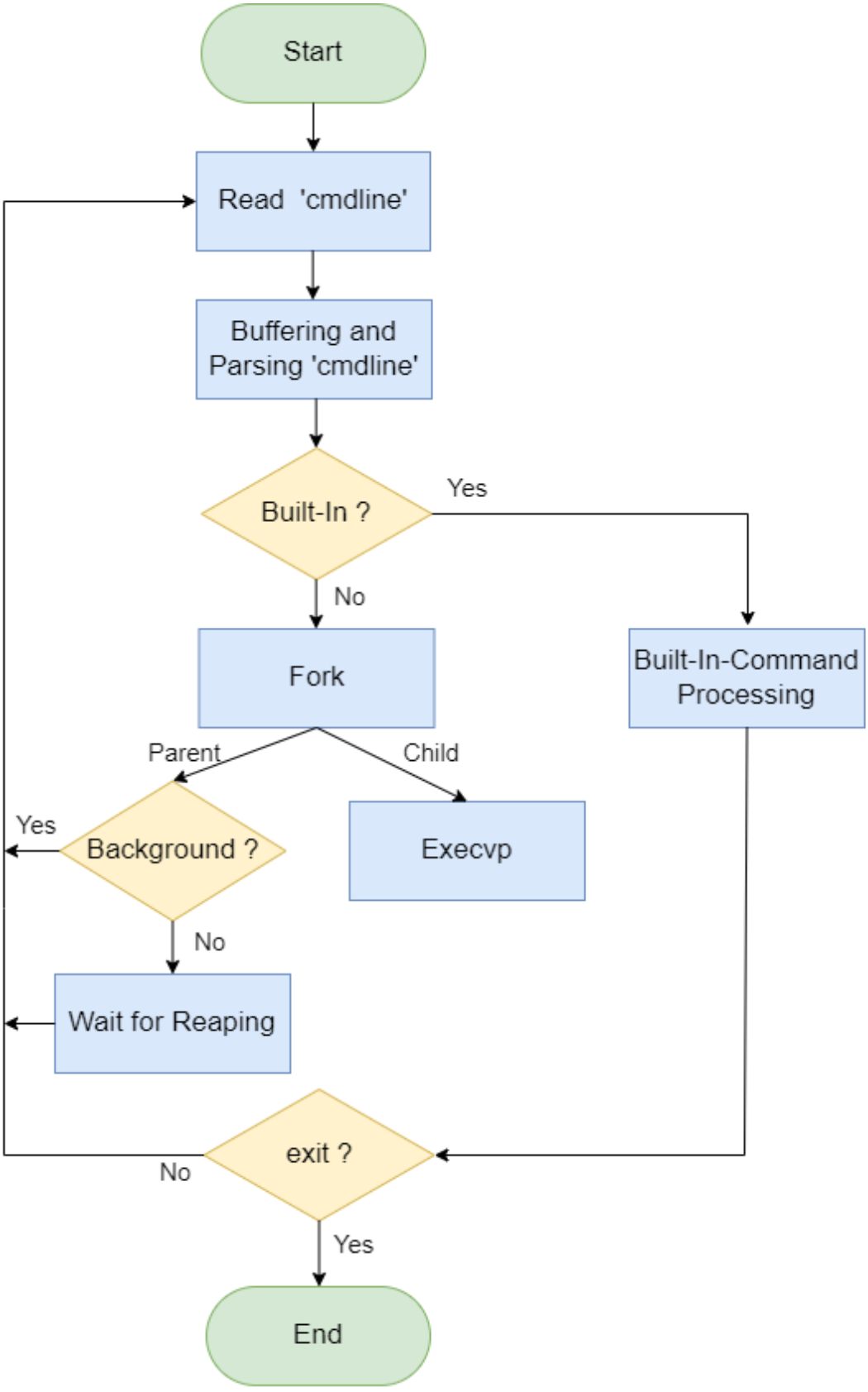
☞



2. Phase 2 (pipeline)



3. Phase 3 (background)



(백그라운드 프로세스의 속성을 보이기 위해 파이프라인은 생략)