Lab 4. A Unix Shell

SNU System Programming Assignment

Jongki Park
SNU TNET Lab.

What You Should Do

1. Implement basic shell functionality
   - Utilize parsed command line
   - External command execution

2. Support for redirection
   - Standard input redirection
   - Standard output redirection

3. Support for pipe
   - Single pipe support
   - Multiple pipe support

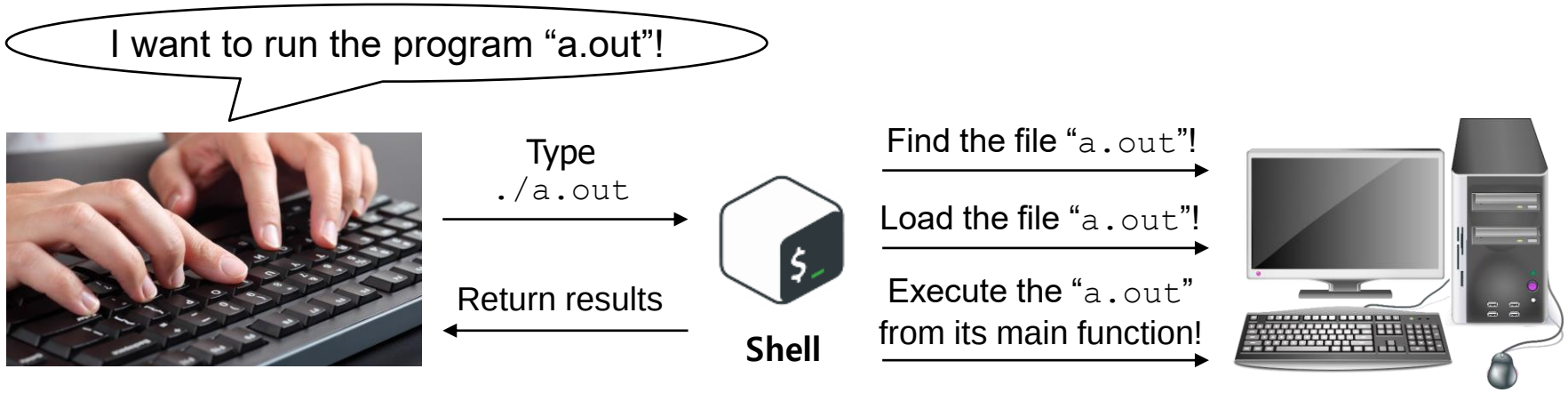
Extra Credit for Background Process Support

4. Support for background process
   - Running a process in the background mode
   - Add job(s) to a background job list
   - Handle the process when the background process terminates

Basic Shell Functions

What Is a Shell?

- Shell: a program that executes commands typed by a user



A shell =

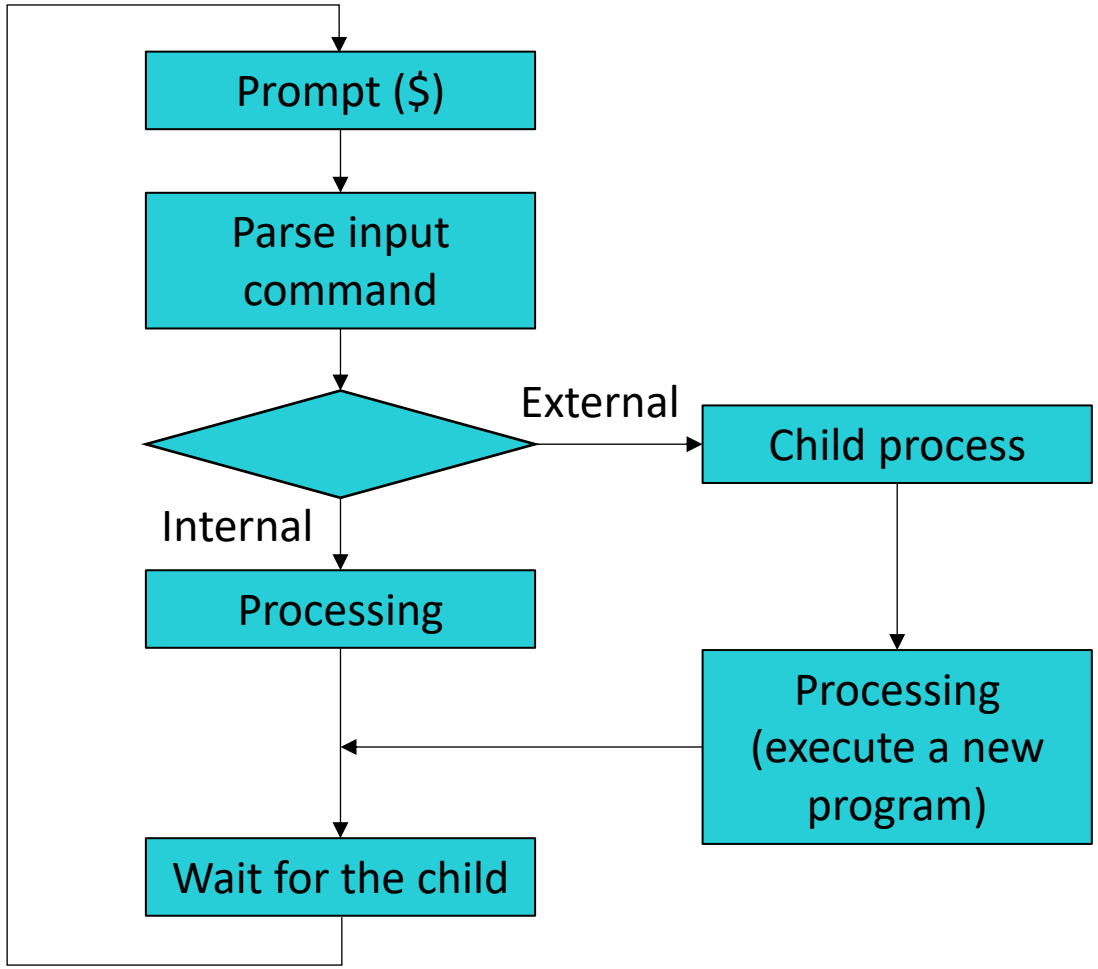
```
while(1) {
```

- Receives a input command string from the user
- Interprets the command
- Executes the command requested by user

```
}
```

Basic Shell Functionality

- Command interpreter
  - Execute commands requested by the user
- Basic logic
  - Display the prompt & parse input commands
  - External commands: fork and exec at child process
  - Internal commands: process without fork and exec
  - Parent process (Shell) waits for the child to end



System Calls for Basic Shell Functions

- `fork()` creates a child process that duplicates the parent
  - Manipulating fds after `fork()` does not affect the fds of the other process

- `execvp()` finds, loads, and executes an external command in current process
  - `Execvp()` automatically searches directories in `$PATH` (env. var.) for the executable binary
  - The first parameter is the binary name without any path ("`ls`" not "`/usr/bin/ls`")

- `waitpid()` is called by the parent process to reap a terminated child process
  - Parent blocks until a state of a child is changed to `EXITED`
  - `SIGCHLD` signal is sent to the parent when its child terminates
  - `SIGCHLD` signal handler of parent calls `waitpid()` to reap the terminated child

A Minimal Shell for Executing a Single Command

- Simple shell code
- Notice
  - Return value/error checking is ***mandatory*** when doing the assignment to ensure proper error handling
  - The code here is simplified just for easy understanding

```
fgets(command, sizeof(command), stdin);
command[strlen(command)-1] = 0;

token = strtok(command, " ");
if (token == NULL) {exit(-1);}
```

① **Get the command**

② **Parse(Tokenize) command**

```
arguments[0] = token;
for (i = 1; i < 10; i++) {
  token = strtok(NULL, " ");
  if (token == NULL)
    break;
  arguments[i] = token;
}
arguments[i] = NULL;
```

③ **Compose argument array**

```
pid = fork();
```

④ **Fork child process**

```
if (pid == 0){
  execvp(arguments[0], arguments);
}
```

⑤ **Child process: changes its execution image to arguments[0]**

```
/* Parent process */
pid = wait(NULL);
```

⑥ **Parent process : waits for exit of the child**

Your TO-DOs for snush: Basic Shell Functions

Functions for Basic Shell Functions

- No need to implement handling internal (built-in) commands
- Two built-in commands are provided
  - `cd` command
    - Moves to the specified directory
    - If no argument is provided, use the env. variable (`HOME`) to navigate to the home directory
  - `exit` command to exit the shell `snush`
- You need to implement `fork_exec()`
- Important system calls to utilize
  - `fork(), execvp(), waitpid()`
- Important `snush` functions to utilize ()
  - `build_command()`: a wrapper for `build_command_partial()`
    - Uses a full range of token indices
  - Do not modify `build_command()` & `build_command_partial()`

Implementing Basic Shell Functions

- Create a child process
- Convert the tokenized input into an argument array for an external command
- Do appropriate actions for supporting I/O redirection or a pipe
- Execute the command in the child process
- Make sure the parent process waits properly
- The return value of `fork_exec()` should be the child's `PID`

Support for Redirection

System Calls for Redirection

- Unix allows programmatic way to redirect `stdin, stdout or stderr`
- How?
  - Use `open()` or `create()` to open a file for reading or writing, creating the file
  - Use `close()` to close unused file descriptors
  - Use `dup2()` to duplicate the file descriptor to any target descriptor, including `stdin`, `stdout` or `stderr`
- For this assignment, support only `stdin/stdout` redirection (no `stderr`)

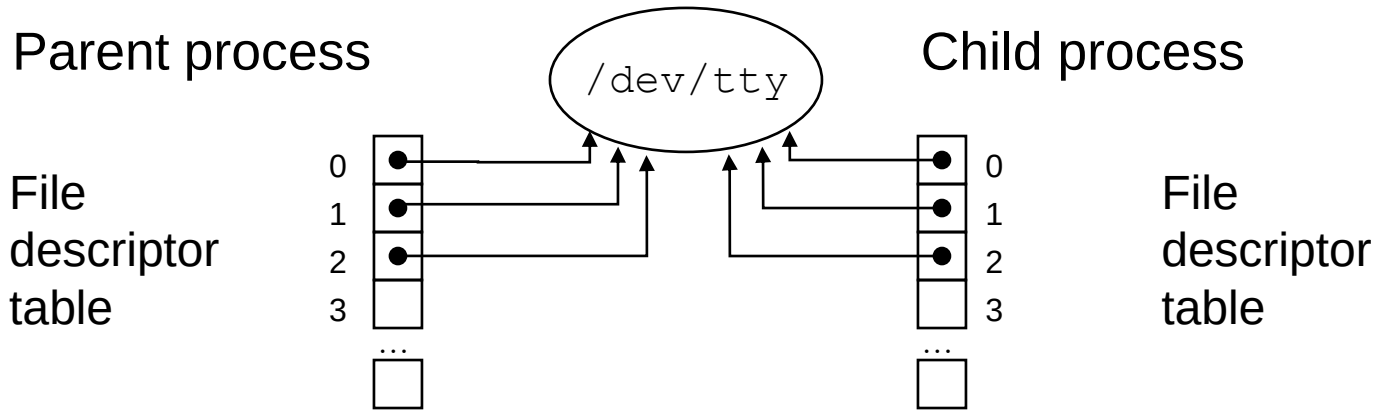
Simple Redirection Trace (1)

- Simple code for implementing "`someprogram > somefile`"
- A new child has the identical file descriptor table as parent's
  - The first three file descriptors of a child point to the same terminal as parent's

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", program_args);
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



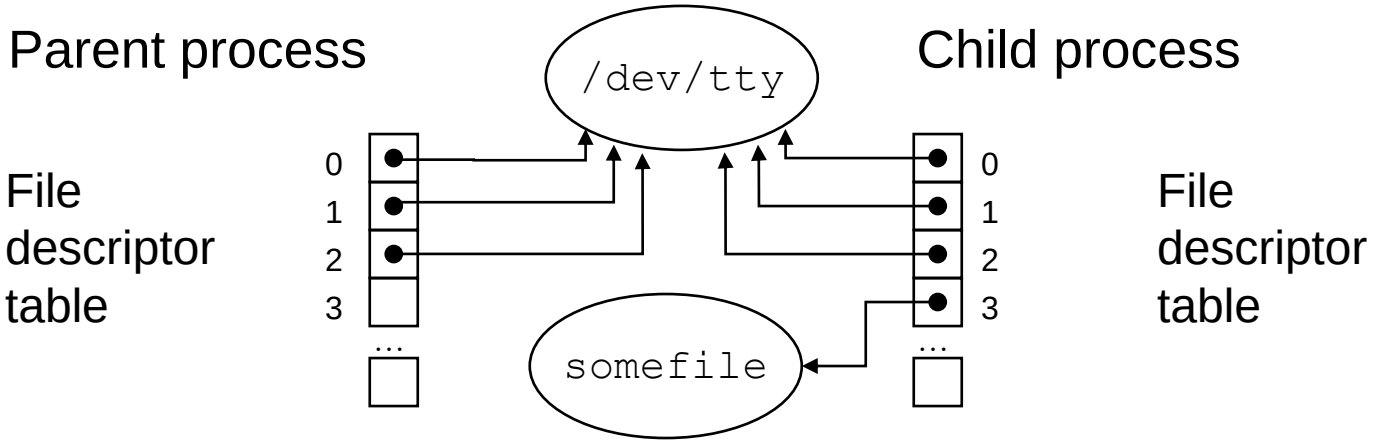
Simple Redirection Trace (2)

- Child process creates `somefile`

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", program_args);
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



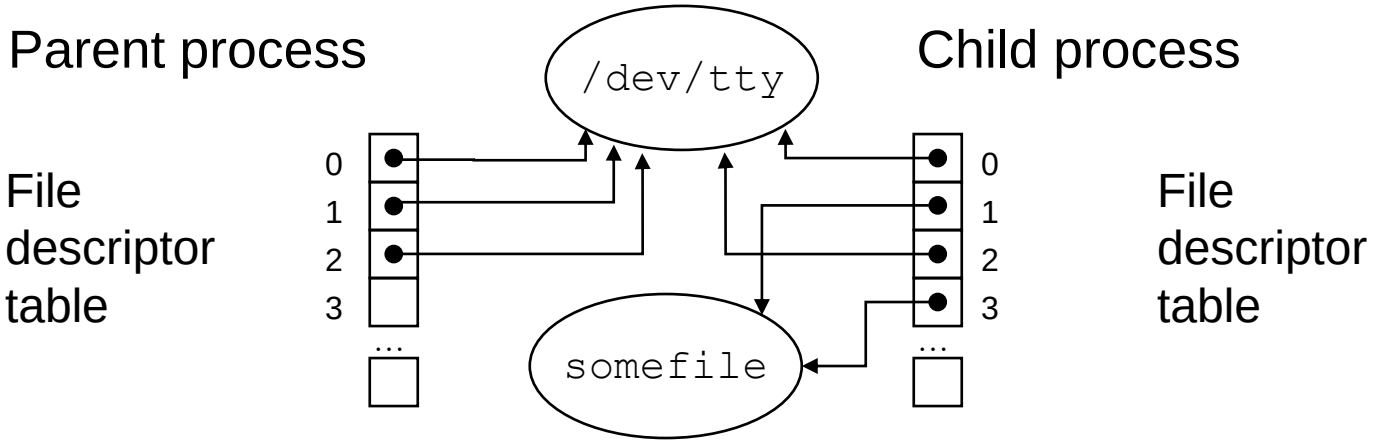
Simple Redirection Trace (3)

- `fd` and file descriptor number `1` can be used interchangeably after the successful call of `dup2()`

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", program_args);
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



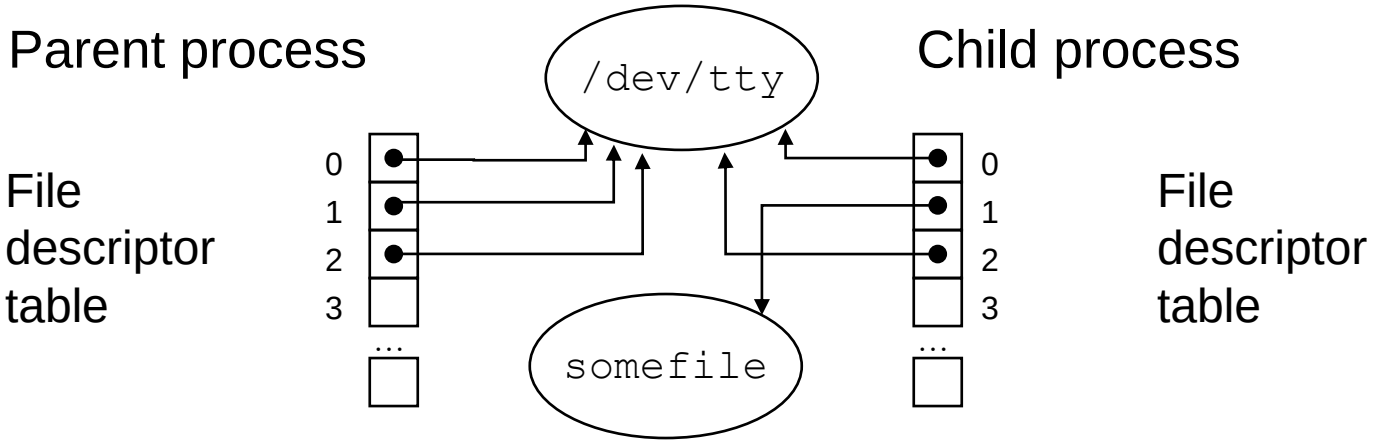
Simple Redirection Trace (4)

- Closes `fd` (index 3 at file descriptor table)

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", program_args);
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



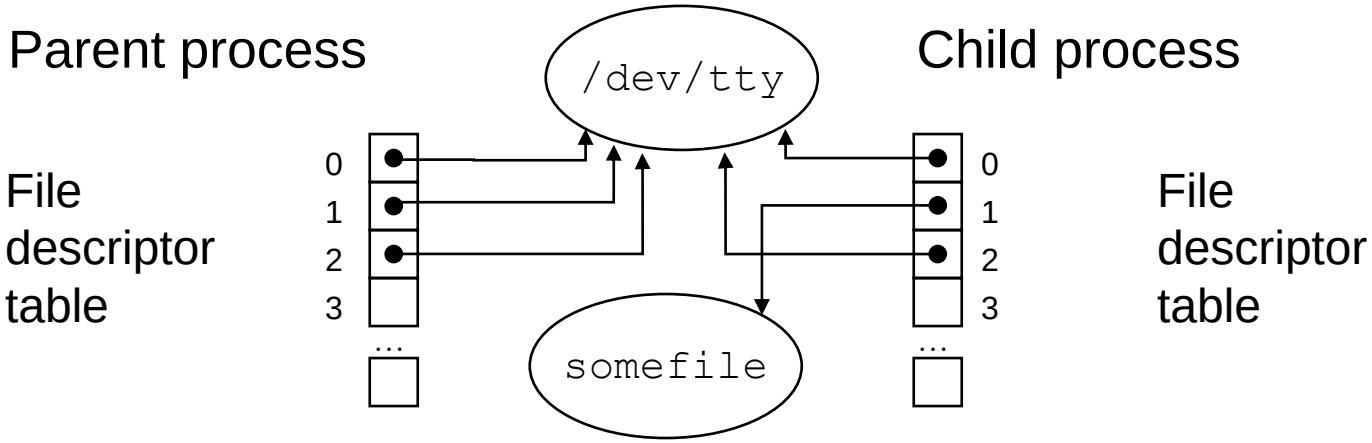
Simple Redirection Trace (5)

- someprogram executes with stdout redirected to somefile

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", program_args);
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



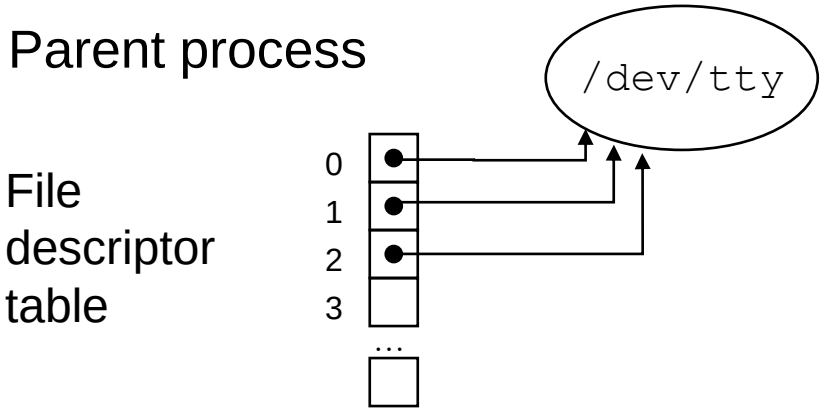
Simple Redirection Trace (6)

- `someprogram` exits
- Parent process returns from `wait()` and proceeds

```
pid = fork();

if (pid == 0){
  /* Child process */
  fd = creat("somefile", 0640);
  dup2(fd, 1);
  close(fd);
  execvp("someprogram", "program_args");
  fprintf(stderr, "exec failed\n");
  exit(EXIT_FAILURE);
}

/* Parent process */
pid = wait(NULL);
```



Your TO-DOs for snush: Redirection

Functions for Redirection

- You need to implement `redout_handler()`
- Sample implementation of `redin_handler()` is provided in the skeleton code
- Important system calls to utilize
  - `open()`
  - `close()`
  - `dup2()`

Implementing Redirection in snush

- Redirect its input and output to a specified file before the command is executed
- `stdin` redirection: opens a file (w/ read-only perm.), replaces `stdin` with the fd
- `stdout` redirection: replaces `stdout` with the redirected fd
  - Set appropriate permissions for the output file
- No need to support `stderr` redirection

Support for Pipe

System calls for Pipe

- A pipe allows two processes on the same machine to exchange data
  - `stdout` of the previous program flows into `stdin` of the next program
- How?
  - Use `pipe()` to create a pair of fds (2 fds) that are connected to each other
    - `pipe()` creates a unidirectional communication line (one for read-only, the other for write-only)
  - Use `fork()` to create a child process that shares the two fds with the parent
  - Use `dup2()` to duplicate any fd, including `stdin`, `stdout` or `stderr`
- Pipe descriptors are file descriptors in UNIX

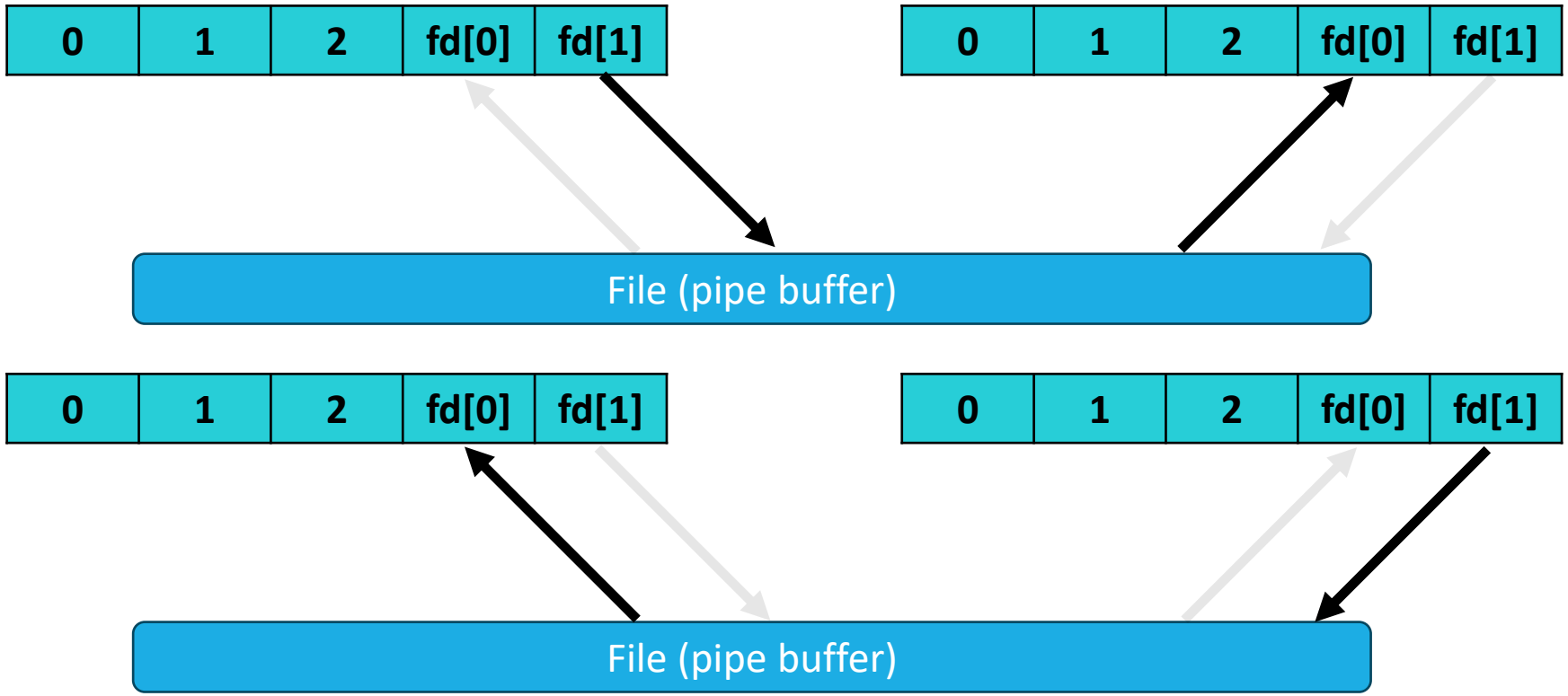
Example Use of Pipes

- The file "`student_ids.txt`" contains many student entries
- Find all students with the last name "Park"
  - `$ grep Park student_ids.txt`
- List the results in order of student id
  - `$ grep Park student_ids.txt | sort -n`

```
jongki@sp04:~$ cat student_ids.txt
202400001 Junghan Yoon
202400003 Seongjong Bae
202400009 Jongki Park
202400005 Juyoung Park
jongki@sp04:~$ grep Park student_ids.txt
202400009 Jongki Park
202400005 Juyoung Park
jongki@sp04:~$ grep Park student_ids.txt | sort -n
202400005 Juyoung Park
202400009 Jongki Park
jongki@sp04:~$
```

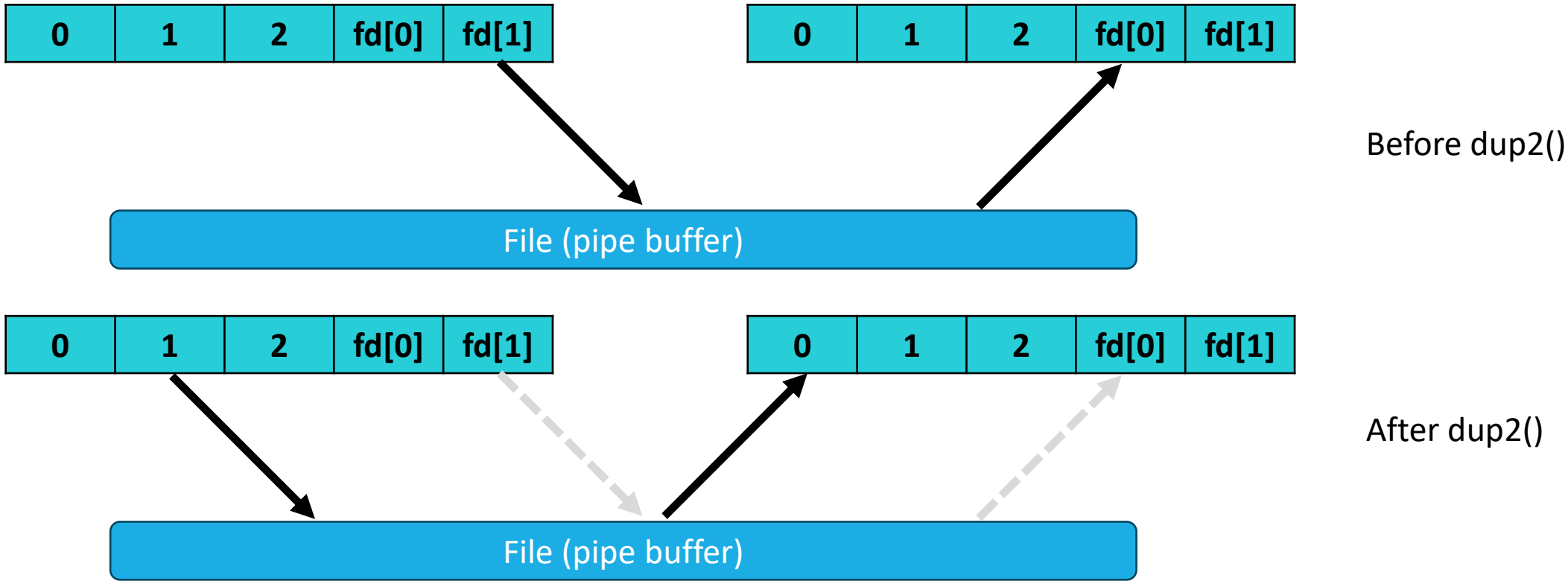
Unidirectional Communication

- `pipe()` creates a communication line (fd[0] for read-only, fd[1] for write-only)
- `pipe(), fork()`
  - Executing `fork()` after `pipe()` inherits the same file descriptors pointing to the same file



Duplicate to Other File Descriptors

- `dup2()`
  - Use `dup2()` to duplicate `stdin`, `stdout`, or other file descriptors to the file descriptors created by `pipe()`
  - The output of the process can be passed to the input of the next process



Simple Pipe example

- Example code
  - Inter Process Communication between the child and the parent

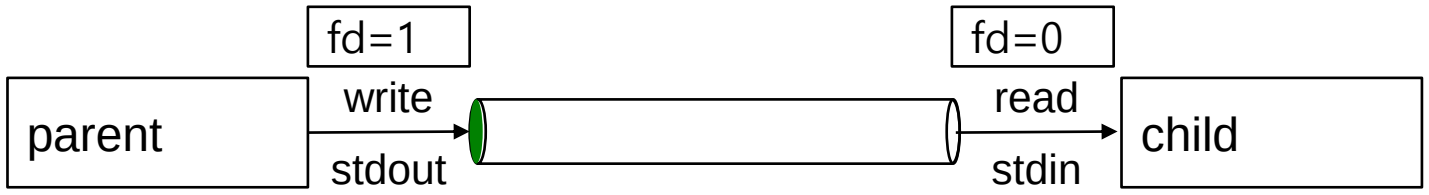
```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);

pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    … read from stdin …
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    … write to stdout …
    wait(&status);
}
```

① Create pipe read: fd[0], write: fd[1]

② Child makes stdin (0) the read side of the pipe

③ Parent makes stdout (1) the write side of the pipe



Your TO-DO for snush: Pipe

Functions for Pipe

- You need to implement `iter_pipe_fork_exec()`
- Important system calls to utilize
  - `fork()`
  - `dup2()`
  - `close()`
  - `execvp()`
  - `waitpid()`
- Important `snush` functions to utilize (Do not modify the functions listed below)
  - `dynarray_get()`: get tokens of a specific index, where index starts at 0
  - `dynarray_get_length()`: get total number of tokens
  - `build_command_partial()`: construct the command array passed as argument to `execvp()`

Implementing Pipe in snush

- Decompose the token array based on the pipe symbol ('|')
  - Note that the pipe symbol cannot be preceded by an output ('>') or input ('<') redirection operator
- Set up pipes
- Create child processes and run each command on each child
  - Create a child process for each command in the pipe
  - Convert the tokenized input to an argument array for external command execution
  - Execute a specified command at the child process
- The output of the previous process becomes the input of the next process
- Make sure the parent process waits properly
- `iter_pipe_fork_exec`() should return the `PID` of the first child

Support for Background Execution

Example for Background Process Execution

- Run tasks in the background and work on other commands simultaneously
- Useful for long-running processes (e.g., file downloads, backups, or large data processing)
- Child sends `SIGCHLD` signal to parent process when the child process exits

Simple Background Execution

- If parent process wants to wait for the child to finish, call `waitpid()`
- Otherwise, do not wait for the child to end its execution

```
pid = fork();

if (pid < 0) {
  perror("fork failed");
  exit(EXIT_FAILURE);
  }else if (pid == 0) {
    execl("/bin/sleep", "sleep", "20", NULL);
    // execlp("sleep", "sleep", "20", NULL);

    perror("execlp failed");
    exit(EXIT_FAILURE);
  }else {
    if (!background) {
      waitpid(pid, &cstatus, 0);
    }else {
       printf("Background: child process (PID: %d)\n", pid);
    }
    exit(EXIT_SUCCESS);
  }
```

① Create a child process using fork()

② If it is a child process,
change its execution image to another

③ If it is a parent process, decide whether to wait or not

④ If it is needs a background execution,
a parent process do not wait for the child to exit

Your TO-DO for snush: Background Execution

Functions for Background Execution

- You need to implement `sigzombie_handler()`
- Important system calls to utilize
  - `waitpid()`: set the third argument of `waitpid()`, the options field, to `WNOHANG` in `sigzombie_handler()`
- Remove the process ID from the list of background jobs
  - `bg_array[]`: global variable for the list of background jobs

Implementing Background Execution in snush

- Background processes are added to the background process array by the parent process
- The parent process does not wait for these background processes to complete
- Child sends `SIGCHLD` signal to parent process when the child process exits
- Signal handler function for `SIGCHLD` receives and removes each background process from the array upon their exit

Final Guidelines for snush

Input Command Line Usage Rules

- It is allowed to use symbols and commands without spaces in between
- The background execution symbol '&' cannot appear in the middle of the command line
- A redirection after a pipe is not valid
- There must be a target after a redirection symbol
- Nested redirections are valid only when input redirection '<' comes first, followed by output redirection '>'
- A pipe symbol cannot precede an input redirection symbol

Notice

- Do not change the name and the prototype of the skeleton code
- What to submit
  - Directory name should be `202400000_assign4`
  - Place the entire source codes in a single directory `202400000_assign4`
  - Make a gzipped tar file for submission

Deadline

- Deadline: ~ 2024. 11. 28 21:00
  - 0 Points if deadline is missed
  - 0 Points for copying
- Contact
  - Lab 4 TA e-mail: jkipark@snu.ac.kr
  - TA mailing list: snu-sysp@googlegroups.com

- Directory name should be `202400000_assign4`
- Place the entire source codes in a single directory `202400000_assign4`
- Make a gzipped tar file for submission

Test cases for self-checking

- All test cases for scoring will be provided, but scores may vary for each test case

- We provide test cases in the `self_check` directory:
  - Redirection input test
  - Redirection output test
  - Finding execution file test
  - Single pipe test
  - Multiple pipe test
  - Multiple pipe and output redirection test
  - Slow pipe test
  - Multiple slow pipe and output redirection test
  - Interrupt on single process test
  - Interrupt on multiple process test
  - Background execution test (Extra credit)

Appendix

snush Function for Parsing Input Command Line

- Take advantage of the command line parser that is already implemented
  - `lex_line()`: gets a command from stdin, parses, and store the array of token pointers
  - `syntax_check()`: checks if generated tokens are in an executable token format
  - Do not modify `lex_line()` and `syntax_check()`

- Do not modify these functions

The Result of Parsing in snush

- Set env. variable (`DEBUG`) before running your shell to see the tokenized output
  - `$ export DEBUG=1`
  - `$ unset DEBUG` (to turn it off)

- Example of how the lex_line() function works
  - `$ ls | sort > output &`

- Syntax check
  - Function `syntax_check()`
  - Performs an analysis to see if the tokenization with the `lex_line()` is in the executable form

```
% ls | sort >output &
[0] TOKEN_WORD("ls")
[1] TOKEN_PIPE(|)
[2] TOKEN_WORD("sort")
[3] TOKEN_REDIRECTION_OUT(>)
[4] TOKEN_WORD("output")
[5] TOKEN_BACKGROUND(&)
```

Token
Type : TOKEN_WORD
Value : ls

Token
Type : TOKEN_PIPE
Value : \0

Token
Type : TOKEN_WORD
Value : sort

Token
Type : TOKEN_REDIRECTION_OUT
Value : \0

Token
Type : TOKEN_WORD
Value : output

Token
Type : TOKEN_BG
Value : \0

snush Functions for Checking the Presence of Pipe or Background

- Pipe / background command check
  - `count_pipe()`: counts the number of pipes in the command
  - `check_bg()`: checks if background process control operator '`&`' exists

- Do not modify these functions

END