PA2 Part 2:
Extending the Shell

SKK UNIVERSITY

PA2

- **Goal**
  - Final goal of this assignment is to create a shell with redirection and pipe support.
  - For now, we will extend our shell by
    - » Pipe and redirection support
    - » Job control
      - » Using concepts like process groups, sessions, background processes, etc.
      - » Handling signals
    - » Adding other built-in commands (bg, fg, kill, cd, exit, and pwd)
  - Hint: https://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html (don't copy the code)

PA2

- **Supplement resources/guides (don't copy the code)**
  - https://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html
  - https://www.linusakesson.net/programming/tty/
    - » Focuses on tty, which we will not learn, instead of shell, but explains job control and signals very well.

Shell

SKK UNIVERSITY

Shell

- **Shell**
  - A shell reads user input, evaluates the input, and then return the result.
    - » When evaluating the input, the shell can first lex the input, and then parse the result to easily execute the commands.
  - For PA2, you will be implementing a shell that is heavily based on bash.
    - » Thus, you can use bash to test the behaviour of your shell but note that not all features in bash are found in your shell (e.g. variables, expansion, “~”, etc.).

Shell

- **Grammar**
  - The inputs passed to the shell follow this grammar:

```
input ::= pipeline ["&"];
pipeline ::= commands | pipeline "|" commands ;
commands ::= command |
        command redirection_operator pathname |
        command redirection_in_operator pathname redirection_out_operator pathname ;
redirection_operator ::= redirection_in_operator | redirection_out_operator ;
redirection_in_operator ::= "<" ;
redirection_out_operator ::= ">" | ">>" ;
command ::= (executable | builtin) {args}  ;
args ::= { option | option_with_argument | argument } ;
executable ::= executable_in_path | prefixed_self_implemented_executable | path ;
prefixed_self_implemented_executable ::= "pa2_" , self_implemented_executable ;
self_implemented_executable ::= "head" | "tail" | "cat" | "cp" | "mv" | "rm" ;
builtin ::= "cd" | "pwd" | "exit" | "fg" | "bg" | "jobs";
path ::= ["./" | "../" | "/"] pathname ;
```

  - Grammar is in EBNF ([...] = optional, | = or, {...} = ...*)

Shell

- **Input**
  - User enters input up to 200 bytes.
  - This input can contain whitespace, but it is ignored.
  - Before being processed, this input can be lexed and parsed.
    - » An example function to lex/tokenize the input can be found in the skeleton code for week 8.
- **Unknown command**
  - If the shell encounters an unknown command, it should print “COMMAND: command not found”

Shell

- **Pipeline**
  - Set of concurrent processes called subprocesses that are connected through standard streams
    - » $ command1 | command2 | command3
      - » command1's output is command2's input
      - » command2's output is command3's input
    - » These processes are independent
      - » `$ echo 1 | cat | exit | echo 2 | cat` returns 2 and does not exit the shell. exit only exits the subprocess.
    - » Do not run these commands "sequentially"; first create a process for each command and wait for all of them to finish
  - Implemented using pipes
  - Associated with a job.

Shell

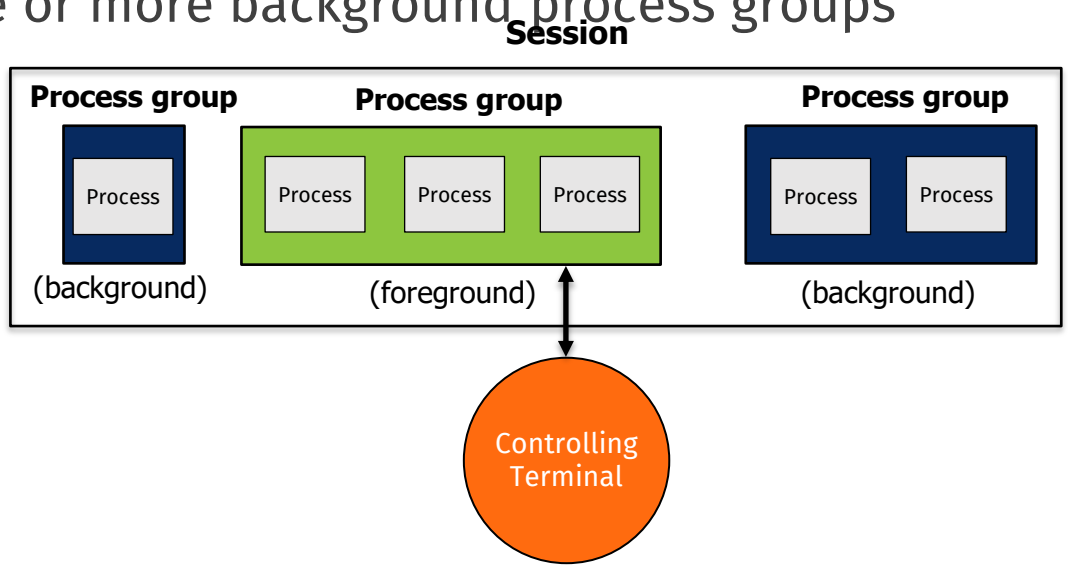
- **Redirection**
  - Input redirection (<)
    - » Replaces standard input of process with file in [pathname]
  - Output redirection (>, >>)
    - » Replaces standard output of process with file in [pathname]
    - » If redirected with >, the file is initialized and rewritten
    - » If redirected with >>, append from the end of file (if it exists)
  - If an error occurs due to the file, print “SHELL_NAME: pathname: strerr(errno)”

Job Control

SKK UNIVERSITY

Session (recall)

- **Collection of one or more process groups**
  - Session can have a single terminal
    - » This terminal is called a controlling terminal
  - Process group within a session can be divided into:
    - » A single foreground process group
    - » One or more background process groups



Jobs

- **Terminology**
  - Job: “A set of processes, comprising a shell pipeline, and any processes descended from it, that are all in the same process group.” - POSIX Issue 7
  - Job can be a background group or foreground group.
  - Job can have one or more concurrent process(es) via a pipeline (|).
  - A job table is used to keep track of all the jobs, which has a limit of 8192.
  - Note that this means that shell PGID != Job PGID.

Jobs

- **This is how bash holds processes and jobs in the actual code. (jobs.h). This is just to give you an idea on how it is done in actual applications.**

```
typedef struct process {
  struct process *next; /* Next process in the pipeline.  A circular chain. */
  pid_t pid;       /* Process ID. */
  WAIT status;     /* The status of this command as returned by wait. */
  int running;     /* Non-zero if this process is running. */
  char *command;   /* The particular program that is running. */
} PROCESS;

typedef struct job {
  char *wd;   /* The working directory at time of invocation. */
  PROCESS *pipe;   /* The pipeline of processes that make up this job. */
  pid_t pgrp;     /* The process ID of the process group (necessary). */
  JOB_STATE state; /* The state that this job is in. */
  int flags;      /* Flags word: J_NOTIFIED, J_FOREGROUND, or J_JOBCONTROL. */
#if defined (JOB_CONTROL)
  COMMAND *deferred;   /* Commands that will execute when this job is done. */
  sh_vptrfunc_t *j_cleanup; /* Cleanup function to call when job marked JDEAD */
  PTR_T cleanarg;   /* Argument passed to (*j_cleanup)() */
#endif /* JOB_CONTROL */
} JOB;
```

Jobs

- Taken from “The TTY demystified”

The following shell interactions...

```
Terminal
lft@shizuku:~$ cat
hello
hello
^Z
[1]+  Stopped                 cat
lft@shizuku:~$ ls | sort
```

...correspond to these processes...

Session 100	Session 101			
Job 100	Job 101	Job 102	Job 103	
XTerm (100)	bash (101)	cat (102)	ls (103)	sort (104)
stdin: -	stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: pipe0
stdout: -	stdout: /dev/pts/0	stdout: /dev/pts/0	stdout: pipe0	stdout: /dev/pts/0
stderr: -	stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0
PPID: ?	PPID: 100	PPID: 101	PPID: 101	PPID: 101
PGID: 100	PGID: 101	PGID: 102	PGID: 103	PGID: 103
SID: 100	SID: 101	SID: 101	SID: 101	SID: 101
TTY: -	TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0

Foreground processes

- **Synchronous commands**
  - Normal jobs that run in the foreground and control the terminal -> the shell no longer runs in the foreground.
  - $ grep aladdin Aladdin.txt | sort > sorted.txt
    - » This single job will run in the foreground.
    - » Each command in the pipeline is referred to as a subprocess.
    - » The first subprocess is the process group leader, but all subcommands are running simultaneously.
  - The exit status is the status of the last command.

Foreground processes

- *int* tcsetpgrp(*int* fd, *pid_t* pgrp);
  - Sets the given **pgid** as the foreground process group with the controlling terminal associated with **fd** (i.e. STDIN_FILENO)
  - Has to be called in both parent and child processes.
  - The shell has to regain the controlling terminal after the child has been terminated or stopped.
  - Returns **SIGTTOU** signal to the shell, which will stop it, so it has to be ignored by the shell.
  - A bit tricky to use --- an example code will be added soon

Background processes

- **& (asynchronous commands)**
  - & can be added to the end of a job to make run in the background (i.e. shell does not wait for the job to finish)
  - $ grep aladdin Aladdin.txt | sort > sorted.txt &
    - » This single job will run in the background.
  - It will print “[JOB ID] LAST_PROCESS_PID” (i.e. [1] 100)
  - Returns 0 as an exit status

Signals

- **Signals**
  - Recall that signals are sent to all processes in the process group.
  - If a job is currently running, the shell is running in the background, so it will not receive the signal.
  - However, it will still receive **SIGCHLD**, which means a child has been stopped or terminated.
    - » This signal can be used to handle (completed/stopped) jobs.
  - You can use the macros **WIFEXITED**, **WIFSIGNALED along with WTERMSIG**, and **WIFSTOPPED along with WSTOPSIG** to determine if the process has terminated without a signal, terminated due to a signal like SIGINT, or stopped due to a signal like SIGTSTP.

Signals

- **Signals**
  - A shell should not terminate or stop when these signals are received by the shell:
    - » SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, SIGTTOU.
  - However, the child processes should still receive these signals (SIG_DFL).

bg

- **Synopsis**
  - bg [JOB NUMBER...]
- **Description**
  - “bg sends jobs to the background, resuming them if they are stopped”
  - If JOB NUMBER is not passed, the most recent job would be put in the background.
  - JOB NUMBER format: %1, %2, %3, etc.
- **Error**
  - If there are no jobs that can be put in the background, print “SHELL_NAME: bg: current: no such job”
  - If the given job is invalid, print “SHELL_NAME: bg: JOB NUMBER: no such job” (still run other valid jobs in bg)

fg

- **Synopsis**
  - fg [JOB NUMBER]
- **Description**
  - fg sends a “job to the foreground, resuming them if it is stopped”
  - If JOB NUMBER is not passed, the most recent job would be put in the foreground.
  - JOB NUMBER format: %1, %2, %3, etc.
- **Error**
  - If there are no jobs that can be put in the foreground, print “SHELL_NAME: fg: current: no such job”
  - If the given job is invalid, print “SHELL_NAME: fg: JOB NUMBER: no such job”

jobs

- **Synopsis**
  - jobs
- **Description**
  - “jobs prints a list of the currently running jobs and their status”
  - Format for each job: [JOB ID]SYMBOL STATE COMMAND
    - » i.e. [1]+ Running sleep 10 &
  - SYMBOL:
    - » + -> Current/most recent job (i.e., if you call fg, this job will be called)
    - » - -> The second-most recent job
    - » NONE -> Other jobs

pwd

- **Synopsis**
  - pwd [OPTION]
- **Description**
  - Outputs the current working directory
  - -L prints logical directory (getenv(“PWD”))
  - -P prints physical directory (getcwd())
  - Prints the logical directory by default

pwd

- *char* *getcwd(*char* buf[.size], *size_t* size);
  - Gets working directory
  - Return value:
    - » Pointer to string on success. (should be buf)
    - » NULL with errno on failure
  - Size can be set to PATH_MAX (limits.h), which is usually set to 4096 bytes.

cd

- **Synopsis**
  - cd [DIRECTORY]
- **Description**
  - “cd changes the current working directory”
  - If DIRECTORY is not passed, change the directory to the home directory (i.e., getenv(“HOME”))
- **Errors**
  - When directory does not exist, “SHELLNAME: cd: DIRECTORY: No such file or directory”

cd

- *int* chdir(*const char* *path);
  - Changes working directory to path.
  - Return value: On success, 0. On error, -1 with errno.
  - Possible Errors:
    - » EACCES: not enough search (execute) permissions for one of the ancestors of path.
    - » ENOENT: directory does not exist
    - » ENOTDIR: ancestor in path is not a directory

exit

- **Synopsis**
  - exit [CODE]
- **Description**
  - exits the shell
  - Exit with CODE if given. If CODE >= 255, exit with 255.
  - Otherwise, exit with the exit status of the last command executed (usually 0, but not always).
- **Errors**
  - When code is not a number, `SHELL_NAME: CODE: numeric argument required` and exit with status of 2
  - When more than one argument is passed, print `SHELL NAME: exit: too many arguments`