PA2 Part 3: Supplemental Material

SK UNIVERSITY

Update

- **2024-05-03**
  - Added slides on how to use tcsetpgrp with setpgid
  - Added slide on an idea on how to implement the parser
  - Added slide on how to execute command
  - Added slide on how to deal with errors in a pipeline
  - Fixed detail in the pipeline slide
  - Decreased JOB_MAX from 2063244 to 100.
  - Added ARG_MAX, which is also 100.

Errata

- **Part 1 was updated.**
  - Made the error message format much easier to understand.
    - » “command: ERROR MESSAGE FROM strerror()” ->
      - » “command: %s” where the string is the return value of strerror().
  - Added _GNU_SOURCE part to basename().
  - Removed the duplicate slide for longopts
- **The “SHELL_NAME” in this PA should be “pa2_shell”.**
  - But the executable is still called pa2.

Parsing

Parsing

- **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, {...} = ...*)

Parsing

- **Structure**
  - Based on this grammar, we can create these:
    - » Job (Input)
      - » Foreground/Background
      - » Commands[N = 100]
    - » Command
      - » Command (argv[0])
      - » Arguments[N = 100]
      - » Redirection (or NULL)
      - » Built-in, self-implemented, in PATH, or pathname (., .., /)?
    - » Redirection
      - » <, >, >>
      - » pathname

Parsing

- **Using Job Structure**
  - There is only one job in one input line.
  - You can extend the previous input structure by adding pgid, job status (foreground, background, stopped), etc.
  - The maximum number of processes in a job is 100 (number from getrlimit(2))
  - You can keep track of all the job structures, which can be used for `jobs`.
    - » In bash, this structure is stored using a linked list, but you can use an array since the limit is 8192 and it is easy to manage job numbers with this
      - » Do note that foreground job does not have a job number

Parsing

- **Implementing parsing**
  - You can use the lex algorithm in *week8* to convert the input into tokens.
  - You do not have to use actual parsing algorithms/techniques, especially if you are not familiar with them.
    - » You can just go through each token since the grammar is simple.
    - » But you can still use RD/packrat if you are familiar with them.
  - There are two main structures (+ redirection, which can be merged with the command structure)
    - » Job: Has an array with the capacity of 100 commands
    - » Command: Has an array with the capacity of 100 arguments
  - So you only have to keep track of two indices: one for command and one for args
    - » And depending on the token, you can change the attribute of the structure
      - » i.e. If & exists at the end, the input is a background job.
      - » i.e., if parser encounters '<', you can set the redirection of the current command to <.

Executing

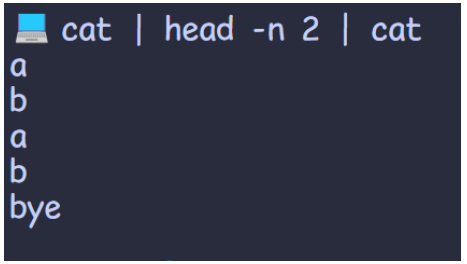
- **Executing the parsed expression.**
  - Now that you parsed the input, you can easily determine if pipe and redirection are needed.
  - You can now easily determine if the command is a built-in shell command, self-implemented executable, an executable in a given path, or an executable in $PATH (please note the difference between the last two!), so you can handle a command differently based on its type (if execvp is needed, path has to be modified, etc.)!
  - Assuming your pa2 is in ./bin, your self-implemented variables should be in ./bin, so you should call ./bin/pa2_EXECUTABLE, when pa2_EXECUTABLE is passed as a command.

Pipeline

SK UNIVERSITY

Pipeline

- **Pipeline**
  - command1 | command2 | command3
    - » command1's input is STDIN (of shell or redirected STDIN)
    - » command1's output is command2's input
    - » command2's output is command3's input
    - » Command3's output is STDOUT (of shell or redirected STDOUT)
  - All of these commands are running concurrently.
    - » If we have "cat | head –n 2 | cat", what will happen?
      - » First line: a -> goes to first cat
      - » Second line: b -> goes to second cat
      - » Head reads two lines and can safely quit
      - » Second cat prints the two line and exits
      - » First cat is still running since it is expecting input.
      - » After pressing enter, it finds out pipe has been closed and exits.



Pipeline

- **Pipe**
  - Short counts are very normal in pipes as seen in the previous example.
    - » When data is available in STDIN, programs will usually process that and then read STDIN again if applicable.
  - If we have `a | b`,
    - » What happens if `a` terminates while `b` is waiting for input?
      - » `b` will receive an EOF in the pipe and then terminate as STDIN has essentially been closed.
    - » What happens if `b` terminates and a attempts to write to the pipe?
      - » `a` will expect an input (read function blocks!), attempt to write the input, but receive SIGPIPE as the pipe has been closed already.
      - » Example: `cat | exit`
        - » Although `cat` exits last, the exit code is the exit code of `exit`, not `cat`

Pipeline

- **Handling errors**
  - What happens if an error occur?
    - » The process will exit and usually print an error to stderr before exiting.
    - » The same process happens as the previous slide (i.e., if `a` terminates, then b terminates as well, but if `b` terminates, then `a` will still expect input)
    - » If the command does not exist, it should still print “COMMAND: command not found”

tcsetpgrp

SK UNIVERSITY

tcsetpgrp – basic example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

```c
int main() {
  pid_t pid;
  int shell_fd = STDIN_FILENO;

  switch (pid = fork()) {
    case -1:
      exit(1);
    case 0: {
      pid_t pgid = setsid();
      tcsetpgrp(shell_fd, pgid);
      execlp("ls", "ls", NULL);
      exit(0);
      break;
    }
    default: {
      pid_t child_pid = pid;
      pid_t child_pgid = getpgid(child_pid);
      pid = getpid();
      pid_t pgid = getpgid(pid);
      tcsetpgrp(shell_fd, child_pgid);
      waitpid(child_pid, NULL, 0);
      tcsetpgrp(shell_fd, pgid);
      puts("Works well!");
    }
  }
  exit(0);
  return 0;
}
```

- tcsetpgrp should be called in both parent and child
- Process should call tcsetpgrp when it wants to control the terminal
- Note: execlp is used here instead of execvp to reduce lines. You can use execvp.

tcsetpgrp – issue

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

```c
int main() {
 pid_t pid;
 int shell_fd = STDIN_FILENO;

 switch (pid = fork()) {
  case -1:
   exit(1);
  case 0: {
   pid_t pgid = setsid();
   tcsetpgrp(shell_fd, pgid);
   execlp("cat", "cat", NULL);
   exit(0);
   break;
  }
  default: {
   pid_t child_pid = pid;
   pid_t child_pgid = getpgid(child_pid);
   pid = getpid();
   pid_t pgid = getpgid(pid);
   tcsetpgrp(shell_fd, child_pgid);
   kill(child_pid, SIGTSTP);
   waitpid(child_pid, NULL, 0);
   tcsetpgrp(shell_fd, pgid);
   puts("Works well!");
  }
 }
 exit(0);

 return 0;
}
```

- This SIGTSTP signal will cause the process not to terminate.
- “Works well” will not be printed.

tcsetpgrp

- **Issue**
  - This example does not work well with SIGTSTP as waitpid does not wait for stopped processes
  - Solution?
    - » WUNTRACED flag in waitpid
      - » Stopped processes are considered as well.
    - » How to differentiate regular termination, SIGINT, SIGTSTP, and other signals? -> Use status from waitpid
      - » WIFEXITED(status): normal termination
      - » WIFSIGNALED(status): terminated due to signal
        - » WTERMSIG(status): returns signal number that caused the process to terminate
      - » WIFSTOPPED(status): stopped (not terminated)
        - » WSTOPSIG(status): returns signal number that caused the process to stop

tcsetpgrp – solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

```c
int main() {
  pid_t pid;
  int shell_fd = STDIN_FILENO;

  switch (pid = fork()) {
    case -1:
      exit(1);
    case 0: {
      pid_t pgid = setsid();
      tcsetpgrp(shell_fd, pgid);
      execlp("cat", "cat", NULL);
      exit(0);
      break;
    }
    default: {
      pid_t child_pid = pid;
      pid_t child_pgid = getpgid(child_pid);
      pid = getpid();
      pid_t pgid = getpgid(pid);
      tcsetpgrp(shell_fd, child_pgid);
      kill(child_pid, SIGTSTP);
      waitpid(child_pid, NULL, WUNTRACED);

      tcsetpgrp(shell_fd, pgid);
      puts("Works well!");
    }
  }
  exit(0);

  return 0;
}
```

- By adding WUNTRACED, “Works well” can now be printed.
- Now that this job has been stopped, we can set the status of this job to “Stopped” and we can move this job to the background or foreground.

tcsetpgrp

- **Moving job back to foreground**
  - Shell has to let go of the terminal and give the control to this job
  - After doing this, you have to pass SIGCONT to the progress group of the job
    - » Why progress group?
      - » So that all the subprocesses in the job (pipeline) can continue
  - And then the shell should wait for the process to finish and then retrieve the terminal afterwards

tcsetpgrp – continue

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

```c
int main() {
  pid_t pid;
  int shell_fd = STDIN_FILENO;

  switch (pid = fork()) {
    case -1:
      exit(1);
    case 0: {
      pid_t pgid = setsid();
      tcsetpgrp(shell_fd, pgid);
      execlp("cat", "cat", NULL);
      exit(0);
      break;
    }
    default: {
      pid_t child_pid = pid;
      pid_t child_pgid = getpgid(child_pid);
      pid = getpid();
      pid_t pgid = getpgid(pid);
      tcsetpgrp(shell_fd, child_pgid);
      kill(child_pid, SIGTSTP);
      waitpid(child_pid, NULL, WUNTRACED);
      tcsetpgrp(shell_fd, pgid);
      puts("Works well!");
      // Continue job
      tcsetpgrp(shell_fd, child_pgid);
      kill(child_pid, SIGCONT);
      waitpid(child_pid, NULL, WUNTRACED);
      tcsetpgrp(shell_fd, pgid);
    }
  }
  exit(0);

  return 0;
}
```

tcsetpgrp

- **However, this creates a new session**
  - We need to keep the process groups in the same session!
  - So we have to use setpgid(pid, pgid) instead of setsid()
    - » pid = child pid
    - » pgid = pid (if new job) or pgid (if stopped job)
    - » So somehow store the pid and pgid in a structure
    - » You have to call this in both the child and parent processes
  - But this introduces a new problem!

```
If  tcsetpgrp()  is called by a member of a background process group
in its session, and the calling process is not blocking or  ignoring
SIGTTOU,  a SIGTTOU signal is sent to all members of this background
process group.
```

tcsetpgrp – issue

```c
int main() {
 pid_t pid;
 int shell_fd = STDIN_FILENO;
 switch (pid = fork()) {
  case -1:
   exit(1);
  case 0: {
   pid_t pid = getpid();
   setpgid(pid, pid);  // or pgid if it exists
   puts("This will stop...");
   tcsetpgrp(shell_fd, pid);
   puts("Now I can call cat! Press Ctrl+D to exit!");
   execlp("cat", "cat", NULL);
   exit(0);
   break;
  }
  default: {
   pid_t child_pid = pid;
   pid_t child_pgid = pid;
   pid = getpid();
   pid_t pgid = getpgid(pid);
   tcsetpgrp(shell_fd, child_pgid);
   kill(child_pid, SIGTSTP);
   waitpid(child_pid, NULL, WUNTRACED);
   tcsetpgrp(shell_fd, pgid);
   // Continue again
   tcsetpgrp(shell_fd, child_pgid);
   puts("Parent was stopped");
   kill(child_pid, SIGCONT);
   waitpid(child_pid, NULL, WUNTRACED);
   puts("Parent will enter a stopped state again...");
   tcsetpgrp(shell_fd, pgid);
   puts("Bye!");
  }
 }
 exit(0);
 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

tcsetpgrp

- **Process will enter a stopped state twice**
  - tcsetpgrp will trigger a SIGTTOU signal if called by a background process group in the same session
    - » Did not happen before as the sessions were different
  - How to handle this?
    - » Ignore SIGTTOU completely in parent
    - » In child,
      - » ignore SIGTTOU before calling tcsetpgrp
      - » BUT, set it to the default behaviour after calling tcsetpgrp
    - » Keep in mind, you have to ignore other signals aside from SIGTTOU too in the parent while keeping the default behaviour in the child!
  - In this way, you can safely use tcsetpgrp with multiple process groups in the same session.