Ceng334

Homework 1 Recitation

Çağrı Utku Akpak

March, 2024

Outline

Introduction

- Implement EShell, an extended version of the standard shell
- Implement four main shell operations on it:
  - Pipeline
  - Sequential
  - Subshell
  - Parallel (Our extension)

Introduction

Objectives

- Learn to use process execution and file descriptors.
- Hands on knowledge with fork, exec, dup2, wait and similar functions.
- Gain understanding of programs are executed.

Background

Shell

- Command line interpreter
- Executes other programs
- Supports pipeline, redirection, batch execution, conditional, sequential and subshell operations
- We are only interested in pipeline, sequential, and subshell operations plus one new operation we defined

Background

Shell - Pipeline

- Commands are chained together
- Output of one command is sent to the next command
- All of the commands should be executed concurrently
- Format:

```
(command1) | (command2) | ... | (commandN)
```

Background

Shell - Pipeline

- Examples:

```
ps aux | grep python | wc -l
ls -1 /etc | tr /a-z/ /A-Z/
cat file.txt | grep search
```

Background

Shell - Sequential

- Commands are separated by a semicolon
- They are executed one after the other
- Next command must wait for the previous one to finish
- Format:

```
{command1} ; {command2} ; ... ; {commandN}
```

Background

Shell - Sequential

- Examples:

```
echo "Hello"; sleep 5; echo "World"
```

Background

Shell - Parallel

- Commands are separated by a comma
- They are executed concurrently
- Similar to pipe, however there is no connection between commands
- Format:

```
{command1} , {command2} , ... , {commandN}
```

Background

Shell - Parallel

- Examples:

```
echo "Hello", sleep 5, echo "World", sleep 10
```

Background

Shell - Subshell

- Commands are encapsulated in parenthesis
- They are executed in a child shell program
- The input and output of the entire subshell can be redirected
- Format:

```
\verb|(sequential or parallel commands)|
```

Background

Shell - Subshell

Examples:

-
  ```
  (echo "Hello"; sleep 3; echo "World")
  (echo "Hello", sleep 2, echo "World")
  ```

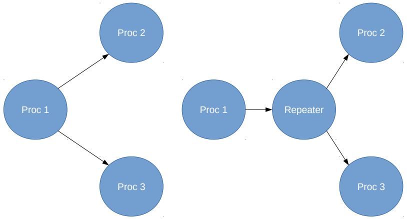
-
  ```
  (echo "Hello"; cat config.txt; cat manifest.txt |
  grep "Ship" | tr /a-z/ /A-Z/) |
  (cat | wc -l; echo "Finished")
  ```

  ```
  (echo "Hello"; cat config.txt; cat manifest.txt |
  grep "Ship" | tr /a-z/ /A-Z/) |(wc -l, wc -c)
  ```

Background

Shell - Subshell

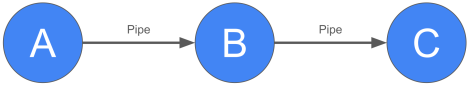
- When there is a redirection to a parallel subshell, all commands should receive the input
- This means there is a need to repeat the input
- Here is a simple graph:



Background

Shell Examples

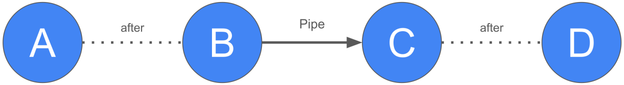
A | B | C



Background

Shell Examples

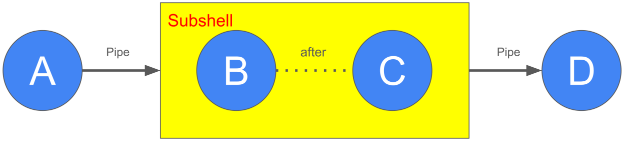
A ; B | C ; D



Background

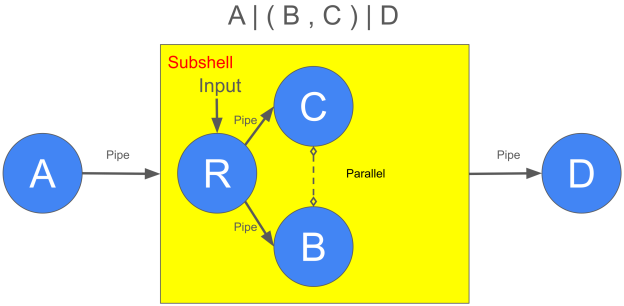
Shell Examples

A | (B ; C) | D



Background

Shell Examples



Extended Shell

- Same as a regular shell with some limitations and one addition
- Parallel execution is added
- Subshells can only be piped to one another
- No nested subshell
- Parallel and sequential execution cannot be mixed

Extended Shell I

- Format for waiting input:

  $/>$

- Executing a single command:

  ```
  /> command [args]*
  ```

- Format for pipeline:

  ```
  /> {command or subshell} | [{command or subshell} |]*
  {command or subshell}
  ```

  Examples:

  ```
  /> ls -l | tr /a-z/ /A-Z/
  /> cat input.txt | grep "log" | wc -c
  ```

Extended Shell II

- The format for sequential execution:

  ```
  /> {command or pipeline} ;[{command or pipeline} ;]*
  {command or pipeline}
  ```

  Examples:

  ```
  /> ls -l | tr /a-z/ /A-Z/ ; echo "Done."
  /> cat input.txt | grep "log" | wc -c ; ls -al /dev
  ```

- The format for parallel execution:

  ```
  /> {command or pipeline} ,[{command or pipeline} ,]*
  {command or pipeline}
  ```

  Examples:

  ```
  /> ls -l | tr /a-z/ /A-Z/ , echo "Done."
  /> cat input.txt | grep "log" | wc -c , ls -al /dev
  ```

Extended Shell III

- The format for subshell operations:

```
/> ({command or pipeline} {; or ,}
[{command or pipeline} {; or ,}]* {command or pipeline})
```

Examples:

```
/> (ls -l | tr /a-z/ /A-Z/ ; echo "Done.")
/> (ls -l | tr /a-z/ /A-Z/ , echo "Done.")
/> (ls -l | tr /a-z/ /A-Z/ , echo "Done.") |
(cat ; echo "Hello"; cat input.txt) |
cat | (wc -c , wc -l)
/> (cat input.txt | grep "c") |
(tr /a-z/ /A-Z/ ; ls -al /dev) |
(cat | wc -l , cat , grep "A")
```

Extended Shell IV

- The command `quit` terminate the program. Example:

```
/> quit
```

Extended Shell

Execution

- If there is a single command, fork a process and execute the commands.
- Afterwards, reap the child process.
- Otherwise, four main scenarios:
    - Pipeline Execution
    - Sequential Execution
    - Parallel Execution
    - Subshell Execution

Pipeline I

1. Assuming there are N number of command or subshells (CS) in a pipeline, create $N-1$ pipes (see pipe)
2. Fork the CS processes (see fork)
3. Redirect their output and input to the correct pipes (see dup2)
4. Execute the CS (for commands: see exec family of functions/ see slide 34 for subshells)
5. Reap all of the CS that have finished executing(see wait family of functions) to prevent zombie processes. This also assumes subshell does the same for all its children (see slide 34).

Pipeline II

```
A | B | C
for (comm in A,B,C) {
pipe_i = pipe()
        fork
           dup(pipe_i-1, 0)  (if i>0)
           dup(pipe_i, 1)    (if i<n)
           exec(comm)
for (process in A,B,C)
    wait  (after forks complete)
```

Pipeline III

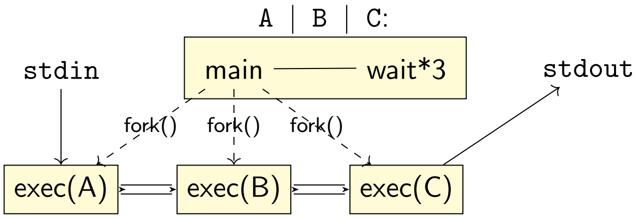


Figure: Pipeline Execution Example

Sequential I

1. For each PC in the list:
    1. If it is a pipeline, execute the pipeline (see slide 25)
    2. Else: Fork a process (see fork)
    3. Execute the command (for commands: see exec family of functions)
    4. Reap the finished process (see wait family of functions).

Sequential II

```
A ; B ; C
for (comm in A,B,C) {
    fork (child)
       exec(comm)
    wait() (Parent)  (synchronously, wait inside the loop)
}
```

Sequential III

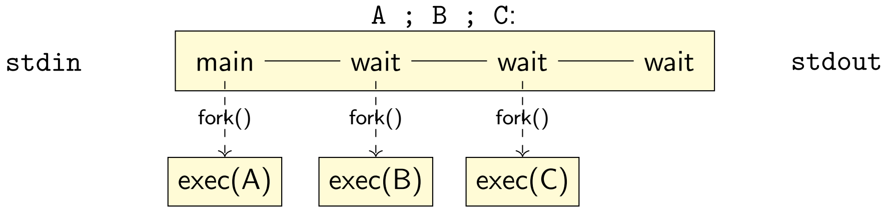


Figure: Sequential Execution Example

Parallel I

1. For each PC in the list:
    1. If it is a pipeline, execute the pipeline (see slide 25). Do not wait for the pipeline to finish.
    2. Else: Fork a process (see fork)
    3. Execute the command (for commands: see exec family of functions)
2. Reap all of the finished processes (see wait family of functions). This also includes the pipelines.

Parallel II

```
A , B , C
for (comm in A,B,C) {
    fork (child)
       exec(comm)
wait() (Parent)  (All of the children, wait outside of the loo
}
```

Parallel III

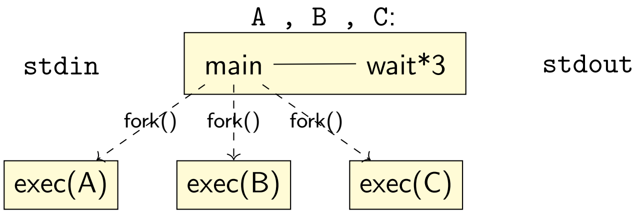


Figure: Parallel Execution Example

Subshell I

1. Fork a process for the subshell (see fork).
2. If it just contains a single command or pipeline, execute it (see slide 25).
3. If it is a sequential execution, complete it (see slide 28).
4. Else (it is a parallel execution):
5. Create N number of pipes for all pipelines or commands (PC) inside the subshell.
6. For each PC in the list:
    1. Redirect the input of the command or the first command in the pipeline to one of the pipes created in step 4.
    2. If it is a pipeline, execute the pipeline (see slide 25). Do not wait for the pipeline to finish.
    3. Else: Fork a process (see fork)
    4. Execute the command (for commands: see exec family of functions)

Subshell II

7. Fork a process for the repeater.
8. Repeater should work like the following:
    1. While until EOF is received from standard input or pipes are no longer open on the other side:
    2. Read stdin
    3. Write the read data to every pipe
    4. When EOF is received, close the pipes
9. Reap all of the finished processes (see wait family of functions). This also includes the pipelines.

```
(A ; B; C)
fork()
    call A;B;C handling code
wait (in subshell parent)
```

Subshell III

```
(A, B, C)
fork()
    for (comm in A,B,C)
       pipe_i = pipe()
       fork()
           dup(pipe_i, 0)
           exec(comm)
    fork()
        repeater code,
        for each line of stdin
           replicate to pipe_*[1]
for (process in A,B,C,repeater):
   wait  (after all forks finish, subshell parent)
```

Subshell IV

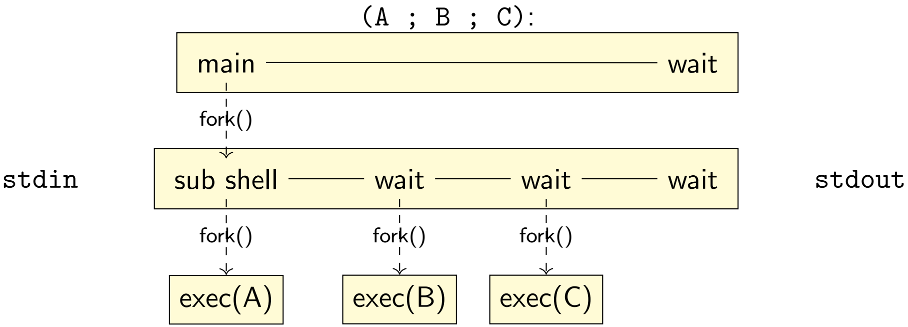
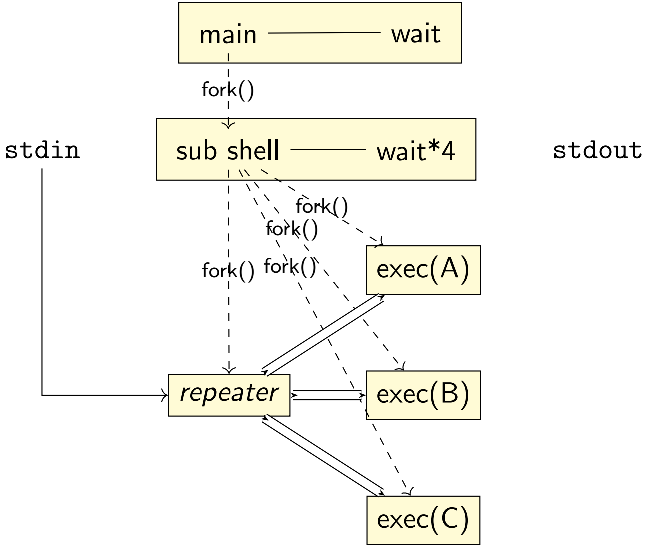


Figure: Subshell Sequential Execution Example

Subshell V

```
(A , B , C):
```

Subshell VI


main
wait
fork()
stdin
sub shell
wait*4
stdout
fork()
fork()
fork()
fork()
exec(A)
repeater
exec(B)
exec(C)

Subshell VII

Figure: Subshell Parallel Execution Example

Input

- Parser will be provided
- Commands are given in a single line
- Executable names may be given as full path or not
- All inputs will be proper. Non-existing executables will not be given. The `quit` command will not be used as an executable inside a subshell, pipeline, sequential, or parallel execution.

Parser I

The parser will fill a struct name parsed_input:

```
typedef enum {
    INPUT_TYPE_NON, INPUT_TYPE_SUBSHELL, INPUT_TYPE_COMMAND, I
} SINGLE_INPUT_TYPE;
typedef enum {
    SEPARATOR_NONE, SEPARATOR_PIPE, SEPARATOR_SEQ, SEPARATOR_P
} SEPARATOR;

typedef struct {
    char *args[MAX_ARGS]; // Null-terminated arguments
} command;
```

Parser II

```
typedef struct {
    command commands[MAX_INPUTS]; // Array of commands
    int num_commands;
} pipeline;

typedef union {
    char subshell[INPUT_BUFFER_SIZE]; // Entire subshell strin
    command cmd;                      // Single command
    pipeline pline;                   // Pipeline of commands
} single_input_union;

typedef struct {
    SINGLE_INPUT_TYPE type; // Type of the inputs
    single_input_union data; // Actual input which is union.
} single_input;
```

Parser III

```
typedef struct {
    single_input inputs[MAX_INPUTS]; // Array of inputs
    SEPARATOR separator; // Separators for the input
    int num_inputs; // Number of inputs
} parsed_input;
```

Parser IV

- The function that parses the input is called `parse_line`.
- There is `free_parsed_input` to clean allocated commands
- `pretty_print` is to nicely print the parsed structure

```
/***
 * Parses one input line and fills the parsed_input struct giv
 * It can handle any number of spaces between arguments and se
 * It has support for single or double-quoted commands and arg
 * It returns 1 if it is a valid input and 0 otherwise.
 * @param line
 * @param input
 * @return
 */
int parse_line(char *line, parsed_input *input);
```

Parser V

```
/***
 * Frees the allocated characters inside the inputs to prevent
 * It is recommended that you use this function after executi
 * @param input
 */
void free_parsed_input(parsed_input *input);
```

Parser VI

```
/***
 * Prints the contents of the parsed_input struct nicely for c
 * You should look at how different inputs are stored to under
 * Please do not forget to delete this before submission to pr
 * @param input
 */
void pretty_print(parsed_input *input);
```

Parser VII

These are important points for the `parsed_input`:

- There can only be one type of separator per line
- If the separator is a pipe, the inputs can only be subshells or commands indicated by the input type
- If the separator is a parallel or sequential separator, the inputs can only be pipelines or commands
- Subshells are recorded as complete strings
- Please do not modify the parser files, as they will be overwritten.

Specifications

- The eshell must terminate when the user gives the *quit* command. Otherwise, your homework will not be graded.
- All processes in a parallel execution or pipeline that are being executed should run in parallel. Therefore, the order of the outputs will be non-deterministic. In other words, executing the same commands may generate different outputs. Output order will not be important during evaluation.
- The eshell should reap all its child processes. It should not leave any zombie processes. This will lose points for each test case it happens in.
- When a subshell, sequential or parallel execution is happening, the eshell must wait for the termination of all the child processes before accepting new commands.
- If output is not redirected to a command or a subshell (via pipeline), it should use stdout as its output. Similarly, if its input is not redirected to another command or subshell (via pipeline), it should

Evaluation I

1. Single Command: 5 pts
   ```
   /> A
   ```
2. Single Pipeline: 10 pts
   ```
   /> A | B | C
   ```
3. Sequential Execution with commands: 5 pts
   ```
   /> A ; B ; C
   ```
4. Sequential Execution with mixed commands and pipelines: 15 pts
   ```
   /> A | B ; C | D | E ; F
   ```
5. Parallel Execution with commands: 5 pts
   ```
   /> A , B , C
   ```
6. Parallel Execution with mixed commands and pipelines: 15 pts
   ```
   /> A | B , C | D | E , F
   ```

Evaluation II

7. Subshell with a single pipeline or sequential or parallel execution: 5 pts
```
/> ( A | B | C | D )
/> ( A ; B ; C ; D )
/> ( A , B , C , D )
```
8. Subshell with mixed types(sequential-pipeline or parallel pipeline): 10 pts
```
/> ( A | B ; C | D | E ; F ; G | H )
/> ( A | B , C | D | E , F , G | H )
```
9. Subshell pipeline with mixed types(sequential-pipeline or parallel pipeline) without the need for repeaters: 15 pts
```
/> ( A | B , C ) | ( D | E ) | ( F ; G ; H | I )
```
10. Subshell pipeline with mixed types(sequential-pipeline or parallel pipeline) with repeaters necessary: 15 pts
```
/> ( A | B , C ) | ( D | E , F ) | ( G , H | I ) | J
```

Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with gcc or g++ on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your work. Any work used from third-party sources will be considered cheating and disciplinary action will be taken under the "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates daily.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

Submission

Submission will be done via ODTUClass. Create a tar.gz file named `hw1.tar.gz` that contains all your source code files with a makefile. The tar file should not contain any directories! The make should create an executable called `eshell`. Your code should be able to be executed using the following command sequence.

```
$ tar -xf hw1.tar.gz
$ make
$ ./eshell
```