lab one + discussion week two

ls, cat, and wc

- ls: list directory contents

```
ls    // Lists all files and directories in the present working directory
ls -a // Lists hidden files as well
ls -l // Lists files and directories with detailed information like permissions, size, owner, etc.
ls -R // Lists files in sub-directories as well
ls [path] // Lists files in the target directory
```

- cat: reads data from the files and gives their content as output

```
cat filename      // view a single file, it will show content of given filename
cat file1 file2   // view multiple files, the content of file1 and file2.
cat -n filename   // view contents of a file preceding with line numbers.
cat [filename-whose-contents-is-to-be-copied] > [destination-filename]
        // Copy the contents of one file to another file
```

- wc: find out number of lines, word count, byte and characters count in the files specified in the file arguments
- options
    - `-l`, --lines - Print the number of lines.
    - `-w`, --words - Print the number of words.
    - `-m`, --chars - Print the number of characters.
    - `-c`, --bytes - Print the number of bytes.
    - `-L`, --max-line-length - Print the length of the longest line.

piping

Name	Command	Description	Example
Redirect to a file	`> *a_file`	Take all output from standard out and place it into filename. Note using >> will append to the file, rather than overwrite it.	`ls > *a_file`
Read from a file	`< *a_file`	Copy all data from the file to the standard input of the program	`echo < *a_file`

Pipe	`program1 | program2`	Take everything from standard out of program 1 and pass it to standard input of program 2	`ls | more`

- if there is no pipe, where does a program dump output or take input from?
    - by default: connected to the terminal, 0 is to the standard inputer to the terminal (keyboard) and 1 is to the standard output from the terminal (console)
- what does `cat test_lab1.py | wc` do?
    - dumps the content of `test1_lab1.py` then counts the line/word/bytes of this dump

lab one overview

- three sub-components
    - read and execute the programs represented by the arguments
    - the execution ordering align with the arguments
    - create a pipe between two adjacent programs
        - connect `argv[i]` 's output to `argv[i+1]` 's input
- to check work
    - compare `program1 | program2 | program3` to `./pipe program1 program2 program3`

read in command line arguments

```
int main(int argc, char *argv[]) {
  return 0;
}
```

- `argc` number of arguments passed in
- `argv` array of actual argument strings
    - `argv[0]` is the program we are running, the rest are the programs we want to pipe together

creating a new process

execlp api

`int execlp(const char * file , const char * arg , ... /*(char*) NULL*/);`

- replaces the current process with a new process

- more convenient than execve, a wrapper function built on top
    - `file` path to new process
        - our executable file name
    - `arg` command line arguments of the new process, NULL terminated
        - `arg[0]` can be either the same as file, or different

fork api

`pid_t fork(void)`

- creates a new process (child process) by duplicating the calling process (parent process)
- child process has (almost) the same state as the parent process, including variables
- example

```
int i = 1, j = 2;
int res = fork();
// child: i = 1, j = 2, res = 0
// parent: i = 1, j = 2, res is pid > 0
```

- on success
    - return <0 means error
    - return 0 to the child process
    - return pid of the child process to the parent process
- distinguishing between child and parent process
    1. `==0` child process
    2. `>0` parents process
    3. `<0` fork() failed
- execution order matters, by default it is non-deterministic

execution ordering

wait api: waiting for process to change state

`wait(int *wstatus)`

- blocks the calling process until any one of its child processes exits or a signal is received
- after wait() finished, the calling process continues its execution

- child process may terminate due to any of these
    - it calls exit();
    - it returns (an int) from main
    - it receives a signal (from the os or another process) whose default action is to terminate
    - wait() returns the pid of the child that terminates

`pid_t waitpid(pid_t pid, int *wstatus, int options);`

- "…until *one of its children* terminates" -> "…until child with this pid terminates"
- `wait(&wstatus) = waitpid(-1, &wstatus, 0)`

macro information about status

- to find information about status, we use WIF… macros
    - `WIFEXITED(status)` child exited normally
    - `WEXITSTATUS(status)` return code when child exits
    - `WIFSIGNALED(status)` child exited because a signal was not caught
    - `WTERMSIG(status)` gives the number of the terminating signal
    - `WIFSTOPPED(status)` child is stopped
    - `WSTOPSIG(status)` gives the number of the stop signal

pipe between adjacent programs

file descriptors

- in unix/linux, everything is a file
- powerful abstraction, only have to consider how to interact with "files"
    - be it a storage device, printer, monitor screen, etc.
- when interacting with them, make system call to kernel to request access to them
- system calls return file descriptors to the user process as a handle (reference) to the underlying file
- when a user process needs to perform i/o to the files, it needs to pass the file descriptor to the kernel via a system call, the kernel will access the file on process's behalf

- standard file desciptors

Integer value	name	Symbol (<unistd.h>)
0	Standard input	STDIN_FILENO
1	Standard output	STDOUT_FILENO
2	Standard error	STDERR_FILENO

redirection

- `>` set a prgram’s stdout to be a specific fd of a file on the disk
- `<` set a program’s stdin to be a specific fd of a file on disk
- connect argv[i]’s output to argv[i+1]’s input
    - why it is not `argv[i]’s stdout fd = argv[i+1]’s stdin fd`
        - permission error of fds
        - programs across the pipe are not parallel
            - i.e. there is no real-time, async communication anyway
            - while waiting, the output has to be temporarily stored somewhere
    - need to create an in-memory buffer

pipe (|)

- in-memory buffer that connects two processes together
- instead of associating the fd for the standard output of the lhs program with an underlying device (i.e. terminal), the descriptor is pointed to an in-memory buffer provided by the kernel
- the other side of this same buffer is associated with the standard input of the rhs program, to consume the output of the lhs program

pipe system call

- pipe is for one-way communication only
    - use a pipe such that one process writes to the pipe, and the other process reads from the pipe
- the pipe can be used by either the parent or child process for reading and writing
    - one process can write to the pipe and another related process can read from it

- if a process tried to read before something is written to the pipe, the process is suspended until something is written

`int pipe(int fds[2]);`

- `fds[0]` will be the file descriptor for the read end of pipe.
- `fds[1]` will be the file descriptor for the write end of pipe.
- returns 0 on success, -1 on error.

```
#include <unistd.h>
int pipefd[2];    // initialize an integer array of size 2
int pipe(pipefd); // creates a pipe, the pipefd passed inside will be
                  // modified to store the read/write fd of the pipe
                  // pipefd[0] is the read end, pipefd[1] is the write end
```

- data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe
- call pipe before fork, if call it after then fork will create 2 pipes
    - if you call pipe in the parent process, so only 1 pipe, how the child is going to know what the pipefd[] is, it cannot set the corresponding fds

io- redirection with dup2

- both dup and dup2 can be used to redirecct a prev_fd to target_fd

```
#include <unistd.h>
int dup2(int target_fd, int prev_fd);
```

- `dup()` uses lowest-numbered unused file descriptor, uncontrollable.
- `dup2()` performs the same task as `dup()`, but it specifically redirects fd.
- if fd was previously open, it is *closed before being reused*; the close is performed silently (i.e., any errors during the close are not reported by `dup2()`).