COMP3511

Lab03: Linux Process and Pipe + Project 1 Introduction

Peter CHUNG (cspeter@cse.ust.hk)

Overview of Linux Process

Notes

- Linux System programming V.S. General C programming
  - General C/C++ programming is very different from Linux system programming
    - Example: `fork()` creates a new process in which you cannot easily write a normal C function to create a new process
- Students are recommended to type in the codes from scratch
  - Example codes are embedded in the slides
    - All sample code is short and easy to type
    - Try to type them, compile them, and then run the programs
    - Students need more practices
  - During the lab, TA may make a few changes to explain how the codes work

Overview: List of system calls used in this lab

- System calls (syscalls) provide an essential interface between a process and the operating system
  - x86_64 (64-bit) Linux kernel has 300+ syscalls
- In this lab, we mainly cover the following syscalls:
  - Process-related:
    - `getpid, getppid, sleep, fork, wait`
  - File-related:
    - `open, read, write, close, pipe, dup, dup2`
  - Execute-related:
    - `execl, execlp, execle, execv, execvp, execvpe`
- You don’t need to use all syscalls in the course project

Getting Process IDs: `getpid, getppid`

```
pid_t getpid(void);
pid_t getppid(void);
```

- Get the process ID of the current and the parent process

- Example:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   pid_t pid = getpid();
   pid_t ppid = getppid();

   printf("Current program PID: %d\n", pid);
   printf("Parent program PID: %d\n", ppid);

   return 0;
}
```

Temporarily suspend a process: `sleep`

```
unsigned int sleep(unsigned int seconds);
```

- `sleep`
  - sleep for a specified number of seconds
- Related system calls:
  - There is another system call, `usleep`, which can be used if the sleep time is between 0 to 1 second
  - `nanosleep` is available in the standard library (`time.h`) to provide even a finer control (in terms of nano second)

- Example

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("The program starts\n");
    sleep(1);
    printf("After 1 second\n");
    return 0;
}
```

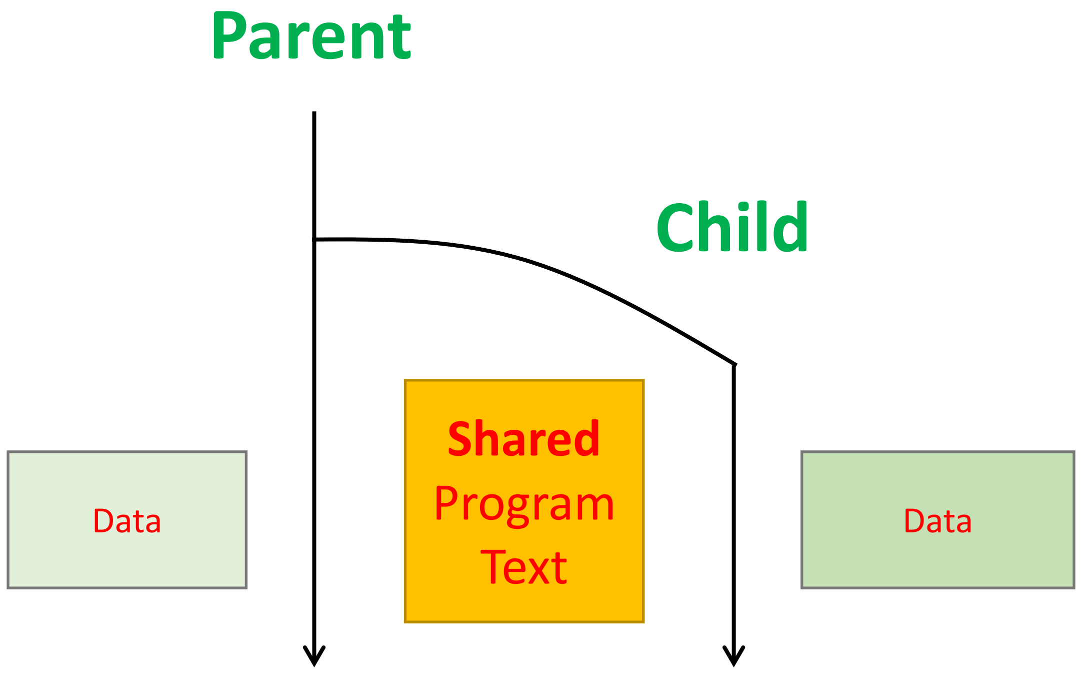
Creating a process: `fork`

```
pid_t fork(void);
```

- Create a child process
  - The child process is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- Returns
  - To child:
    - 0 on success
  - To parent:
    - Process ID of the child process
    - -1 on error, sets `errno`

$\Bigg\}$ The return PID is important for both parent and child processes

Creating a process: `fork`



In the parent:
`pid > 0`

In the child:
`pid == 0`

Waiting a child process: `wait`

```
// wait() will be used in this course
pid_t wait(int *status);

// waitpid() and waited() won’t be used in this course
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- All of them are used to wait for state changes in a child of the calling process.
- Status is the pointer pointing to an **`int`** where return status of the child is stored. You can input 0 (`NULL`) if no return status is needed.

Zombie and Orphan process

- What happens on termination?
  - When a process terminates, it still consumes some system resources (at least, for a short period of time)
  - Entries in various tables & information maintained by the operating system
  - Called a zombie process, waiting for the parent process to reap it
- What if parent does not reap the child process?
  - The child process becomes an orphan process
  - Sooner or later, the orphan process will be adopted and reaped by the `init` process (PID = 1), or be removed when the system shuts down or reboots
  - The situation usually happens when the parent process terminates before the child process

# means the Linux system shell prompt

Example: An orphan process

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

int main() {

   pid_t pid;
   int i;
   /* Create a new process */
   pid = fork();
   if ( pid > 0 ) {
      /* parent process */
      for (i=0; i<10; i++)
         printf("Parent %d\n",i);
      }
   else { /* child process */
      sleep(1);
      for (i=0; i<10; i++)
         printf("Child %d\n",i);
   }
   return 0;
}
```

```
# gcc -o fork_orphan_demo fork_orphan_demo.c
# ./fork_orphan_demo
Parent 0
Parent 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Parent 9
# Child 0
Child 1
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Child 9
```

The parent process terminated here
The control returns to the system shell program

The child process becomes an orphan process, and it won't be able to return to its parent (Press Control-C to terminate it)

# means the Linux system shell prompt

Example: Using `wait()` properly

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   pid_t pid;
   int i;

   /* Create a new process */
   pid = fork();

   if ( pid > 0 ) { /* parent process */
      for (i=0; i<10; i++)
         printf("Parent %d\n",i);
      /* Wait for the child process */
      wait(0);

   } else { /* child process */
      sleep(1);
      for (i=0; i<10; i++)
         printf("Child %d\n",i);
   }
   return 0;
}
```

invokes wait() here

```
# gcc -o fork_with_wait fork_with_wait.c
# ./fork_with_wait
Parent 0
Parent 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Parent 9
Child 0
Child 1
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Child 9
#
```

The parent process finished its print job and waited

The child process returned and the parent process terminated

# means the Linux system shell prompt

Example: A zombie process

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   pid_t pid;
   int i;
   pid = fork();
   if ( pid > 0 ) { // parent
      for (i=0;i<10;i++)
         printf("Parent %d\n", i);
         sleep(10); // sleep for 10 seconds
         wait(0); // take care of the child
         printf("The parent takes care of the child process\n");
   } else { // child
      sleep(1); // make sure the parent print 10 lines first
      for (i=0;i<10;i++)
         printf("Child %d\n", i);
      printf("Now, the child process becomes zombie\n");
   }
   return 0;
}
```

```
# gcc -o fork_zombie_demo fork_zombie_demo.c
# ./fork_zombie_demo

Parent 0
Parent 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Parent 9
Child 0
Child 1
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Child 9
Now, the child process becomes zombie
The parent takes care of the child process
```

Exit status of a process

- When a child process terminates:
  - Open files are flushed and closed
  - Child's resources are de-allocated
    - File descriptors, memory, semaphores, file locks, …
- Parent process is notified via a signal
  - Signal is an inter-process communication mechanism in an operating system
- Exit status is available to parent via **`wait()`**

Voluntary termination	Involuntary termination
Normal exit: exit(0)	Fatal error: divide by 0, core dump, segment fault
Error exit: exit(1)	Killed by another process: kill()

# means the Linux system shell prompt

Example: Checking the exit status

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   int child_status;
   pid_t child_pid ;

   pid_t pid = fork();
   if ( pid == 0 ) { /* child */
      return 0;
   }
   else { /* parent */
      printf("Parent PID %d\n", getpid());
      child_pid = wait(&child_status);
      printf("Child PID %d with status %d\n" ,
        child_pid, child_status);
   }
   return 0;
}
```

Status number 0 means exit without any error

```
# gcc -o fork_wait_status fork_wait_status.c
# ./fork_wait_status
Parent PID 3915
Child PID 3916 with status 0
#
```

File operations

Use these functions in the course project

File operations

- The following file operations are defined in `<fcntl.h>` and `<unistd.h>`

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

  - They are low-level file I/O APIs
  - The returned file descriptor ID can be used in pipe operations and file operations

Example: file operations

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
int main() {
   int fd;
   char buffer[10];

/* Example: Write a text file */
   fd = open("hello.txt", /* output file name */
      O_CREAT | O_WRONLY, /* flag */
      S_IRUSR | S_IWUSR ); /* user permission: 600 */
   write(fd, "hello\n", 6);
   close(fd);

/* Example: Read a text file */
   fd = open("hello.txt", /* input file name */
      O_RDONLY, /* flag */
      S_IRUSR | S_IWUSR); /* user permission: 600 */
      read(fd, buffer, 6);

   printf("%s", buffer);
   close(fd);
   return 0;
}
```

Avoid these functions in the course project

C Standard Library file operations

- There are similar file operations defined in `<stdio.h>`

```
FILE *fopen(const char *path, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream );
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *fp);
```

  - They are high-level I/O APIs

File operations: C Standard Library V.S. System Calls

- Both file-related system calls (i.e., `open, read, write, close`) and C standard library file operations (i.e., `fopen, fread, fwrite, fclose`) can be used to access files
- We should avoid using C standard library file operations in this course project, with the following reasons:
  - `FILE` is a high-level `struct` which is only defined in C standard library
  - `FILE*` cannot be used in pipe operations
    - pipe operations will be introduced in the next few slides

Linux Pipes

# means the Linux system shell prompt

Introduction

- Pipe is a shell feature where the output of one process is made as the input of another process

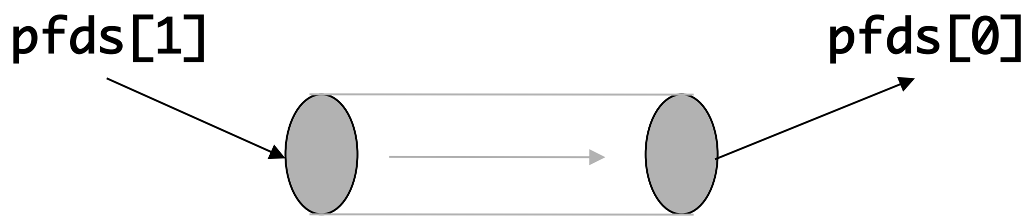
```
# ps aux | grep root | sort | wc -l
```

1. The `ps aux` command will output a list of running processes and the corresponding information.
2. After that, the output will be treated as an input of `grep`, which is a program to match the given pattern `root`
3. The lines will be sorted in an alphabetical order using the `sort` command
4. Finally, we count how lines of the final output

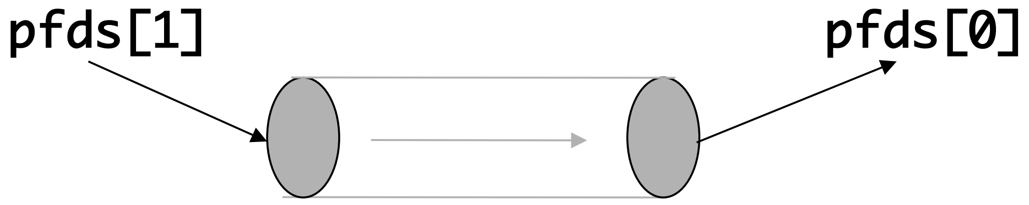
Unnamed pipe: `pipe`

```
#include <unistd.h>
int pipe(int pfds[2]);
```

- **Key concept: treat a pipe as 2 files**
- Create a message pipe
  - Anything can be written to the pipe, and read from the other end
  - Data is received in the order it was sent
  - Operating system enforces mutual exclusion: only one process at a time
  - Accessed by a file descriptor
  - Processes sharing the pipe must have the same parent
- Returns a pair of file descriptors
  - `pfds[0]` is the read end
  - `pfds[1]` is the write end



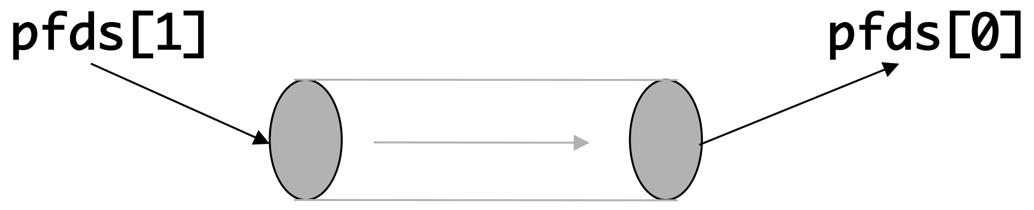
Pipe example (Child => Parent)



```c
int main()
{
   int pfds[2];
   char buf[30];
   pipe(pfds); /* Create a message pipe */
   pid_t pid = fork(); /* 0 (child), non-zero (parent) */
   if ( pid == 0 ) {
      printf("CHILD: writing to pipe\n");
      close(pfds[0]);
      write(pfds[1], "test", 5);
      printf("CHILD: exiting\n");
   } else {
      printf("PARENT: reading from pipe\n");
      close(pfds[1]);
      wait(NULL); /* Wait until the child returns*/
      read(pfds[0], buf, 5);
      printf("PARENT: read \"%s\"\n", buf);
   }
   return 0;
}
```

Note: read/write are blocking I/O operations (i.e., it will block the progress until the read/write is finished)

Pipe example (Parent => Child)



```
int main()
{
   int pfds[2];
   char buf[30];
   pipe(pfds); /* Create a message pipe*/
   pid_t pid = fork(); /* 0 (child), non-zero (parent) */
   if ( pid != 0 ) {
      printf("PARENT: writing to pipe\n");
      close(pfds[0]);
      write(pfds[1], "test", 5);
      wait(NULL); /* Wait until the child returns*/
      printf("PARENT: exiting\n");
   } else {
      printf("CHILD: reading from pipe\n");
      close(pfds[1]);
      read(pfds[0], buf, 5);
      printf(“CHILD: read \"%s\"\n", buf);
   }
   return 0;
}
```

Execute another executable file

- There are 6 different variations to execute a file:

```
execl, execlp, execle, execv, execvp, execvpe
```

- Please note that these functions replace the current process with a new process
    - In other words, after executing one of the above functions, it won't return unless there is an error
    - The return value is -1 if an error occurs

Example: Using `execlp`

- `execlp` is a system call which is useful if you know the number of parameters in advance

```c
#include <stdio.h>
#include <unistd.h>
int main() {
   // Same as running "ls -l -h"
   execlp("ls", "ls", "-l", "-h", NULL);

   // Note: It won’t return
   printf("It won't return. Nothing is shown here\n");

   return 0;
}
```

Example: Using `execvp`

- `execvp` is useful if you do not know the exact number of parameter in compile time. You should read and store the parameters in an array and invoke this function call

```c
#include <stdio.h>
#include <unistd.h>

// Assume that each command line has at most 256 characters
#define MAX_CMDLINE_LEN 256
// Assume that each segment has at most 256 characters
#define MAX_SEGMENT_LENGTH 256

int main() {
        // Same as running "ls -l -h"
        char command[MAX_CMDLINE_LEN] =  "ls";
        char* args[MAX_SEGMENT_LENGTH] = {"ls", "-l", "-h", NULL};
        execvp(command, args);
        return 0;
}
```

Duplicating a file descriptor: `dup`

```c
#include <unistd.h>
int dup(int oldfd);
```

- Create a copy of an open file descriptor: put new copy in first unused file descriptor
- Returns
  - Return value >= 0: success, returns new file descriptor
  - Return value = -1: error, check value of **errno**
- Parameters
  - **oldfd**: the open file descriptor to be duplicated
- Default file descriptor IDs
  - 0 is reserved for `stdin`
  - 1 is reserved for `stdout`
  - 2 is reserved for `stderr`

Example: Using `dup`

- Question: What is the expected output after running the following program?

```
#include <stdio.h> /* For print and fflush */
#include <unistd.h> /* For dup */
#include <sys/types.h>
#include <fcntl.h> /* For open syscall, flags, and user permissions */

int main() {
   int fd;
   fd = open("output.txt", /* output file name */
         O_CREAT | O_WRONLY , /* flags */
         S_IRUSR | S_IWUSR ); /* user permission: 600 */

   close(1); /* Close stdout */
   dup(fd); /* Replace stdout using the new file descriptor ID */

   printf("Hello World!\n"); /* call printf in C standard library */
   fflush(stdout); /* ensure all characters are output from the buffer */
   return 0;
}
```

# means the Linux system shell prompt

Example: A Linux pipe command

```
# ls | wc -l
```

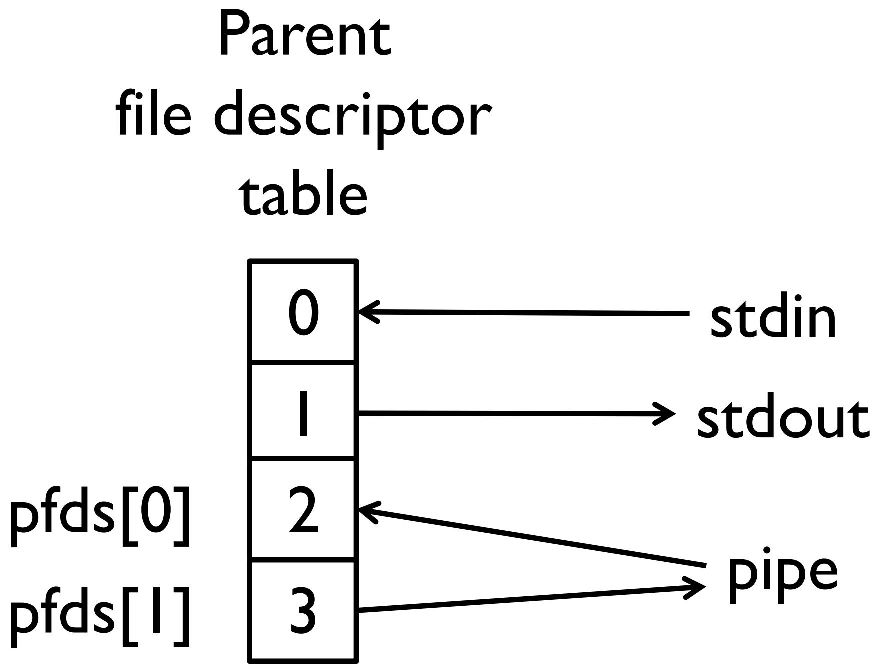
How can we implement a command-line pipe with `pipe()`?

How do we attach the stdout of `ls` to the stdin of `wc`?

Command-line pipe: `ls | wc -l`

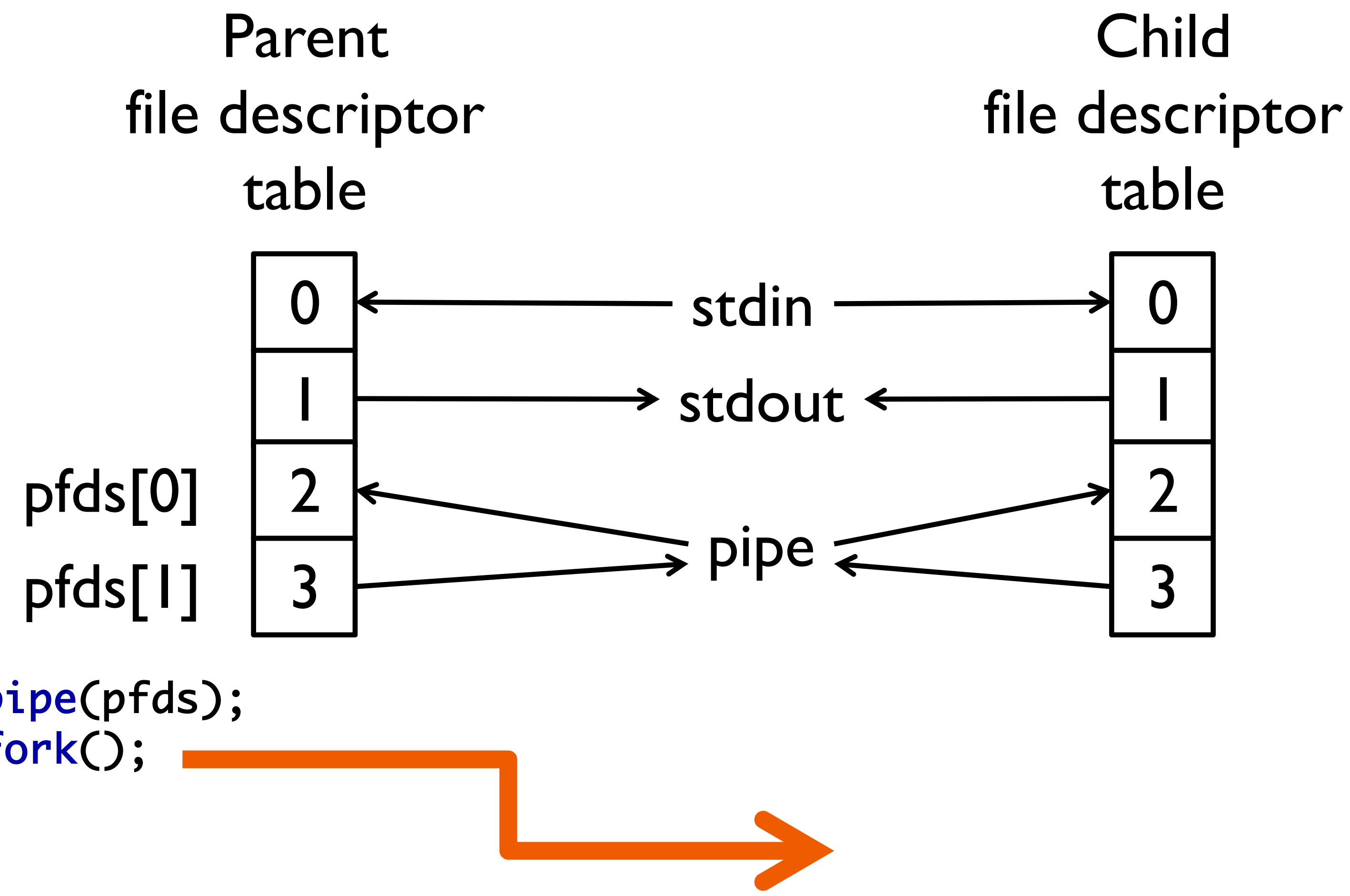
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   int pfds[2];
   pipe(pfds);
   pid_t pid = fork(); /* 0 (child), non-zero (parent) */
   if ( pid == 0 ) { /* The child process*/
      close(1); /* close stdout */
      dup(pfds[1]); /* make stdout as pipe input (1 is the smallest unused file descriptor) */
      close(pfds[0]); /* don't need this */
      execlp("ls", "ls", NULL);
   } else { /* The parent process*/
      close(0); /* close stdin */
      dup(pfds[0]); /* make stdin as pipe output (0 is the smallest unused file descriptor) */
      close(pfds[1]); /* don't need this */
      wait(0); /* wait for the child process */
      execlp("wc", "wc", "-l", NULL);
   }
   return 0;
}
```

Command-line pipe: `ls | wc -l`



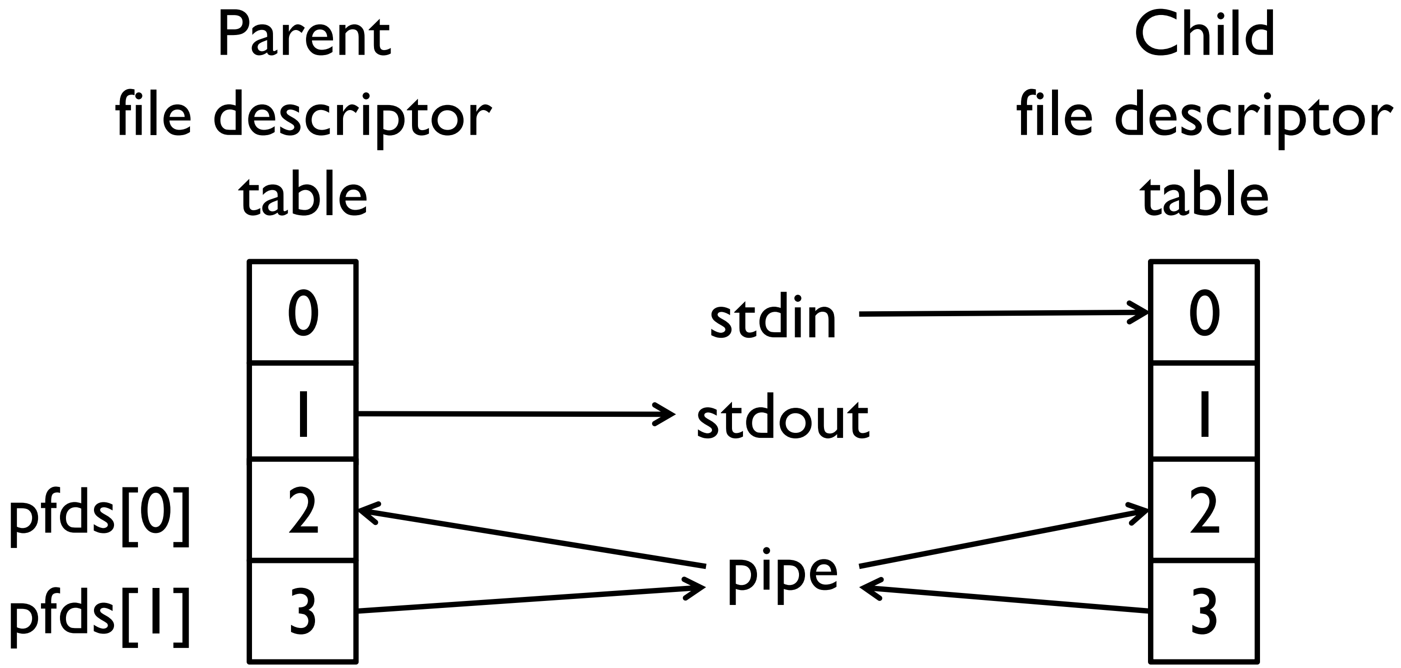
```
pipe(pfds);
```

Command-line pipe: `ls | wc -l`



```
pipe(pfds);
fork();
```

Command-line pipe: `ls | wc -l`

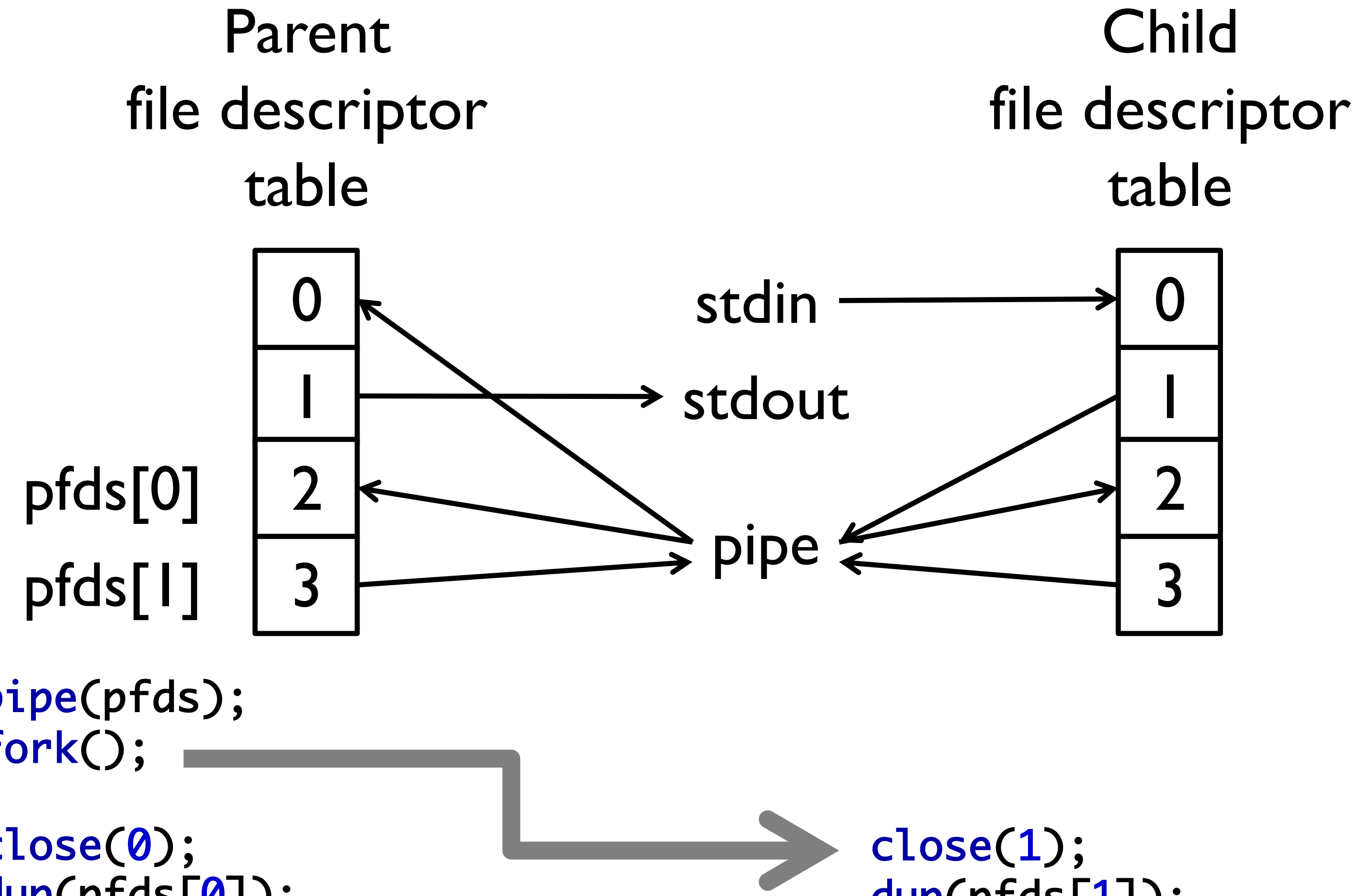


```
pipe(pfds);
fork();

close(0);
```

```
close(1);
```

Command-line pipe: `ls | wc -l`

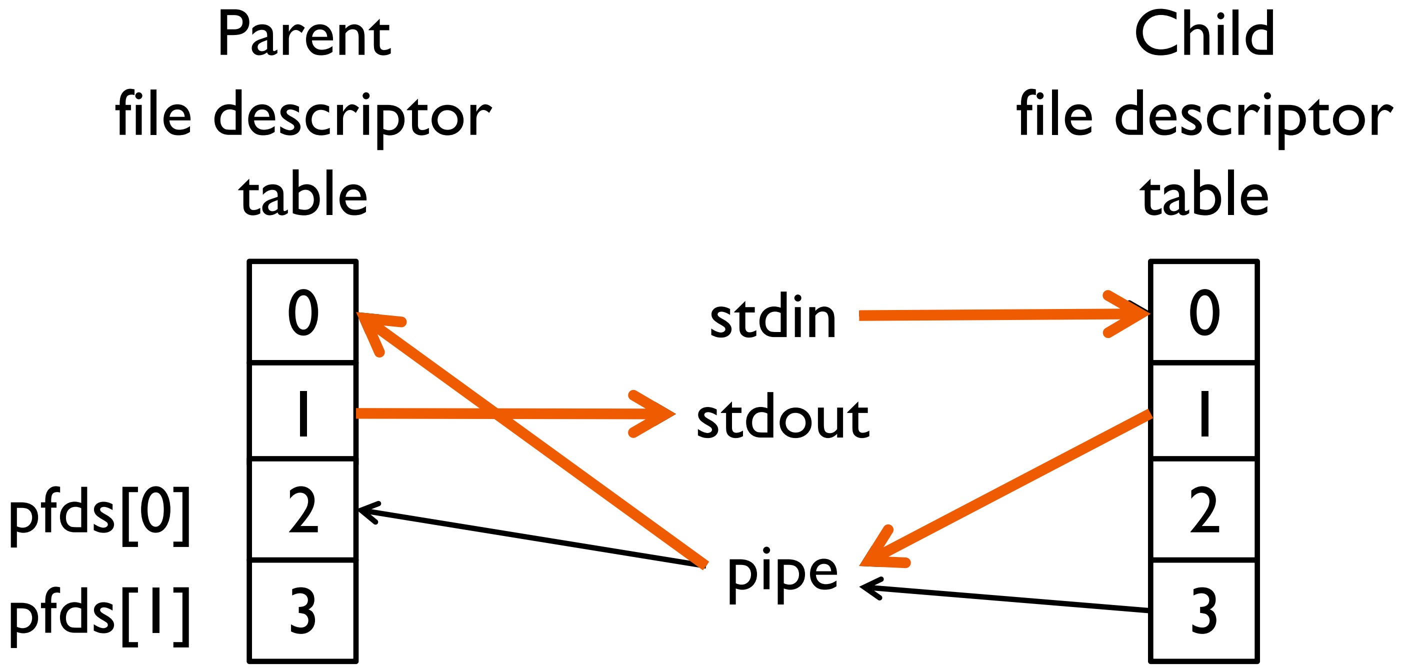


```
pipe(pfds);
fork();

close(0);
dup(pfds[0]);
```

```
close(1);
dup(pfds[1]);
```

Command-line pipe: `ls | wc -l`



```
pipe(pfds);
fork();

close(0);
dup(pfds[0]);
close(pfds[1]);
execlp("wc", "wc", "-l", NULL);
```

```
close(1);
dup(pfds[1]);
close(pfds[0]);
execlp("ls", "ls", NULL);
```

Duplicating a file descriptor: `dup2`

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- Create a copy of an open file descriptor: put new copy in specified location (…after closing **`newfd`**, if it was open)
- Returns
  - Return value >= 0: success, returns new file descriptor
  - Return value = -1: error, check value of **`errno`**
- Parameters
  - **`oldfd`**: the open file descriptor to be duplicated

Command-line pipe: `ls | wc -l` (using `dup2`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   int pfds[2];
   pipe(pfds);
   pid_t pid = fork(); /* 0 (child), non-zero (parent) */
   if ( pid == 0 ) { /* The child process*/
      close(1); /* close stdout */
      dup2(pfds[1],1); /* make stdout as pipe input */
      close(pfds[0]); /* don't need this */
      execlp("ls", "ls", NULL);
   } else { /* The parent process*/
      close(0); /* close stdin */
      dup2(pfds[0],0); /* make stdin as pipe output */
      close(pfds[1]); /* don't need this */
      wait(0); /* wait for the child process */
      execlp("wc", "wc", "-l", NULL);
   }
   return 0;
}
```

Which one should I use? `dup` `dup2`

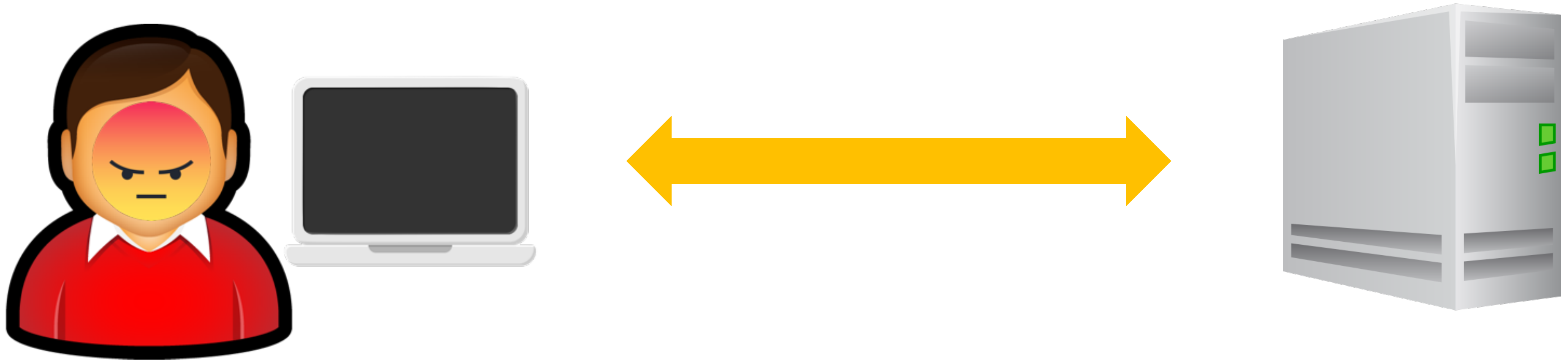
- In most scenario, `dup` and `dup2` can be used interchangeably
- `dup2` is more flexible than `dup`
- Code written using `dup2` is easier to read comparing with `dup`
- In some scenario, you must need to use dup2
  - Example: if you have 2 `close` function calls followed by 2 duplicate operations, it is better to use `dup2` to specify the new locations

Project Introduction

Simplified Linux Shell (Multi-level Pipe)

Students’ Perspective: Why I need to do the programming in a CS Lab 2 machine?

- Students:
  - It is troublesome to upload /download code between CS Lab2 Linux server and my local desktop /laptop computer, and then compile and debug the program in a CS Lab 2 machine



# means the Linux system shell prompt

TA's Perspective: We need to consistently grade 100-200+ programming submissions!

- It is impossible to fairly grade 100+ submissions with different environments:
  - Mac OS: 10+ different versions, 10+ different compiler versions…
  - Windows: 10+ different versions, 10+ different compiler versions…
  - Linux: 100+ different distributions, 10+ different versions…
- For example, there may be multiple `gcc` compilers installed:

```
# ls /usr/local/bin/gcc*
```

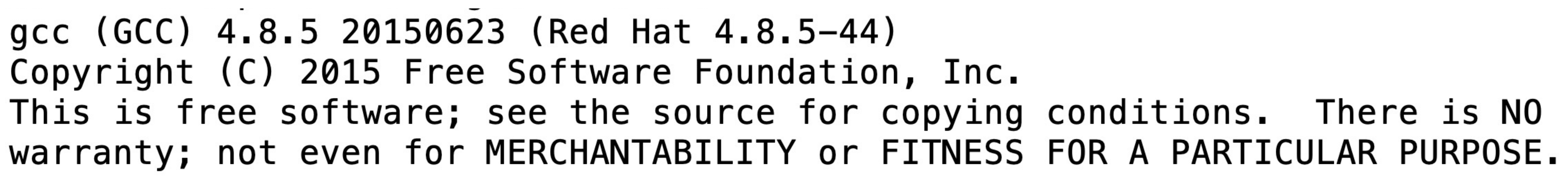
```
/usr/local/bin/gcc10  /usr/local/bin/gcc5  /usr/local/bin/gcc8
/usr/local/bin/gcc11  /usr/local/bin/gcc6  /usr/local/bin/gcc9
/usr/local/bin/gcc4   /usr/local/bin/gcc7
```

# means the Linux system shell prompt

C Compiler in the lab environment

- We use the default `gcc` compiler in a lab 2 machine
- You can check the exact gcc version by the following command:

```
# gcc --version
```



```
gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- We need to grade everything in a consistent lab environment
  - For the course project, `c99` option (which should be flexible enough for project assignments) will be added to compile the source files
  - The sample Linux executable is also compiled in a CS Lab 2 machine

Overview: The sample Linux executable

- In this assignment, you need to implement a command line interpreter that supports multi-level pipe
- Here is a sample usage:

```
$> ./mpipe < in.txt > out.txt
```

./ means the current working directory

< means passing in.txt as the standard input
> means passing the standard output to out.txt

CANNOT use this function in the course project

Restrictions

- You **CANNOT** use the system function defined in `<stdlib.h>`

```
int system(const char *command);
```

- The system function creates a default shell and then process the input command
- The purpose of the project assignment is to help students understand process management and inter-process communication. It is meaningless to directly use the system function to process the whole command

Tip 1: How to extend to a multi-level pipe?

- In the lab, we discussed a 2-level pipe example
- Hint: Can you rewrite the 2-level pipe example to a for-loop like this?

```
for (i=0; i<2; i++) {

// Rewrite the 2-level pipe here



}
```

- Once the loop is correctly implemented, you can easily extend it from a 2-level pipe to a multi-level pipe

# means the Linux system shell prompt

Tip 2: Checking your disk quota

- Unfortunately, CS Lab 2 users only have 100MB of disk quota
- Here are the commands to check the disk quota:

```
# cd ~
# du -ah | sort -rh | less
```

  - The first command goes to your home directory
  - The second command shows your disk usage
  - Press q to quit the `less` viewer
  - Follow-up actions
    - If you disk usage exceeds 100MB, you may need to delete files in your CS Lab 2 machine

Plagiarism

- **DON'T** do any cheating!
- Both parties (i.e., students providing the codes and students copying the codes) will receive 0 marks
- Near the end of the semester, a plagiarism detection software (MOSS) will be used to identify cheating cases
  - MOSS was developed by a research team in Stanford
  - The system is quite robust to detect simple refactoring tricks (e.g., renaming variables, inserting dummy variables, refactoring if-statements /loops, etc.)

Live Demo

The skeleton code

The sample Linux executable program

Q and A