Lab 10 Slides

Redirection & execvp()

exit Function

```
#include<stdlib.h>

void exit(int status);          // The _exit call is often used by the child.
```

Example: exit(EXIT_SUCCESS);

In stdlib.h:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

- The exit() function causes normal process termination and the value of status & 0377 is returned to the parent.
- All open stdio(3) streams are **flushed and closed**.
  (C standard library - from man 3 exit). (informally)
  clean shutdown, flush streams, close files, etc

_*exit* Function – Use in code for the *child*

- **_exit()**
  - - The function _exit() terminates the calling process "immediately".
  - Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a **SIGCHLD** signal.
  - (System call - from man 2 _exit)

    (informally) drop out, files are closed but streams are not flushed

Note:

Child and parent could have buffers with a copy of the unflushed data.

If both call exit(), the pending stdio buffers to be **flushed twice**.

Thus, child should call _exit() instead.

_*exit* Function – Use in code for the *child*

- There are two functions in Lab10 that will require the use of _exit
  - process_input
  - handle_redir

execvp Function

```
#include<unistd.h>

int execvp(const char *filename, *const argv[] );
```

Returns nothing on success, or -1 on error.

Example: return_value = execvp(argv[0], argv);

open call (1 of 5)

- Opening a file informs the kernel that an application wants to access a file
- Allows the kernel to set aside resources
- Returns file descriptor on success, or -1 on error

open call (2 of 5)

Call:

```
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *pathname, int flags, … /* mode_t mode */);
```

Example:

```
/* Open new or existing file for reading and writing, truncating
   to zero bytes; file permissions read+write for owner, nothing
   for all others */

fd = open("myfile", O_RDWR | O_CREAT |
                  O_TRUNC,  S_IRUSR | S_IWUSR);
if (fd == -1)
    perror("Error opening file");
```

Note: a const char *pathname means that the program can't change the data that pathname points to through the pathname pointer!

open call (3 of 5)

- Flags indicating access type:
  - O_RDONLY : read only
  - O_WRONLY : write only
  - O_RDWR: read/write
  - O_CREAT: create the file if doesn't exist
  - O_APPEND: write at end
  - O_TRUNC: Truncate exist file to zero length
  - etc.
- Can also bitwise – inclusive – or them
  - i.e. O_WRONLY | O_APPEND

  - **See: Table 4-3 (in LPI, page 74)**

open call (4 of 5)

- Different mode values (file permissions)

- S_IRUSR: read permission, owner
- S_IWUSR: write permission, owner
- S_IROTH: read permission, others
- S_IWOTH: write permission, others
- etc

Note: for more information, please do a

man 2 open to get all modes values.

open call (5 of 5)

- Open returns a small integer called a ***file descriptor (fd)***
- Application passes this value back to the kernel in subsequent requests to work with a file

- Each process created starts with three open files:
  - **0: standard input (stdin)**
  - **1: standard output (stdout)**
  - **2: standard error (stderr)**

<inistd.h> contains constants: STDIN_FILENO, STDOUT_FILENO, STDERR_FILEN

<stdio.h> includes: stdin, stdout, stderr

dup2 Function

```
#include <unistd.h>

int dup2(int oldfd, int newfd);

                        Returns (new) file descriptor on success, or -1 on error
```

Example: **dup2**(fd, 1); // Redirect from stdout (1) to the file (fd).

More in 9-Unix, slide 44-46

close - Closing Files

- Closing a file tells the kernel it may free resources associated with managing the file

- **The Call**:

```
#include <unistd.h>

int close (int fd);          Returns 0 on success,

  -1 on error
```

Examples:

```
if (close(fd) == -1) {
    perror("close file error\n");
    exit(EXIT_FAILURE);
}
```

```
int returnvalue = close(fd);
if (returnvalue == -1) {
    perror("close file error\n");
    exit(EXIT_FAILURE);
}
```

Dealing with Errors

You have at least two choices:

- *Use a fprintf* for non-system call errors, as in the redirection code.
  - *Example:* fprintf(stderr, "No command \n");

- *Use perror function* for system calls errors.
  - *Example:* perror("Error executing command \n");

The book shows functions named: errExit or usageErr or fatal

They will not work for us unless we include the appropriate code from the text book.

Error Checking - without calling the function twice

Examples: Either pattern is acceptable. Don't mix and match.

```
if (close(fd) == -1) {
    perror("close file error\n");
    exit(EXIT_FAILURE);
}
```

```
int returnvalue = close(fd);
if (returnvalue == -1) {
    perror("close file error\n");
    exit(EXIT_FAILURE);
}
```

perror System Call

```
#include <stdio.h>

void perror (const char *msg);
```

perror also will print the associated *errno*

More information in the text on pages 48-50

Review

Debugging

What is gdb?

- **gdb** is the GNU Project debugger
- **gdb** provides some helpful functionality
  - Allows you to stop your program at any given point.
  - You can examine the state of your program when it’s stopped.
  - Change things in your program, so you can experiment with correcting the effects of a bug.
- Also a command-line program

Using gdb:

- Compile with the **–g** flag to set up for debugging
- To start gdb with your hello program type:

  gdb HelloProg
- When gdb starts, your program is not actually running.
- You have to use the ***run*** command to start execution.
- Before you do that, you should place some break points.
- Once you hit a break point, you can examine any variable.

Useful gdb commands

run *command-line-arguments*

Begin execution of your program with arguments

break *place*

place can be the name of a function or a line number

For example: **break main** will stop execution at the first instruction of your program

delete *N*

Removes breakpoints, where *N* is the number of the breakpoint

step

Executes current instruction and stops on the next one

Gdb commands cont.

next

Same as **step** except this doesn't step into functions

print *E*

Prints the value of any variable in your program when you are at a breakpoint, where *E* is the name of the variable you want to print
print/x var (i.e p/x S_IFREG where x is the hex value), other options include: d (decimal), o (octal), t(two - binary), etc.

help *command*

Gives you more information about any command or all if you leave out command

quit

Exit gdb

GDB debugger with *fork* *(1 of 2)*

GDB Commands using with fork	Description
(gdb) set follow-fork-mode (child or parent)	Set debugger response to a program call of fork. follow-fork-mode can be: parent - the original process is debugged after a fork child - the new process is debugged after a fork The unfollowed process will continue to run. By default, the debugger will follow the parent process.
(gdb) set detach-on-fork (on or off)	Specifies whether GDB should debug both parent and child process after a call to **fork() -** Default is on: The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

GDB debugger with *fork* *(2 of 2)*

GDB Commands using with fork	Description
(gdb) catch fork	Catch calls to fork.
(gdb) info inferiors	Display IDs of currently known inferiors.
(gdb) inferior N	Use this command to switch between inferiors. The new inferior ID must be currently known (See above command).

Lab 10 Slides

The End