Operating Systems Lab

Lab 03

Solution

Objectives:

- Understanding the fork, exec and wait syscall Lecture#17
- Understanding the concept of process tree, fan and chain Lecture#18
- IO Redirection Lecture#08

Process Management

1) Explain the relationship between the parent and child processes created by *fork*(). What resources are shared between them? Write down any 2 use cases of *fork*() syscall.

Solution

Parent-Child Process Relationship (fork())

Memory Copy:

- The child process gets a copy of the parent's memory (stack, heap, data).
- They operate independently—changes in one do not affect the other.

Shared File Descriptors:

- File descriptors are shared between parent and child.
- If one process closes a file descriptor, it becomes unavailable to the other.

Concurrent Execution:

- Both processes run simultaneously.
- In the parent, fork() returns the child's PID.
- In the child, fork() returns 0.

Shared Resources

File Descriptors: Shared between parent and child; changes in one affect the other.

Signal Handlers: Inherited by the child process.

Environment Variables: Copied to the child; changes are independent.

Memory-Mapped Files: Shared, allowing both processes to access the same memory-mapped regions.

Use Cases of the fork() syscall

Creating Background Processes:

- fork() is used to create child processes that run independently of the parent. This is useful for tasks like running a server daemon in the background, allowing the parent process to continue executing while the child handles client requests.

Parallel Execution:

- fork() enables creating multiple processes to execute tasks concurrently. For example, in a shell, fork() is used to create a new process to execute a command while the parent shell continues to run, allowing multiple commands to be executed in parallel.

2) Explain the output of the following code.

```
#include
<stdio.h>
#include
<unistd.h>
#include
<stdlib.h> int
main()
{
int  cpid;
cpid =
fork(); if
(cpid == 0)
{
printf("child here.\n");
printf("CHILD pid = %d\n",
getpid());
printf("CHILD ppid = %d\n", getppid());
}
else
{
printf("parent  here.\n");
printf("PARENT pid = %d\n",
getpid()); printf("PARENT ppid = %d\
n", getppid()); sleep(2);
}

return 0;
}
```

Solution:

if (cpid == 0) block is executed by the child process, while the else block is executed by the parent process.

Output from the Child Process: If the process is the child:

child here.

CHILD pid = [child's PID]

CHILD ppid = [parent's PID]

- getpid() returns the PID of the child process, and getppid() returns the PID of the parent process.

Output from the Parent Process: If the process is the parent:

parent here.

PARENT pid = [parent's PID]

PARENT ppid = [grandparent's PID or 1 if the parent is the initial process]

- getpid() returns the PID of the parent process, and getppid() returns the PID of its parent process (which can be the shell or process that started it).

Assuming the child process has a PID of 1234 and the parent has a PID of 1233, the output could be:

child here.

CHILD pid = 1234

CHILD ppid = 1233

parent here.

PARENT pid = 1233

PARENT ppid = 1232

Summary

- The output indicates that the child and parent processes are executing independently.
- Each process retrieves and prints its own PID and its parent's PID, demonstrating the hierarchical relationship created by fork(). The output order may vary, but both processes will complete their execution independently.

3) What will be the output of the following code.

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

int main(void){

write(1, "I am learning OS", 17);
write(1, "I know what is syscall",
23);
write(1, "I am going to run the echo command", 35);
execl("/usr/bin/echo", "echo", "i am here", NULL);
```

```
write(1, "Should i be printed on screen or not",
37); return 0;
}
```

Solution *:*

Assuming the execl() call is successful, the output of the program will be:

I am learning OS
I know what is syscall
I am going to run the echo command
i am here

First three write calls:

- These will execute and print their respective strings to the screen.

execl("/usr/bin/echo", "echo", "i am here", NULL):

- This replaces the current process with the echo command, which prints "i am here" to the terminal.
- After this, the process running your program is replaced, and no more code in your program will run, unless the execl call fails.

Fourth write call (after execl):

- This line will not be executed if the execl is successful because the current process is replaced by the echo process.
- If the execl fails for some reason, this line will be printed to the screen.

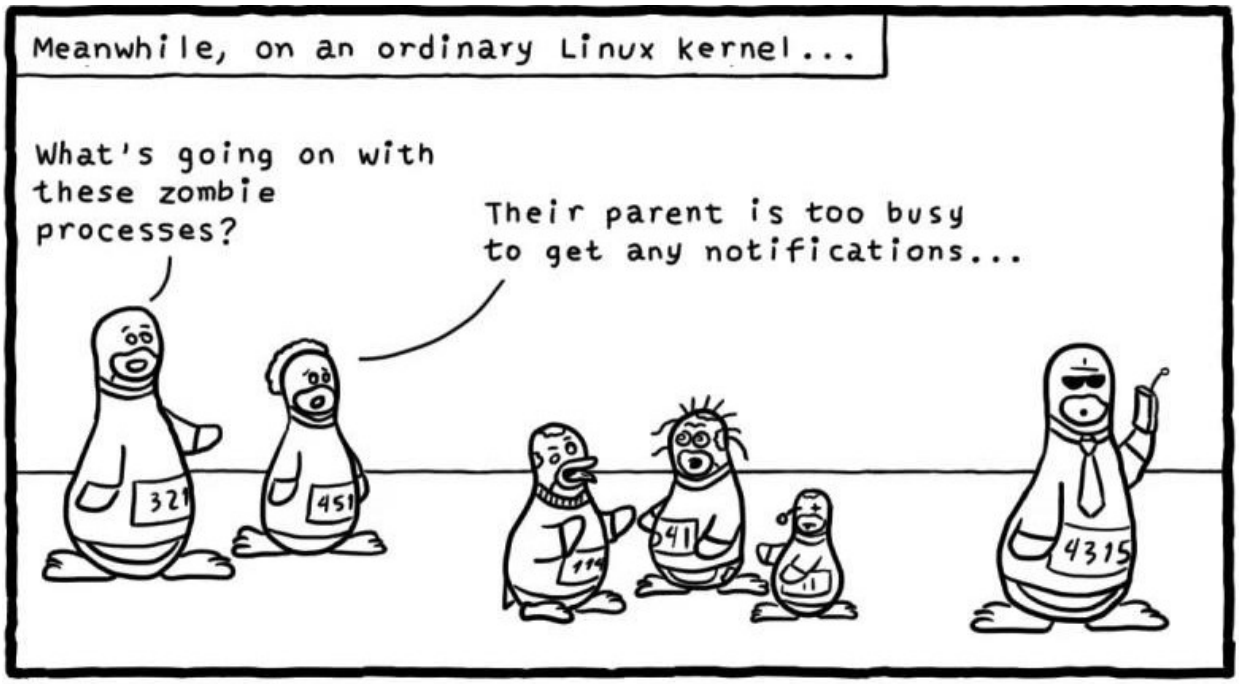
4) Write down a C program which will print the contents of present working directories using `execlp`.

Solution

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

int main(void) {
      // Execute
the pwd command
using execlp
    execlp("pwd"
, "pwd", NULL);

    // If execlp
fails, print an
error message
    perror("exec
lp failed");
          return
EXIT_FAILURE;
}
```



5) What are Orphan and Zombie processes? How can we see how many zombie processes there are in our system?

Solution

Orphan Processes:

- An orphan process is a child process whose parent process has terminated or finished execution before it.

- When a parent process exits, any child processes it has created are adopted by the init process (process ID 1), which becomes their new parent. This prevents the child process from becoming a zombie.
- Orphan processes continue to run independently of their original parent.

Zombie Processes:

- A zombie process is a process that has completed its execution but still has an entry in the process table.
- This happens when a child process finishes executing but the parent process has not yet called wait() to read the child's exit status. As a result, the child's information remains in the process table, marking it as a zombie.
- Zombie processes do not consume system resources (CPU or memory) but do occupy a slot in the process table.

Commands to Check for Zombies: ps aux | grep Z, ps aux | grep 'Z' | wc -l, top, and pstree -p.

Check Point

6) What will be the output of this C Program:

```
int main(){
for (int i=1;i<=4;i+
+){ fork();
fprintf(stderr, "%s\n","ARIF");
}
exit(0);
}
```

Solution

- First fork() call: You now have 2 processes (parent and child).
- Second fork() call: Each of the 2 processes from the first call forks again, so you now have 4 processes.
- Third fork() call: Each of the 4 processes forks, leading to 8 processes.
- Fourth fork() call: Each of the 8 processes forks again, resulting in 16 processes.

Total “Arif” wiil be printed 2+4+8+16=30 times.

7) What will be the output of this C Program:

```
int main(){
if (fork()||
fork()) fork();
printf("1 ");
```

```
}
```

Solution

First fork(): Creates 1 child process.

Parent Process:

- The first fork() returns a positive value, so the second fork() is not executed (short-circuit).
- The parent calls fork() inside the if block, creating another child.

First Child Process:

- The first fork() returns 0, making the condition false.
- The second fork() is executed, creating another child.
- No further fork() calls are made by this child.

Understanding the Output:

- The fork() system call is used multiple times, creating 5 processes (1 parent and 4 children).
- Each process executes printf("1 ");, resulting in 5 prints of "1 ".
- Due to concurrent execution, the order of the prints may vary, but the output will consist of 5 "1"s, as each process prints "1".
- **Thus, the output will be:**
- 1 1 1 1 1

8) What will the output of the following code? Does the parent reap the child successfully or the child becomes a zombie or orphan? Provide proof to whatever you think is the case.

```
int main(){
int  cpid;
cpid =
fork();
switch(cpid)
{
case 0:
printf("I am not a zombie process"); break;
default:
while(1)
;
}
}
```

Process Creation: The fork() creates a child process.

If cpid == 0 (child process), it print

I am not a zombie process

The parent process enters an infinite loop (while(1)), meaning it does not terminate or call wait() to reap the child.

Zombie Process:

Since the parent does not call wait() to collect the child's exit status, the child process will become a zombie after it finishes executing.

Proof:

The child process terminates after printing the message, but since the parent is stuck in the infinite loop and does not call wait(), the child remains in the process table as a zombie until the parent process is killed.

IO Redirection

1) *What is the Per Process File Descriptor Table (PPFDT), and how does it manage file descriptors for individual processes in Unix-like operating systems?*

Solution

The Per Process File Descriptor Table (PPFDT) is a table maintained by the operating system for each process, containing entries for all open files, sockets, pipes, etc. It maps file descriptors (integers) to actual open file entries in the system-wide file table.

Management:

Each process has its own PPFDT, isolating file descriptors between processes.

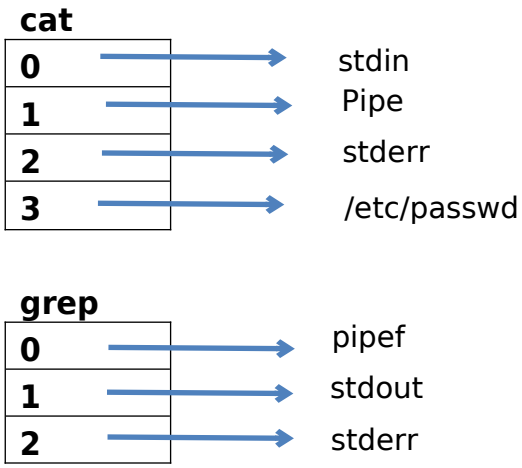
A file descriptor (e.g., 0 for stdin, 1 for stdout, 2 for stderr) in a process points to an entry in the system-wide file table that holds the file metadata, including file position and access mode.

When a process opens a file, the OS adds an entry to the PPFDT, associating the file descriptor with the file.

*2) Draw PPFDTs of **cat** and **grep** in the following command.*

cat /etc/passwd | grep root

Sloution



3) Perform the following tasks:

a. *Write a single command to copy the contents of the file **/etc/passwd** into **output.txt** without using copy.*

Solution

cat /etc/passwd > output.txt

b. *Find all the files named *libc.so* in your root directory (using **find** command) and redirect the output to the file libc_locations.txt and errors to the file /dev/null.*

Solution

find / -name '*libc.so*' > libc_locations.txt 2>/dev/null

4) *Save the following source code as hacking.c. Compile and make executable of the hacking.c and perform I/O redirection operations as described below:*

a) Redirect the output to a file named work_hard.txt.

b) Redirect the error to a file named failed.txt.

c) Redirect the stdout and stderr to a file called screen_copy.txt using copy descriptor.

```
#include <stdio.h>

#include<unistd.h>

#include<fcntl.h>

int main(void) {
int fd = open("/tmp/fake", O_RDONLY);
perror("ARM: Can't open file");
printf("Ever wanted to be a Hacker?\
n");
printf("If Yes, Work hard and learn how OS throws errors to other
files.\n");    return 0;
}
```

Solution

gcc -o hacking hacking.c

a)./hacking > work_hard.txt

b)./hacking 2> failed.txt

c)./hacking > screen_copy.txt 2>&1

5) Interpret the meaning of following if input file f1.txt exist or does not exist

- $ cat f1.txt > f2.txt
- $ cat f1.txt 1>f2.txt 2> f2.txt
- $ cat 0< f1.txt 1>> f2.txt 2>> f2.txt
- $ cat 0< f1.txt 1>> f2.txt 2> &1
- $ 2> errors.txt cat f1.txt > f2.txt
- $ tee 0< f1.txt f2.txt f3.txt

Solution

$ cat f1.txt > f2.txt:

- If f1.txt exists: The contents of f1.txt will be copied to f2.txt. If f2.txt does not exist, it will be created. If f2.txt exists, it will be overwritten.
- If f1.txt does not exist: An error message will be displayed on the terminal (cat: f1.txt: No such file or directory), and f2.txt will not be modified.

$ cat f1.txt 1>f2.txt 2> f2.txt:

- If f1.txt exists: Standard output (stdout) is redirected to f2.txt (the contents of f1.txt), and any error messages (stderr) are also redirected to f2.txt. This will cause confusion because both output and errors will be written to the same file, potentially mixing the contents.
- If f1.txt does not exist: The error message (cat: f1.txt: No such file or directory) will be written to f2.txt, and the original content of f2.txt will be overwritten by the error message.

$ cat 0< f1.txt 1>> f2.txt 2>> f2.txt:

- If f1.txt exists: The input comes from f1.txt (0< f1.txt), and the output (stdout) is appended to f2.txt. Any error messages (stderr) are also appended to f2.txt, so both successful output and errors will be appended to f2.txt.
- If f1.txt does not exist: The error message (cat: f1.txt: No such file or directory) will be appended to f2.txt.

$ cat 0< f1.txt 1>> f2.txt 2> &1:

- If f1.txt exists: The input comes from f1.txt, and the output (stdout) is appended to f2.txt. The error messages (stderr) are also redirected to the same location as stdout (f2.txt), so both output and errors will be appended to f2.txt.
- If f1.txt does not exist: The error message (cat: f1.txt: No such file or directory) will be appended to f2.txt, as both stdout and stderr are redirected to the same file.

$ 2> errors.txt cat f1.txt > f2.txt:

- If f1.txt exists: The contents of f1.txt will be copied to f2.txt (stdout), and any error messages will be written to errors.txt.
- If f1.txt does not exist: The error message (cat: f1.txt: No such file or directory) will be written to errors.txt, and f2.txt will not be modified.

$ tee 0< f1.txt f2.txt f3.txt:

- If f1.txt exists: The contents of f1.txt will be read as input, and the tee command will write the contents to both f2.txt and f3.txt. If the files do not exist, they will be created; if they exist, they will be overwritten.
- If f1.txt does not exist: An error will be displayed (tee: f1.txt: No such file or directory), and neither f2.txt nor f3.txt will be modified.

Bonus Task

1) Write a C program and perform the following tasks.
    a. Your program should take exactly one command line argument
    b. Do fork and run the given cat program using execve syscall in the child process with user argument.
    c. Your parent process should wait for the child process.
    d. Once the child process is terminated, print the id of the terminated child process.

Solution

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

int main(int argc, char *argv[]) {
    // a. Ensure exactly one command line argument is provided
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // b. Fork the process
    int pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        // c. Run the "cat" program using execve with the user-provided argument
```

```
        char *args[] = {"/bin/cat", argv[1], NULL};
        execve("/bin/cat", args, NULL);

        // If execve fails
        perror("execve failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        // c. Wait for the child process to terminate
        int status;
        int terminated_pid = wait(&status);

        if (terminated_pid == -1) {
            perror("wait failed");
            exit(EXIT_FAILURE);
        }

        // d. Print the ID of the terminated child process
        printf("Child process with PID %d terminated.\n", terminated_pid);
    }

    return 0;
}
```