HW#1
Advanced Operating Systems, Fall 2024

Lee Ming Fa (CBB110213)
Department of Computer Science and Information Engineering
National Pingtung University

1. Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

   Solution: Please refer to List 1 (q1.c):

Listing 1: q1.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     int x=100;
9     int rc = fork();
10    if (rc < 0) {
11        // fork failed; exit
12        fprintf(stderr, "fork␣failed\n");
13        exit(1);
14    } else if (rc == 0) {
15        // child (new process)
16        printf("hello,␣I␣am␣child␣(pid:%d)\n", (int) getpid());
17        printf("x␣was␣%d␣", x);
18        x=222;
19        printf(",but␣I␣have␣changed␣it␣to␣222...\n");
20        printf("Now,␣x␣is␣%d!\n", x);
21    } else {
22        // parent goes down this path (original process)
23        printf("hello,␣I␣am␣parent␣of␣%d␣(pid:%d)\n",
24               rc, (int) getpid());
25        printf("x␣was␣%d␣", x);
26        x=111;
27        printf(",but␣I␣have␣changed␣it␣to␣111...\n");
28        printf("Now,␣x␣is␣%d!\n", x);
29    }
30    return 0;
31 }
```

Its execution results are as follows:

```
1 $ cc q1.c
2 $ ./a.out
3 hello, I am parent of 10963 (pid:10962)
4 x was 100, but I have changed it to 111...
5 Now, x is 111!
6 hello, I am child (pid:10963)
7 x was 100, but I have changed it to 222...
8 Now, x is 222!
9 $
```

As we can see in Line 8 of Listing 1, we have declared a variable x with a value of 100. Lines 14-20 and 22-28 show the code for the child process and the parent process, respectively. In Lines 17 and 25, we print out the value of x for the child and the parent process, which both are 100 (as shown in lines 4 and 7 of the execution results). It is because the child process is created with the same content as the parent

– including the data segment as well as the stack. Later, in Lines 18 and 26, we changed the values of x in the child and the parent to 222 and 111, respectively. We also print out the changed values in Lines 20 and 28, which are 222 and 111. This shows that the forked child process has the same contents as its parent when it was created. However, after that, the child and the parent are two independent processes.

2. Write a program that opens a file (with the open() system call) and then calls fork() to create a new process. Can both the child and parent access the file descriptor returned by open()? What happens when they are writing to the file concurrently, i.e., at the same time?

Solution:

Consider the following program:

Listing 2: q2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <string.h>
6 #include <sys/types.h>
7 #include <sys/wait.h>
8
9 int main() {
10     //fd : file descripter
11     int fd = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
12     if (fd < 0) {
13         perror("open failed");
14         exit(EXIT_FAILURE);
15     }
16
17     int rc = fork();
18     if (rc < 0) {
19         perror("Fork failed");
20         exit(EXIT_FAILURE);
21     } else if (rc == 0) {
22         const char *child_message = "Hello from child!\n";
23         write(fd, child_message, strlen(child_message));
24         printf("Child wrote to file.\n");
25     } else {
26         const char *parent_message = "Hello from parent!\n";
27         write(fd, parent_message, strlen(parent_message));
28         printf("Parent wrote to file.\n");
29         wait(NULL);
30     }
31
32     close(fd);
33     return 0;
34 }
```

Its execution results are as follows:

```
1 $ cc q2.c
2 $ ./a.out
```

```
3 Parent wrote to file.
4 Child wrote to file.
5 $
```

The output.txt content:

```
1 Hello from parent!
2 Hello from child!
```

As shown in Listing 2, the program's main idea is to create a parent and child process to write messages into a file named `output.txt`. In Line 10, the `open()` function is used to open or create the file `output.txt`. The flags `O_CREAT` indicate that the file should be created if it does not already exist, `O_WRONLY` opens the file in write-only mode, and `O_TRUNC` clears the contents of the file if it already exists. The permission mode `0644` allows the owner to read and write the file, while others can only read it. Lines 17-20 handle error messages. If `fork()` fails, the program will display an error message and exit. Lines 20-24 represent the child process segment, which writes a message to the file and prints a success message. Lines 24-29 represent the parent process segment, which functions similarly to the child process. It writes a message to the file and prints a success message as well. As we can see in `output.txt`, the messages are written successfully.

3. Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

Solution:

Listing 3: q3.c

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

int main() {
    int pipefd[2];
    pipe(pipefd);
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello\n");
        write(pipefd[1], "finished", 8);
        close(pipefd[1]);
    } else {
        char buffer[4];
        read(pipefd[0], buffer, 8);
        printf("goodbye\n");
        close(pipefd[0]);
    }
    return 0;
}
```

Its execution results are as follows:

```
$ cc q3.c
$ ./a.out
hello
goodbye
$
```

As we can see in Line 7 of the code, we use pipe() to create a pipeline for inter-process communication, where pipefd[0] is the read end and pipefd[1] is the write end of the pipe. In Line 8, we call fork() to create a child process.

Lines 12-16 handle the child process. In Line 13, the child prints "hello", and in Line 14, it writes "finished" to the pipe using write(pipefd[1], "finished", 8), which sends the message to the parent process through the pipe. The child then closes the write end of the pipe in Line 15.

Lines 16-20 deal with the parent process. In Line 18, the parent waits for messege from the child by calling read(pipefd[0], buffer, 8), which blocks until the child sends its message. After receiving the message, the parent prints "goodbye" in Line 19, representing the message was received.

From upon example, we can know how the parent and child process switch the process by "finished" message. :)

4. Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?

Solution:

Listing 4: q4.c

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

int main() {
    printf("Child process is going to run /bin/ls\n");
    int rc = fork();
    if (rc == 0) {
        printf("Running execl\n");
        execl("/bin/ls", "ls", NULL);
        perror("excel error QAQ.");
        exit(1);
    }
    rc = fork();
    if (rc == 0) {
        printf("Running execle\n");
        char * env[] = {"User=MingfaUser", NULL};
        execle("/bin/ls", "ls", NULL,env);
        perror("excel error QAQ.");
        exit(1);
    }
    rc = fork();
    if (rc == 0) {
        printf("Running execlp\n");
        execlp("ls", "ls", "-l", NULL);
```

```
26          perror("excelp error QAQ.");
27          exit(1);
28      }
29      rc = fork();
30      if (rc == 0) {
31          printf("Running execv\n");
32          char *args[] = {"ls", "-l", NULL};
33          execv("/bin/ls", args);
34          perror("excev error QAQ.");
35          exit(1);
36      }
37      rc = fork();
38      if (rc == 0) {
39          printf("Running execvp\n");
40          char *args[] = {"ls", "-l", NULL};
41          execvp("ls", args);
42          perror("execvp error QAQ.");
43          exit(1);
44      }
45      return 0;
46
47      printf("All Processes finished.");
48  }
```

Its execution results are as follows:

```
1  $ cc q4.c
2  $ ./a.out
3  Child process is going to run /bin/ls
4  Running execl
5  Running execle
6  Running execlp
7  Running execv
8  Running execvp
9  a.out           q1.c            q3.c            q3_explain.c    q4.c
10 a.out           q1.c            q3.c            q3_explain.c    q4.c
11 total 104
12 total 104
13 total 104
14 -rwxr-xr-x  1 limingfa  staff  33488 10  1 15:31 a.out
15 -rw-r--r--@ 1 limingfa  staff    811  3 10  2022 q1.c
16 -rw-r--r--@ 1 limingfa  staff    487 10  1 14:39 q3.c
17 -rw-r--r--  1 limingfa  staff    854 10  1 14:01 q3_explain.c
18 -rw-r--r--  1 limingfa  staff   1179 10  1 15:28 q4.c
19 -rwxr-xr-x  1 limingfa  staff  33488 10  1 15:31 a.out
20 -rw-r--r--@ 1 limingfa  staff    811  3 10  2022 q1.c
21 -rwxr-xr-x  1 limingfa  staff  33488 10  1 15:31 a.out
22 -rw-r--r--@ 1 limingfa  staff    487 10  1 14:39 q3.c
23 -rw-r--r--  1 limingfa  staff    854 10  1 14:01 q3_explain.c
24 -rw-r--r--@ 1 limingfa  staff    811  3 10  2022 q1.c
25 -rw-r--r--  1 limingfa  staff   1179 10  1 15:28 q4.c
26 -rw-r--r--@ 1 limingfa  staff    487 10  1 14:39 q3.c
27 -rw-r--r--  1 limingfa  staff    854 10  1 14:01 q3_explain.c
28 -rw-r--r--  1 limingfa  staff   1179 10  1 15:28 q4.c
```

As we can see from the example, there are many different styles of `exec`. I think the basic functions are:

- **Parameter Transfer:** Some functions, like `execl()` and `execle()`, allow you to directly list parameters, which is suitable for a small number of known parameters. Others, like `execv()` and `execvp()`, use an array, making it easier to handle an uncertain number of parameters.
- **Control of Environment Variables:** Functions such as `execle()` and `execvpe()` allow you to set environment variables, which is really useful when you need to change the environment during execution.
- **Path Searching:** Functions like `execvp()` and `execvpe()` can automatically search for executable files in the `PATH` environment variable, so you don't need to write the full path.

With those functions, we can use the command more wisely. :)

5. Now write a program that uses wait() to wait for the child process to finish in the parent. What does wait() return? What happens if you use wait() in the child?

Solution:

First Example will show the parent process with wait() function, waiting for child process.

Listing 5: q5-1.c

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

int main() {
    int rc = fork();

    if (rc < 0) {
        fprintf(stderr, "Fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("Child process (PID: %d) is running.\n", getpid());
        sleep(2);
        printf("Child process (PID: %d) is done.\n", getpid());
        exit(0);
    } else {
        printf("Parent process (PID: %d) is waiting for child process (PID: %d) to finish.\n", getpid(), rc);
        int status;
        printf("wait the child process PID: %d\n",wait(&status));
        if(wait(&status)){
            printf("Child process exited with wait(&status) returned: %d\n", wait(&status));
        }
        printf("Parent process (PID: %d) is done. : )\n", getpid());
    }

    return 0;
}
```

Its execution results are as follows:

```
1 $  cc q5_1.c
2 $  ./a.out
3 Parent process (PID: 27404) is waiting for child process (PID: 27405) to finish.
4 Child process (PID: 27405) is running.
5 Child process (PID: 27405) is done.
6 wait the child process PID: 27405
7 Child process exited with wait(&status) returned: -1
8 Parent process (PID: 27404) is done. : )
```

As we can see the first example has printed out the return value waiting for child process (on the line 6). It actually return the Pid from the child process. After that it will return the -1 as the child process is finished (on the line 7).

Second Example will show the child process with wait() function, waiting for child's child process (not exist).

Listing 6: q5-2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <errno.h>
6
7 int main() {
8     int rc = fork();
9     if (rc < 0) {
10        fprintf(stderr, "fork failed\n");
11        exit(1);
12    } else if (rc == 0) {
13        printf("Child process is running\n");
14        int status;
15        printf("wait(&status): %d\n",wait(&status));
16        if(wait(&status)){
17            printf("No more child process here.\n");
18        }
19    } else {
20        wait(NULL);
21    }
22    return 0;
23 }
```

Its execution results are as follows:

```
1 $  cc q5_2.c
2 $  ./a.out
3 Child process is running
4 wait(&status): -1
5 No more child process here.
```

As we can see in lines 12-19, the child segment is designed to wait for its child process to finish. However, since there are no child processes created after the child process, the `wait(&status)` function will return -1, indicating that there are no child processes left to wait for.

6. Write a slight modification of the previous program, this time using waitpid() instead of wait(). When

would waitpid() be useful?

Solution:

Listing 7: q6.c

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

int main() {
    int rc = fork();

    if (rc < 0) {
        perror("Fork failed");
        exit(1);
    } else if (rc == 0) {
        printf("Child process is running. PID=%d\n", getpid());
        sleep(2);
    } else {
        int status;
        int waited_pid = waitpid(rc, &status, 0);
        printf("Parent process is running. PID=%d\n", getpid());
        printf("waitpid() returned: %d\n", waited_pid);
    }

    return 0;
}
```

Its execution results are as follows:

```
$  cc q6.c
$  ./a.out
Child process is running. PID=36153
Parent process is running. PID=36149
waitpid() returned: 36153
```

The getpid() function will be useful for identifying a specific process. If we want to know the process ID of the currently running process, we can call getpid(), which returns its own process ID. This can help in debugging or when we need to manage processes, as we can keep track of which process is executing certain tasks. For example, if a parent process needs to wait for a particular child process, it can use getpid() to know its own ID and ensure it's managing the right child process effectively.

7. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf() to print some output after closing the descriptor?

Solution:

Listing 8: q7.c

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

```

```c
int main() {
    int rc = fork();

    if (rc < 0) {
        perror("Fork failed");
        exit(1);
    } else if (rc == 0) {
        close(STDOUT_FILENO);
        printf("This message will not be printed to the terminal.\n");
        exit(0);
    } else {
        int status;
        waitpid(rc, &status, 0);
        printf("Child process has finished. :)\n");
    }

    return 0;
}
```

Its execution results are as follows:

```
$  cc q7.c
$  ./a.out
Child process has finished. :)
```

In the child process (when rc == 0), nothing will be printed out after closing the standard output until the process finishes and control switches back to the parent process.(when rc == 1)

8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the pipe() system call.

Solution:

Listing 9: q8.c

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

int main() {
    int pipeEnd[2];
    if (pipe(pipeEnd) == -1) {
        perror("pipe failed");
        exit(1);
    }

    int rc1 = fork();

    if (rc1 < 0) {
        perror("Fork failed for first child");
        exit(1);
    } else if (rc1 == 0) {
        close(pipeEnd[0]);
        const char *message = "Hello from child 1";
        write(pipeEnd[1], message, strlen(message) + 1);
```

```
22          close(pipeEnd[1]);
23      }
24
25      int rc2 = fork();
26
27      if (rc2 < 0) {
28          perror("Fork failed for second child");
29          exit(1);
30      } else if (rc2 == 0) {
31          close(pipeEnd[1]);
32          char buffer[100];
33          read(pipeEnd[0], buffer, sizeof(buffer));
34          printf("Child 2 received: %s\n", buffer);
35          close(pipeEnd[0]);
36      }
37
38      close(pipeEnd[0]);
39      close(pipeEnd[1]);
40
41      return 0;
42  }
```

Its execution results are as follows:

```
1 $  cc q8.c
2 $  ./a.out
3 Child 2 received: Hello from child 1
```

In the child process (when rc1 == 0), it closes the read end of the pipe (pipeEnd[0]) to ensure it only writes. It then prepares a message, "Hello from child 1", and writes this message to the write end of the pipe (pipeEnd[1]). After writing, it closes the write end of the pipe and exits.

A second child process is created with another fork() call (rc2). If the fork fails, an error message is printed. In the second child process, it closes the write end of the pipe (pipeEnd[1]) to ensure it only reads. It then declares a buffer to hold the incoming message and reads from the read end of the pipe (pipeEnd[0]). After successfully reading the message, it prints the received message to the console and closes the read end before exiting.

After all the process ended, close the writing and reading ends. :)