HW#1
Advanced Operating Systems, Spring 2022

CHUANG, CHIA-CHUN (CBB107045)
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) and its execution results are as follows:

```
1 $ cc 5-1.c
2 $ ./a.out
3 set    x = 111, (pid:8659)
4 parent x = 111 (pid:8659)
5 parent after changed x = 333 (pid:8659)
6 child  x = 111 (pid:8660)
7 child after changed x = 222 (pid:8660)
8
9 $
```

And consider the following programs:

Listing 1: q1.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<stdlib.h>
4 int main()
5 {
6         int x = 111;
7         printf("set x = %d, (pid:%d)\n", x, (int)getpid());
8         int rc = fork();
9         if (rc < 0) {
10                fprintf(stderr, "fork Error!");
11                exit(1);
12        }
13        else if (rc == 0) {
14                printf("child x = %d (pid:%d)\n", x, (int)getpid());
15                x = 222;
16                printf("child after changed x = %d (pid:%d)\n", x, (int)getpid());
17        }
18        else {
19                printf("parent x = %d (pid:%d)\n", x, (int)getpid());
20                x = 333;
21                printf("parent after changed x = %d (pid:%d)\n", x, (int)getpid());
22        }
23        return 0;
24 }
```

In Line 7, we have declared a variable x with value 111. As we can see that in Lines 13-17 and 18-22 show the code for the child process and the parent process, respectively. In Lines 14and 19, we print out the value of x for the child and the parent process, which both are 111(as shown in lines 4 and 6 of the execution results). It is because the child process is created with the same content of the parent – including the data segment as well as the stack. Later, in Lines 15 and 20, we changed the values of x in the child and the parent to 222 and 333, respectively. We also print out the changed values in Lines 16 and 21, which are 222 and 333. This can be shown that the forked child process has the same contents of its parent at the time 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: Please check the following results:

```
1 $ cc 5-2.c
2 $ ./a.out
3 $ cat 5.2.txt
4 parent come here to check
5 child wrote something here
6 $
```

And consider the following programs:

Listing 2: q2.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<stdlib.h>
4 #include<fcntl.h>
5 #include<sys/wait.h>
6 #include<string.h>
7 int main()
8 {
9         int f = open("./5.2.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
10        int rc = fork();
11        if (rc < 0) {
12                fprintf(stderr,"fork Error!");
13                close(f);
14                exit(1);
15        }
16        else if (rc == 0) {
17                char * s = "child wrote something here\n";
18                write(f, s, strlen(s));
19        }
20        else {
21                char * s = "parent come here to check\n";
22                write(f, s, strlen(s));
23                wait(NULL);
24                close(f);
25        }
26        return 0;
27 }
```

We opened the 5.2.txt text file on line 9 as the file required by the title, As we can see that in Lines 16-19 and 20-25 show the code for the child process and the parent process, respectively. In lines 17and 21 are the text we are going to write into the file, and do the writing action at lines 18 and 22. As shown in lines 4 and 5 of the execution results ,child and parent access can open to the file, it can be seen that Both child and parent access can perform write actions at the same time.

3. Write another program using fork(). The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?

   Solution: Please check the following results:

```
1 $ cc 5-3.c
2 $ ./a.out
3 hello
4 goodbye
5 $
```

And consider the following programs:

Listing 3: q3.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<unistd.h>
4  int main()
5  {
6          int rc = vfork();
7          if (rc < 0) {
8                  fprintf(stderr, "fork Error!");
9                  exit(1);
10         }
11         else if (rc == 0) {
12                 printf("hello\n");
13                 exit(1);
14         }
15         else {
16                 printf("goodbye\n");
17         }
18         return 0;
19 }
```

In Line 6, we use the vfork() formula. vfork is mainly used to ensure that the child process runs first, and the parent process can be scheduled to run after it calls exec or exit. As we can see that in Lines 11- 14 and 15-17 show the code for the child process and the parent process, respectively. We add exit to the subroutine line 13 to end the vfork and can see in the execution result lines 3 and 4 that the child process is more successful than the parent process to execute early

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: Please check the following results:

```
1 $ cc 5-4.c
2 $ ./a.out
3 anaconda-post.log   cbb105021   cbb108042   lib          mnt    run    tmp
```

```
bin                 cbb108010   dev         lib64         opt    sbin   usr
boot                cbb108016   etc         lost+found    proc   srv    var
cbb105014           cbb108038   home        media         root   sys
$
```

And consider the following programs:

Listing 4: q4.c

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

int i = 0;
const int MAX = 6;
int main()
{
        char * s = "/bin/ls";
        char * ss = "ls";
        char * s2 = "/";
        char * sv[] = { ss, s2, NULL };
        int rc = fork();
        for(i=0;i<MAX;i++) {
                if (rc < 0) {
                    fprintf(stderr, "fork failed");
                    exit(1);
                }
                else if (rc == 0) {
                    switch(i) {
                            case 0:
                                execl(s, ss, s2, NULL);
                                break;
                            case 1:
                                execle(s, ss, s2, NULL);
                                break;
                            case 2:
                                execlp(s, s, s2, NULL);
                                break;
                            case 3:
                                execv(s, sv);
                                break;
                            case 4:
                                execvp(ss, sv);
                                break;
                            case 5:
                                execvpe(ss, sv);
                                break;
                            default: break;
                    }
                }
                else {
                    wait(NULL);
                }
        }
        return 0;
}
```

As we can see that in Lines 19- 41 and 42-44 show the code for the child process and the parent process, respectively. We try to use 6 different execl variants in the child process, such as the execution result lines 3-6 As shown in the row, all variants can be executed

5. 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: Please check the following results:

```
1 $ cc 5-5.c
2 $ ./a.out
3 childpid:17941 wc:-1 rc:0
4 parentpid:17940 wc:17941 rc:17941
5 $
```

And consider the following programs:

Listing 5: q5.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<sys/wait.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8         int rc = fork();
9         int wc = wait(NULL);
10        if(rc < 0) {
11                fprintf(stderr, "fork failed");
12                exit(1);
13        }
14        else if (rc == 0) {
15                printf("child");
16        }
17        else {
18                printf("parent");
19        }
20        printf("pid:%d wc:%d rc:%d\n", (int)getpid(), wc, rc);
21        return 0;
22 }
```

Declare a wc variable to store the return value of wait, As we can see that in Lines 14-16 and 17-19 show the code for the child process and the parent process, respectively. parent process use wait() to wait for the child process to return id, child process itself does not need to wait for fork return, so the wait value is -1 (the execution result lines 3 and 4 are visible)

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

Solution: Please check the following results:

```
1 $ cc 5-6.c
2 $ ./a.out
3 childpid:18829 wc:-1 rc:0
4 parentpid:18828 wc:18829 rc:18829
5 $
```

And consider the following programs:

Listing 6: q6.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<sys/wait.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8         int rc = fork();
9         int wc = waitpid(rc, NULL, 0);
10        if(rc < 0) {
11                fprintf(stderr, "fork failed");
12                exit(1);
13        }
14        else if (rc == 0) {
15                printf("child");
16        }
17        else {
18                printf("parent");
19        }
20        printf("pid:%d wc:%d rc:%d\n", (int)getpid(), wc, rc);
21        return 0;
22 }
```

We can see that in Lines 14-16 and 17-19 show the code for the child process and the parent process, respectively. According to the execution result lines 3 and 4, we can know that waitpid works when the process itself has a child process

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: Please check the following results:

```
1 $ cc 5-7.c
2 $ ./a.out
3 $ ./a.out
4 $
5 $
```

And consider the following programs:

Listing 7: q7.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<stdlib.h>
4
5 int main()
6 {
7         int rc = fork();
8         if (rc < 0) {
9                 fprintf(stderr, "fork failed");
10                exit(1);
11        }
12        else if (rc == 0) {
13                close(STDOUT_FILENO);
14                printf("output child\n");
15        }
16        wait(NULL);
17        return 0;
18 }
```

We can see that in Lines 12-15 show the code for the child process , According to the execution result nothing is printed out, we can know that child closes standard output (STDOUT FILENO). There is no way to print anything when it is closed

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: Please check the following results:

```
1 $ cc 5-8.c
2 $ ./a.out
3 child0 out (child input) in the child1
4 $
```

And consider the following programs:

Listing 8: q8.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<stdlib.h>
4
5 int main() {
6         int pi[2];
7         int p = pipe(pi);
8         if(p < 0) {
9                 fprintf(stderr, "pipe failed");
10                exit(1);
```

```
11          }
12          int i = 0;
13          int rc[2];
14          char buf[256];
15          for(i=0;i<2;i++) {
16                  rc[i] = fork();
17                  if (rc[i] < 0) {
18                          fprintf(stderr, "fork failed");
19                          exit(1);
20                  }
21                  else if (rc[i] == 0) {
22                          switch(i) {
23                                  case 0:
24                                          dup2(pi[1], STDOUT_FILENO);
25                                          puts("child input");
26                                          break;
27                                  case 1:
28                                          dup2(pi[0], STDIN_FILENO);
29                                          gets(buf);
30                                          printf("child0 out (%s) in the child1\n", buf
                                                );
31                                          return 2;
32                          }
33                          break;
34                  }
35          }
36          waitpid(rc[0], NULL, 0);
37          waitpid(rc[1], NULL, 0);
38          return 0;
39  }
```

Declare an array of pi in line 6 to distinguish two different child processes, line 7 can execute two different processes, line 14 declares a variable buf used to store strings, As we can see that in Lines 23 and 27-31 show the code for the child process and the other child process, respectively. In line 25, use puts to store the text to be output into buf and use gets in line 29, and line 30 outputs the content of the obtained string