OS实验三——进程通信实验

一、观察fork1.c代码并解释

1. 代码展示

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

int value = 5;  // where?

// 定义全局变量，被后续创建的两个进程共享

int main() {
    int i;  // where?
    // 定义局部变量
    pid_t pid;
    for (i = 0; i < 2; i++) {  // How many new processes and printfs£¿
        pid = fork();
        if (pid == 0) {
            value += 15;
            printf("Child: value = %d\n", value);
        } else if (pid > 0) {
            wait(NULL);  // 使父进程等待子进程结束
            printf("PARNET: value = %d\n", value);
            exit(0);  // Notice£ºWhat will happen with or without this line?
            //   使父进程正常退出返回0，没有的话父进程会继续执行
        }
    }
}
```

2. 结果输出

```
[root@GYF os_week13]# ./fork1
Child: value = 20
Child: value = 35
PARNET: value = 20
PARNET: value = 5
```

3. 现象解释

> 复制状态信息都只是对父进程当前状态信息的拷贝。

1. 进行第一次循环时，i=0，value=5，父进程P0调用fork函数创建子进程P1，子进程P1复制父进程P0当前状态的信息（value=5，i=0）并返回自身的pid给父进程P0的pid。父进程P0执行到else if的时候阻塞在wait（NULL）处等待P1的返回。
2. P1在被创建后，此时i=0，value=5，会再一次调用fork函数，得到的结果为0放入P1的pid中。然后执行if的代码块，value+=15，此时value=20，并打印输出结果。之后i++，进入第二次循环。P1进入第二次循环后，会创建新子进程P2，P2复制P1的当前状态信息（value=20，i=1），并返回自身的pid给P1的pid。父进程P1执行到else if的时候阻塞在wait（NULL）处等待P2的返回。
3. P2只会进行一次循环，调用fork函数，得返回值0给pid，然后执行if代码块，value+=15，并打印输出结果。之后i++，退出循环，由main函数返回。此时P1得知P2运行完毕，继续执行打印自身现

在的value值，最后exit（0）返回。

4. P0知道P1结束运行后，接着运行打印自身的value值，然后执行exit（0）返回。如果没有exit语句，就会开始第二轮循环，创建新子进程P3。

二、管道通信

1. 代码展示

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    int pid;
    int pipe1[2];
    int pipe2[2];
    int x;
    if (pipe(pipe1) < 0) {
        perror("failed to create pipe1");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipe2) < 0) {
        perror("failed to create pipe2");
        exit(EXIT_FAILURE);
    }
    pid = fork();
    if (pid < 0) {
        perror("failed to create new process");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // 子进程=>父进程：子进程通过pipe2[1]进行写
        // 子进程<=父进程：子进程通过pipe1[0]读
        // 因此，在子进程中将pipe1[1]和pipe2[0]关闭
        close(pipe1[1]);  // 关闭1的写
        close(pipe2[0]);  // 关闭2的读
        do {
            read(pipe1[0], &x, sizeof(int));
            printf("child %d read: %d\n", getpid(), x++);
            write(pipe2[1], &x, sizeof(int));
        } while (x <= 9);
        close(pipe1[0]);
        close(pipe2[1]);
    } else {
        // 父进程<=子进程:父进程从pipe2[0]读取子进程传过来的数
        // 父进程=>子进程:父进程将更新的值通过pipe1[1]写入，传给子进程
        // 因此，父进程会先关闭pipe1[0]和pipe2[1]端口
        close(pipe1[0]);
        close(pipe2[1]);
        x = 1;
        do {
            write(pipe1[1], &x, sizeof(int));
            read(pipe2[0], &x, sizeof(int));
            printf("parent %d read: %d\n", getpid(), x++);
        } while (x <= 9);
        close(pipe1[1]);
        close(pipe2[0]);
    }
```

```
    return EXIT_SUCCESS;
}
```

2. 创建Makefile文件

```
srcs=ppipe.c
objs=ppipe.o
opts=-g -c
all:ppipe
ppipe: $(objs)
    gcc $(objs) -o ppipe
ppipe.o: $(srcs)
    gcc $(opts) $(srcs)
clean:
    rm ppipe *.o
```

3. 运行Makefile得到可执行文件并执行

```
[root@GYF os_week13]# make
gcc -g -c ppipe.c
gcc ppipe.o -o ppipe
[root@GYF os_week13]# ./ppipe
child 4671 read: 1
parent 4670 read: 2
child 4671 read: 3
parent 4670 read: 4
child 4671 read: 5
parent 4670 read: 6
child 4671 read: 7
parent 4670 read: 8
child 4671 read: 9
parent 4670 read: 10
[root@GYF os_week13]#
```

三、独立实验

1. 代码展示

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

int fx(int x) {
    if (x == 1)
        return 1;
    return x * fx(x - 1);
}

int fy(int y) {
    if (y == 1 || y == 2)
        return 1;
    return fy(y - 1) + fy(y - 2);
}

int main() {
    pid_t pid_fx, pid_fy;
    int x, y;
    printf("Input x and y:\n");
    scanf("%d %d", &x, &y);
```

```
    // 创建第一个子进程计算 f(x)
    pid_fx = fork();
    if (pid_fx == 0) {
        printf("f(x) = %d\n", fx(x));
        exit(0);
    }

    // 创建第二个子进程计算 f(y)
    pid_fy = fork();
    if (pid_fy == 0) {
        printf("f(y) = %d\n", fy(y));
        exit(0);
    }

    wait(NULL);
    wait(NULL);

    printf("f(x, y) = %d\n", fx(x) + fy(y));

    return 0;
}
```

2. 结果输出

```
[root@GYF os_week13]# gcc -o function function.c
function.c: 在函数‘main’中：
function.c:38:5: 警告：隐式声明函数‘wait’ [-Wimplicit-function-declaration]
   38 |     wait(NULL);
      |     ^~~~
[root@GYF os_week13]# ./function
Input x and y:
3 4
f(x) = 6
f(y) = 3
f(x, y) = 9
[root@GYF os_week13]#
```

3. 现象解释

父进程P0创建两个子进程P1和P2，分别执行递归操作和计算斐波那契数列的值的操作，但是该代码疑似逻辑上有点问题。在计算完两个子进程后没有对计算结果进行保存，而是选择在父进程内再计算一遍，理论上应该可以把值直接保存好，避免重复计算。