Homework 5: Processes

Name: Christian Magnus Engebretsen Heimvik Kingo ID: 2025318314

Question 1

What we expect here is that both the parent and the child process share all memory regions initially, but upon one of them being modified, it will get copied. Thus, as soon as we change one of the variables, both copies wil operate on their own copy. This holds for all 3 memory regions: static (stored in proram code), stack and heap.

Here is the written code:

```
//Q1 global static variables
int static_variable = 10;

int q1(void){
    int* heap_variable = (int*)malloc(sizeof(int));
    *heap_variable = 10;
    int stack_variable = 10;
    int pid = fork();
    if(pid>0){
        printf("The parent process entered increment\n");
        for(int j = 0; j<5;j++){
            printf("Mother inc: %d %d %d\n",stack_variable++,
(*heap_variable)++, static_variable++);
        }
        wait(NULL);
        return 0;
    } else {
        printf("The child process entered increment\n");
        for(int j = 0; j<5;j++){
            printf("Child inc: %d %d %d\n", stack_variable--,
(*heap_variable)--,static_variable--);
        }
        return 0;
    }
    return 1;
}
```

And the output is:

```
The parent process entered increment
Mother inc: 10 10 10
Mother inc: 11 11 11
Mother inc: 12 12 12
Mother inc: 13 13 13
Mother inc: 14 14 14
```

```
The child process entered increment
Child inc: 10 10 10
Child inc: 9 9 9
Child inc: 8 8 8
Child inc: 7 7 7
Child inc: 6 6 6
```

And we see that upon any modifiacations to their respective memory regions, they get a new copy of the memory region, and are thus no longer operating on the same data (COW - Copy On Write).

Question 2

What we expect here, is that upon the creation of the child process, the file descriptors get duplicated (not like memory access), and any modifications on either of them, will only affect the file descriptor in question. However, as we havnt applied any form of sychronization, and the scheduling is undeterministic, we could expect some simultaneous/uncoordinated writing to file.

Here is the code:

```
int q2(void){
    //Open a file in the same file directory as the current one
    int fd = open("testFile.txt",O_RDWR|O_CREAT,0666); //READ/WRITE access,
create the file upon no find, everyone can access
    pid_t pid = fork();
    if(pid>0){
        const char* parent_msg = "parentwrittentofile\n";
        write(fd,parent_msg,sizeof(char)*strlen(parent_msg));
        if(close(fd)){
            printf("Parent could not close file!");
        }
        wait(NULL);
    } else {
        const char* child_msg = "CHILDWRITTENTOFILE\n";
        write(fd,child_msg,sizeof(char)*strlen(child_msg));
        if(close(fd)){
            printf("Child could not close file!");
        }
    }
    return 0;
}
```

Upon running the code, we get the following output:

```
parentwrittentofile
CHILDWRITTENTOFILE
```

Which was not exactly what I expected, but close. As it turns out, the kernel imposes restritions on writing to files, to ensure all access to files to be atomic actions. So the relative order between the parent and the child is arbitary, but the indivitual letters is not.

Question 3

Here i used signals to do some IPC (Inter Process Communication) with the signal api. If the parent arrives first, it simply prints and sends a SIGUSR1 to the child, saying "im done". The child the executes the set signal handler, which unlocks its printing. The child is not allowed to print unless it has received the signal from the parent. Running it 10 times shows deterministic behaviour every time.

Here is the code:

```
int childPrinted = 0;

void childHandler(){
    printf("Goodbye from child!\n");
    childPrinted = 1;
}

int q3(void){
    //We now want to do the writing to terminal, and sychronize hello and goodbye between parent/child, without wait()
    //We can do it two ways, using waitpid() or global variable. waitpid() will be used here
    pid_t pid = fork();
    if(pid > 0){
        //Parent process
        printf("Hello from parent!\n");
        kill(pid,SIGUSR1);

    } else {
        signal(SIGUSR1,childHandler);
        while(!childPrinted);
    }
    return 0;
}
```

Some of the printouts:

```
➜  HW05 ./hw05
Hello from parent!
Goodbye from child!
➜  HW05 ./hw05
Hello from parent!
Goodbye from child!
➜  HW05 ./hw05
Hello from parent!
Goodbye from child!
➜  HW05 ./hw05
```

```
Hello from parent!
Goodbye from child!
```

Question 4

To call the exec and let the child process be replaced by /bin/ls we i simply forked out a child and let the parent wait, for then to setup the arguments for the child and call execl. Note the convenience of separating the fork and the execl as we can set up the enviornment and args for the new process, before calling exec.

Here is the code:

```
int q4(void){
    pid_t pid = fork();
    if(pid > 0){
        //Parent process
        wait(NULL);
        printf("Child process terminated");

    } else {
        //Child process
        const char* path = "/bin/ls";
        execl(path, "ls",NULL);

    }
    return 0;
}
```

The output was:

```
➜  HW05 ./hw05
hw05  hw05.c  hw5.md  q1  q2.c  testFile.txt
Child process terminated%
```

This is expected. There are different versions of exec, as one might need to specify a list of arguments, or a argument list right isnide the function call. Also, whether the path is relative, absolute or in relation to the $PATH variable is given by the different versions of exec.

Question 5

A program that uses wait() to wait for the child process to terminate is presented below. The wait() statement returns the pid of any child process it waited for and got terminated, and -1 if there are no child processes to wait for (that is, for any child process with no other child processes under it, it will return -1).

Here is the code:

```c
int q5(void){
    pid_t pid = fork();
    if(pid > 0){
        //Parent process
        printf("Parent wait returned: %d\n",wait(NULL));
        printf("Child process terminated");
        exit(0);

    } else {
        //Child process
        printf("Child wait returned: %d\n",wait(NULL));
        exit(0);
    }
    return 0;
}
```

The output was:

```
➜  HW05 ./hw05
Child wait returned: -1
Parent wait returned: 50111
Child process terminated%
```

Just as expected (the child process has no process to wait for, hence -1).