Practice: SIGCHLD orphan handler sleep pause waitpid

I was learning about *signal handler*, *sleep*, *pause* and *waitpid*. To better understand these concepts, I wrote the code:

Let's say I've installed two signal handler before `fork()`,

- child process print its ID in an infinite loop,
- parent process print its ID then sleep for 15s to get enough time waiting the child process caught SIGINT;
- In `sigint_handler`,
    - if the calling process is child, reset the default action associated with the `sig` by `signal(sig, SIG_DFL)`, then send `sig` to the calling process.
    - else print parent process ID or message `Who am I???`

To ensure the signal handlers are executed, 3 methods are used here: *sleep*, *pause*, and *waitpid*:

1. sleep

```
#include "csapp.h"

pid_t pid_child;
pid_t pid_parent;

void sigint_handler(int sig) {
        pid_t pid_sig = getpid(); //get the process ID of the calling process.
        if (pid_sig == pid_child) {
        printf("In sigint_handler, called by child: %d\n", pid_child);
        signal(sig, SIG_DFL);
        kill(pid_sig, sig);
        }
        else if(pid_sig == pid_parent) {
                printf("In sigint_handler, called by parent: %d\n", pid_parent);
        } else {
                printf("Who am I???\n");
        }
}

void sigchld_handler(int sig) {
        printf("In sigchld_handler, called by process: %d\n", getpid());
}
int main() {
        int status;
        Signal(SIGINT, sigint_handler);
        Signal(SIGCHLD, sigchld_handler);
        if ( (pid_child = fork() ) == 0) {
                pid_child = getpid();
                printf("Child ID: %d\n", pid_child);
```

```
                while(1) {
                        printf("In while loop... Child ID: %d  Parent ID is:
%d\n", getpid(), getppid());
                        sleep(1);
                }
        }
        pid_parent = getpid();
        printf("Parent ID: %d\n", pid_parent);
        sleep(10);
}
```

If we type `Ctrl + C` after executing the program 3s :

```
Parent ID: 19192
Child ID: 19193
In while loop... Child ID: 19193  Parent ID is: 19192
In while loop... Child ID: 19193  Parent ID is: 19192
In while loop... Child ID: 19193  Parent ID is: 19192
^CIn sigint_handler, called by child: 19193
In sigint_handler, called by parent: 19192
```

Here is why both child and parent called *sigint_handler*.

> If you do `fork` (without further `exec*`) after a signal handler has been registered, the same signal handler will be used in **both** parent and child processes. That is, if you do something other than `exit` in your `SIGINT` handler, neither parent nor child will exit (how `SIGINT` was sent is irrelevant here).
>
> If you mean a `SIGINT` sent from the *terminal* (by `vintr` character which is usually `Ctrl+C`): **it will be received by processes using the *terminal* as controlling terminal.** That is, unless you detach child or parent from controlling terminal, both will react to `Ctrl+C` by calling your `SIGINT` handler.

But why there is no message about *sigchld_handler* ? I got the message about *sigint_handler* called by child, then the child process was terminated, and kernel should have sent a SIGCHLD to parent process. You know, kernel send SIGCHLD to parent process when a process terminated normal or **abnormal**. That's weird...

The description of *sleep()* form man7.org can explain this well:

> `sleep()` causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or **until a signal arrives which is not ignored**.

In a nutshell, *sleep()* returned after we sent SIGINT to parent process by typing `Ctrl + C`.

What if parent exited before child ?

Let's say, the number of real-time seconds specified in seconds have elapsed:

```
Parent ID: 19770
Child ID: 19771
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
In while loop... Child ID: 19771  Parent ID is: 19770
hzhen@haipeng-pc:~/playground/csapp/ecf$ In while loop... Child ID: 19771
Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
^C
hzhen@haipeng-pc:~/playground/csapp/ecf$ In while loop... Child ID: 19771
Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In while loop... Child ID: 19771  Parent ID is: 1
In sigint_handler, called by child: 19771
```

After parent process exited, the child process was adopted by *init* process which ID is 1. Still remember that we installed `sigchld_handler` in parent process which have already gone away , the `init` process adopted the `orphan` which now running in the background (that's why the child won't response to the keyboard interrupt).

When we send `SIGINT` by `kill -2 ID` to the child process, child called `sigchld_handler` and that's what we have seen form terminal.

I strongly recommend you to read this article **Zombie vs Orphan vs Daemon Processes**.

2. pause

```
#include "csapp.h"

pid_t pid_child;
pid_t pid_parent;

void sigint_handler(int sig) {
        pid_t pid_sig = getpid(); //get the process ID of the calling process.
        if (pid_sig == pid_child) {
        printf("In sigint_handler, called by child: %d\n", pid_child);
        signal(sig, SIG_DFL);
        kill(pid_sig, sig);
        }
        else if(pid_sig == pid_parent) {
```

```
                printf("In sigint_handler, called by parent: %d\n", pid_parent);
        } else {
                printf("Who am I???\n");
        }
}

void sigchld_handler(int sig) {
        printf("In sigchld_handler, called by process: %d\n", getpid());
}
int main() {
        int status;
        Signal(SIGINT, sigint_handler);
        Signal(SIGCHLD, sigchld_handler);
        if ( (pid_child = fork() ) == 0) {
                pid_child = getpid();
                printf("Child ID: %d\n", pid_child);
                while(1) {
                        printf("In while loop... Child ID: %d  Parent ID is:
%d\n", getpid(), getppid());
                        sleep(1);
                }
        }
        pid_parent = getpid();
        printf("Parent ID: %d\n", pid_parent);
        pause();
}
```

If we type `Ctrl + C` after executing the program 4s :

```
Parent ID: 19395
Child ID: 19396
In while loop... Child ID: 19396  Parent ID is: 19395
In while loop... Child ID: 19396  Parent ID is: 19395
In while loop... Child ID: 19396  Parent ID is: 19395
In while loop... Child ID: 19396  Parent ID is: 19395
^CIn sigint_handler, called by child: 19396
In sigint_handler, called by parent: 19395
```

There is no message about *sigchld_handler* which isn't what we expected.

By reading the description of *pause()*, thing getting clearer:

> The *pause()* function suspends the calling thread until delivery of a signal whose action is either to execute a signal handler or to terminate the process.
>
> If the action is to terminate the process, *pause()* doesn't return. If the action is to execute a signal handler, *pause()* returns after the signal handler returns.

Therefore, parent process invoked `sigint_handler` after receiving SIGINT. The handler only print a message, so `pause()` returned after the `sigint_handler` returned, then the parent process exited. Meanwhile, the child process terminated after receiving `SIGINT`, and kernel sent SIGCHLD to the parent process. But parent process have already exited, the `sigchld_handler` will never be called ...

You can add another `pause()` to catch SIGCHLD after `pause()`, like :

```
pause(); // catch SIGINT
pause(); // catch SIGCHLD
```

3. waitpid

```
#include "csapp.h"

pid_t pid_child;
pid_t pid_parent;

void sigint_handler(int sig) {
        pid_t pid_sig = getpid(); //get the process ID of the calling process.
        if (pid_sig == pid_child) {
        printf("In sigint_handler, called by child: %d\n", pid_child);
        signal(sig, SIG_DFL);
        kill(pid_sig, sig);
        }
        else if(pid_sig == pid_parent) {
                printf("In sigint_handler, called by parent: %d\n", pid_parent);
        } else {
                printf("Who am I???\n");
        }
}

void sigchld_handler(int sig) {
        printf("In sigchld_handler, called by process: %d\n", getpid());
}
int main() {
        int status;
        Signal(SIGINT, sigint_handler);
        Signal(SIGCHLD, sigchld_handler);
        if ( (pid_child = fork() ) == 0) {
                pid_child = getpid();
                printf("Child ID: %d\n", pid_child);
                while(1) {
                        printf("In while loop... Child ID: %d  Parent ID is:
%d\n", getpid(), getppid());
                        sleep(1);
                }
        }
        pid_parent = getpid();
        printf("Parent ID: %d\n", pid_parent);
        waitpid(-1, &status, 0);
        if(WIFSIGNALED(status))
                printf("Child: %d terminated with status: %d\n", pid_child,
WTERMSIG(status));
        if(WIFEXITED(status))
                printf("Child: %d terminated with status: %d\n", pid_child,
WEXITSTATUS(status));

}
```

If we type `Ctrl + C` after executing the program 2s :

```
Parent ID: 19669
Child ID: 19670
In while loop... Child ID: 19670  Parent ID is: 19669
In while loop... Child ID: 19670  Parent ID is: 19669
^CIn sigint_handler, called by child: 19670
In sigint_handler, called by parent: 19669
In sigchld_handler, called by process: 19669
Child: 19670 terminated with status: 2
```

This is what we expected. Child process terminated with status 2 (which is the number of SIGINT).

Reference

pause

Zombie vs Orphan vs Daemon Processes