TD1 - Processes and signals

Exercise 1 - Who am I?

```
int main ()
{
    pid_t child = fork();
    if (child == 1) {
        perror("fork() error");
        exit(1);
    }

    // à ajouter pour la question (b)
    // printf("My PID is %d.\n", getpid());

    if (child == 0) {
        printf("Child process: my PID is %d.\n", getpid());

        // à enlever pour la question (b)
        exit(0);
    }

    printf("Now my PID is %d.\n", getpid());
    exit(0);
}
```

a. What is the output of this program ?

> This program outputs the child and parent PID, like so:
>
> ```
> Child process: my PID is 4321.
> Now my PID is 1234.
> ```
>
> The order of the prints being arbitrary.

b. What is the output, if we add line 10 and remove line 16 ?

> This program outputs the child PID 3 times and the parent PID twice, because the child process doesn't terminate until the last line, like so:
>
> ```
> My PID is 4321.
> My PID is 1234.
> Child process: my PID is 4321.
> Now my PID is 4321.
> Now my PID is 1234.
> ```

> The order of the prints being arbitrary.

Exercise 2 - Bob

```c
int sunny_days_in_London(int year) { /* 6 heures de calcul */ }
int second_year_presence(int day) { /* 6 heures de calcul */ }

int main ()
{
    int captains_age = 0;
    pid_t cpid = fork();

    if (cpid != 0)
        captains_age += sunny_days_in_London(1305);
    else
        captains_age += second_year_presence(1305);

    printf(
        "Le capitaine a %d ans et mon PID est %d\n",
        captains_age,
        (int) getpid()
    );

    return 0;
}
```

Explain Bob's mistake and propose a way to solve the problem using `exit()` and `wait()`. Detail the actions of both processes.

> Bob didn't make the child process exit after doing the calculation, so there are going to be 2 differents prints.

> Also, Bob forgot to wait for the child process to end before printing out the result, and to check for an exit code to check for errors.

> This program is going to output two lines, containing the 2 different calculations.

> To solve the problem, we could add `exit(age);` in the `if`, and a `wait()` before the `printf` statement.

Corrected program

```c
int sunny_days_in_London(int year) { /* 6 heures de calcul */ }
int second_year_presence(int day) { /* 6 heures de calcul */ }

int main ()
{
```

```c
    int captains_age = 0;
    int status;

    pid_t cpid = fork();

    if (cpid != 0) {
        captains_age += sunny_days_in_London(1305);
        status = 0;
        exit(captains_age); // or return age;
    } else
        captains_age += second_year_presence(1305);

    wait(&status);
    WIFEXITED(&status) {
        captains_age += WEXITSTATUS(&status);
    }

    printf(
        "Le capitaine a %d ans et mon PID est %d\n",
        captains_age,
        (int) getpid()
    );

    return 0;
}
```

Exercise 3 - Firefork()

```c
int main(int argc, char ** argv)
{
    pid_t id;
    int i, N = 0;

    if (argc > 1) N = atoi(argv[1]);

    for (i = 0; i < N; i++) {
        id = fork();

        if (id == 1) {
            perror("fork() error"); exit(1);
        }

        printf(
            "I am %d, son of %d.\n",
            getpid(),
            getppid()
        );
    }

    printf("%d out.\n", getpid());
```

```
    return 0;
}
```

Give a generic formula for the number of processes depending on N.

> This program doesn't check if you're the parent or a child process : so this is going to fork from the parent and the childs. Now let's make an example :

> Let's say we execute `./fforks 3` :

> `atoi('3') = 3` so the loop is going to go 3 times.

> 1st iteration :
> fork 1 child

> 2nd iteration :
> Both the parent and the child fork a process.
> We now have 4 total processes.

> 3rd iteration :
> Each 4 of the processes fork a child.
> We now have a total of 8 processes.

> You've probably noticed the pattern : this program makes `O(N)` child processes.

> This type of program is called a ***fork bomb***.

Exercise 4 - world! Hello,

```
int main ()
{
    pid_t id = fork();

    if (id == 0) {
        printf("Hello, ");
        exit(0);
    }

    // <-- TROU

    printf("world!");
    return 0;
}
```

We want to complete this program to make sure that the words are always printed in the right order, independently of scheduling decisions.

The system call `int sleep(int n)` puts the program in the waiting state for `n` seconds. Can we complete the program by calling `sleep(1)`? `sleep(100)`? Explain your answer.

> Completing the program by adding a `sleep` system call might work, but is totally unreliable, no matter the amount of seconds slept.

Can we complete the program with a wait() call? Substantiate your answer.

> Yes, a `wait` system call is the right option here : it allows the parent process to wait for a `SIGCHILD` signal, which is sent when the child process has finished execution.

> It is reliable and this way, we can reliably execute our program in a organized and pretictable way.

Exercise 5 - To infinity in eight seconds.

The following program increments a counter of type `unsigned int` and stops after the maximal value is reached and the counter goes back to zero.

```
volatile unsigned int count = 0;
volatile int step = 1;
//
// < TROU 1
//

int main() {
    //
    // < TROU 2
    //
    for (count = 1; count > 0; count += step);
    return 0;
}
```

We declare the variables `count` and `step` as `volatile` to prevent compiler optimisation (which otherwise might remove the loop entirely).

Complete the code so that count is reset to 1 when the process receives the SIGUSR1 signal.

```
volatile unsigned int count = 0;
volatile int step = 1;

void hnd1(int sig) { // signal handler
    count = 1;
}

int main() {
    // bind SIGUSR1 to handler hnd1
    signal(SIGUSR1, &hnd1);

    for (count = 1; count > 0; count += step);
    return 0;
}
```

Complete the code so that the increment step changes sign when the process receives SIGUSR2.

```
volatile unsigned int count = 0;
volatile int step = 1;

void hnd1(int sig) { // signal handler
    count = 1;
}

void hnd2(int sig) {
    step = -step;
}

int main() {
    // bind SIGUSR1 to handler hnd1
    signal(SIGUSR1, &hnd1);
    signal(SIGUSR2, &hnd2);

    for (count = 1; count > 0; count += step);
    return 0;
}
```

To ensure that our program behaves correctly, we want to watch the counter. A new system call comes in handy:

```
#include <unistd.h>
unsigned int alarm(unsigned int nb_sec);
```

The system call `alarm()` sets a timer that will send a `SIGALRM` signal to the calling process `nb_sec` seconds later. Any `alarm()` call cancels and replaces the previously set alarm; `alarm(0)` cancels the current alarm and does not set a new one.

Complete the code so that the current value of count is printed every second.

```
#include <unistd.h>

volatile unsigned int count = 0;
volatile int step = 1;

void hnd1(int sig) { // signal handler
    count = 1;
}

void hnd2(int sig) {
    step = -step;
}
```

```c
void hnd3(int sig) {
    printf("Count : %d", count);
    alarm(1);
}

int main() {
    // bind SIGUSR1 to handler hnd1
    signal(SIGUSR1, &hnd1);
    signal(SIGUSR2, &hnd2);
    signal(SIGALRM, &hnd3);

    alarm(1);

    for (count = 1; count > 0; count += step);
    return 0;
}
```

Exercise 6 - Et mon courroux, coucou!

Let us go back to Exercise 4. As we know, the parent process receives the `SIGCHLD` signal whenever its child process terminates. Let us see if we can use this to ensure the correct order of execution.

The primitive `int pause(void)` puts the calling process to sleep until it receives a signal which is neither ignored nor blocked. Can we complete the program with a `pause()` call? Explain your answer.

> No, because with the program as is, the `SIGCHILD` signal will get ingored by the `pause()` system call.

Can we complete the program with a `pause()` call and a simple handler for `SIGCHLD`, for example, a function that does nothing?

> Yes, but we need to make sure that our handler is defined before the `fork()`.

```c
void hnd(int sig) {}

int main()
{
    signal(SIGCHILD, hnd);
    // rest of program
}
```

Bob is not happy with our answers. What if we used our signal handler to avoid calling `pause()` if the signal has already arrived? To do that, we would need a global `flag` variable:

```c
volatile int flag = 0;
void handler_chld (int sig) { flag = 1; }
```

```c
int main ()
{
    signal(SIGCHLD, handler_chld);
    if (fork() == 0) { printf("Hello"); exit(0); }
    while (!flag) pause();
    printf(" world");
    return 0;
}
```

Why is it important to install the handler before calling `fork()`?

> Because otherwiser the handler might not be defined in the parent process when the child process exits.

Why is it important to test the flag in a `while` loop?

> It's important to do so because if the parent evaluates `flag` before the child process has ended it will get past the instruction and exit.

Does Bob's program guarantee that the words are printed in the right order?

> Yes, `while(!flag)` waits for the end of the child process before resuming execution. It is however very demanding for the machine : see `busy waiting`.

Exercise 7 - One, two, many.

Let us execute the following program :

```c
volatile int count = 0;
void handler(int sig) { count++; }

int main()
{
    int i;
    signal(SIGUSR1, handler);

    if (fork() == 0) {
        for (i = 0; i < 256; i++)
            kill(getppid(), SIGUSR1);
        exit(0);
    }
    wait(NULL);
    printf("Final: %d\n", count);
    return 0;
}
```

What is the output of this program? Explain the answer.

> `man 7 signal` gives us the following definition for `SIGUSR1` :

```
Signal      Norm     Action  Note
─────────────────────────────────────────
SIGUSR1     P1990    Term    User signal 1
```

> The `action` attribute is `term`, meaning XYZ...