Timothé Belcour
TP1

TP2 – Processus

Exercice 1

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

int carre(int x) {
    return x * x;
}

int main(void) {
    printf("[fonction] 5^2 = %d\n", carre(5));

    pid_t pid = fork();
    if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); }

    if (pid == 0) {
        exit(42);
    } else {
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf("[processus] code de retour du fils = %d\n", WEXITSTATUS(status));
        }
    }
    return 0;
}
```

```
timothebelcour@MacBook-Air TP2 % ./ex1
[fonction] 5^2 = 25
[processus] code de retour du fils = 42
```

La valeur de retour d'une fonction est transmise au sein du même processus elle permet à une fonction de renvoyer un résultat directement à celui qu'il l'appelle exemple : int r = carre(5);.

À l'inverse, le code de retour d'un processus est la valeur renvoyée au processus parent lors de la fin du programme. Il s'obtient via return dans main() ou exit(), et peut être récupéré par le parent grâce à wait().

Ainsi, la valeur de retour d'une fonction concerne la logique interne du programme, tandis que le code de retour d'un processus indique l'état de terminaison d'un programme à son parent.

Timothé Belcour
TP1

Exercice 2

1) Les différences de temps entre ces 5 commandes s'expliquent par la gestion mémoire et le moment d'initialisation de la chaîne.

- cmd2 est la plus lente car la chaîne est recopiée à chaque appel.
- cmd1 et cmd3 utilisent un pointeur, donc aucune copie n'est effectuée.
- cmd4 est statique : initialisée une seule fois, elle est plus rapide.
- cmd5 est une macro : le compilateur optimise tout, d'où la vitesse maximale.

2) Si la chaîne est plus longue, seul cmd2 devient plus lent, car c'est la seule version qui effectue une copie à chaque exécution.

Timothé Belcour
TP1

Exercice 3

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s cmd1 [cmd2 ...]\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("[parent] PID=%d, PPID=%d\n", getpid(), getppid());

    for (int i = 1; i < argc; ++i) {
        pid_t pid = fork();
        if (pid == -1) { perror("fork"); continue; }

        if (pid == 0) {
            printf("[fils-pre-exec] CMD=%s PID=%d PPID=%d\n", argv[i], getpid(), getppid());
            execlp(argv[i], argv[i], (char*)NULL);
            perror("execlp");
            _exit(127);
        }
    }
    int status;
    pid_t wpid;
    while ((wpid = wait(&status)) > 0) {
        if (WIFEXITED(status)) {
            printf("[wait] PID=%d exit=%d\n", wpid, WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("[wait] PID=%d signal=%d\n", wpid, WTERMSIG(status));
        }
    }
    return 0;
}
```

```
timothebelcour@MacBook-Air TP2 % ./ex3 ls pwd whoami

[parent] PID=78059, PPID=77437
[fils-pre-exec] CMD=ls PID=78060 PPID=78059
[fils-pre-exec] CMD=pwd PID=78061 PPID=78059
[fils-pre-exec] CMD=whoami PID=78062 PPID=78059
/Users/timothebelcour/BUT/Année 2/S1/Architecture des système/TP2
[wait] PID=78061 exit=0
timothebelcour
[wait] PID=78062 exit=0
ex1                     ex3                     ex5.c
ex1.c                   ex3.c                   TP2-processus.pdf
[wait] PID=78060 exit=0
```

Le programme crée un processus fils pour chaque commande donnée en argument grâce à fork() puis exécute chaque commande avec execlp().
Chaque fils affiche la commande correspondante ls, pwd, whoami, puis affiche son PID et son PPID.

Le père attend la fin de tous ses fils avec wait() et affiche leur code de sortie et pour finir il affiche exit=0 s'ils ont réussi.

Le résultat montre que les trois fils s'exécutent en parallèle, héritent du même parent et que le père synchronise correctement leur terminaison.

Exercice 4

Le programme affiche deux fois *«* Bonjour *»* car ce message est écrit avant le fork().
Lors de la duplication du processus, le fils reçoit une copie du tampon d'affichage du père, et les deux affichent donc le même texte.
Les adresses de la variable n sont identiques, car le fork() duplique la mémoire du processus.
En revanche, les valeurs de n peuvent être différentes, puisque le père et le fils modifient chacun leur propre copie de cette variable.
L'ordre d'affichage peut varier selon la planification du système, mais l'ensemble du comportement observé est logique et conforme au fonctionnement du fork().

Timothé Belcour
TP1

Exercice 5

```c
#define _XOPEN_SOURCE 700
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

static int open_out(const char *path) {
    int fd = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) { perror(path); exit(EXIT_FAILURE); }
    return fd;
}

int main(void) {

    pid_t p2 = fork();
    if (p2 == -1) { perror("fork P2"); exit(EXIT_FAILURE); }

    if (p2 == 0) {
        int f = open_out("f");
        if (dup2(f, STDOUT_FILENO) == -1) { perror("dup2 P2 stdout"); exit(EXIT_FAILURE); }
        close(f);
        dprintf(STDOUT_FILENO, "[P2] stdout -> f\n");
        _exit(0);
    }

    pid_t p3 = fork();
    if (p3 == -1) { perror("fork P3"); exit(EXIT_FAILURE); }

    if (p3 == 0) {
        int g = open_out("g");
        int h = open_out("h");
        if (dup2(g, STDOUT_FILENO) == -1) { perror("dup2 P3 stdout"); exit(EXIT_FAILURE); }
        if (dup2(h, STDERR_FILENO) == -1) { perror("dup2 P3 stderr"); exit(EXIT_FAILURE); }
        close(g); close(h);
        dprintf(STDOUT_FILENO, "[P3] stdout -> g\n");
        dprintf(STDERR_FILENO, "[P3] stderr -> h\n");
        _exit(0);
    }

    int status;
    while (wait(&status) > 0) {  }

        printf("[P3] stdout redirigé vers g.txt (PID=%d, PPID=%d)\n", getpid(), getppid());
        fprintf(stderr, "[P3] stderr redirigé vers h.txt (PID=%d, PPID=%d)\n", getpid(), getppid());
        _exit(0);
    }
    int status;
    while (wait(&status) > 0);

    printf("[P1] stdout sur le terminal (PID=%d)\n", getpid());
    fprintf(stderr, "[P1] stderr sur le terminal (PID=%d)\n", getpid());

    return 0;
}
```

Timothé Belcour
TP1

```
timothebelcour@mac Architecture-des-systeArchitecture-des-systeme % cd TP2
timothebelcour@mac TP2 % gcc ex5.c -o ex5

timothebelcour@mac TP2 % ./ex5

[P1] PID=1911 (parent), PPID=82851
[P1] stdout sur le terminal (PID=1911)
[P1] stderr sur le terminal (PID=1911)
```

Le programme crée deux processus fils à partir d'un parent.
Le premier fils (P2) redirige sa sortie standard (stdout) vers le fichier f.txt.
Le second (P3) redirige sa sortie standard vers g.txt et sa sortie d'erreur (stderr) vers h.txt.
Le parent (P1) conserve ses sorties normales sur le terminal.
Ainsi, chaque processus écrit dans une destination différente, ce qui montre comment on peut rediriger les flux d'entrée/sortie à l'aide de la fonction dup2().