TP1 ex 2

```c

/*
 * TP1 ex 2
 * Programme qui crée un processus fils.
 * Le père affiche les nombres impairs, le fils affiche les nombres pairs,
 * les deux en parallèle avec une pause d'une seconde entre chaque affichage.
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork(); // Création d'un processus fils

    if (pid == 0) {
        // Code exécuté par le processus fils
        printf("Processus fils (pairs) :\n");

        for (int i = 0; i < 5; i++) {
            printf("%d ", 2 * i); // Affiche les nombres pairs
            sleep(1); // Pause de 1 seconde
        }

        printf("\n");
    } else {
        // Code exécuté par le processus père
        printf("Processus père (impairs) :\n");

        for (int i = 0; i < 5; i++) {
            printf("%d ", 2 * i + 1); // Affiche les nombres impairs
            sleep(1); // Pause de 1 seconde
        }

        printf("\n");
    }

    return 0; // Terminaison des deux processus
}
```

TP1 ex 3

```c
/*
 * TP1 ex 3
 * Programme qui crée un processus fils, exécute la commande `ps` dans le fils,
 * et attend dans le père que le fils termine pour afficher son code de retour.
 */

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


int main() {
    pid_t pid;
    int status;

    pid = fork(); // Création d'un processus fils
    if (pid == 0) {
        // Processus fils : exécute la commande `ps`
        execlp("ps", "ps", NULL);
        exit(0); // Si execlp échoue (par sécurité)

    } else {

        // Processus père : attend la fin du processus fils
        waitpid(pid, &status, 0);

        // Vérifie si le processus fils s'est terminé correctement
        if (WIFEXITED(status)) {
            printf("Le processus fils s'est terminé avec le code de retour : %d\n", WEXITSTATUS(status));
        } else {
            printf("Le processus fils ne s'est pas terminé normalement.\n");
        }
    }

    return 0;
}
```

TP2

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

/*
 * Programme qui crée deux threads.
 * Le premier thread affiche des points à intervalles réguliers.
 * Le second attend qu'un utilisateur entre un caractère, puis arrête le programme.
 *
 * Utilise un thread détaché pour que le thread puisse fonctionner indépendamment
 * du thread principal, sans nécessiter d'appel à pthread_join pour libérer ses ressources.
 */
void points(void arg) {
    setbuf(stdout, NULL); // Désactive le buffering pour un affichage immédiat
    while (1) {
        printf(".");
        sleep(1); // Pause de 1 seconde
    }
    return NULL;
}

void entree(void arg) {
    char c;
    printf("Tapez un caractère : ");
    c = getchar(); // Lit un caractère
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_attr_t attr;

    printf("Les 2 threads sont lancés\n");

    // Initialise les attributs de thread
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // Threads détachés

    // Un thread détaché permet au système de libérer automatiquement ses ressources
    // lorsqu'il termine, sans besoin d'appeler pthread_join.

    // Crée le thread 2
    if (pthread_create(&thread2, &attr, entree, NULL) != 0) {
        perror("Échec de la création du thread 2");
        exit(1);
    }

    // Crée le thread 1
    if (pthread_create(&thread1, &attr, points, NULL) != 0) {
        perror("Échec de la création du thread 1");
        exit(1);
    }

    // Attendre que l'utilisateur entre un caractère (via thread 2)
    pthread_join(thread2, NULL);

    printf("On va s'arrêter là\n");

    // Annule le thread 1 (qui affiche les points en boucle)
    pthread_cancel(thread1);

    pthread_attr_destroy(&attr); // Nettoie les attributs
    return 0;
}
```

TP3

```c
#include <stdio.h>
#include <pthread.h>

/*
 * Programme qui crée deux threads incrémentant un compteur partagé.
 * Utilise un mutex pour synchroniser les threads et éviter les conflits
 * lors de l'accès au compteur global.
 *
 * Le mutex (verrou) garantit qu'une seule section critique (l'accès au compteur)
 * est exécutée à la fois, assurant l'intégrité des données en environnement multithread.
 */

unsigned long cpt = 0; // Compteur global
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex pour synchronisation

void compteur(void arg) {
    for (int i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&mutex); // Verrouille l'accès au compteur
        cpt++; // Incrémente le compteur
        pthread_mutex_unlock(&mutex); // Libère le verrou
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Crée deux threads qui partagent le même compteur
    pthread_create(&thread1, NULL, compteur, NULL);
    pthread_create(&thread2, NULL, compteur, NULL);

    // Attend la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Affiche la valeur finale du compteur
    printf("Valeur finale du compteur = %lu\n", cpt);

    pthread_mutex_destroy(&mutex); // Détruit le mutex
    return 0;
}
```

TP4

```c
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <stdlib.h>

/*
 * Programme qui utilise un pipe pour communiquer entre un processus père et un fils.
 * Le père lit des caractères au clavier et les envoie au fils via le pipe.
 * Le fils reçoit les caractères, les convertit en majuscules et les affiche.
 */
int main() {
    int fd[2]; // Tableau pour le pipe (fd[0] = lecture, fd[1] = écriture)
    pid_t pid;
    char c;

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1); // Erreur si la création du pipe échoue
    }

    pid = fork(); // Création d'un processus fils
    if (pid == -1) {
        perror("fork");
        exit(1); // Erreur si la création du processus échoue
    }

    if (pid == 0) {
        // Code du processus fils
        close(fd[1]); // Le fils ferme l'écriture dans le pipe puisqu'on ne va pas écrire

        while (read(fd[0], &c, 1) > 0) {
            putchar(toupper(c)); // Convertit et affiche en majuscules
        }

        close(fd[0]); // Le fils ferme la lecture une fois terminé

        fflush(stdout);
    } else {
        // Code du processus père
        close(fd[0]); // Le père ferme la lecture dans le pipe puisqu'on ne va pas lire

        while ((c = getchar()) != EOF) {
            write(fd[1], &c, 1); // Écrit les caractères lus dans le pipe
        }

        close(fd[1]); // Ferme l'écriture une fois terminé
    }

    return 0;
}
```

TP5 ex 5

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

/*
 * Programme qui crée un processus fils et utilise des signaux pour
 * communiquer entre le père et le fils. Le père envoie un signal SIGUSR1
 * après un délai de 5 secondes, et le fils attend ce signal pour terminer.
 */

// Variable globale pour synchroniser l'attente du signal dans le processus fils
volatile sig_atomic_t attente = 0;

// Gestionnaire pour le signal SIGUSR1
// Ce gestionnaire est appelé lorsque le signal SIGUSR1 est reçu
void traiter_sigusr1(int sig) {
    attente = 1; // Change la variable globale pour indiquer que le signal a été reçu
}

int main() {
    pid_t pid;

    // Création du processus fils
    pid = fork(); // Le processus père se divise en deux (père et fils)

    if (pid == -1) {
        // En cas d'erreur lors de la création du processus
        perror("Erreur lors du fork");
        exit(EXIT_FAILURE);

    } else if (pid == 0) {
        // Code exécuté par le processus fils

        // Préparation du gestionnaire de signal pour SIGUSR1
        struct sigaction sa;
        sa.sa_handler = traiter_sigusr1; // Spécifie la fonction qui gère SIGUSR1
        sa.sa_flags = 0; // Aucun drapeau spécifique pour l'action
        sigemptyset(&sa.sa_mask); // Aucun signal ne sera masqué lors de l'exécution du gestionnaire
        sigaction(SIGUSR1, &sa, NULL); // Associe le gestionnaire de signal à SIGUSR1

        printf("Fils : en attente du signal SIGUSR1...\n");

        // Boucle d'attente du signal
        while (!attente) {
            // Attente active : boucle vide tant que 'attente' est 0
        }

        // Une fois le signal reçu, 'attente' devient 1 et on sort de la boucle
        printf("Fils : signal reçu, fin du processus fils.\n");
        exit(EXIT_SUCCESS); // Le fils termine son exécution
```

```c
    } else {
        // Code exécuté par le processus père

        printf("Père : PID du fils = %d\n", pid); // Affiche le PID du processus fils

        // Le père attend 5 secondes avant d'envoyer le signal
        sleep(5); // Attente de 5 secondes

        // Envoi du signal SIGUSR1 au processus fils
        printf("Père : envoi du signal SIGUSR1 au fils.\n");
        kill(pid, SIGUSR1); // Envoi du signal SIGUSR1 au fils

        // Le père attend la fin du processus fils avant de continuer
        wait(NULL); // Attente de la terminaison du processus fils
        printf("Père : fils terminé, fin du processus père.\n");
    }

    return 0; // Fin du programme
}
```

Flag de signal :

- `SA_RESTART` : Si cette option est définie, certaines fonctions bloquantes (comme `read()` ou `wait()`) seront automatiquement redémarrées si elles sont interrompues par le signal.

Masque de signal :

- `sa_mask` est un masque de signaux qui seront **bloqués** pendant l'exécution du gestionnaire de signal.
- Par exemple, tu peux empêcher qu'un autre signal soit reçu pendant que le gestionnaire est en cours d'exécution.

Exemple :

```c
    struct sigaction sa;
    sa.sa_handler = traiter_sigusr1;
    sa.sa_flags = SA_RESTART;  // Redémarre les appels système interrompus
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGINT);  // Bloque SIGINT pendant le gestionnaire
    sigaction(SIGUSR1, &sa, NULL);
```