Exercices

Création de processus

Père attend fin fils

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

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
#define SLEEP_TIME 30

int main(void)
{
    pid_t pid_fils;
    pid_t ret_wait;
    int etat;

    // On affiche le pid du processus pere
    printf("Je suis le processus pere de pid %d\n", getpid());
    // On cree un processus fils
    pid_fils = fork();

    switch (pid_fils) {
    // Si le pid du fils est 0, on est dans le processus fils
    case 0:
        printf("*** FILS ***\n");
        // On affiche le pid du processus fils
        printf("Processus fils de pid %d\n", getpid());
        // On affiche le pid du processus pere
        printf("Pere de pid %d\n", getppid());
        printf("Je vais dormir 30 secondes ...\n");
        // On fait dormir le processus fils pendant 30 secondes
        sleep(SLEEP_TIME);
        printf("Je me reveille\n");
        printf("Je termine mon execution par un `return EXIT_SUCCESS`\n");
        // On termine le processus fils
        return EXIT_SUCCESS;
    // Si le pid du fils est -1, il y a eu une erreur au fork
    case -1:
        perror("Le fork a echoue");
        return EXIT_FAILURE;
    // Sinon, on est dans le processus pere
    default:
        printf("*** PERE ***\n");
        // On affiche le pid du processus pere
        printf("Processus pere de pid %d\n", getpid());
        // On affiche le pid du processus fils
        printf("Fils de pid %d\n", pid_fils);
        printf("J'attends la fin de mon fils...\n");
```

```
        // On attend la fin du processus fils
        ret_wait = wait(&etat);
        // On affiche le pid du processus fils qui est termine
        printf("Mon fils de pid %d est termine\n", ret_wait);
        // On affiche l'etat de retour du processus fils
        printf("Son etat etait : %04x\n", etat);
    }
    return EXIT_SUCCESS;
}
```

Un père et deux fils

```
int main(){
    pid_t pid1 = fork();
    if (pid1 >0){
        printf("je suis le pere %d \n",getpid());
        pid_t pid2 = fork();
        if (pid2 == 0) {
            printf("je suis le fils n°2 mon PID est : %d \n",getpid());
            sleep(2);
        }else if (pid1 >0){
            pid_t temp1 = wait(NULL);
            printf("mon fils n°%d est mort\n",temp1);
            pid_t temp2 = wait(NULL);
            printf("mon fils n°%d est mort\n",temp2);
        }else{
        printf("ERREUR dans fork\n");
    }
    }else if(pid1 == 0){
        printf("je suis le fils n°1 mon PID est : %d \n",getpid());
        sleep(3);
    }else{
        printf("ERREUR dans fork\n");
    }
    return 0;
}
```

Signaux

Synchronisation comptage père-fils

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

void empty_funct(int){}

int main(void){
    // Ecrire un programme qui crée deux processus, père et fils. Le père affiche les nombres entiers impairs compris entre 1 et 100, alors que le fils affiche les entiers pairs compris dans le même intervalle. Synchroniser les processus à l'aide des signaux pour que le résultat d'affichage soit : 1 2 3 ... 100
    pid_t pid_pere = getpid();

    // On capture le signal SIGUSR1
    signal(SIGUSR1, empty_funct);

    printf("Je suis le processus pere de pid %d\n", getpid());
    // On cree un processus fils
    pid_t pid_fils = fork();

    switch (pid_fils) {
    // Si le pid du fils est 0, on est dans le processus fils
    case 0:
        for(int j = 2; j <= 100; j+=2){
            printf("%d\n", j);
            kill(pid_pere, SIGUSR1);
            pause();
        }
        printf("end of child process\n");
        return EXIT_SUCCESS;
    // Si le pid du fils est -1, il y a eu une erreur au fork
    case -1:
        perror("Le fork a echoue");
        return EXIT_FAILURE;
    // Sinon, on est dans le processus pere
    default:
        for(int i = 1; i <= 100; i+=2){
            printf("%d\n", i);
            kill(pid_fils, SIGUSR1);
            pause();
        }
        kill(pid_fils, SIGUSR1);
        printf("end of parent process\n");
    }
```

```
    return EXIT_SUCCESS;
}
```

Roulette Russe

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

#define TAILLE_MAX 2

int KILLED = 0;

void empty_funct(){}

void living(){KILLED++;}

int lire_valeur(const char *path){
    FILE *fichier;
    char chaine[TAILLE_MAX];
    int valeur;
    fichier = fopen(path, "r");
    if (fichier != NULL) {
        /* On lit au maximum TAILLE_MAX caractères du fichier, ce qui est lu est
stocké dans `chaine` */
        fgets(chaine, TAILLE_MAX, fichier);
        fclose(fichier);
        valeur = atoi(chaine);
    }
    return valeur;
}

void ecrire_valeur(const char *path, int valeur){
    FILE *fichier = fopen(path, "w");
    if (fichier != NULL) {
        fprintf(fichier, "%d", valeur);
        fclose(fichier);
    }
}

int main(void){
    // On capture le signal SIGUSR1 (reveil)
    signal(SIGUSR1, empty_funct);

    // On capture le signal SIGUSR2 (fils mort ou vivant)
    signal(SIGUSR2, living);

    // On print le pid du pere
    printf("Je suis le processus pere de pid %d\n\n", getpid());
```

```
    /*---------On cree 6 processus fils--------------*/
    pid_t pids[6];
    int nprocesses = 0;
    while(nprocesses < 6)
    {
        // on cree un processus fils et on stocke son pid dans le tableau
        pids[nprocesses] = fork();
        switch (pids[nprocesses]) {
        // Si le pid du fils est 0, on est dans le processus fils
        case 0 :
            ;//On ne fait rien car on ne peut affecter directement après un :
            // on stocke son id
            int id = nprocesses+1;
            // on attend le signal du père pour jouer
            pause();
            // on lit la valeur du barillet
            int valeur_roulette = lire_valeur("barillet.txt");
            // Si la valeur du barillet est égale à son id, il se tue et réveille
le père
            if (valeur_roulette == id){
                //kill itself
                printf("\nLe processus %d est mort\n\n", id);
                kill(getppid(), SIGUSR2);
                kill(getpid(), SIGKILL);
            // Sinon, il envoie un signal au père pour dire qu'il est toujours
vivant
            } else {
                //send signal that still alive
                kill(getppid(), SIGUSR1);
            }
            return EXIT_SUCCESS;
        case -1 :
            perror("Le fork a echoue");
            return EXIT_FAILURE;
        default :
            //on boucle
            nprocesses++;
        }
    }


    /*-------------On joue----------------*/
    // add the number in the file
    char path[13]="barillet.txt";
    // on tire un nombre random entre 1 et 6 après avoir initialisé le pseudo-
random
    srand(time(NULL));
    int nombre = (rand() % 6) + 1;
    printf("La valeur du barillet est %d\n\n", nombre);
    // on écrit le nombre dans le fichier
    ecrire_valeur(path, nombre);
    nprocesses = 0;
```

```
    while (nprocesses < 6)
    {
        // Si aucun processus n'est mort, on fait jouer les fils les uns après les
autres
        if(KILLED == 0) {
            // on reveille le fils
            printf("On fait jouer le processus %d de pid %d \n", nprocesses+1,
pids[nprocesses]);
            kill(pids[nprocesses], SIGUSR1);
            pause();
        } else {
            // Sinon, on tue tous les processus restants
            printf("On met fin au jeu pour le processus %d de pid %d \n",
nprocesses+1, pids[nprocesses]);
            kill(pids[nprocesses], SIGKILL);
        }
        nprocesses++;
    }
    return EXIT_SUCCESS;
}
```

Tubes

Serveur et client

```c
/* Objectif :
* cree un tube /tmp/fifo
* Ouvre le tube en lecture
* Attend un texte du client
* Affiche a l'ecran le texte du client
* Se met en attente d'un nouveau texte du client
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#define SIZE 100

int main() {
    // Crée le tube
    char path[] = "/tmp/fifo";
    int mode = 0666;
    mkfifo(path, mode);
    printf("Tube créé\n\n");

    // Ouvre le tube en lecture
    FILE *tube = NULL;
    tube = fopen(path, "w+");
    if (tube == NULL) {
        printf("Erreur d'ouverture du tube\n\n");
        exit(1);
    }
    printf("Tube ouvert en lecture, en attente d'un client...\n\n");

    // Attend un texte du client
    char buffer[SIZE];
    fgets(buffer, SIZE, tube);
    printf("Buffer : %s\n\n", buffer);
    for(int i = 0; i < 10; i++) {
        fgets(buffer, SIZE, tube);
        if(buffer[0] != '\0') {
            // Affiche à l'écran le texte du client
            printf("%s\n",buffer);
            if (buffer[0] == 'q' && buffer[1] == '\0') {
                fclose(tube);
                unlink(path);
                exit(0);
            }
            // Se met en attente d'un nouveau texte du client
```

```
            fseek(tube,0,0);
            fputs("\0",tube);
        }
    }
}
```

```c
/* Objectifs :
* Ouvre en ecriture seule le tube /tmp/fifo. Echec si le tube n'existe pas
* Ecrit un texte passé en paramètre dans le tube
*/
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 100

int main(int argc, char const *argv[])
{
    if(argc!=2){printf("Usage : ./client <message>\n");exit(1);}
    FILE *fic=NULL;
    fic=fopen("/tmp/fifo","w");
    if(fic==NULL){printf("Erreur ouverture !\n");exit(1);}
    //int fputs(const char *s, FILE *stream

    fputs(argv[1],fic);
    //fseek(fic,0,0);
    fputs("\0",fic);


    fclose(fic);
    return 0;
}
```

Files de messages

Serveur et client

Serveur

```c
#include "calcul.h"

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// Declaration des variables globales
int msg_id_send;
int msg_id_receive;

/**
 * \fn void raz_msg()
 * \brief Fonction de suppression de la file de message
 * Supprime la file de message à la reception de n'importe quel signal
 */
void raz_msg() {
    printf("Suppression de la file de message!\n");
    msgctl(msg_id_send, IPC_RMID, NULL);
    msgctl(msg_id_receive, IPC_RMID, NULL);
    exit(0);
}

/**
 * \fn int calculation_mode()
 * \brief Fonction de calcul
 * \return EXIT_SUCCESS si le calcul s'est bien passé, EXIT_FAILURE sinon
*/
int calculation_mode(){
    msg_struct msg;
    int i_sig;
    int result;

    // Ouverture de la file de message
    key_t clef_send = ftok("calcul.h", 1);

    /* Creation de la file de message
    * IPC_CREAT = creation de la file de message
    * 0660 = droits d'acces (dialogue avec les processus d’un même ensemble
d’applications)
```

```
    */
    msg_id_send = msgget(clef_send,IPC_CREAT | 0660);
    // Verification de l'ouverture de la file de message
    if(msg_id_send == -1){
        printf("Erreur lors de la création de la file de message\n");
        return EXIT_FAILURE;
    } else {
        printf("File de message créée avec succès\nSERVEUR: pret!\n");
    }

    /* Pour tous les signaux, on appelle la fonction raz_msg
    * NSIG = nombre de signaux du systeme
    */
    for (i_sig = 0 ; i_sig < NSIG ; i_sig++) {
        signal(i_sig, raz_msg);
    }
    msg_struct reception;
    while (1) {
        /* reception */
        /* Ouverture de la file
        * reception = structure de reception (vide au début)
        * sizeof(client_charact) = taille de la structure de reception
        * 1 = type de message à recevoir
        * 0 = bloquant
        */
        if(msgrcv(msg_id_send, &reception,sizeof(client_calculation),1L,0) == -1){
            perror("Erreur lors de la reception du message\n");
            return EXIT_FAILURE;
        }
        printf("SERVEUR: reception d'une requete de la part de: %d, contenant
l'operation suivante:\n %d %c %d\n\n", reception.client_c.pid,
reception.client_c.op1, reception.client_c.op, reception.client_c.op2);

        /* preparation de la reponse */
        msg.type=reception.client_c.pid;
        msg.client_c.pid=getpid();

        // Calcul du resultat
        switch (reception.client_c.op)
        {
        case '/':
            result=reception.client_c.op1 / reception.client_c.op2;
            break;
        case 'x':
            result=reception.client_c.op1 * reception.client_c.op2;
            break;
        case '+':
            result=reception.client_c.op1 + reception.client_c.op2;
            break;
        case '-':
            result=reception.client_c.op1 - reception.client_c.op2;
            break;
        default:
            printf("Erreur opération non reconnue\n");
```

```
            exit(1);
            break;
        }

        msg.client_c.result=result;

        /* envoi de la reponse */
        msgsnd(msg_id_send, &msg, sizeof(client_calculation),0);
    }

    return EXIT_SUCCESS;
}

/**
 * \fn int translation_mode()
 * \brief Fonction de traduction du message en majuscule
 * \return EXIT_SUCCESS si la traduction s'est bien passée, EXIT_FAILURE sinon
*/
int translation_mode(){
    msg_struct msg, reception;
    int i_sig;

    // Ouverture des deux files de message
    key_t clef_receive = ftok("calcul.h", 1);
    key_t clef_send = ftok("calcul.h", 2);

    /* Creation de la file de message
    * IPC_CREAT = creation de la file de message
    * 0660 = droits d'acces (dialogue avec les processus d’un même ensemble
d’applications)
    */
    msg_id_receive = msgget(clef_receive,IPC_CREAT | 0660);
    msg_id_send = msgget(clef_send,IPC_CREAT | 0660);

    // Verification de l'ouverture de la file de message
    if(msg_id_send == -1 || msg_id_receive == -1){
        printf("Erreur lors de la création de la file de message\n");
        return EXIT_FAILURE;
    } else {
        printf("File de message créée avec succès\nSERVEUR: pret!\n");
    }

    /* Pour tous les signaux, on appelle la fonction raz_msg
    * NSIG = nombre de signaux du systeme
    */
    for (i_sig = 0 ; i_sig < NSIG ; i_sig++) {
        signal(i_sig, raz_msg);
    }

    while (1) {
        /* Ouverture de la file
        * reception = structure de reception (vide au début)
        * sizeof(client_charact) = taille de la structure de reception
        * 1 = type de message à recevoir
```

```
        * 0 = bloquant
        */
        if(msgrcv(msg_id_receive, &reception,sizeof(client_translation),2L,0) ==
-1){
            perror("Erreur lors de la reception du message\n");
            return EXIT_FAILURE;
        }
        printf("SERVEUR: reception d'une requete de type %ld de la part de %d :
%s\n", reception.type, reception.client_t.pid, reception.client_t.string);

        /* preparation de la reponse */
        msg.type=reception.client_t.pid;
        msg.client_t.pid=getpid();

        // Arrêt du serveur si on reçoit un '@'
        if(reception.client_t.string[0] == '@' && reception.client_t.string[1] ==
'\0'){
            printf("Réception d'un '@', arrêt du serveur...\n");
            strcpy(msg.client_t.string,"Arrêt du serveur");
            msgsnd(msg_id_send, &msg, sizeof(client_translation),0);
            exit(EXIT_SUCCESS);
        }
        else {
            int i = 0;
            // On traduit la chaine en majuscule
            for(i = 0; reception.client_t.string[i] != '\0'; i++){
                msg.client_t.string[i] = toupper(reception.client_t.string[i]);
            }
            msg.client_t.string[i] = '\0';

            /* envoi de la reponse */
            printf("SERVEUR: envoie de la reponse au client %d : %s\n",
msg.client_t.pid, msg.client_t.string);
            msgsnd(msg_id_send, &msg, sizeof(client_translation),0);
        }
    }

    return EXIT_SUCCESS;
}

/**
 * \fn int main(int argc, char const *argv[])
 * \brief Fonction principale du serveur
 * \param argc Nombre d'arguments
 * \param argv Tableau des arguments
 * \return EXIT_SUCCESS si le serveur s'est bien lancé, EXIT_FAILURE sinon
*/
int main(int argc, char const *argv[]){
    if(argc == 2 && strcmp(argv[1],"-c") == 0){
        printf("Calculation mode\n");
        calculation_mode();
    } else if(argc == 2 && strcmp(argv[1],"-t") == 0){
        printf("Translation mode\n");
        translation_mode();
```

```
    } else {
        printf("Usage:\nCalculation mode : %s -c\nTranslation mode : %s -t\n",
argv[0], argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Client

```
#include "calcul.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/**
 * \fn int calculation_mode(char const *argv[])
 * \brief Fonction de calcul
 * \param argv[] : tableau des arguments pour récupérer les opérandes et
l'opérateur
 * \return EXIT_SUCCESS si le calcul s'est bien passé, EXIT_FAILURE sinon
*/
int calculation_mode(char const *argv[]){
    // Declaration des variables
    int msg_id;
    msg_struct msg;

    // Ouverture de la file de message
    key_t clef = ftok("calcul.h", 1);
    msg_id = msgget(clef, 0);

    // Verification de l'ouverture de la file de message
    if(msg_id == -1){
        printf("Erreur lors de la création de la file de message\n");
        return EXIT_FAILURE;
    } else {
        printf("File de message ouverte avec succès\n");
    }

    // Affichage de l'operation effectuée
    printf("CLIENT %d: preparation du message contenant l'operation suivante: %d
%c %d\n", getpid(), atoi(argv[2]), argv[3][0], atoi(argv[4]));

    // Initialisation du message
    msg.type = 1;
    // Un client est identifié par son pid
```

```
    msg.client_c.pid = getpid();
    msg.client_c.op1 = atoi(argv[2]);
    msg.client_c.op = argv[3][0];
    msg.client_c.op2 = atoi(argv[4]);
    // Initialisation du resultat à 0 pour éviter les erreurs de lecture
    msg.client_c.result = 0;

    /* Envoie du message
    * options = 0
    * on récupère la taille du message grace au sizeof() de la structure
client_charact, donc on ne prend pas en compte le type du message dans la taille
    */
    msgsnd(msg_id, &msg, sizeof(msg.client_c), 0);

    // Reception de la reponse du serveur, d'identifiant le pid du client
    msgrcv(msg_id, &msg, sizeof(msg.client_c), getpid(), 0);
    // Affichage du resultat
    printf("CLIENT: resultat recu depuis le serveur %d : %d\n", msg.client_c.pid,
msg.client_c.result);

    return EXIT_SUCCESS;
}

/**
 * \fn int translation_mode(char const *argv[])
 * \brief Fonction de traduction
 * \param argv[] : tableau des arguments pour récupérer le texte à traduire
 * \return EXIT_SUCCESS si la traduction s'est bien passée, EXIT_FAILURE sinon
*/
int translation_mode(char const *argv[]){
    // Declaration des variables
    int msg_id_send, msg_id_receive;
    msg_struct msg;

    // Creation des clefs (1 ou 2 = identifiant)
    key_t clef_send = ftok("calcul.h", 1);
    key_t clef_receive = ftok("calcul.h", 2);
    // Ouverture de la file de message (0 = options)
    msg_id_send = msgget(clef_send, 0);
    msg_id_receive = msgget(clef_receive, 0);

    // Verification de l'ouverture de la file de message
    if(msg_id_send == -1 || msg_id_receive == -1){
        printf("Erreur lors de la création de la file de message\n");
        return EXIT_FAILURE;
    } else {
        printf("File de message ouverte avec succès\n");
    }

    // Affichage de l'operation effectuée
    printf("CLIENT %d: preparation du message contenant le texte suivant : %s\n",
getpid(), argv[2]);

    // Initialisation du message
```

```
    msg.type = 2;
    // Un client est identifié par son pid
    msg.client_t.pid = getpid();

    // Recopie de la chaine de caractere
    strncpy(msg.client_t.string, argv[2], strlen(argv[2]));

    /* Envoie du message
    * options = 0
    * on récupère la taille du message grace au sizeof() de la structure
client_charact, donc on ne prend pas en compte le type du message dans la taille
    */
    printf("Message send : %d\n",msgsnd(msg_id_send, &msg,
sizeof(client_translation), 0));

    // Reception de la reponse du serveur, d'identifiant le pid du client
    printf("Message received : %ld\n",msgrcv(msg_id_receive, &msg,
sizeof(client_translation), getpid(), 0));
    // Affichage du resultat
    printf("CLIENT: reponse recu depuis le serveur %d : %s\n", msg.client_t.pid,
msg.client_t.string);

    return EXIT_SUCCESS;
}

/**
 * \fn int main(int argc, char const *argv[])
 * \brief Fonction principale
 * \param argc : nombre d'arguments
 * \param argv[] : tableau des arguments
 * \return EXIT_SUCCESS si le programme s'est bien passé, EXIT_FAILURE sinon
*/
int main(int argc, char const *argv[])
{
    // Verification des arguments
    if(argc == 5 && strcmp(argv[1],"-c") == 0){
        // Verification de l'absence de division par 0
        if(atoi(argv[4]) == 0 && argv[3][0] == '/'){
            printf("Division par 0 impossible\n");
            return EXIT_FAILURE;
        }
        printf("Calculation mode\n");
        calculation_mode(argv);
    } else if(argc == 3 && strcmp(argv[1],"-t") == 0){
        if(strlen(argv[2]) > 80){
            printf("Texte trop long : longueur maximale = 80 caractères\n");
            return EXIT_FAILURE;
        }
        printf("Translation mode\n");
        translation_mode(argv);
    } else {
        printf("Usage:\nCalculation mode : %s -c operande1 {+|-|x|/}
operande2\nTranslation mode : %s -t \"text\"\n", argv[0], argv[0]);
        return EXIT_FAILURE;
```

```
    }

    return EXIT_SUCCESS;
}
```

Segments de mémoire

Segment

```c
#include "segment_memoire.h"

#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <stdio.h>
#include <wait.h>
#include <signal.h>
#include <string.h>

#define NOM "exo2.c"
#define SLEEP_TIME 2
#define TAILLE 100
#define ID 1

void empty_funct(){}

/**
 * \brief Cree un segment de memoire
 * \param nom le nom du fichier associe
 * \param taille la taille du segment memoire a creer
 * \param id associe au nom pour la construction d'une cle identifiant pour le
segment memoire (on utilisant la fonction ftok)
 * \return l'identificateur du segment, ou -1 en cas d'erreur
 * int shmget(key_t cle, int taille, int option)
*/
int cree_segment(char* nom, size_t taille, int id){
    // Création de la clef avec le fichier et l'id passés en paramètre
    printf("### Création de la clef segment_key sur le fichier %s, et d'id %d
###\n", nom, id);
    key_t segment_key = ftok(nom, id);

    // Création du segment
    printf("### Création d'un segment de mémoire partagé avec la clef %d, de
taille %ld ###\n\n", segment_key, taille);
    int segment_id = shmget(segment_key, taille, IPC_CREAT | 0660);

    // Renvoie de l'identifiant segment
    return segment_id;
}

/**
 * \fn int afficher_info_segment(int shmid)
 * \brief Affiche toutes les informations concernant un segment de memoire
 * \param shmid l'identificateur du segment
```

```c
 * \return -1 en cas d'erreur
 * int shmctl (int shmid, int op, struct shmid_ds *p_shmid)
 * struct shmid_ds {
    struct ipc_perm shm_perm;       Appartenace et permissions
    size_t          shm_segsz;      Taille segment en octets
    time_t          shm_atime;      Heure dernier attachement
    time_t          shm_dtime;      Heure dernier détachement
    time_t          shm_ctime;      Heure dernière modification
    pid_t           shm_cpid;       PID du créateur
    pid_t           shm_lpid;       PID du dernier shmat(2)/shmdt(2)
    shmatt_t        shm_nattch;     Nombre d'attachements actuels
    };
*/
int afficher_info_segment(int shmid){
    // Initialisation de la structure de reception
    struct shmid_ds* buffer = malloc(sizeof(struct shmid_ds));

    // Reception de l'information
    int index = shmctl(shmid, IPC_STAT, buffer);

    // Affichage des informations
    printf("##### Informations sur la structure : #####\n");
    printf("Taille segment en octets\t\t%zu\n", buffer->shm_segsz);
    printf("Heure dernier attachement\t\t%ld\n", buffer->shm_atime);
    printf("Heure dernier détachement\t\t%ld\n", buffer->shm_dtime);
    printf("Heure dernière modification\t\t%ld\n", buffer->shm_ctime);
    printf("PID du créateur\t\t\t\t%d\n", buffer->shm_cpid);
    printf("PID du dernier shmat(2)/shmdt(2)\t%d\n", buffer->shm_lpid);
    printf("Nombre attachements actuels\t\t%ld\n\n",buffer->shm_nattch);

    // Retour
    return index;
}

int detruire_segment(int id){
    struct shmid_ds p_shmid; // Structure temporaire non allouée afin d'eviter des
erreurs, détruite à la fin de la fonction
    int res = shmctl(id,IPC_RMID,&p_shmid); // Appel a shmctl avec IPC_RMID pour
demander la suppression du segment id.
    return res; // Renvoi du code de retour de shmctl
}

int main(int argc, char const *argv[])
{
    if(argc != 2){
        printf("Utilisation : ./exo2 <chaine>\n");
        exit(EXIT_FAILURE);
    }

    pid_t pid;
    char *mem;
    int shmid;
    int taille_segment = strlen(argv[1]);
```

```c
    /* Création de :
     * Une clef à partir de NOM et ID
     * Un segment d'id shmid à partir de la clef et de TAILLE */
    shmid = cree_segment(NOM, taille_segment, ID);
    if(shmid == -1){
        printf("Echec de la création, fin du programme\n");
        exit(EXIT_FAILURE);
    }

    switch(pid = fork()) {
    case -1: // ECHEC
        perror("fork");
        return -1;
    case 0: // FILS
        sleep(SLEEP_TIME);
        /*s'attacher au segment et affichage de son contenu */
        mem = shmat(shmid, NULL,SHM_RDONLY);
        if(mem ==  (void *)-1){
            printf("## Fils n'a pas réussi a s'attacher ##\n");
            exit(EXIT_FAILURE);
        }
        printf("Message reçu :\t%s\n",mem);
        kill(getppid(),SIGUSR1);
        break;
    default: // PERE
        signal(SIGUSR1,empty_funct);
        // Attachement du segment de memoire partagee
        mem = shmat(shmid, NULL, 0);

        // Ecriture dans le segment
        strncpy(mem, argv[1], taille_segment);

        // Attente de la fin du fils
        pause();

        // Détachement du segment mémoire
        shmdt(mem);

        // Destruction du segment
        if(detruire_segment(shmid) == -1){
            printf("Echec destruction, fin du programme\n");
            exit(EXIT_FAILURE);
        }
        else{
            printf("### Destruction : OK ###\n");
        }
    }

    return 0;
}
```

Sémaphores

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

#define ASCENSEUR 0
#define SEGMENT 1

void empty_funct(){}

void sortie(){
    printf("Le fils de PID %d est bien arrivé\n", getpid());
    exit(0);
}

/**
 * \brief Cree un segment de memoire
 * \param nom le nom du fichier associe
 * \param taille la taille du segment memoire a creer
 * \param id associe au nom pour la construction d'une cle identifiant pour le
segment memoire (on utilisant la fonction ftok)
 * \return l'identificateur du segment, ou -1 en cas d'erreur
 * int shmget(key_t cle, int taille, int option)
*/
int cree_segment(char* nom, size_t taille, int id){
    // Création de la clef avec le fichier et l'id passés en paramètre
    printf("### Création de la clef segment_key sur le fichier %s, et d'id %d
###\n", nom, id);
    key_t segment_key = ftok(nom, id);

    // Création du segment
    printf("### Création d'un segment de mémoire partagé avec la clef %d, de
taille %ld ###\n\n", segment_key, taille);
    int segment_id = shmget(segment_key, taille, IPC_CREAT | 0660);

    // Renvoie de l'identifiant segment
    return segment_id;
}

int detruire_segment(int id){
    struct shmid_ds p_shmid; // Structure temporaire non allouée afin d'eviter des
erreurs, détruite à la fin de la fonction
    int res = shmctl(id,IPC_RMID,&p_shmid); // Appel a shmctl avec IPC_RMID pour
demander la suppression du segment id.
    return res; // Renvoi du code de retour de shmctl
}
```

```c
int main(int argc, char const *argv[]){

    // Lecture des arguments
    if(argc != 2){
        printf("Utilisation : ./main <nombre_ouvriers>\n");
        exit(0);
    }
    int n = atoi(argv[1]);

    // Création du semaphore
    int sem_id;
    key_t sem_key = ftok("main.c", 0);
    // Identifiant key, 1 = un sémaphore, IPC_CREAT = créé le sémaphore
    sem_id = semget(sem_key, 2, IPC_CREAT | 0666);

    // Initialisation du sémaphore de l'ascenseur à 2, et de celui du segment de
mémoire à 1
    if(semctl(sem_id, ASCENSEUR, SETVAL, 2) == -1 || semctl(sem_id, SEGMENT,
SETVAL, 1) == -1){
        printf("Echec à l'initialisation du sémaphore\n");
        exit(EXIT_FAILURE);
    }

    // Création du segment de mémoire partagée
    int shmid;
    char nom[7]="main.c";
    int taille_segment = 5;
    int id = 1;
    shmid = cree_segment(nom, taille_segment, id);


    //Creation des fils
    char *mem;
    int i = 0;
    pid_t son = 1;
    while(i<n && son!=0){
        i++;
        son = fork();
    }
    // Code des fils
    if (son == 0){
        signal(SIGUSR1,sortie);
        printf("Je suis le fils de PID : %d et de PPID :
%d\n",getpid(),getppid());

        struct sembuf ascenseur_buffer = {ASCENSEUR, -1, 0};
        struct sembuf segment_buffer = {SEGMENT, -1, 0};

        // Modifier les sémaphores de l'ascenseur puis du buffer (l'ordre est
important !)
        semop(sem_id, &ascenseur_buffer, 1);
        semop(sem_id, &segment_buffer, 1);

        mem = shmat(shmid, NULL,0);
```

```
        if(mem ==  (void *)-1){
            printf("## Fils n'a pas réussi a s'attacher ##\n");
            exit(EXIT_FAILURE);
        }

        // Ecriture dans le segment
        char str[5];
        sprintf(str, "%d", getpid());
        strncpy(mem, str, 5);
    }
    //Code du père
    else{
        int first_child, second_child;
        struct sembuf segment_buffer = {SEGMENT, 1, 0};
        struct sembuf ascenseur_buffer = {ASCENSEUR, 2, 0};

        signal(SIGUSR2,empty_funct);
        // Attachement du segment de memoire partagee
        mem = shmat(shmid, NULL, 0);

        // En attente du premier fils
        pause();

        for (int i = 0; i < n; i+=2){
            // Gestion premier fils
            first_child = atoi(mem);

            // Libération du segment mémoire
            semop(sem_id, &segment_buffer, 1);

            // En attente du deuxième  fils
            pause();
            second_child = atoi(mem);

            // Fin des deux fils et libération de l'ascenceur et du segment
            printf("Montée de %d et %d\n", first_child, second_child);
            kill(first_child,SIGUSR1);
            kill(second_child,SIGUSR1);
            semop(sem_id, &ascenseur_buffer, 1);
            semop(sem_id, &segment_buffer, 1);
            pause();
        }

        // Si il y avait un nombre impaire de fils
        if (i != n){
            // Montée du dernier fils
            first_child = atoi(mem);
            printf("Montée de %d\n", first_child);
            kill(first_child,SIGUSR1);

            // Redéfinition de la structure buffer de l'ascenseur pour ajouter une
place (pas deux, car un seul fils est monté cette fois)
            ascenseur_buffer = {ASCENSEUR, 1, 0};
```

```
            // Libération de l'ascenseur et du segment
            semop(sem_id, &ascenseur_buffer, 1);
            semop(sem_id, &segment_buffer, 1);
        }

        // Détachement du segment
        shmdt(mem);

        // Destruction du segment
        if(detruire_segment(shmid) == -1){
            printf("Echec destruction, fin du programme\n");
            exit(EXIT_FAILURE);
        }
        else{
            printf("### Destruction : OK ###\n");
        }

        // Sortie du programme (pas besoin de détacher/supprimer/etc les
sémaphores)
    }
}
```