Compilation exercices C : Master 1 UPMC

Disclaimer

L'ensemble des exercices présent dans ce recueil sont réalisés par un étudiant en informatique.

Il est de ce fait possible que certains exercices soient faux, néanmoins l'auteur s'est appliqué à essayer de les réaliser correctement.

Nicolas SALLERON

Table des matières

Prise en main

Création de processus

/*2 Création de processus
A l'aide de la fonction standard fork, définir la fonction
int nfork (int n)
qui crée n processus fils et renvoie :
le nombre de processus fils créés, pour le processus père ;
0 pour les processus fils ;
-1 dans le cas où aucun fils n'a été créé.
Si la création d'un processus fils échoue, la fonction n'essaie pas de créer les autres fils. Dans ce cas, la fonction renvoie le nombre de fils qu'elle a pu créer (ou –1, si aucun fils n'a été créé). On pourra compléter cette fonction par une fonction main l'appliquant à un entier pas trop grand pour la tester.
Exemple d'appel :
$PWD/bin/nfork
*/

```
int nfork(int n){
	int nbFils = 0;
	pid_t fils;
	while( nbFils < n){
		if((fils = fork()) ==-1){
			perror("fork"); exit(1);
		}else if(fils ==0){
			printf("fils %d\n",nbFils);
			exit(1);
		}
		nbFils++;
	}
}

int main(int argc,char *argv[]){
	if(argc != 2){
		printf("Pas assez d'arguments");
		exit(0);
	}else{

		printf("Début\n\n");
		nfork(atoi(argv[1]));
		int n = 0;
		while(n<atoi(argv[1])){
			wait();
			n++;
		}
		printf("Fin\n");


	}
}
```

Attente de processus

/*5 Attente de processus

Écrire un programme qui crée deux processus fils fils1 et fils2. Chaque fils crée un fils, fils1.1 et fils2.1 respectivement. Ces 4 processus ne font qu'imprimer leur PID et PPID. Dans le cas du processus fils2, il imprime aussi le PID de son frère aîné fils1. Les processus n'ayant pas de fils se terminent aussitôt, mais un processus qui a des fils (y compris le processus principal) ne se termine qu'après ceux-ci. On utilisera la fonction wait pour réaliser cette attente, à l'exclusion de toute autre méthode (fichiers, fonction sleep etc).
Exemple d'appel :
$PWD/bin/mon_frere
*/

```c
int main(int argc,char *argv[]){
	pid_t fils[2];
	int i, j;
	printf("Père : %d\n",getpid());
	for(i = 0;i<2;i++){
		int valFork;
		if((fils[i] = fork()) == -1){
			perror("fork");
			exit(-1);
		}

		if(fils[i] == 0){ // Les fils
			if(i == 1)	//fils 2
				printf("Fils 2 \tMon PID : %d, PPID : %d, PID de mon frere : %d\n",getpid(),getppid(), fils[0]);
			else		//fils 1
				printf("Fils 1 \tMon PID : %d, PPID : %d\n",getpid(),getppid());

			pid_t pfils;
			if((pfils = fork()) == -1){
				perror("fork");
				exit(-1);
			}
			if(pfils == 0){	//les petits fils

				printf("Fils %d.1 - Mon PID : %d, PPID : %d\n",i+1,getpid(),getppid());
				exit(1);
			}else{
				wait();
				printf("Mon petit fils est mort\n");
				exit(1);
			}
		}
	}//for
	sleep(1);
	for(j = 0;j<2;j++){
		waitpid(fils[j],NULL,1);
		printf("Mon fils %d est mort\n", fils[j]);
	}
}//main
```

Recopie de fichier

```
/*
1 Recopie de fichier
Écrire en C un programme qui prend en argument deux noms de fichier et recopie intégralement le contenu du premier dans le second, en utilisant les fonctions POSIX open, read et write. On donnera au deuxième fichier les droits en lecture et écriture pour soi (à l'aide du 3e argument de open ou en appelant la fonction chmod). On veillera à dénoncer (avec la fonction perror) les cas d'erreur suivants :
   la ligne de commande ne contient pas exactement 2 noms ;
   le premier nom ne désigne pas un fichier régulier et accessible en lecture ;
   le second ne peut être créé (répertoire inaccessible en écriture, ou entrée déjà existante dedans).
En cas de réussiste, le programme 0 sinon il retourne la valeur de errno.
Exemple d'appel :
$PWD/bin/mycp src/mycp.c cp.c
*/

int main(int argc,char *argv[]){
	struct stat buf;
	int fd1,fd2;
	char buffer[1];
	int k = 1;
	int n;
	printf("Argc %d\n",argc);
	if(argc != 3){
		printf("Il manque des arguments\n");
		exit(1);
	}
	printf("Récupération stat\n");
	//Récupération des informations du fichier
	if(stat(argv[1],&buf)!=0){
		perror("Echec stat : ");
		exit(0);
	}
	printf("Vérification si le fichier est régulier\n");
	//Vérification si le fichier est un fichier régulier
	if(S_ISREG(buf.st_mode)){
		printf("Le fichier est regulier\n");
	}else{
		printf("Le fichier n'est pas un fichier régulier, sortie..\n");
		exit(0);
	}
	printf("Fin de vérification\n");
	//Ouverture du premier fichier
	if((fd1=open(argv[1],O_RDONLY,buf.st_mode)) ==-1){
		printf("Probleme lecture\n");
		printf("%s",strerror(errno));
	}
	//Ouverture ou création du second fichier
	if((fd2=open(argv[2],O_CREAT, buf.st_mode)) == -1){
		printf("Problème dans la création\n");
		printf("%s\n",strerror(errno));
	}

	//Recopie des fichiers
	while((n=read(fd1,&buffer,k)) > 0){
		write(fd2,&buffer, k);
	}
	//Fermeture des fichiers
	close(fd1);
	close(fd2);
}
```

Remontée de valeurs

/*
Écrire en C un programme prenant en argument un nombre n et un nom de fichier. Le processus principal doit créer n processus fils, à l'aide de fork. Chaque processus fils produit une valeur aléatoire qu'il insère dans le fichier donné en 2e argument, à destination du processus principal. La valeur aléatoire est calculée par :
(int) (10*(float)rand()/ RAND_MAX)
De son côté, le processus principal doit attendre la terminaison de tous ses fils, puis extraire toutes les valeurs du fichier pour ensuite les additionner et enfin afficher la somme résultante. On pourra utiliser waitpid appliqué aux résultats de fork pour attendre les terminaisons, puis utiliser lseek. On contrôlera les cas d'erreurs comme à l'exercice précédent.
Exemple d'appel :
$PWD/bin/remonte 8 aleas
*/

```c
int desc_cible = 0, N = 0, i = 0;
pid_t *tabFils;
char *ValeurLecture;
FILE *ptFile;
int valTot;
char c;
int main(int argc, char *argv[]) {
	//Vérification du nombre d'arguments
	if(argc != 3)
		perror("-> Le nombre de paramètres est incorrect.\n");
	//Récupération du nombre de fils
	N = atoi(argv[1]);
	//Création d'un tableau pour connaitre le PID de mes fils
	tabFils = malloc(sizeof(pid_t)*N);
		if( (desc_cible = open(argv[2],O_CREAT|O_RDWR,0700)) == -1){ //Vérification si l'ouverture est correcte
		perror("-> Problème d'ouverture en lecture du fichier \n");
		return EXIT_FAILURE;
	}
	   if((ptFile = fdopen(desc_cible, "w+"))==NULL){ //Ouverture pour pouvoir manipuler des caractères simplement
		perror("-> Probleme fdopen : ");
		return EXIT_FAILURE;
	}
	for (i=0;i<N;i++) {
		//Fork
		if((tabFils[i] = fork()) == -1){
			perror("-> Problème de fork");
			return EXIT_FAILURE;
		}
		if(tabFils[i]==0){ //Dans le fils

			//printf("Fils : %d, père : %d\n", getpid(),getppid());

			//Génération de la valeur aléatoire
			int valeur = (10*(float)rand()/ RAND_MAX);
			//printf("-> valeur : %d\n",valeur);
			//Ecriture
			if(fprintf(ptFile, "%d", valeur)==-1){
				perror("-> Problème de fprintf :");
				fclose(ptFile);
				return EXIT_FAILURE;
			}
			return EXIT_SUCCESS;
		}
	}
		for (i=0;i<N;i++) {//Attente de l'ensemble des fils
		waitpid(tabFils[i],NULL,0);
	}
	lseek(desc_cible, 0, SEEK_SET); //Placement en début de fichier
	//Lecture et addition entier par entier
	while ((c = fgetc(ptFile)) != EOF) {
		valTot += (c - '0');
	}
	fclose(ptFile);
	printf("=>Valeur totale : %d\n",valTot);
	return EXIT_SUCCESS;
}
```

Fonctions POSIX vs. fonctions C

```
/*
4 Fonctions POSIX vs. fonctions C
Écrire en C deux fonctions qui lisent caractère par caractère un fichier passé en argument, et qui affichent chaque caractère dès qu'il est lu. Ces fonctions utiliseront 3 processus partageant le même descripteur et agissant à l'identique (l'identité de chaque processus est affichée avec le caractère lu). La première utilisera les standards POSIX open et read, la seconde fopen et fgetc. Vous écrirez un unique programme qui, selon que son premier argument est -p ou -C appliquera l'une ou l'autre de ces fonctions sur le fichier donné en 2e argument. Vous contrôlerez les cas d'erreur comme précédent. Quelle différence observez-vous entre les deux modes d'appels ?
Exemple d'appel :
$PWD/bin/lectures -p src/lectures.c; $PWD/bin/lectures -C src/lectures.c
*/
int c(char *file){
	FILE* ptFile;
	pid_t fils;
	char c;
	if((ptFile = fopen(file, "r+")) == NULL){
		perror("-> Problème ouverture"); return EXIT_FAILURE;
	}
	for (int i = 0;i<3;i++) {

			if((fils = fork()) ==-1){
				perror("fork"); exit(EXIT_FAILURE);
			}else if(fils ==0){		//Dans le fils
				while ((c=fgetc(ptFile))!= EOF) {
						printf("Moi : %d / Père %d LIBC : %c\n",getpid(), getppid(),c);
				}
				return EXIT_SUCCESS;
			}
	}
	return EXIT_SUCCESS;

}
int p(char *file){
	int fd;
	char c;
	pid_t fils;
	if((fd = open(file, O_RDONLY,0600)) == -1){
		perror("-> Problème ouverture"); return EXIT_FAILURE;
	}
	for (int i = 0;i<3;i++) {
		if((fils = fork()) ==-1){
			perror("fork"); exit(EXIT_FAILURE);
		}else if(fils ==0){		//Dans le fils
			while (read(fd, &c, sizeof(char))>0) {
				printf("Moi : %d / Père %d POSIX : %c\n",getpid(),getppid(),c);
			}
			return EXIT_SUCCESS;
		}
	}
	return EXIT_SUCCESS;
}
int main(int argc, char *argv[]) {

	if(argc != 3){
		perror("Pas assez d'arguments.");
	}
	if(strcmp(argv[1], "-p") == 0){
		printf("Passage mode POSIX\n");
		p(argv[2]);

	}else if(strcmp(argv[1], "-C") == 0){
		printf("Passage mode C\n");
		c(argv[2]);
	}else {
		printf("Argument incorrect.\n");
		exit(EXIT_FAILURE);
	}
}
```

Fonctions sur fichiers

Liens et fichiers

//Ecrivez un programme un C qui lit sur la ligne de commande deux chemins absolus dans l'arborescence Unix, et teste s'il s'agit du même fichier.
//On utilisera la fonction stat. Testez votre programme sur différents cas, notamment des liens symboliques ou non symboliques.

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

	int fd1, fd2;
	struct stat stat1, stat2;

	if(argc != 3){
		perror("Problème pour le nombre d'arguments.\n"); return EXIT_FAILURE;
	}
	printf("Argument 1 : %s\n", argv[1]);
	printf("Argument 2 : %s\n", argv[2]);

	if(access(argv[1], R_OK) == -1)
		return EXIT_FAILURE;
	if(access(argv[2], R_OK) == -1)
		return EXIT_FAILURE;
	if((fd1 = open(argv[1],O_RDWR,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}
	if((fd2 = open(argv[2],O_RDWR,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}

	if (lstat(fd1,&stat1) == -1){
		perror("Problème dans le fstat\n");
	}
	if (lstat(fd2,&stat2) == -1){
		perror("Problème dans le fstat\n");
	}

	//if the inodes and device numbers are equal, it is impossible for the two paths to refer to different files

	if(stat1.st_dev == stat2.st_dev && stat1.st_ino == stat2.st_ino){
		printf("Les fichiers partage le même numéro de device et le même numéro d'inode\n");
	}else{
		printf("Les fichiers ne partagent pas le même numéro d'inode et/ou de device\n");
	}

	return EXIT_SUCCESS;
}
```

Droits sur un fichier

```
//Ecrire un programme qui permet d'effacer, de renommer ou de changer les droits d'un fichier existant. Le programme reçoit en argument :
//      type d'opération :
//      "E" ou "e" pour effacer
//      "R" ou "r" pour renommer
//      "C" ou "c" pour changer les droits
//      nom du fichier
//      nom du fichier à renommer ou nouveaux droits :
//      "R" ou "r" (read-only / lecture seulement) // + écriture pour le propriétaire du fichier R pour GRP et OTH
//      "W" ou "w" (read-write / lecture-ecriture)      // RW partout
//      Observations :
//      Le programme doit vérifier que le deuxième argument n'est pas un répertoire (utiliser la fonction stat).
//      pour le changement de droits en "read-only" n'oubliez pas de donner le droit d'écriture au propriétaire du fichier.
int efface(char *filename){
    printf("je suis dans efface\n");
    //Il faut supprimer le référencement vers le fichier
    if(unlink(filename) == -1){
        perror("La suppression à échoué !\n");
        return EXIT_FAILURE;
    }return EXIT_SUCCESS;
}
int renomme(char *nameOld, char *nameNew){
    printf("ancien : %s\n",nameOld);
    printf("nouveau : %s\n",nameNew);
    if(rename(nameOld, nameNew) == -1){
        perror("Le renommage à échoué !\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
int change(char *filename, char *mode){
    int result = 0;
    if((strcmp(mode, "R") == 0) || (strcmp(mode, "r") == 0))
        result = chmod(filename, S_IREAD|S_IWUSR);
    else if ((strcmp(mode, "W")  == 0)|| (strcmp(mode, "w") == 0))
        result = chmod(filename, S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
    else {
        printf("Problème argument R/W");
    }
    if(result == -1){
        perror("Problème chmod\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
int main(int argc, char *argv[]) {
    int fd1;
    struct stat stat1;
    if(argc == 1)
        return EXIT_FAILURE;
    if((fd1 = open(argv[2],O_RDWR,0700)) == -1){
        perror("Problème d'ouverture du fichier\n");
        return EXIT_FAILURE;
    }
    if (fstat(fd1,&stat1) == -1){
        perror("Problème dans le fstat\n");
        return EXIT_FAILURE;
    }
    if(S_ISDIR(stat1.st_mode)){ //Vérification si le deuxième argument n'est pas un répertoire
        perror("C'est un directory\n");
        return EXIT_FAILURE;
    }
    if((strcmp(argv[1], "E") == 0) || (strcmp(argv[1], "e") == 0))
        efface(argv[2]);
    if ((strcmp(argv[1], "R")  == 0)|| (strcmp(argv[1], "r")  == 0))
        renomme(argv[2],argv[3]);
    if ((strcmp(argv[1], "C")  == 0)|| (strcmp(argv[1], "c") == 0))
```

```
        change(argv[2],argv[3]);
    return EXIT_SUCCESS;
}
```

Remplacement dans un fichier

```
//Ecrire un programme qui reçoit au moins trois arguments :
//	un nom de fichier à créer ;
//	un mot quelconque ;
//	une suite de mots quelconques.
//	Le programme doit créer le fichier et écrire la suite de mots dans le fichier, ainsi que dans le flux de sortie.
//	Ce même fichier est ensuite parcouru en utilisant la fonction read et la lecture s'arrête après le premier mot.
//	On remplace alors le deuxième mot de la suite par le mot donné en deuxième argument.
//	On suppose que le remplaçant et le remplacé sont de même longueur.
//	Exemple d'appel :
//	$PWD/bin/remplacedansfichier texte toi a moi de jouer

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

	int fd;
	if(argc < 3)
		return EXIT_FAILURE;

	if((fd = open(argv[1],O_CREAT|O_RDWR|O_TRUNC,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}
	int i = 3;
	while (argv[i] != NULL) {
		if(write(fd, argv[i], strlen(argv[i])) == -1){
			perror("Probleme write");
			return EXIT_FAILURE;
		}
		printf("%s ",argv[i]);
		if(write(fd, " ", strlen(" ")) == -1){
			perror("Probleme write");
			return EXIT_FAILURE;
		}
		i++;
	}
	printf("\n");

	lseek(fd, 0, SEEK_SET);
	char c;
	while (read(fd, &c, sizeof(char))>0) {
		if(c == ' '){
			if(write(fd, argv[2], strlen(argv[2])) == -1){
				perror("Write pb ");
			}
			break;
		}
	}
}
```

Redirection

```
//En utilisant la fonction dup2, écrire une fonction Rediriger_stdout redirigeant la sortie standard vers un fichier donné en argument. On considère que le fichier n'existe pas.
//Ecrire ensuite une deuxième fonction Restaurer_stdout qui restaure la sortie vers le terminal. Écrire enfin une fonction main prenant en argument un nom de fichier, et qui appelle 3 fois la fonction printf, les deux premiers appels encadrant un appel à Rediriger_stdout sur le fichier indiqué, et les deux derniers un appel à Restaurer_stdout. Qu'observez-vous dans le flux de sortie et le fichier ?
//
//Exemple d'appel :
//$PWD/bin/rediriger trace.txt

int Rediriger_stdout(int desc){

	if ( dup2(desc,1) == -1)	//Fermeture de l'écran et placement de desc dans le tableau au niveau du descripteur 1;
		perror("Problème dup2");
	return EXIT_SUCCESS;
}

int Restaurer_stdout(int std_out){

	if ( dup2(std_out,1) == -1)
		perror("Problème dup2");
	return EXIT_SUCCESS;
}


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

	int fd;

	if(argc < 1)
		return EXIT_FAILURE;

	if((fd = open(argv[1],O_CREAT|O_RDWR|O_TRUNC,0700))==-1)
		perror("pb ouverture");

	int std_old = dup(STDOUT_FILENO);



	printf("Apl 1 \n");
	Rediriger_stdout(fd);
	printf("Apl 2\n");
	Restaurer_stdout(std_old);
	printf("Apl 3\n");

	return EXIT_SUCCESS;


}
```

Fonction grep étendue

//On considère le fichier liste-rep.c founi en annexe qui liste le contenu d'un répertoire. Écrire une variante de ce programme qui cherche la chaîne donnée en premier argument dans tous les fichiers du répertoire donné en 2e argument. Il affiche le nom de chaque fichier qui contient la chaîne de caractères recherchée, ou "Aucun fichier valide" si la chaîne n'est présente dans aucun des fichiers du répertoire.
//
//N.B : Vous pouvez utiliser la fonction strstr de la bibliothèque string.h pour trouver si une chaîne de caractères est présente dans une autre.
//
//Exemple d'appel :
//bin/extended-grep if src

```
char buff_path [TAILLE_PATH];
DIR *pt_Dir;
struct dirent* dirEnt;

int main (int argc, char* argv []) {
	if (argc ==  3) {
		//Répertoire donné en 2ème argument.
		memcpy (buff_path,argv[2],strlen(argv[2]));
	}else
		return EXIT_FAILURE;
	if ( ( pt_Dir = opendir (buff_path) ) == NULL) {
		if (errno ==  ENOENT ) {
			/* répertoire n'existe pas - créer le répertoire */
			if (mkdir ( buff_path, S_IRUSR|S_IWUSR|S_IXUSR) == -1) {
				perror ("erreur mkdir\n");
				exit (1);
			}
			else
				return 0;
		}
		else {
			perror ("erreur opendir \n");
			exit (1);
		}
	}
	char *ret;
	int i = 0;
	/* lire répertoire */
	while ((dirEnt= readdir (pt_Dir)) !=NULL) {

		ret = strstr(dirEnt->d_name, argv[1]);
		if(ret != NULL){
			printf ("%s\n", dirEnt->d_name);
			i++;
		}

		//qui cherche la chaîne donnée en premier argument dans tous les fichiers du répertoire donné en 2e argument.
		//printf("String : %s\n", dirEnt->d_name);
	}
	if(i==0)
		printf("Aucun fichier correspondant à \"%s\" dans le répertoire.",argv[1]);
	closedir (pt_Dir);

	return 0;
}
```

Inverseur de contenu en utilisant lseek

```
/*
2 Inverseur de contenu en utilisant lseek

Écrire un programme qui prend en argument un nom de fichier, le lit caractère par caractère pour l'écrire de manière inversée dans un autre fichier. Votre programme doit impérativement utiliser la fonction lseek pour modifier l'offset lors de la lecture.

Si le premier fichier contient "fichier ok" alors le deuxième devra contenir "ko reihcif" (on ne testera évidemment pas sur un fichier contenant un palindrome comme radar etc).
Exemple d'appel :
bin/inverser-fichier src/inverser-fichier.c
*/

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

	int fd,fd2;
	char c;
	int nbChar;

	if(argc < 2)
		return EXIT_FAILURE;

	if((fd = open(argv[1],O_RDONLY,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}
	if((fd2 = open("./result.txt",O_CREAT|O_RDWR|O_TRUNC,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}

	while (read(fd, &c, sizeof(char))>0)
		nbChar++;
	printf("nombre de caractère : %d\n",nbChar);
	lseek(fd, 0, SEEK_SET);
	lseek(fd2, nbChar, SEEK_END);
	while (read(fd, &c, sizeof(char))>0){
		printf("%c\n",c);
		write(fd2, &c, sizeof(char));
		lseek(fd2, --nbChar, SEEK_SET);
	}
}
```

Inverseur de contenu en utilisant pread

```
/*
3 Inverseur de contenu en utilisant pread

Même exercice que précédemment, mais en utilisant pread à la place de lseek.
Exemple d'appel :
bin/inverser-pread src/inverser-pread.c
*/

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

	int fd,fd2;
	char c;
	int nbChar;

	if(argc < 2)
		return EXIT_FAILURE;

	if((fd = open(argv[1],O_RDONLY,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}
	if((fd2 = open("./result.txt",O_CREAT|O_RDWR|O_TRUNC,0700)) == -1){
		perror("Problème d'ouverture du fichier\n");
		return EXIT_FAILURE;
	}

	while (read(fd, &c, sizeof(char))>0)
		nbChar++;
	printf("nombre de caractère : %d\n",nbChar);
	int init = 0;
	while (pread(fd, &c, sizeof(char), init++)>0) {
		pwrite(fd2, &c, sizeof(char), nbChar--);
	}


}
```

Signaux et processus

Chaîne de processus

/*
1 Chaîne de processus

A l'aide de la fonction fork, ecrire un programme qui crée une chaîne de processus telle que le processus initial (celui du main) crée un processus qui à son tour en crée un second et ainsi de suite jusqu'à la création de N processus (en plus du processus initial). Au moment de sa création, le dernier processus de la chaîne affiche le Pid de tous les autres processus y compris celui du processus initial. Chacun des autres processus attend la terminaison de son fils, puis affiche son propre Pid (à l'aide de getpid), celui de son père (à l'aide de getppid) et celui de son fils avant de se terminer.

On souhaite de plus que le dernier processus créé génère une valeur aléatoire entre 0 et 100. Pour générer cette valeur aléatoire utilisez :

(int)(rand () /(((double) RAND_MAX +1) /100))

Écrire le programme de sorte que le processus initial affiche cette valeur aléatoire avant de se terminer
Exemple d'appel :
$PWD/bin/chaine_proc 10
*/

```c
pid_t *Proc;
int main(int argc, char *argv[]) {

	if(argc != 2){
		perror("exiting...");
		return 0;
	}
	int n = atoi(argv[1]) + 1;
	int i = 0;
	int j = 0;
	int status;
	Proc = malloc(n*sizeof(pid_t));
	for(i = 0;i<n;i++){
		if((Proc[i] = fork()) != 0){//Père
			//printf("Mon PID : %d\n",getpid());
			break;
		}else{
			srand(getpid()); //A savoir
			Proc[i] = getppid();
			//printf("%d",i);
			if((i+1)==n){		//On est dans le dernier
				for(j=0;j<n;j++)
					printf("PID du processus %d : %d\n",j,Proc[j]);
				status = (int)(rand () /(((double) RAND_MAX +1) /100));
				//status = 15;
				printf("Valeur aléatoire dernier fils : %d\n",status);
				exit(status);
			}
		}


	}
	//Attente du fils nouvellement crée
	waitpid(Proc[i], &status, NULL);
	if(i==1){
		printf("Valeur aléatoire : %d\n",WEXITSTATUS(status));
		return EXIT_SUCCESS;
	}

	exit(WEXITSTATUS(status));
}
```

La fonction kill

```
/*
2 La fonction kill
On reprend l'exercice précédent, mais on s'interdit d'utiliser les fonctions wait et assimilées, ni bien sûr le signal SIGCHLD. A la place on utilisera les fonctions kill, sigaction et sigsuspend. Comment résoudre alors le problème en s'assurant qu'aucun processus ne se termine avant que tous les autres ne soient créées ?
Remarque : on ne demande plus de récupérer la valeur aléatoire ici.
Exemple d'appel :
$PWD/bin/kill_proc 10
*/
pid_t *Proc;
void sigTrt(int sig){
	if(sig == SIGUSR1)
		printf("Fin d'un fils.\n");
}

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

	if(argc != 2){
		perror("exiting...");
		return 0;
	}
	int n = atoi(argv[1]) + 1;
	int i = 0;
	int j = 0;
	int status;

	Proc = malloc(n*sizeof(pid_t));
	sigset_t sig;
	sigfillset(&sig);
	sigprocmask(SIG_SETMASK, &sig, NULL); //On bloque les sigaux;
	struct sigaction action;
	action.sa_flags = 0;
	action.sa_mask = sig;
	action.sa_handler = sigTrt;
	sigaction(SIGUSR1, &action, NULL);

	for(i = 0;i<n;i++){
		if((Proc[i] = fork()) != 0){//Père
			//printf("Mon PID : %d\n",getpid());
			break;
		}else{
			Proc[i] = getppid();
			//printf("%d : PID : %d\n",i, getpid());
			if((i+1)==n){		//On est dans le dernier
				Proc[i] = getpid();
				for(j=0;j<n;j++)
					printf("PID du processus %d : %d\n",j,Proc[j]);
				kill(getppid(),SIGUSR1);
				break;
			}
		}
	}

	sigdelset(&sig, SIGUSR1);
	//Attente du fils nouvellement crée
	if(i+1 != n){
		sigsuspend(&sig);
		printf("PID : %d\n",getpid());
		if(getpid()==Proc[0]){
			kill(getppid(),SIGUSR1);
		}
	}
}
```

Les Signaux SIGSTOP, SIGCONT, SIGCHLD

```
//On reprend encore le même exercice, mais nous voulons que tous les processus, à l'exception du processus initial, soient suspendus par un signal SIGSTOP.
//Lorsqu'ils le sont tous, le processus initial affiche : Tous les descendants sont suspendus. L'exécution de ces processus doit alors reprendre pour que ceux-ci se terminent. Lorsque tous se sont terminés, le programme initial affiche Fin du programme. De nouveau, il faut répondre sans utiliser les fonctions de la famille wait.
pid_t *Proc;
int i = 0;
pid_t pere;
int k = 1;
int status = 0;
void sigTrt(int sig){
	if(sig == SIGCHLD){
		if(i==0){
			if(status == 0){
				printf("Tous les descendants sont suspendus.\n");
				kill(Proc[0], SIGCONT);
			}
			if(status == 1){printf("Fin programme.\n");}
			status++;
		}else {
			if(status == 0){
				printf("Je m'arrête.\n");
				status++;
				kill(getpid(), SIGSTOP);
			}
			if(status == 1){ kill(Proc[i+1], SIGCONT);}
		}
	}
}
int main(int argc, char *argv[]) {
	if(argc != 2){
		perror("exiting...");
		return 0;
	}
	int n = atoi(argv[1]) + 1;
	int j = 0;
	Proc = malloc(n*sizeof(pid_t));
	sigset_t sig;
	sigemptyset(&sig);
	struct sigaction action;
	action.sa_flags = 0;
	action.sa_mask = sig;
	action.sa_handler = sigTrt;
	sigaction(SIGCHLD, &action, 0);
	printf("Père : %d\n",getpid());
	for(i = 0;i<n;i++){
		if((Proc[i] = fork()) != 0){ break; //père
		}else{
			Proc[i] = getppid();
			if((i+1)==n){		//On est dans le dernier
				Proc[i] = getpid();
				for(j=0;j<n;j++)
					printf("PID du processus %d : %d\n",j,Proc[j]);
				printf("Je m'arrête 1er.\n");
				kill(getpid(),SIGSTOP);   //Traitement signal synchrone
				printf("Reprise\n");
				exit(NULL);
			}
		}
	}
	sigsuspend(&sig);
	if(i==0) sigsuspend(&sig);
}
```

Synchronisation de processus

//Nous avons un processus P1 qui crée un processus P2 (fils de P1) qui à son tour crée un processus P3 (fils de P2 et petit-fils de P1).
//Lorsque le processus P3 est créé, il envoie un signal à son grand-père, le processus P1, pour lui signaler sa création, puis se termine juste après.
// Quand son père, le processus P2, prend connaissance de la terminaison de P3, il envoie un signal à P1, son père, pour signaler la mort de son fils.
//Après P2 se termine lui aussi. Le processus P1 doit traiter les événements dans l'ordre décrit ci-dessus. Autrement dit, il doit premièrement traiter la délivrance du signal de P3 en affichant le message « Processus P3 créé », ensuite la délivrance du signal de P2 en affichant « Processus P3 terminé » et à la fin afficher « Processus P2 terminé » lorsqu'il prend connaissance de la mort de son fils.

//Programmez une telle synchronisation.

```
void sigTrt(int sig){
	if(sig == SIGUSR1)
		printf("Processus P3 crée\n");
	if(sig == SIGUSR2)
		printf("Processus P3 terminé\n");
}

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

	pid_t P1, P2, P3;
	int status;
	struct sigaction action;
	sigset_t ens_sig;

	sigfillset(&ens_sig);
	sigdelset(&ens_sig, SIGUSR1);
	sigdelset(&ens_sig, SIGUSR2);
	action.sa_mask = ens_sig;
	action.sa_flags = 0;
	action.sa_handler = sigTrt;
	sigaction(SIGUSR1,&action,NULL);
	sigaction(SIGUSR2,&action,NULL);
	P1 = getpid();

	if((P2 = fork()) == -1)
		perror("fork");
	if(P2 == 0){		//P2

		if((P3 = fork()) == -1)
			perror("fork");

		if(P3 == 0){		//P3
			//printf("fils P3\n ->kill vers %d\n",P1);
			kill(P1, SIGUSR1); // Le processus se termine
		}else {		//P2
			wait(&status);		//Connaissance mort P3
			kill(P1, SIGUSR2);		//Information vers P1
			//printf("P2 : Mon fils est mort !\n -> kill vers %d",P1);

		}
	}else{		//P1
		sigsuspend(&ens_sig);
		wait(&status);
		printf("Processus P2 terminé\n");
		//sleep(5);
	}


}
```

Attente ordonnée de processus

```
//On reprend l'énoncé de l'exercice "Attente de processus" où un processus crée deux fils créant chacun un processus.
//On ajoute la contrainte que le processus fils 1 ne peut se terminer qu'après les affichages réalisés par son frère fils 2 et son fils fils 1.1.
//Comme auparavant, les processus ne se terminent qu'après envoi des messages de leur fils respectifs. Trouvez une solution n'utilisant que les signaux SIGUSR1 et SIGUSR2, à l'exclusion de tout autre moyen (Wait, Sleep etc).
int main(int argc,char *argv[]){
	pid_t fils[2];
	int i, j;
	int status;
	printf("Père : %d\n",getpid());
	sigset_t sig;
	sigfillset(&sig);
	sigprocmask(SIG_SETMASK, &sig, NULL); //On bloque tout les sigaux;
	for(i = 0;i<2;i++){
		int valFork;
		if((fils[i] = fork()) == -1){
			perror("fork");
			exit(-1);
		}
		if(fils[i] == 0){ // Les fils
			pid_t pfils;
			if((pfils = fork()) == -1){
				perror("fork");
				exit(-1);
			}
			if(pfils == 0){		//les petits fils
			 printf("Fils %d.1 - Mon PID : %d, PPID : %d\n",i+1,getpid(),getppid());
				kill(getppid(), SIGUSR1);
				exit(1);
			}else{
				if(i == 1){		//Fils 2
					while(1){
						//printf("Pas encore...\n");
						sigpending(&sig);
						if(sigismember(&sig, SIGUSR1))		//Il doit attendre la fin de son fils
							break;
					}
					printf("Fils 2 \tMon PID : %d, PPID : %d, PID de mon frere : %d\n",getpid(),getppid(), fils[0]);
					kill(fils[0], SIGUSR2);
				}
				if(i == 0){ //Fils 1
					while(1){
						sigpending(&sig);
						if(sigismember(&sig, SIGUSR1)&&sigismember(&sig, SIGUSR2))
							break;
					}
					printf("Fils 1 \tMon PID : %d, PPID : %d\n",getpid(),getppid());
				}
				exit(1);
			}
		}
	}//for
	for(j = 0;j<2;j++){
		waitpid(fils[j],&status,NULL);
		printf("Mon fils %d est mort\n", fils[j]);
	}
}//main
```

Introduction aux processus légers

Création de threads

//Ecrire un programme créeant N processus légers (à l'aide de pthread_create) et passant en paramètre à chacune son numéro de création compris entre 0 et N. Chacune affichera son numéro de création et son identité (utiliser pthread_self). Ensuite elle se termine, avec pthread_exit, en retournant son numéro de création multiplié par 2. De son côté, le programme principal doit attendre leur terminaison (à l'aide de pthread_join) en affichant la valeur renvoyée par chaque.

//Exemple d'appel :
//$PWD/bin/thread_create 10

//

```
void *funcThread(void *arg){
	int pt =  (int)malloc(sizeof(int));
	printf("Numéro de création %d et identité %ld\n",(*(int*)arg),(long)pthread_self());
	pt = (*(int*)arg)<<1;
	pthread_exit((void*)pt); return NULL;
}

pthread_t *tab;
int main(int argc, char *argv[]) {

	int i = 0;
	int *pi;
	int *valeur;

	if(argc != 2)
		return EXIT_FAILURE;

	tab = malloc(atoi(argv[1])*sizeof(pthread_t));

	for (i =0;i<atoi(argv[1]);i++) {
		//passage de l'argument par le pointeur
		pi = malloc(sizeof(int));
		*pi = i;	//Pointeur vers variable i;
		pthread_create(&tab[i],NULL,funcThread,pi);
	}
	for (i =0;i<atoi(argv[1]);i++) {
		if(pthread_join(tab[i],(void**)&valeur) !=0){
			printf("pthread_join\n");
			exit(1);
		}
		printf("La valeur de %ld est %d\n",(long)tab[i], ((int)valeur));
	}



}
```

Exclusion mutuelle de threads

```
//Modifier le programme de l'exercice précédent pour que chaque thread affiche non plus son numéro de création, mais une valeur aléatoire entre 0 et 10. Pour cela, utilisez la fonction rand de la façon suivante :
//
//      (int) (10*((double)rand())/ RAND_MAX)
//
//      De plus, cette valeur aléatoire sera ajoutée à une variable globale, initialisée à zéro par le programme principal. On veillera évidemment à éviter les accès concurrents à cette variable, en utllisant les fonctions de la famille pthread_mutex_lock. Après terminaison de toutes les threads, le programme afficher la valeur finale de cette variable.
//
//      Exemple d'appel :
//      $PWD/bin/thread_rand 10

pthread_t *tab;
int val;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *funcThread(void *arg){
        pthread_mutex_lock(&mutex);
        val += *((int*)arg);        //(déférencement) (//Cast vers *entier) Pointeur
        printf("Valeur de val : %d\n",val);
        pthread_mutex_unlock(&mutex);
        pthread_exit((void*)0); return NULL;
}

int main(int argc, char *argv[]) {
        int i = 0;
        int *pi;
        int *valeur;
        val  = 0;
        if(argc != 2)
                return EXIT_FAILURE;

        tab = malloc(atoi(argv[1])*sizeof(pthread_t));

        for (i =0;i<atoi(argv[1]);i++) {
                //passage de l'argument par le pointeur
                pi = malloc(sizeof(int));
                *pi = (int) (10*((double)rand())/ RAND_MAX);
                pthread_create(&tab[i],NULL,funcThread,pi);
        }
        for (i =0;i<atoi(argv[1]);i++) {
                if(pthread_join(tab[i],(void**)&valeur) !=0){
                        printf("pthread_join\n");
                        exit(1);
                }
                //printf("La valeur de %d est %d\n",tab[i], valeur);
        }
        printf("Valeur finale : %d",val);
}
```

Réveil de threads

```
//Modifier le programme précédent pour que la valeur finale soit affichée non plus par le programme principal, mais par une nouvelle thread créée au départ. Celle-ci, après sa création, doit se bloquer en attendant que la somme de toutes les valeurs aléatoires soit complétée.
//La dernière thread à ajouter sa valeur aléatoire utilisera pthread_cond_signal pour signifier à la première qu'elle peut afficher la valeur de la globale.
//A VOIR CETTE PARTIE
//On utilisera une variable statique dans la fonction appelée à la création de la thread pour compter le nombre de ses appels, et repérer ainsi le dernier appel (le problème peut se résoudre sans d'autres variables globales que celle additionnant les valeurs aléatoires).
//Exemple d'appel :
//$PWD/bin/thread_wait 10
pthread_t *tab;
int valeur = 0;
int val_thread_max = 0;
int val_thread_courant = 0;
int condition =0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void *funcThread(void *arg){
	pthread_mutex_lock(&mutex);
	val_thread_courant++;
	valeur += (int) (10*((double)rand())/ RAND_MAX);
	printf("Valeur de val : %d\n",valeur);
	if(val_thread_courant == val_thread_max){	//On est dans le dernier Thread;
		condition = 1;
		pthread_cond_signal(&cond);
	}
	pthread_mutex_unlock(&mutex);
	pthread_exit((void*)0); return NULL;
}
void *funcAttente(int *arg){
	pthread_mutex_lock(&mutex);
	//ici il n'y a qu'un seul thread qui doit attendre.
	while (!condition) {
		pthread_cond_wait(&cond, &mutex);
	}
	pthread_mutex_unlock(&mutex);
	printf("Valeur finale : %d\n",valeur);
	pthread_exit((void*)0);
}

int main(int argc, char *argv[]) {
	int i = 0;
	int *pi;
	int *valeur;
	if(argc != 2)
		return EXIT_FAILURE;

	val_thread_max = atoi(argv[1]);
	tab = malloc((atoi(argv[1])+1)*sizeof(pthread_t));

	//Création du premier Thread
	pthread_create(&tab[0],NULL,funcAttente,NULL);

	for (i =1;i<atoi(argv[1])+1;i++) {
		//passage de l'argument par le pointeur
		pi = malloc(sizeof(int));
		*pi = i;	//Pointeur vers variable i;
		pthread_create(&tab[i],NULL,funcThread,pi);
	}
	for (i =1;i<atoi(argv[1])+1;i++) {
		if(pthread_join(tab[i],(void**)&valeur) !=0){
			printf("pthread_join\n");
			exit(1);
		}
	}
}
```

Détachement

```
//Création

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

//Dans le thread

    pthread_detach(pthread_self());
```

Synchronisation par broadcast (barrière)

```
//Une barrière est un mécanisme de synchronisation. Elle permet à N threads de prendre rendez-vous en un point donné de leur exécution.
//Dès que l'une d'entre elles atteint la barrière, elle reste bloquée jusqu'à ce que toutes les autres y arrivent. Lorsque toutes sont arrivées, chacune peut alors reprendre son exécution.
//Ecrire une fonction, qu'on nommera wait_barrier prenant en argument un entier N, permettant à N threads de se synchroniser sur une barrière. Testez votre programme avec la thread suivante :
//void* thread_func (void *arg) {
//        printf ("avant barriere\n");
//        wait_barrier (((int *)args)[0]);
//        printf ("après barriere\n");
//        pthread_exit ( NULL);
//}
//En exécutant votre programme avec 2 threads, il devra afficher :
//avant barrière
//avant barrière
//après barrière
//après barrière
//En d'autres termes, on veut que tous les messages « avant barrière » soient affichés avant les messages « après barrière ».
//Exemple d'appel :
//$PWD/bin/thread_broadcast 10
int val_thread_max;
int val_inside_barrier=0;
int condition = 0;
int *args;
pthread_t *tabThread;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void wait_barrier(int arg){
        val_inside_barrier++;
        pthread_mutex_lock(&mutex);
        if (val_inside_barrier == arg) {
                condition = 1;
                pthread_cond_broadcast(&cond);
        }else {
                while(!condition){  pthread_cond_wait(&cond, &mutex); }
        }
        pthread_mutex_unlock(&mutex);
        return;
}
void* thread_func (void *arg) {
        printf ("avant barriere\n");
        wait_barrier(((int *)args)[0]);
        printf ("après barriere\n");
        pthread_exit ( NULL);
}
int main(int argc, char *argv[]) {
        int i = 0;
        int *pi;
        int *valeur;
        if(argc != 2)
                return EXIT_FAILURE;
        val_thread_max = atoi(argv[1]);
        tabThread = malloc(val_thread_max *sizeof(pthread_t));
        args = malloc(val_thread_max *sizeof(int));
        for (i =0;i<val_thread_max;i++) {
                //passage de l'argument par le pointeur
                args[i] = val_thread_max - i;
                pi = malloc(sizeof(int));
                *pi = i;
                pthread_create(&tabThread[i],NULL,thread_func,pi);
        }
        for (i = 0; i < val_thread_max; i++) {
                if(pthread_join(tabThread[i],NULL ) !=0){
                        printf("pthread_join\n");
                        exit(1);
                }
        }}
```

Processus légers et fichiers

N Threads pour N fichiers

//Ecrire un programme C prenant plusieurs noms de fichiers en argument.

//Il doit créer autant de Threads que de fichiers, et les lancer en parallèle.
//La i-ème Thread créée doit appliquer la fonction ci-dessus sur le i-ème fichier de la liste des fichiers, et transmettre au programme principal le résultat de cette fonction.
//Le programme principal attend la terminaison de chaque Thread et teste son retour. S'il n'est pas nul, il affiche le nom du fichier posant problème sur le flux de sortie.
//Au final, le programme sort avec comme code de retour le nombre de fichiers qui ont posé problème (donc 0 si tout c'est bien passé).

//Pourquoi un void**?
//Chaque thread possède sa propre pile, dans la pile se trouve toute les variables locales.
//int retour dans le thread est locale, elle sera donc dans la pile. Ce sera donc perdu, car on va désallouer la pile.
//Erreur de seg : car on essaye d'accéder à une variable non allouée !
//SOLUTION : MALLOC pour réserver une zone mémoire dans le tas.
//Le retour ne sera plus un entier, mais un pointeur d'entier.
//Le thread va donc écrire sa valeur de retour dans le tas.
//Pour la récupérer, il faut dans l'apl a join, pouvoir modifier le pointeur.
//Il faut donc donner l'adresse de la variable status. Grâce à ce pointeur, join peut modifier pour le faire pointer sur la variable pointeur du thread;

```
void *funcThread(void *arg){
	int *t = 0;
	t = malloc(sizeof(int));
	printf("ValArg : %s\n",(char*)arg);
	*t = upper((char*)arg);
	pthread_exit((void*)t);
}

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

	int i;
	if(argc < 2)
		return EXIT_FAILURE;

	pthread_t* tabThread;
	tabThread = malloc((argc-1)*sizeof(pthread_t));

	char *pchar = 0;
	for (i = 1;i<argc;i++) {
		pchar = argv[i];
		pthread_create(&tabThread[i-1], NULL, funcThread, (void*)pchar);
		printf("thread %d\n",i);
	}

	//Attente de l'ensemble des threads;
	void *valRet = 0;
	int valRetour = 0;
	for (i = 1;i<argc;i++) {
			pthread_join(tabThread[i-1], (void**)&valRet);
			if((*(int*)valRet) !=0){
				printf("Le fichier %s pb \n",argv[i]);
				valRetour++;
			}
			free(valRet);
	}

	return valRetour;
}
```

N fichiers , N-K Threads

//On considère maintenant que le nombre de threads créées est inférieur au nombre de fichiers à traiter. Dès qu'une Thread a converti un fichier avec succès, elle doit passer à un autre fichier s'il en reste, et sinon se terminer. Si un fichier pose problème, elle se termine tout de suite en indiquant le fichier fautif au programme principal.
//Ecrire un nouveau programme C programmant cette stratégie, le premier argument sur la ligne de commande étant le nombre de Thread permis, les suivants étant les fichiers, en nombre supérieur. Le programme principal doit attendre la fin des Threads, tester leur retour et afficher les fichiers ayant posé problème. On notera qu'il peut y avoir des fichiers non examinés si toutes les Threads ont rencontré un fichier à problème.

```
//Nombre de fichiers restant
int *nbFichierRestant, *nbFichierMax;
//Nombre de threads
pthread_t* tabThread;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void *funcThread(void *arg){
	int *t = 0;
	t = malloc(sizeof(int));
	char ** arguments = (char**)arg;
	pthread_mutex_lock(&mutex);
	while (*nbFichierRestant>2) {		//Cad sans nom fichier et sans nb thread
		//printf("Valeur de nbFichierMax : %d et nbFichierRestant : %d\n",*nbFichierMax,*nbFichierRestant);
		printf("Thread : %ld - Traitement du fichier : %s\n",(long)pthread_self(),arguments[*nbFichierMax -
*nbFichierRestant+2]);
		*nbFichierRestant = *nbFichierRestant - 1;
		//printf("Valeur de nbFichierMax : %d et nbFichierRestant : %d\n",*nbFichierMax,*nbFichierRestant);
		pthread_mutex_unlock(&mutex);
		//Traitement du fichier
		*t = upper(arguments[(*nbFichierMax - *nbFichierRestant)+1]);
		pthread_mutex_lock(&mutex);
		printf("Thread : %ld - Fin traitement valeur de t: %d\n",(long)pthread_self(),*t);
		//Vérification de la valeur de *t retour possible
		if(*t!=0){
			pthread_mutex_unlock(&mutex); //unlock car sortie
			printf("Thread : %ld - fin du Thread\n",(long)pthread_self());
			pthread_exit((void*)arguments[(*nbFichierMax - *nbFichierRestant)+1]);
		}
	}
	printf("Thread : %ld - fin du Thread correct\n",(long)pthread_self());

	*t = 0; //Dans ce cas tout est OK
	pthread_mutex_unlock(&mutex);
	pthread_exit((void*)t);
}
int main(int argc, char *argv[]) {
	int i;
	if(argc < 2)
		return EXIT_FAILURE;

	tabThread = malloc((argc-1)*sizeof(pthread_t));  //On alloue l'emplacement du int dans le tas
	nbFichierMax = malloc((argc-1)*sizeof(int));
	nbFichierRestant = malloc((argc-1)*sizeof(int));
	*nbFichierMax = *nbFichierRestant = argc;

	for (i = 0;i<atoi(argv[1]);i++) {
		pthread_create(&tabThread[i], NULL, funcThread, (void *)argv); //On balance le tableau de char *
		printf("thread %d\n",i);
	}
	//Attente de l'ensemble des threads;
	void *valRet = 0;
	int valRetour = 0;
	for (i = 0;i<atoi(argv[1]);i++) {
			pthread_join(tabThread[i], (void**)&valRet);
			if((*(int*)valRet) != 0){
				printf("Le fichier %s pb \n",(char*)valRet);
				valRetour++;
			}
	}
	return valRetour;
}
```

Un producteur et un consommateur

```
/*
3 Un producteur et un consommateur
Nous voulons faire communiquer une thread Producteur et une thread Consommateur en utilisant une pile de taille fixe (un tableau de 100 caractères). Les valeurs empilées sont des caractères. La thread Producteur utilise la fonction Push() pour empiler un caractère au sommet de la pile et la thread Consommateur utilise la fonction Pop() pour désempiler une valeur du sommet de la pile. Une variable globale stack_size contrôle le sommet de la pile.
Programmez les fonctions Push() et Pop() décrites ci-dessus pour faire communiquer les threads Producteur et Consommateur, le corps de ces deux fonctions reposant respectivement sur les deux séquences de code définies par les deux macros suivantes, fournies dans le fichier .h en annexe :
#define PRODUCTEUR int c;  while((c = getchar()) != EOF){ push(c); }
#define CONSOMMATEUR while(1) { putchar(pop()); fflush(stdout); }
Écrire ensuite le programme main utilisant ces deux fonctions.
Exemple d'appel :
echo "123456789" | bin/producteur_consommateur
*/
int pile[SIZE];  //Tableau de 100 éléments
int stack_size = -1;  //Pour le moment il n'y a aucun élément dans mon tableau, il est vide.
/* Les conditions */
pthread_mutex_t mutex_stack = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_pop  = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_push  = PTHREAD_COND_INITIALIZER;
void push(int car) /* empiler 1 caractère */
{
   /* Verrouillage des opérations de la thread */
   pthread_mutex_lock(&mutex_stack);
   /* Si le tableau est plein */
   while (stack_size == SIZE){
        printf("%ld | -> Attente consommateur\n", (long ) pthread_self());
       /* Le consommateur doit prendre des caractères */
       /* On attend qu'il nous réveil pour produire des caractères */
       pthread_cond_wait(&cond_pop,&mutex_stack);
   }
   /*On est ici si nous sommes autorisé à produire */
   stack_size++;  /* Note : Dans le premier passage, il passe de -1 à 0 */
   pile[stack_size] =, car ;
   if((int),car == 10)
     printf("%ld | Empilation du caractère de fin\n",(long ) pthread_self());
   else
     printf("%ld | Empilation de : %c\n",(long ) pthread_self(), pile[stack_size]);
   /* Incrémentation de la taille du tableau */
   /* Si c'est le premier caractère dans le tableau */
   if (stack_size == 0){
     /* Nous pouvons réveiller le consommateur pour qu'il consomme */
     printf("%ld | Réveil du consommateur \n", (long ) pthread_self());
     pthread_cond_signal(&cond_push);  // On envoie le signal sur ce que lui attend, push cad nous.
   }

   /* Fin des opérations de la thread */
   pthread_mutex_unlock(&mutex_stack);
}

int  pop(){ /* dépiler 1 caractère */
   int car;
    /* Verrouillage des opérations de la thread */
   pthread_mutex_lock(&mutex_stack);
   /* Si le tableau est vide */
   while (stack_size == -1){
       /* Le producteur doit produire des caractères */
       /* On attend qu'il nous réveil pour consommer des caractères */
       printf("%ld | Le consommateur s'endort\n", (long ) pthread_self());
       pthread_cond_wait(&cond_push,&mutex_stack);
       //printf("%ld | Valeur de stacksize au retour de condwait : %d \n", (long ) pthread_self(),stack_size);
   }
   /* On est ici si nous sommes autorisé à consommer */,
   car = pile[stack_size];
   stack_size --;
   if((int)car == 10)
```

```
    printf("%ld | Dépilation du caractère de fin\n",(long ) pthread_self());
  else
    printf("%ld | Dépilation de : %c\n",(long ) pthread_self(),car);
  /* Nous pouvons réveiller le producteur pour qu'il produit */
  /* Il reste un emplacement dans le tableau */
  if (stack_size == (SIZE-1)){
    /* Nous pouvons réveiller le producteur pour qu'il produit */
    printf("%ld | Réveil du producteur\n",(long ) pthread_self());
    pthread_cond_signal(&cond_pop);
  }
   /* Fin des opérations de la thread */
  pthread_mutex_unlock(&mutex_stack);
  return car;
}
void *producteur()
{
  int c;
  while((c = getchar()) != EOF)
  {
     push(c);
  }

  pthread_exit ((void*)0);
}
void *consommateur()
{
/* Version original */
//    while(1)
//    {
//      putchar(pop());
//              fflush(stdout);
//    }
  while(1)
  {
    //putchar(pop());
    pop(); //<- pour démonstration (l'affichage est inclut dans la fonction pop
              fflush(stdout);
  }
  pthread_exit ((void*)0);
}

int main (int argc, char* argv[])
{
  int *status;
  pthread_t pth1, pth2;
  stack_size = -1;
  /* producteur */
  if (pthread_create (&pth1, NULL, producteur, (void*)0) != 0) {
     perror("pthread_create \n");
     exit (1);
  }
  /* consommateur */
  if (pthread_create (&pth2, NULL, consommateur, (void*)0) != 0) {
     perror("pthread_create \n");
     exit (1);
  }
  /* consommateur */
  if (pthread_join(pth1, (void**) &status) != 0) {
     printf ("pthread_join");
     exit (1);
  }
  /* producteur */
  if (pthread_join(pth2, (void**) &status) != 0) {
     printf ("pthread_join");
     exit (1);
  }

  return 0;
 }
```

Des producteurs et des consommateurs

```
#define PRODUCTEUR int c;  while((c = getchar()) != EOF){ push(c); }
#define CONSOMMATEUR while(1) { putchar(pop()); fflush(stdout); }
/*
4 Des producteurs et des consommateurs
Reprenez l'exercice précédent en faisant échanger plusieurs producteurs et plusieurs consommateurs de façon concurrente, toujours à travers une seule pile. Votre programme prendra en argument deux nombres : le nombre de producteurs suivi du nombre de consommateurs.
*/
#define STACK_SIZE 2
int pile[STACK_SIZE];  //Tableau de 100 éléments
int stack_size = -1;  //Pour le moment il n'y a aucun élément dans mon tableau, il est vide.
/* Les conditions */
pthread_mutex_t mutex_stack = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_pop  = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_push  = PTHREAD_COND_INITIALIZER;
void push(int car) /* empiler 1 caractère */
{
                        /* Verrouillage des opérations de la thread */
                        pthread_mutex_lock(&mutex_stack);

                        /* Si le tableau est plein */
                        while (stack_size == STACK_SIZE){
                                printf("%ld | -> Attente consommateur\n", (long ) pthread_self());
                                 /* Le consommateur doit prendre des caractères */
                                 /* On attend qu'il nous réveil pour produire des caractères */
                                 pthread_cond_wait(&cond_pop,&mutex_stack);
                        }

                        /*On est ici si nous sommes autorisé à produire */
                        stack_size++;  /* Note : Dans le premier passage, il passe de -1 à 0 */
                        pile[stack_size] =, car ;
                        if((int),car == 10)
                                printf("%ld | Empilation du caractère de fin\n",(long ) pthread_self());
                        else
                                printf("%ld | Empilation de : %c\n",(long ) pthread_self(), pile[stack_size]);
                        /* Incrémentation de la taille du tableau */
                        /* Si c'est le premier caractère dans le tableau */
                        if (stack_size == 0){
                                /* Nous pouvons réveiller le consommateur pour qu'il consomme */
                                printf("%ld | Réveil du consommateur \n", (long ) pthread_self());
                                pthread_cond_broadcast(&cond_push);  // On envoie le signal sur ce que lui attend, push cad nous.
                        }
                        /* Fin des opérations de la thread */
                        pthread_mutex_unlock(&mutex_stack);
}

int pop(){ /* dépiler 1 caractère */
                        int car;
                         /* Verrouillage des opérations de la thread */
                        pthread_mutex_lock(&mutex_stack);
                        /* Si le tableau est vide */
                        while (stack_size == -1){
                                 /* Le producteur doit produire des caractères */
                                /* On attend qu'il nous réveil pour consommer des caractères */
                                 printf("%ld | Le consommateur s'endort\n", (long ) pthread_self());
                                 pthread_cond_wait(&cond_push,&mutex_stack);
                        }
                        /* On est ici si nous sommes autorisé à consommer */,
                        car = pile[stack_size];
                        stack_size --;
                        if((int)car == 10)
                                printf("%ld | Dépilation du caractère de fin\n",(long ) pthread_self());
                        else
                                printf("%ld | Dépilation de : %c\n",(long ) pthread_self(),car);

                        /* Nous pouvons réveiller le producteur pour qu'il produit */
```

```
                        /* Il reste un emplacement dans le tableau */
                        if (stack_size == (STACK_SIZE-1)){
                                /* Nous pouvons réveiller le producteur pour qu'il produit */
                                printf("%ld | Réveil du producteur\n",(long ) pthread_self());
                                pthread_cond_broadcast(&cond_pop);
                        }

                         /* Fin des opérations de la thread */
                        pthread_mutex_unlock(&mutex_stack);
                        return car;
}
void *producteur()
{
        PRODUCTEUR
        pthread_exit ((void*)0);
}
void *consommateur()
{
        CONSOMMATEUR
        pthread_exit ((void*)0);
}
pthread_t *TabStack;
int main (int argc, char* argv[])
{
                        int i = 0;
                        TabStack = malloc(STACK_SIZE*sizeof(char));

                        if(argc<3)
                                return EXIT_FAILURE;


                        int nbProducteur = atoi(argv[1]);
                        int nbConsommateur = atoi(argv[2]);

                        printf("Nombre total : %d\n", (nbConsommateur + nbProducteur));
                        //printf("Nombre total de l'entrée : %ld",strlen(STDIN_FILENO));
                        pthread_t *thread = malloc((nbConsommateur + nbProducteur) * sizeof(pthread_t));

                        for(i = 0; i<nbConsommateur;i++)
                                pthread_create(&thread[i], NULL, consommateur, NULL);

                        for(i = 0; i<nbProducteur;i++)
                                pthread_create(&thread[i+nbConsommateur], NULL, producteur, NULL);
                        for(i = 0; i<(nbConsommateur+nbProducteur);i++)
                                printf("Valeur du tableau : %ld\n", (long) thread[i]);

                        for(i = 0; i<(nbConsommateur+nbProducteur);i++)
                                pthread_join(thread[i], NULL);
        }
```

Chaine de threads et signaux

```
//On désire créer une chaîne de N threads (la Thread principale crée une Thread, qui à son tour en crée une autre, et ainsi de suite N fois) qui fonctionne de la manière suivante.	OK
//Au démarrage du programme, la Thread principale masque tous les signaux, démarre la chaîne de création puis attend que toutes les Threads soient créées avant d'afficher "Tous mes descendants sont créés".	OK
//Après leur création, toutes les Threads sauf la principale se bloquent en attendant que celle-ci les libère. Parallèlement, la Thread principale se bloque en attente d'un signal SIGINT émis par l'utilisateur avec un CTRL+C.
// A la délivrance de ce signal, elle se débloque et débloque les autres Threads de la chaîne puis attend enfin que toutes se soient terminées avant d'afficher "Tous mes descendants se sont terminés".
//NB : Seule la Thread principale doit pouvoir être interrompue par un signal. On rappelle que chaque Thread gère un masque de signaux qui lui est propre.
int threadCreation = 1;
int signalPasRecu = 1;
pthread_mutex_t threadCrea = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condCreation = PTHREAD_COND_INITIALIZER;
pthread_t *tabThread;

void *funcThread(void* arg){
	pthread_mutex_lock(&threadCrea);
	printf("Valeurs de pMaxThread : %d\n",(*(int*)arg));
	if((*(int*)arg>0)){ //On doit encore faire N Thread;
		(*(int*)arg)--;
		pthread_create(&tabThread[(*(int*)arg)], NULL, funcThread, (void*)arg);
	}else{		//On est au dernier Thread

		printf("Passage dans la fin \n");
		pthread_cond_broadcast(&condCreation);
		threadCreation = 0;

	}

	printf("Attente de la création de toutes les threads \n");
	/* ------ Attente de la création de toutes les threads ------*/
	while(threadCreation){
		pthread_cond_wait(&condCreation, &threadCrea);
	}

	//printf("Attente du dévérouillage par le thread 0 \n");
	/* ------ Attente du dévérouillage par le Thread 0 (main) ------*/
	while(signalPasRecu){
		pthread_cond_wait(&condCreation, &threadCrea);
	}
	pthread_mutex_unlock(&threadCrea);
	pthread_exit((void*)0);
}
int main(int argc, char *argv[]) {
	int i = 0;
	int maxThread;
	int *pmaxThread;
	/* ------ Vérification des arguments ------*/
	if(argc<2)
		return EXIT_FAILURE;
	/* ------ Masquage des signaux ------*/
	sigset_t ens; int sig;
	sigfillset(&ens);
	pthread_sigmask(SIG_SETMASK,&ens,NULL);
	/* ------ Récupération nombre chaine thread ------*/
	maxThread = atoi(argv[1]);
	pmaxThread = malloc(sizeof(int));
	*pmaxThread = maxThread-1;
	tabThread = malloc(maxThread * sizeof(pthread_t));
	/* ------ Creation chaine de pmaxThread ------*/
	pthread_create(&tabThread[maxThread], NULL, funcThread, (void*)pmaxThread);
	/* ------ Attente fin creation thread et message ------*/
	pthread_mutex_lock(&threadCrea);
	while(threadCreation){
		pthread_cond_wait(&condCreation, &threadCrea);
	}
```

```
	pthread_mutex_unlock(&threadCrea);
	printf("Tous mes descandants sont crées.\n ");
	/* ------ Attente signal user ------*/
	sigemptyset(&ens);
	sigaddset(&ens, SIGINT);
	int valReturn = 0;
	while(1){
		valReturn = 0;
		sigwait(&ens, &valReturn);
		/* ------ Réveil de l'ensemble des Threads ------*/
		if(valReturn == SIGINT){
			pthread_cond_broadcast(&condCreation);
			signalPasRecu = 0;
			break;
		}
	}

	/* ------ Attente de l'ensemble des Threads ------*/
	for(i = 0; i<maxThread;i++){
		pthread_join(tabThread[i], NULL);
	}
	printf("Tous mes descandants sont fini.\n ");

}
```

Arborescence de Threads

//On souhaite écrire un variante de la fonction précédente, de sorte que toutes les Threads d'un niveau L soient créées avant de commencer à créer celles du niveau L+1. Pour cela, écrire une fonction main qui
//prend sur sa ligne de commande la profondeur de Thread désirée ;
//lance une première Thread qui se bloque en attendant que toutes celles du niveau courant soient créées (utiliser la formule demandée à la question précédente) ;
//lance la variante de la fonction thread_func désirée ;
//affiche pour finir le nombre total de Threads créées (à partir d'une variable incrémentée à chaque création).
//Exemple d'appel :

```
int cond =0;
pthread_cond_t condthread = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int max = 0;
void* thread_func(void* arg) {
        int i, nb;
        int *param;
        int *lvl = (int*)arg;
        pthread_t *tid;
        nb = (*lvl)+1;
        pthread_mutex_lock(&mutex);
        while (!cond) {
            pthread_cond_wait(&condthread, &mutex);
        }
        cond = 0;
        pthread_mutex_unlock(&mutex);
        if (*lvl < max) {
            param = (int*)malloc(sizeof(int));
            *param = nb;
            tid = calloc(nb, sizeof(pthread_t));
            printf("%d cree %d fils\n", (int)pthread_self(), nb);
            pthread_mutex_lock(&mutex);
            for (i = 0; i < nb; i++) {
                pthread_create((tid+i), 0, thread_func, param);
            }
            pthread_cond_broadcast(&condthread);
            pthread_mutex_unlock(&mutex);
            cond = 1;
            for (i = 0; i < nb; i++)
                pthread_join(tid[i], NULL);
        }
        if (*lvl > 1)
            pthread_exit ( (void*)0);

        return (void*)0;
    }
int main(int argc, char *argv[]) {
    if(argc<2)
        return EXIT_FAILURE;

    max = atoi(argv[1]);
    printf("Passage\n");
    int *pi = 0;
    int i = 1;
    pi = malloc(sizeof(int));
    *pi = i;
    pthread_t pthread;
    printf("Passage\n");
    pthread_mutex_lock(&mutex);
    pthread_create(&pthread, NULL, thread_func,(void*)pi );
    pthread_cond_broadcast(&condthread);
    cond = 1;
    pthread_mutex_unlock(&mutex);
    pthread_join(pthread, NULL);
}
```

Pipes et tubes

Tube et majuscules

```
//1 Tube et majuscules
//
//On souhaite écrire un programme qui mette en majuscules les chaînes de caractères entrées par l'utilisateur via le terminal. Ce programme lance 2 processus :
//
//le processus père créé un tube avec la fonction C pipe et un processus fils, récupère les messages utilisateur en les lisant sur l'entrée standard, puis les transmet à son fils ;
//le processus fils lit les messages de son père, les transcrit en majuscules avec la fonction C toupper, puis les redirige vers la sortie standard.
//Exemple d'appel :
//echo abcd.ext | $PWD/bin/pipe_maj

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

	int tube[2];
	char toto;
	int fin;
	pipe(tube);

	pid_t fils = 0;

	if((fils = fork()) == -1){
		perror("fork : \n");
	}
	if(fils != 0){//père
		close(tube[0]);
		while(!fin)
			if(scanf("%c",&toto)!=0){
				write(tube[1], &toto, sizeof(char));
			}
		close(tube[0]);
		close(STDIN_FILENO);
	}else{
		close(tube[1]);
		dup2(tube[0], STDIN_FILENO);
		while(!fin)
			if(read(STDIN_FILENO, &toto, sizeof(char))!=0){
				printf("%c",toupper(toto));
				fflush(stdout);
			}
		close(tube[0]);
		close(STDIN_FILENO);
		printf("fin ! \n");
	}


}
```

Tube nommé et majuscules

```
//On se propose de reprendre l'exercice précédent, mais au moyen d'un processus serveur dont les clients sont sans lien de parenté avec lui. L'utilisateur doit pouvoir lancer le serveur puis, éventuellement à travers une autre fenêtre de terminal, lancer un programme client qui s'adresse au serveur grâce à un tube nommé, fourni par la fonction C mkfifo.
//
//Dans cette partie, il s'agit d'écrire le serveur. Il prend sur la ligne de commande le nom du tube à créer, et le crée. Le serveur se met à l'écoute sur ce tube et affiche la transcription en majuscules dans son propre flux de sortie. Il doit pouvoir être interrompu par un ^C, et doit alors fermer le tube nommé et le détruire.
//
//Exemple d'appel :
//$PWD/bin/serveur_maj minmaj &

int fdread;
char *file;
int ecoute = 1;
char c;


void *func(int sig){
	printf("Interruption du serveur\n");
	close(fdread);
	unlink(file);
	ecoute = 0;
	return 0;
}


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

	if(argc<2){
		perror("Arg invalide : \n");
		exit(2);
	}

	//Copy de l'@ du fichier
	file = malloc(sizeof(argv[1]));
	strcpy(file, argv[1]);

	//Signaux
	sigset_t sig;
	sigfillset(&sig);
	sigdelset(&sig, SIGINT);
	sigprocmask(SIG_SETMASK, &sig, &sig); //Retour ancien masque

	struct sigaction action;
	action.sa_flags = 0;
	action.sa_mask = sig;
	action.sa_handler = (void *)func;
	sigaction(SIGINT, &action, 0);

	if(mkfifo(argv[1], S_IRUSR|S_IWUSR) == -1)
		perror("mkfifo");

	if((fdread = open(argv[1],O_RDONLY)) == -1){
		perror("open ");
		exit(2);
	}
	while(ecoute){
		read(fdread, &c, sizeof(char));
		printf("%c",toupper(c));
	}
}
```

Tube nommé et minuscules

```
/*
3 Tube nommé et minuscules

Dans cette partie, il s'agit d'écrire le client qui prend en argument le nom du tube nommé sur la ligne de commande. Il lit les messages tapés par l'utilisateur dans le flux d'entrée, puis les réécrit sur le tube à chaque retour chariot.
Exemple d'appel :
$PWD/bin/client_maj minmaj
*/

int fdwrite;
char *file;
int ecoute = 1;
size_t len;
char *c;

void *func(int sig){
	printf("Interruption du serveur\n");
	close(fdwrite);
	unlink(file);
	ecoute = 0;
	return 0;
}


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

	if(argc<2){
		perror("Arg invalide : \n");
		exit(2);
	}

	//Copy de l'@ du fichier
	file = malloc(sizeof(argv[1]));
	strcpy(file, argv[1]);

	//Signaux
	sigset_t sig;
	sigfillset(&sig);
	sigdelset(&sig, SIGINT);
	sigprocmask(SIG_SETMASK, &sig, &sig); //Retour ancien masque

	struct sigaction action;
	action.sa_flags = 0;
	action.sa_mask = sig;
	action.sa_handler = (void *)func;
	sigaction(SIGINT, &action, 0);

	//mkfifo(argv[1], S_IRUSR|S_IWUSR);

	if((fdwrite = open(argv[1],O_WRONLY)) == -1){
		perror("open ");
		exit(2);
	}

	while(ecoute){
		read(STDIN_FILENO, &c, 1);
		write(fdwrite, &c, 1);
	}

}
```

Introduction à la communication inter-processus (IPC)

Remontée par partage de mémoire

```
/*
1 Remontée par partage de mémoire
Définir les macros nécessaires au canevas fourni pour que les valeurs soient remontées au processus principal via un segment de mémoire partagée,disponible en POSIX à l'aide des fonctions C shm_open, shm_unlink, mmap, munmap et ftruncate.
*/
int main(int argc, char *argv[]){
 int id, n, i, total, res;
 int *ptr = NULL;
 int *pids;
 n = (argc < 2) ? 0 : strtol(argv[1], NULL, 10);
 if (n <= 0) {
  fprintf(stderr, "Usage: %s nombre\n", argv[0]);
  exit(EXIT_FAILURE);
 }
 /* Demande un descripteur de ressource partagee, eventuellement deja creee. */
 if ((id = shm_open("monshm",O_RDWR | O_CREAT,0600)) == -1) {
  perror("Echec de l'allocation du descripteur\n");
  exit(errno);
 }
 /* Allocation pour stocker n entiers*/
 if(ftruncate(id,sizeof(int)*n) == -1){
  fprintf(stderr, "Echec d'allocation ftruncate\n");
 }
 /* Allocation pour stocker n entiers*/
 if ((ptr = mmap(NULL, sizeof(int)*n, PROT_READ|PROT_WRITE, MAP_SHARED, id, 0)) == MAP_FAILED){
  fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
  //exit(errno);
 }
 //Une fois shm_open ftruncate mmap fait, la mémoire est alloué et segmenter correctement.
 pids = malloc(n * sizeof(int));
 for(i=0; i<n; i++){
  int pid = fork();
  if (pid == -1){
   perror("fork");
   return -1;
  } else if (pid) {
   pids[i] = pid;
  } else {
   srand(time(NULL)*i);
   /* Ecriture dans la ressource partagee */
   ptr[i] = (int) (10*(float)rand()/ RAND_MAX);
   exit(EXIT_SUCCESS);
  }
 }
 for(i=0; i<n; i++){
  int status;
  waitpid(pids[i], &status, 0);
 }
 total = 0;
 for(i=0; i<n; i++){
  /* Lecture dans la ressource partagee */
  res = ptr[i];
  printf("pid %d envoie %d\n", pids[i], res);
  total += res;
 }
 free(pids);
 printf("total: %d\n", total);
 /* Liberation de la ressource */
 munmap(&res, sizeof(int)*n);
 if (shm_unlink("monshm")) {
  fprintf(stderr, "Ressource partagee mal rendue\n");
  exit(EXIT_FAILURE);
 }
 return EXIT_SUCCESS;
}
```

Remontée par file de messages

```
/*
3 Remontée par file de messages

Adaptez l'exercice précédent cette fois à l'utilisation d'une file de messages pour envoyer chaque valeur aléatoire. Vous utiliserez les fonctions C mentionnées dans le cours. Il faudra parfois définir une variable supplémentaire dans les macros, et à l'inverse certaines macros ne produiront rien, file et mémoire partagée étant des techniques sensiblement différentes. Des avertissements de variables inutilisées pourront en résulter à la compilation.
Exemple d'appel :
$PWD/bin/remonte_ipc 4
*/

int main(int argc, char *argv[]){
	int id, n, i, total, res;
	int *ptr = NULL;
	int *pids;
	mqd_t mqdesc;
	struct mq_attr attr;
	mq_unlink("./file");
	ptr = malloc(n*sizeof(int));
	n = (argc < 2) ? 0 : strtol(argv[1], NULL, 10);
	if (n <= 0) {
		fprintf(stderr, "Usage: %s nombre\n", argv[0]);
		exit(EXIT_FAILURE);
	}

	if ((mqdesc = mq_open("./file", O_RDWR|O_CREAT, 0666,&mqattr) == -1)) {
		perror("Echec de l'allocation du descripteur\n");
		exit(errno);
	}
	for(i=0; i<n; i++){
		int pid = fork();
		if (pid == -1){
			perror("fork");
			return -1;
		} else if (pid) {
			if (mq_getattr(mqdesc, & attr) != 0) {
				perror("mq_getattr");
				exit(EXIT_FAILURE);
			}
			taille = attr.mq_msgsize;
			mq_receive(mqdesc,&ptr[i], taille, NULL);
			total += *ptr;
			pids[i] = pid;
			printf("Message : %d\n",*ptr);
		} else {
			srand(time(NULL)*i);
			/* Ecriture dans la file */
			mq_send(mqdesc,(int) (10*(float)rand()/ RAND_MAX),sizeof(int), 0);
			//ptr[i] = (int) (10*(float)rand()/ RAND_MAX);
			exit(EXIT_SUCCESS);
		}
	}
	for(i=0; i<n; i++){
		int status;
		waitpid(pids[i], &status, 0);
	}
	total = 0;
	for(i=0; i<n; i++){
		/* Lecture dans la ressource partagee */
		res = ptr[i];
		printf("pid %d envoie %d\n", pids[i], ptr[i]);
	}
	free(pids);
	printf("total: %d\n", total);
	return EXIT_SUCCESS;
}
```

Introduction aux sémaphores

Barrière par sémaphores

/*
1 Barrière par sémaphores
Une barrière est un mécanisme de synchronisation. Elle permet à N processus de prendre rendez-vous en un point donné de leur exécution. Quand un des processus atteint la barrière, il reste bloqué jusqu'à ce que tous les autres arrivent à la barrière. Lorsque les N processus sont arrivés à la barrière, chacun peut alors reprendre son exécution.
Sans utiliser de compteur partagé, programmez la fonction wait_barrier prenant en argument un entier N et qui permet à N processus de se synchroniser sur une barrière. Écrire ensuite un programme main créant par fork autant de processus que demandé sur la ligne de commande, chacun exécutant le code suivant :
void process (int NB_PCS) {
  printf ("avant barrière);
  wait_barrier (NB_PCS);
  printf ("après barrière);
  exit (0);
}
L'affichage devra être le suivant :
avant barrière
avant barrière
après barrière
après barrière
En d'autres termes, on veut que tous les messages « avant barrière » soient affichés avant les messages « après barrière ». On utilisera les fonctions POSIX sem_wait et sem_post, ainsi que sem_close et sem_unlink pour finir.
Exemple d'appel :
bin/posix_barrier 4
*/

```c
int nbProcess = 0;
sem_t *barrier;
void wait_barrier(int NB_PCS){
	int i = 0;
	if(NB_PCS != nbProcess)
		sem_wait(barrier);
	else {
		for (i=0;i<nbProcess;i++) {
			printf("Déblocage de i : %d",i);
			sem_post(barrier);
		}

	}
	sem_close(barrier);
}
void process (int NB_PCS) {
	printf ("avant barrière");
	wait_barrier (NB_PCS);
	printf ("après barrière");
	exit (0);
}
int main(int argc, char *argv[]) {

	nbProcess = atoi(argv[1]);
	int i;

	if((barrier = sem_open("/sem", O_CREAT|O_EXCL|O_RDWR,0666,0))){
		if(errno != EEXIST){
			perror("sem_open");
			exit(1);
		}
	}

	for( i = 0; i<nbProcess;i++){
		if(fork() != 0)
			break;
	}
	printf("Valeur de i = %d\n",i);
	process(i);
}
```

Encore des producteurs et des consommateurs

```
#define PRODUCTEUR int c;  while((c = getchar()) != EOF){ push(c); }
#define CONSOMMATEUR while(1) { putchar(pop()); fflush(stdout); }
/*
3 Encore des producteurs et des consommateurs
Reprenez l'exercice Processus légers et fichiers en remplaçant les Threads par des processus, et en construisant la pile dans un segment de mémoire partagée.
Exemple d'appel :
echo "123456789" | bin/prod_conso_partagees 5 3
*/
#define STACK_SIZE 2
sem_t *mutexpush, *mutexpop, *mutexCondPush, *mutexCondPop;
char *tabStack;
struct pile{
	char ptr[STACK_SIZE];
	int stack_size;
};
struct pile *stack;
void push(int car) /* empiler 1 caractère */
{
		printf("%d | Lancement du push \n",getpid());
		/* Verrouillage des opérations */
		sem_wait(mutexpush);
		/* Si le tableau est plein */
		while (stack->stack_size == STACK_SIZE){
			printf("%d | -> Attente consommateur\n", getpid());
			 /* Le consommateur doit prendre des caractères */
			 /* On attend qu'il nous réveil pour produire des caractères */
			sem_wait(mutexCondPush);
		}
		printf("%d | Autorisé à empiler\n",getpid());
		/*On est ici si nous sommes autorisé à produire */
		printf("%d | Valeur de stack : %d \n",getpid(),stack->stack_size);
		stack->stack_size++;  /* Note : Dans le premier passage, il passe de -1 à 0 */
		stack->ptr[stack->stack_size] =, car ;
		if((int),car == 10)
			printf("%d | Empilation du caractère de fin\n",getpid());
		else
			printf("%d | Empilation de : %c\n", getpid(),stack->ptr[stack->stack_size]);
		/* Si c'est le premier caractère dans le tableau */
		if (stack->stack_size == 0){
			/* Nous pouvons réveiller le consommateur pour qu'il consomme */
			printf("%d | Réveil du consommateur \n", getpid());
			sem_post(mutexCondPop);
		}
		/* Fin des opérations de la thread */
		sem_post(mutexpush);
}
int pop(){ /* dépiler 1 caractère */
		int car;
		printf("%d | Lancement de pop\n",getpid());
		 /* Verrouillage des opérations */
		sem_wait(mutexpop);
		/* Si le tableau est vide */
		while (stack->stack_size == -1){
			 /* Le producteur doit produire des caractères */
			 /* On attend qu'il nous réveil pour consommer des caractères */
			printf("%d | Le consommateur s'endort\n", getpid());
			//printf("%d | Valeur de stack : %d \n",getpid(), stack->stack_size);
			sem_wait(mutexCondPop);
			printf("%d | Reprise - Valeur de stack : %d\n",getpid(),stack->stack_size);

		}
		printf("%d | Autorisé à dépiler\n",getpid());
		/* On est ici si nous sommes autorisé à consommer */,
		car =stack->ptr[stack->stack_size];
		stack->stack_size --;
		if((int),car == 10)
```

```
                printf("%d | Depilation du caractere de fin\n",getpid());
        else
                printf("%d | Dépilation de : %c\n",getpid(),car);
        /* Nous pouvons réveiller le producteur pour qu'il produit */
        /* Il reste un emplacement dans le tableau */
        if (stack->stack_size == (STACK_SIZE-1)){
                /* Nous pouvons réveiller le producteur pour qu'il produit */
                printf("%ld | Réveil du producteur\n",getpid());
                sem_post(mutexCondPush);
        }
         /* Fin des opérations de la thread */
        sem_post(mutexpop);
        return car;
}
void *producteur()
{
    PRODUCTEUR
}
void *consommateur()
{
    CONSOMMATEUR

}
int main (int argc, char* argv[])
{
        int i = 0;
        int shmDesc = 0, shmDescStack = 0;
        if(argc<3)
                return EXIT_FAILURE;
        int nbProducteur = atoi(argv[1]);
        int nbConsommateur = atoi(argv[2]);
        printf("Nombre total : %d\n", (nbConsommateur + nbProducteur));
        //Vérification de la présence du descripteur;
        shm_unlink("monshm");
        shm_unlink("monshmStack");
        sem_unlink("mutexpushsem");
        sem_unlink("mutexpopsem");
        sem_unlink("mutexcondsempop");
        sem_unlink("mutexcondsempush");
        /*      POUR LA PILE PARTAGEE       + COMPTEUR      */
        /* Ouverture de la pile et création dans la mémoire */
        if ((shmDesc = shm_open("monshm",O_RDWR | O_CREAT,0600)) == -1) {
                perror("Echec de l'allocation du descripteur\n");
                exit(errno);
        }
        /* Allocation pour stocker STACK_SIZE entiers*/
        if(ftruncate(shmDesc,sizeof(char)*STACK_SIZE) == -1){
                fprintf(stderr, "Echec d'allocation ftruncate\n");
        }
        /* Segmentation pour stocker STACK_SIZE entiers*/
        if ((stack = mmap(NULL, sizeof(char) * STACK_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, shmDesc, 0))
== MAP_FAILED){
                fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
                //exit(errno);
        }
        /* Initialisation */
        stack->stack_size = -1;
        /* Ouverture des mutex */
        if((mutexpush = sem_open("mutexpushsem",O_CREAT|O_EXCL|O_RDWR,0666,1)) == SEM_FAILED){
                if(errno != EEXIST){
                        perror("sem_open");
                        exit(1);
                }
        }
        if((mutexpop = sem_open("mutexpopsem",O_CREAT|O_EXCL|O_RDWR,0666,1)) == SEM_FAILED){
                if(errno != EEXIST){
                        perror("sem_open");
                        exit(1);
                }
```

```
        }
        if((mutexCondPop = sem_open("mutexcondsempop",O_CREAT|O_EXCL|O_RDWR,0666,0)) == SEM_FAILED){
                if(errno != EEXIST){
                        perror("sem_open");
                        exit(1);
                }
        }
        if((mutexCondPush = sem_open("mutexcondsempush",O_CREAT|O_EXCL|O_RDWR,0666,0)) == SEM_FAILED){
                if(errno != EEXIST){
                        perror("sem_open");
                        exit(1);
                }
        }
        int *pids = malloc((nbConsommateur + nbProducteur) * sizeof(int));
        for(i = 0; i<nbConsommateur;i++){
                printf("PID : %d | nbConsommateur : %d\n",getpid(),i);
                if(fork()==0)//Consommateur
                {       printf("PID : %d | lancementConsommateur : %d\n",getpid(),i);
                        consommateur();
                }
        }
        for(i = nbConsommateur; i<(nbProducteur+nbConsommateur);i++){
                printf("PID : %d | nbProducteur : %d\n",getpid(),i);
                if(fork()==0)//Producteur
                {       printf("PID : %d | lancementProducteur :%d\n",getpid(),i-nbConsommateur);
                        producteur();
                }
        }

        wait(NULL);       //Attention il y aura des zombies...
}
```

Synchronisation d'affichage entre processus en utilisant des sémaphores

```
int nbProcess = 0;
int nbProcessCreate = 0;
sem_t *sem;

//Nous considérons qu'un processus crée N processus fils. Chaque processus fils possède un identifiant unique (1, …, N).
//Chacun commence par s'endormir pendant (N % identifiant) secondes.
//A son réveil il affiche son identifiant et son Pid, cependant ces affichages doivent être faits par ordre croissant d'identifiant.
//Pour assurer une telle synchronisation, vous devrez utiliser des sémaphores anonymes.

//Le processus père doit attendre la terminaison de tous ses fils avant de se terminer lui aussi.

void process(int i ){
	//printf("Valeur de i : %d\n",i);
	sleep(i%getpid());
	if(i>0)
		sem_wait(&sem[i-1]);
	printf("PID : %d identifiant : %d\n",getpid(),i);
	if(i<nbProcess-1)
		sem_post(&sem[i]);

}


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

	nbProcess = atoi(argv[1]);
	printf("Père : %d\n",getpid());
	int i;
	sem = malloc(sizeof(sem_t)*nbProcess);


	for( i = 0; i<nbProcess-1;i++){
		sem_init(&sem[i], 0, 0);
	}


	nbProcessCreate = nbProcess;
	for( i = 0; i<nbProcess-1;i++){
		nbProcessCreate--;
		if(fork() != 0)
			break;
	}
	process(i);

	wait(NULL);



}
```

Une messagerie instantanée en mémoire partagée

Un serveur de messagerie instantanée

```
int idServeur, idClient;
struct myshm *shmServeur, *shmClient;
struct message messageClient;
char tabClient[MAX_USERS][TAILLE_MESS];
int i = 0, nbUserCourant = 0, emplacementVide = 0;
char utilisateur[TAILLE_MESS];
/*
1 Un serveur de messagerie instantanée
Le serveur est appelé avec en ligne de commande l'identifiant par lequel il sera connu de tous les clients. Au lancement, le programme
serveur crée un segment de mémoire partagée dont le nom est son identifiant. Le serveur lit ensuite les requêtes de ses clients dans ce
segment. Les requêtes auront toutes le même format décrit dans le ".h" fourni en annexe, savoir
struct message {
  long type;
  char content[1024];
}
où
   type permet de connaître la sémantique de la requête :
      connexion,
      diffusion de message,
      déconnexion ;
   content fournit des informations supplémentaires suivant le champ type :
      identifiant du client,
      contenu du message à diffuser.
Lorsqu'il traite une requête de connexion, le serveur mémorise l'identifiant du client dans un tableau. Une requête de déconnexion
amène le serveur à ôter le client de ce tableau. Pour pouvoir diffuser des informations vers ses clients, le serveur utilise des segments de
mémoire partagée créés par ces derniers.
Exemple d'appel :
bin/chat_server semserver &

*/
int main(int argc,char *argv[]){

      shm_unlink(argv[1]);
      /* Segment de mémoire serveur */
      if ((idServeur = shm_open(argv[1],O_RDWR| O_EXCL| O_CREAT,0777)) == -1) {
            perror("Echec de l'allocation du descripteur\n");
            exit(errno);
      }
      /* Allocation pour stocker 1 entiers*/
      if(ftruncate(idServeur,sizeof(struct myshm)) == -1){
            fprintf(stderr, "Echec d'allocation ftruncate\n");
      }
      /* Segmentation pour stocker 1 entiers*/
      if ((shmServeur = mmap(NULL, sizeof(struct myshm), PROT_READ|PROT_WRITE, MAP_SHARED, idServeur, 0)) ==
MAP_FAILED){
            fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
            //exit(errno);
      }
      sem_init(&shmServeur->sem,1,1);
      printf("Fin d'initialisation du serveur.\n");
      while(1){
            sem_wait(&shmServeur->sem);
            while(shmServeur->write != 0){ //Traitement de la requête
             //Récupération du message
             messageClient = shmServeur->messages[shmServeur->write-1];
              switch(messageClient.type){
                case 1: //Connexion
                  printf("Connexion de %s!\n",messageClient.content);
                  printf("Nombre d'user : %d\n",nbUserCourant);
                  //Ajout de l'utilisateur
                  for(i = 0 ; i<nbUserCourant;i++){
                        if(!strcmp(tabClient[i],"")){
                          emplacementVide = 1;
                          strcpy(tabClient[i],messageClient.content);
```

```
                }
            }
            if(!emplacementVide){
            strcpy(tabClient[nbUserCourant],messageClient.content);
            nbUserCourant++;
            }
            //Affichage de l'utilisateur
            for(i=0;i<nbUserCourant;i++)
            printf("User connecté: %s\n",tabClient[i]);
                break;
            case 2:     //Diffusion des messages vers les utilisateurs connectés.
            for(i = 0;i<nbUserCourant;i++){
                                    if(!strcmp(tabClient[i],""))     //Dans le cas d'un troue dans la table.
                continue;
        printf("Envoi vers \"%s\" de :%s\n",tabClient[i],messageClient.content);
                            if ((idClient = shm_open(tabClient[i],O_RDWR,0)) == -1) {
                perror("Echec de l'allocation du descripteur\n");
                exit(errno);
                            }

                            if ((shmClient = mmap(NULL, sizeof(struct myshm), PROT_READ|PROT_WRITE,
MAP_SHARED, idClient, 0)) == MAP_FAILED){
                                fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
                            }
                            sem_wait(&shmClient->sem);
                                shmClient->messages[shmClient->write] = messageClient;
                                shmClient->write++;
                                shmClient->read++;
                            sem_post(&shmClient->sem);
                            shmServeur->read++;
                            shmServeur->nb++;
                        }//for
                        break;
                    case 3:     //Déconnexion

                        strcpy(utilisateur,messageClient.content);

                        printf("Déconnexion de: %s\n",utilisateur);

                        for(i=0;i<nbUserCourant;i++)
        if(!strcmp(tabClient[i],utilisateur)){      //On a trouvé l'utilisateur
          printf("Suppression de %s\n",utilisateur);
                                for(;i<nbUserCourant-1;i++){
         printf("Déplacement de %s vers %s\n",tabClient[i+1],tabClient[i]);
                                strcpy(tabClient[i],tabClient[i+1]);
                                }
                                memset(tabClient[i],0,sizeof(tabClient[i]));
                            }
                        break;
                }
                shmServeur->write--;
            }
            sem_post(&shmServeur->sem);
        }
}
```

Un client de messagerie instantanée

```
/*
2 Un client de messagerie instantanée
Le client de messagerie instantanée doit d'abord créer deux segments de mémoire partagées, à l'aide de shm_open, permettant de dialoguer avec le serveur, un pour l'émission et l'autre pour la réception. Il prend en ligne de commande les deux identifiants à donner aux deux appels à cette fonction.
Ensuite, il permet l'accès en lecture-écriture à ces segments à l'aide de la fonction mmap.
Enfin, il lance deux Threads :
   l'un qui lit les envois du serveur sur le premier segment, et en fait l'écho sur le flux sortie standard ;
   l'autre qui lit les entrées de l'utilisateur sur le flux d'entrée standard, et qui les écrit sur l'autre segment, à destination du serveur.
A son initialisation, le client aura aussi installé un gestionnaire du signal ^C afin de permettre à l'utilisateur de terminer le programme proprement, notamment pour libérer les ressources partagées.
Exemple d'appel :
bin/chat_client semclient semserver
*/
int idClient, idServeur;
struct myshm *shmClient, *shmServeur;
char nomClient[TAILLE_MESS];
char nomServeur[TAILLE_MESS];
char textToSend[TAILLE_MESS];
void funcThreadEcriture(void *arg){
	struct message messageClient;
	/* Phase d'enregistrement sur le serveur */
	sem_wait(&shmServeur->sem);
	messageClient.type =  1;
	strcpy(messageClient.content,nomClient);
	shmServeur->messages[shmServeur->write] = messageClient;
	shmServeur->write++;
	sem_post(&shmServeur->sem);
	printf("Fin d'enregisrement sur le serveur de %s\n",nomClient);
	while(1){
		//printf("Votre texte : ");
		//read(0,&textToSend,sizeof(textToSend));
		scanf("%s",&textToSend);
		if(strlen(textToSend)!=0){
			//sem_wait(&shmServeur->sem);
			sem_wait(&shmClient->sem);
			messageClient.type = 2;
			strcpy(messageClient.content,textToSend);

			shmServeur->messages[shmServeur->write] = messageClient;
			shmServeur->write++;

			shmClient->nb++;
			sem_post(&shmClient->sem);
			sem_post(&shmServeur->sem);
		}
	}
}
void funcThreadLecture(void *arg){
	struct message messageClient;
	while(1){

		sem_wait(&shmClient->sem);
		if(shmClient->write != 0){
			messageClient = shmClient->messages[shmClient->write-1];
			printf("Recu : %s\n",messageClient.content);
			shmClient->read++;
			shmClient->nb++;
			shmClient->write--;
		}
		sem_post(&shmClient->sem);
	}
}
void sigIntTrt(int sig){
	struct message messageClient;
	//On se désenregistre
	sem_wait(&shmServeur->sem);
```

```
        messageClient.type = 3;
        strcpy(messageClient.content,nomClient);
        shmServeur->messages[shmServeur->write] = messageClient;
        shmServeur->write++;
        sem_post(&shmServeur->sem);
        munmap(&shmClient,sizeof(struct myshm));
        shm_unlink(nomClient);
        printf("Fin de la conversation \n");
        exit(0);
}
int main(int argc,char *argv[]){
        pthread_t threadEcriture, threadLecture;
        sigset_t ens;
        sigfillset(&ens);
        sigdelset(&ens,SIGINT);
        struct sigaction action;
        action.sa_mask = ens;
        action.sa_flags = 0;
        action.sa_handler =sigIntTrt;
        sigaction(SIGINT,&action,NULL);
        shm_unlink(argv[1]);
        printf("Valeur argv1 : %s, argv2 : %s\n",argv[1], argv[2]);
        /* Segment de mémoire client */
        if ((idClient = shm_open(argv[1],O_RDWR | O_CREAT,0600)) == -1) {
                perror("Echec de l'allocation du descripteur\n");
                exit(errno);
        }
        /* Allocation pour stocker 1 entiers*/
        if(ftruncate(idClient,sizeof(struct myshm)) == -1){
                fprintf(stderr, "Echec d'allocation ftruncate\n");
        }
        /* Segmentation pour stocker 1 entiers*/
        if ((shmClient = mmap(NULL, sizeof(struct myshm), PROT_READ|PROT_WRITE, MAP_SHARED, idClient, 0)) ==
MAP_FAILED){
                fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
                //exit(errno);
        }
        printf("fin ouverture client\n");
        strcpy(nomClient,argv[1]);
        strcpy(nomServeur,argv[2]);
        /* Segment de mémoire serveur */
        if ((idServeur = shm_open(argv[2],O_RDWR,0)) == -1) {
                perror("Echec de l'allocation du descripteur\n");
                exit(errno);
        }
        /* Segmentation pour stocker 1 entiers*/
        if ((shmServeur = mmap(NULL, sizeof(struct myshm), PROT_READ|PROT_WRITE, MAP_SHARED, idServeur, 0)) ==
MAP_FAILED){
                fprintf(stderr, "Echec de l'allocation de la ressource partagee\n");
                //exit(errno);
        }

        sem_init(&shmClient->sem,1,1);

        pthread_create(&threadEcriture,(void *)0,(void*)funcThreadEcriture,(void *)0);
        pthread_create(&threadLecture,(void *)0,(void*)funcThreadLecture,(void *)0);
        pthread_join(threadEcriture,(void*)0);

}
```

Introduction aux sockets

Remontée de valeurs par communication distante

/*
1 Remontée de valeurs par communication distante

Reprendre l'exercice Remontée de valeurs par partage de mémoire pour que les valeurs aléatoires transmises par les fils au père passent non plus par un segment de mémoire partagée, mais par datagrammes UDP. Le programme devra créer une socket UDP et la rendre accessible par bind, en la nommant grâce à la structure sockaddr_un propre au domaine af_unix des sockets. Il l'utilisera à la fois pour les envois (par sendto) et les réceptions (par recvfrom) de datagrammes (évidement d'autres solutions seraient possibles). Le programme prendra en premier argument le nom de la socket, et en deuxième argument le nombre de processus à créer.

Remarque : afin de permettre plusieurs exécutions de ce programme, veiller avant sa terminaison à appliquer unlink sur le nom de la socket afin de le rendre disponible.
Exemple d'appel :
$PWD/bin/remonte_udp sockudp 4
*/

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

	int id, n, i, total, res;
	int *result = NULL;
	int *pids;
	struct sockaddr_un si_me, si_other;
	socklen_t lenslt = (unsigned) sizeof(struct sockaddr_un);

	n = (argc < 2) ? 0 : strtol(argv[2], NULL, 10); //argc inférieur à 2? oui n = 0 non n = strtol

	if (n <= 0) {
		fprintf(stderr, "Usage: %s nombre\n", argv[0]);
		exit(EXIT_FAILURE);
	}

	unlink(argv[1]);

	/* Créatiion descripteur */
	if ((id = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {
		close(id);
		perror("Echec de création de la socket\n");
		exit(errno);
	}

	/* Bind le nom au socket */
	memset(&si_me,'\0',sizeof(struct sockaddr_un));
	si_me.sun_family = AF_UNIX;
	strcpy (si_me.sun_path, argv[1]);

	if (bind(id,(struct sockaddr *) &si_me, sizeof(si_me))==-1){
		close(id);
		perror("Echec du bind\n");
		exit(errno);
	}

	pids = malloc(n * sizeof(int));
	result = malloc(n * sizeof(int));

	for(i=0; i<n; i++){

		int pid = fork();

		if (pid == -1){

			perror("fork");
			return -1;

		} else if (pid) {
```

```
                pids[i] = pid;
                int val;

                if (recvfrom(id, &val, sizeof(int), 0,(struct sockaddr *)&si_me, &lenslt)==-1){
                        //close(id);
                        perror("père : Echec rcv");
                        exit(errno);
                }
                result[i] = val;
                printf("père val : %d\n",val);
                val+=1;

                printf("fin i=%d\n",i);
        } else {
                srand(time(NULL)*i);
                /* Ecriture puis lecture dans la socket */
                int val = (int) (10*(float)rand()/ RAND_MAX);
                printf("%d | val : %d\n",getpid(),val);

                if (sendto(id, &val, sizeof(int), 0,(struct sockaddr *)&si_me, lenslt ) == -1) {
                        perror("fils sending datagram message");
                        exit(errno);
                }

                printf("%d | val reçu : %d\n",getpid(),val);

                exit(EXIT_SUCCESS);
        }
    }

    for(i=0; i<n; i++){
        int status;
        waitpid(pids[i], &status, 0);
    }

    total = 0;

    for(i=0; i<n; i++){
        /* Lecture dans la ressource partagee */
        res = result[i];
        printf("pid %d envoie %d\n", pids[i], res);
        total += res;
    }

    free(pids);

    printf("total: %d\n", total);
    return EXIT_SUCCESS;
}
```

Serveur d'environnement

```
/*
2 Serveur d'environnement

On souhaite réaliser un mini-serveur d'environnement qui communique par UDP sur un port dont le numéro est donné sur la ligne de commande du serveur à son démarrage. Ce qu'on appelle ici un environnement est une liste de couples identificateur, valeur. Les identificateurs et les valeurs sont de type chaîne de caractères.

Le serveur reconnait deux opérations :

    set(identificateur, valeur) : pour fixer la valeur d'un identificateur ;
    get(identificateur) : pour obtenir la valeur d'un identificateur.

Pour éviter d'avoir à définir et à gérer une structure de données, on pourra utiliser directement l'environnement du processus serveur via les fonctions getenv et setenv.
Exemple d'appel :
$PWD/bin/env_serveur 2001 &
*/
#define BUFSIZE 1024

int main(int argc, char **argv) {

        int sockfd; /* socket */
        int portno; /* port d'écoute */


        struct sockaddr_in serveraddr; /* server's addr */
        socklen_t servlen = sizeof(serveraddr); /* taille de l'adresse client */
        struct hostent *hostp; /* client host info */
        char buf[BUFSIZE]; /* message buf */
        char *hostaddrp; /* dotted decimal host addr string */
        int optval; /* flag value for setsockopt */
        int n; /* message byte size */
        char *REP = "DONE";
        char *USER;
        char *VARIABLE;
        /*
         * Vérification ligne de commande
         */
        if (argc != 2) {
                fprintf(stderr, "usage: %s <port>\n", argv[0]);
                exit(1);
        }
        portno = atoi(argv[1]);

        /*
         * socket: création de la socket
         */
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sockfd < 0)
                perror("socket ");


        /*
         * préparation de l'adresse d'attachement
         */

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
        serveraddr.sin_port = htons((unsigned short)portno);

        /*
         * bind: attachement de la socket
         */
        if (bind(sockfd, (struct sockaddr *) &serveraddr, sizeof(struct sockaddr_in)) < 0)
                perror("bind");

        printf("Socket UDP sur port : %d\n",ntohs(serveraddr.sin_port));
```

```
	/*
	 * boucle
	 */
	while (1) {

		/*
		 * recvfrom: réception d'un datagramme UDP d'un client
		 */
		printf("Caractères dans le buff: %lu\n",strlen(buf));
		bzero(buf, BUFSIZE);

		printf("-> Vidage, caractères dans le buff: %lu\n",strlen(buf));
		n = recvfrom(sockfd, buf, BUFSIZE, 0,(struct sockaddr *) &serveraddr, &servlen);
		printf("Caractères dans le buff avant traitement: %lu\n",strlen(buf));

		if (n < 0)
			perror("recvfrom ");


		//Traitement du message
		char *token = strtok(buf, " ");
		printf("token : %s\n",token);
		if(!strcmp(token, "S")){

			USER = strtok(NULL, " ");
			VARIABLE = strtok(NULL, " ");
			setenv(USER, VARIABLE, 1);
			n = sendto(sockfd, REP, strlen(REP), 0, (struct sockaddr *) &serveraddr, servlen);
				if (n < 0)
					perror("sendto");

			printf("Variable env : %s\n",getenv(USER));


		}
		if (!strcmp(token, "G")){

			USER = strtok(NULL, " ");
			USER[strlen(USER)-1]= 0;		//On retire le caractère \n de l'exemple
			VARIABLE = getenv(USER);
			if(strlen(VARIABLE)!=0)
				printf("VARIABLE : %s\n",VARIABLE);
			n = sendto(sockfd, VARIABLE, strlen(VARIABLE), 0, (struct sockaddr *) &serveraddr, servlen);
			if (n < 0)
				perror("sendto");

		}
		if (!strcmp(token, "Q\n")){
			n = sendto(sockfd, REP, strlen(REP), 0, (struct sockaddr *) &serveraddr, servlen);
				if (n < 0)
					perror("sendto");
		}


	}
}
```

Un client pour l'environnement

```
/*
3 Un client pour l'environnement
Le client du programme précédent prend sur la ligne de commande l'adresse du serveur et son port. Ensuite il lit sur le flux d'entrée
une suite de requêtes dont chacune doit avoir l'une des formes suivantes :
   S identificateur valeur, pour un Set ;
   G identificateur, pour un Get.
   Q, pour quitter le client.
Le client construit le message correspondant à la requête et envoie ce message au serveur en utilisant une socket et le protocole UDP.
Il attend alors la réponse du serveur et l'envoie sur le flux de sortie.
Exemple d'appel :
echo "S USER moi;G USER;Q;"|tr ";" "\n"|$PWD/bin/env_client 127.0.0.1 2001
*/
#define BUFSIZE 1024
int main(int argc, char **argv) {
	int sockfd, portno, n;
	socklen_t serverlen;
	struct sockaddr_in serveraddr;
	struct hostent *server;
	char *hostname;
	char buf[BUFSIZE];
	if (argc != 3) {/* vérification de la commande */
		fprintf(stderr,"usage: %s <hostname> <port>\n", argv[0]);
		exit(0);
	}
	hostname = argv[1];
	portno = atoi(argv[2]);
	sockfd = socket(AF_INET, SOCK_DGRAM, 0); /* creation du socket */
	if (sockfd < 0)
		perror("socket") ;
	server = gethostbyname(hostname);   /* récupération nom serveur */
	if (server == NULL) {
		fprintf(stderr,"host %s\n", hostname);
		exit(0);
	}
	bzero((char *) &serveraddr, sizeof(serveraddr)); /* création adresse serveur */
	serveraddr.sin_family = AF_INET;
	bcopy((char *)server->h_addr, (char *)&serveraddr.sin_addr.s_addr, server->h_length);
	serveraddr.sin_port = htons(portno);
	while(1){
		/* message */
		bzero(buf, BUFSIZE);
		printf("message: ");
		//fgets(buf, BUFSIZE, stdin);
		read(STDIN_FILENO,buf,BUFSIZE);
		/* envoie */
		serverlen = sizeof(serveraddr);
		n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr*) &serveraddr, serverlen);
		if (n < 0)
			perror("ERROR in sendto");
		bzero(buf, BUFSIZE); /* On réinitialise le serveur à 0 */
		/* On reçoit et l'on précise la taille du buffer /!\ */
		n = recvfrom(sockfd, buf, BUFSIZE, 0, (struct sockaddr*) &serveraddr, &serverlen);
		if (n < 0)
			perror("ERROR in recvfrom");
		printf("Echo from server: %s\n", buf);
	}
	return 0;
}
```

Réception d'un fichier par socket

```
#define TAILLE 256
struct sigaction action;
/*
4 Réception d'un fichier par socket
On souhaite écrire un programme attendant sur un port une demande de connexion de la part d'un client. Ce programme prend en ligne de commande le numéro du port sur lequel il attend les demandes de connexion. Lorsqu'une connexion s'ouvre, il lit la première ligne envoyée et considère que c'est le nom d'un fichier ; il crée alors dans son répertoire d'exécution un fichier vide portant ce nom. Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé précédemment.
Le programme récepteur stocke les fichiers recopiés dans son répertoire d'exécution.
Exemple d'appel :
$PWD/bin/recvfile 2000 &
*/
/* Pour la capture du signal sigchild */
void finFils(int sig){
	printf("Fin d'un fils\n");
	wait(NULL);
}
int main(int argc, char *argv[]) {
	struct sockaddr_in adr;
	socklen_t adr_len = sizeof(struct sockaddr_in);
	int connexion, n;
	int socketfd, fichierfd;
	char NOM_FICHIER[TAILLE];
	char DATA[4*TAILLE];
	if(argc <2){
		fprintf(stderr, "usage : %s <port>",argv[0]);
		exit(-1);
	}

	action.sa_handler = finFils;

	sigaction(SIGCHLD, &action, NULL);

	/* Création de la socket */
	if((socketfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
		perror("Création socket problème ");
		exit(-1);
	}

	/* Préparation de l'adresse d'attachement */
	adr.sin_family = AF_INET;
	adr.sin_addr.s_addr = htonl(INADDR_ANY);
	adr.sin_port = htons(atoi(argv[1]));

	/* Attachement de la socket */
	if (bind(socketfd, (struct sockaddr *)&adr,adr_len)==-1){
		perror("Attachement impossible ");
		exit(-1);
	}

	/* ouverture du service */
	if(listen(socketfd, 10) == -1){
		perror("listen impossible ");
		exit(-1);
	}

	bzero(&NOM_FICHIER, TAILLE);
	/* Boucle de traitement */
	while(1){
		connexion = accept(socketfd, (struct sockaddr *)&adr, &adr_len);
		if (connexion == -1) {
			if(errno == EINTR) continue; //Intéruption d'appel, on continue
			else {		//Trop grave, on quitte.
				perror("accept");
				exit(-1);
			}
		}
```

```
        if(fork() == 0){
            //Lorsqu'une connexion s'ouvre, il lit la première ligne envoyée et considère que c'est le nom d'un fichier
            memset(NOM_FICHIER, 0, TAILLE);
            n = read(connexion,&NOM_FICHIER,TAILLE);
            //n = recvfrom(connexion, &NOM_FICHIER, TAILLE , 0, (struct sockaddr *) &adr, &adr_len);
            if (n < 0)
                perror("ERROR in recvfrom");



            /* Modification du nom du fichier volontaire, car même répertoire */
            printf("Nom fichier : %s\n",NOM_FICHIER);
            NOM_FICHIER[strlen(NOM_FICHIER)+1] = NOM_FICHIER[strlen(NOM_FICHIER)];
            NOM_FICHIER[strlen(NOM_FICHIER)] = '2';
            printf("Nom fichier : %s\n",NOM_FICHIER);

            //il crée alors dans son répertoire d'exécution un fichier vide portant ce nom.
            if((fichierfd = open(NOM_FICHIER, O_CREAT|O_WRONLY, 0666)) == -1){
                perror("open ");
                close(connexion);
                exit(-1);
            }

            //Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé précédemment.
            while(1){
                n = read(connexion, &DATA,4*TAILLE);
                printf("Impression data %d\n",n);
                if (n < 0)
                    perror("ERROR in recvfrom");
                if(n == 0)
                    break;
                if(write(fichierfd, DATA, 4*TAILLE)==-1)
                    perror("write ");
                /* Ne pas oublier d'effacer le buffer */
                memset(DATA, 0, 4*TAILLE);
            }

            printf("Fin d'écriture du fichier \n");
            exit(1);

        }
        close(connexion);

    }


}
```

Envoi d'un fichier par socket à un récepteur

```
#define TAILLE 256
/*
On souhaite écrire un programme transmettant au programme précédent le contenu d'un fichier, au moyen d'une connexion TCP.
Le programme prend en ligne de commande :
   l'adresse à laquelle le récepteur attend les demandes de connexion ;
   le numéro du port sur lequel le récepteur attend les demandes de connexion ;
   le nom du fichier à recopier.
Lorsqu'il obtient sa connexion avec le programme récepteur, il envoie d'abord le nom du fichier, puis le contenu de celui-ci.
Exemple d'appel :
$PWD/bin/sendfile 127.0.0.1 2000 makefile
*/
int main(int argc, char *argv[]) {
	struct sockaddr_in adr;
	socklen_t adr_len = sizeof(struct sockaddr_in);
	int connexion, n, port;
	int socketfd, fichierfd;
	struct hostent *hp; // pour l'adresse du serveur
	char NOM_FICHIER[TAILLE];
	char DATA[4*TAILLE];
	if(argc <4){
		fprintf(stderr, "usage : %s <addr> <port> <filename>",argv[0]);
		exit(-1);
	}
	/* recherche de l'adresse Internet du serveur */
	if((hp = gethostbyname(argv[1]))== NULL){
		fprintf(stderr, "machine %s incconue \n",argv[1]);
		exit(2);
	}
	/* Création de la socket */
	if((socketfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
		perror("Création socket problème ");
		exit(-1);
	}
	/* Préparation de l'adresse d'attachement */
	adr.sin_family = AF_INET;
	memcpy(&adr.sin_addr.s_addr, hp->h_addr, hp->h_length);
	adr.sin_port = htons(atoi(argv[2]));
	/* ouverture du service */
	if(connect(socketfd, (struct sockaddr *)&adr,adr_len) == -1){
		perror("connection impossible ");
		exit(-1);
	}
	fprintf(stdout, "Connexion réussi\n");
	/*Ouverture du fichier */
	fichierfd = open(argv[3], O_RDONLY, 0666);
	/*Vidage et copie dans le buffer pour le nom */
	memset(NOM_FICHIER, 0, TAILLE);
	strcpy(NOM_FICHIER, argv[3]);
	/*Ecriture sur la socket*/
	//n = write(socketfd, NOM_FICHIER, TAILLE);
	n = write(socketfd, "LIST ", sizeof("LIST "));
	if (n < 0)
		perror("ERROR in sendto");
	/* Boucle de traitement */
	//while((n=read(fichierfd,&DATA,4*TAILLE)) != 0){
	while((n=read(socketfd,&DATA,4*TAILLE)) != 0){
		printf("%s",DATA);
		//n = write(socketfd, DATA, 4*TAILLE);
		if (n < 0)
			perror("ERROR in sendto");
		/* Ne pas oublier de remettre à 0 le buffer */
		memset(DATA, 0, 4*TAILLE);
	}
	shutdown(socketfd, SHUT_RDWR);
	close(socketfd);
	close(fichierfd);
}
```

Un mini-serveur FTP

```
/*
1 Un mini-serveur FTP

Dans cette question il s'agit d'écrire le serveur, qui reçoit en ligne de commande le numéro de port où l'appeler, et le répertoire où entreposer les fichiers envoyés par les clients. Les connexions se feront en TCP. On considère qu'un serveur ne peut traiter qu'un client à la fois.
Exemple d'appel :
$PWD/bin/ftp_server 2000 /tmp &
*/

#define TAILLE 256
struct sigaction action;


/* Pour la capture du signal sigchild */
void finFils(int sig){
	printf("Fin d'un fils\n");
	wait(NULL);
}

int main(int argc, char *argv[]) {
	struct sockaddr_in adr;
	socklen_t adr_len = sizeof(struct sockaddr_in);
	int connexion, n;
	int socketfd, fichierfd;
	char NOM_FICHIER[TAILLE];
	char NOM_REPERTOIRE[TAILLE];
	char DATA[4*TAILLE];
	char DATALIST[4*TAILLE];
	char *CMD;
	char *NOM;
	char CMD_BRUT[TAILLE];
	int tube[2];
	if(argc <3){
		fprintf(stderr, "usage : %s <port> <repertory>",argv[0]);
		exit(-1);
	}

	action.sa_handler = finFils;

	sigaction(SIGCHLD, &action, NULL);

	/* Création de la socket */
	if((socketfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
		perror("Création socket problème ");
		exit(-1);
	}

	/* Préparation de l'adresse d'attachement */
	adr.sin_family = AF_INET;
	adr.sin_addr.s_addr = htonl(INADDR_ANY);
	adr.sin_port = htons(atoi(argv[1]));

	/* Attachement de la socket */
	if (bind(socketfd, (struct sockaddr *)&adr,adr_len)==-1){
		perror("Attachement impossible ");
		exit(-1);
	}

	/* ouverture du service */
	if(listen(socketfd, 10) == -1){
		perror("listen impossible ");
		exit(-1);
	}

	strcpy(NOM_REPERTOIRE, argv[2]);
	printf("%s\n",NOM_REPERTOIRE);
```

```
bzero(&NOM_FICHIER, TAILLE);
/* Boucle de traitement */
while(1){
	connexion = accept(socketfd, (struct sockaddr *)&adr, &adr_len);

	memset(CMD_BRUT,0,TAILLE);

	if (connexion == -1) {
		if(errno == EINTR) continue; //Intéruption d'appel, on continue
		else {		//Trop grave, on quitte.
			perror("accept");
			exit(-1);
		}
	}

	if(fork() == 0){

		//Lorsqu'une connexion s'ouvre, il lit la première ligne envoyée et considère que c'est le nom d'un fichier
		n = read(connexion,&CMD_BRUT,TAILLE);
		if (n < 0)
			perror("read ");

		CMD = strtok(CMD_BRUT, " ");
		printf("CMD : %s\n",CMD);
		NOM = strtok(NULL, " ");
		printf("NOM : %s\n",NOM);

		if(!strcmp(CMD,"LIST")){
			printf("PASSAGE\n");
			pipe(tube); //Création d'un tube pour communication (récupération du résultat de ls);
			if(fork() == 0){
				dup2(tube[1], STDOUT_FILENO);
				dup2(tube[1], STDERR_FILENO); //On place également l'erreur dans le flux de sortie;
				close(tube[0]);
				close(tube[1]);
				execlp("ls", "ls","-l", NOM_REPERTOIRE,NULL);
				perror("execlp");
				exit(2);

			}
			close(1);
			wait(NULL); //Attente du fils
			read(tube[0], DATALIST, 4*TAILLE);

			//On envoi la réponse au client maintenant
			n = write(connexion, DATALIST, 4*TAILLE);
			if (n < 0)
				perror("write ");


		}else if(!strcmp(CMD,"UPLOAD")){

			strcat(NOM_FICHIER, NOM_REPERTOIRE);
			strcat(NOM_FICHIER, NOM);

			//NOM_FICHIER[strlen(NOM_FICHIER)-1] = '2';

			printf("NOM FINAL : %s",NOM_FICHIER);


			//il crée alors dans son répertoire d'exécution un fichier vide portant ce nom.
			if((fichierfd = open(NOM_FICHIER, O_CREAT|O_WRONLY, 0666)) == -1){
				perror("open ");
				close(connexion);
				exit(-1);
			}

			//Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé
précédemment.
```

```
                while(1){
                        n = read(connexion, &DATA,4*TAILLE);
                        printf("Impression data %d\n",n);
                        if (n < 0)
                                perror("read ");
                        if(n == 0)
                                break;
                        if(write(fichierfd, DATA, 4*TAILLE)==-1)
                                perror("write ");
                        /* Ne pas oublier d'effacer le buffer */
                        memset(DATA, 0, 4*TAILLE);
                }

                printf("Fin d'écriture du fichier \n");

        }else if(!strcmp(CMD,"DOWNLOAD")){

                strcat(NOM_FICHIER, NOM_REPERTOIRE);
                strcat(NOM_FICHIER, NOM);
                /*Ouverture du fichier */
                fichierfd = open(NOM_FICHIER, O_RDONLY, 0666);

                /* Boucle de traitement */
                while((n=read(fichierfd,&DATA,4*TAILLE)) != 0){

                        n = write(connexion, DATA, 4*TAILLE);
                        if (n < 0)
                                perror("write");

                        /* Ne pas oublier de remettre à 0 le buffer */
                        memset(DATA, 0, 4*TAILLE);
                }
                close(fichierfd);

        }
        close(connexion);
        exit(1);

    }
    close(connexion);

}
```

Un mini-client FTP

```
#define TAILLE 256
/*
Le client prend sur la ligne de commande l'adresse IP du serveur et son numéro de port. Il s'y connecte immédiatement, et en cas de réussite rentre dans une boucle de lecture ligne par ligne des requêtes de l'utilisateur au clavier. Pour chaque ligne, il vérifie que la requête demandée est bien l'une des trois indiquée dans l'énoncé, si oui l'envoie au serveur, attend sa réponse et l'affiche dans le flux de sortie.
Exemple d'appel :
$PWD/bin/ftp_client 127.0.0.1 2000
*/
int main(int argc, char *argv[]) {
	struct sockaddr_in adr;
	socklen_t adr_len = sizeof(struct sockaddr_in);
	int connexion, n, port;
	int socketfd, fichierfd;
	struct hostent *hp; // pour l'adresse du serveur
	char NOM_FICHIER[TAILLE];
	char *CMD;
	char CMD_BRUT[TAILLE];
	char CMD_COMPLET[TAILLE];
	char *REP_COURANT = "./";
	char *NOM_EXTRAIT;
	char DATA[4*TAILLE];
	if(argc <3){
		fprintf(stderr, "usage : %s <addr> <port> <filename>",argv[0]);
		exit(-1);
	}
	/* recherche de l'adresse Internet du serveur */
	if((hp = gethostbyname(argv[1]))== NULL){
		fprintf(stderr, "machine %s inconnue \n",argv[1]);
		exit(2);
	}
	/* Préparation de l'adresse d'attachement */
	adr.sin_family = AF_INET;
	memcpy(&adr.sin_addr.s_addr, hp->h_addr, hp->h_length);
	adr.sin_port = htons(atoi(argv[2]));
	while (1) {
		/* Lecture CMD */
		memset(CMD_BRUT, 0, TAILLE);
		memset(CMD_COMPLET, 0, TAILLE);
		memset(NOM_FICHIER, 0, TAILLE);
		/* Création de la socket */
		if((socketfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
			perror("Création socket problème ");
			exit(-1);
		}
		/* ouverture du service */
		if(connect(socketfd, (struct sockaddr *)&adr,adr_len) == -1){
			perror("connection impossible ");
			exit(-1);
		}
		n = read(STDIN_FILENO, CMD_BRUT, TAILLE);
		if(n<0)
			perror("lecture ");
		/* TRAITEMENT CHAINE */
		CMD_BRUT[strlen(CMD_BRUT)-1] = 0; //On retire le \n
		CMD = strtok(CMD_BRUT, " ");
		printf("CMD : %s, taille %lu\n",CMD, strlen(CMD));
		NOM_EXTRAIT = strtok(NULL, " ");
		printf("NOM_EXTRAIT : %s\n",NOM_EXTRAIT);
		if(!strcmp(CMD, "LIST")){
			/* Ecriture de la commande sur socket */
			n = write(socketfd, CMD, 4*TAILLE);
			if (n < 0)
				perror("write ");
			printf("Passage list\n");
			while((n=read(socketfd,&DATA,4*TAILLE)) != 0){
				printf("%s\n",DATA);
```

```
                //n = write(socketfd, DATA, 4*TAILLE);
                if (n < 0)
                        perror("read ");

                /* Ne pas oublier de remettre à 0 le buffer */
                memset(DATA, 0, 4*TAILLE);
        }
    }else if (!strcmp(CMD, "UPLOAD")){
        printf("Passage upload\n");
        /* Ecriture de la commande sur socket */
        strcat(CMD_COMPLET, CMD);
        strcat(CMD_COMPLET," ");
        strcat(CMD_COMPLET, NOM_EXTRAIT);
        n = write(socketfd, CMD_COMPLET, 4*TAILLE);
        if (n < 0)
                perror("write ");
        /* Ouverture du fichier */
        strcat(NOM_FICHIER, REP_COURANT);
        strcat(NOM_FICHIER, NOM_EXTRAIT);
        printf("-> Ouverture de %s\n",NOM_FICHIER);
        //il crée alors dans son répertoire d'exécution un fichier vide portant ce nom.
        if((fichierfd = open(NOM_FICHIER, O_RDONLY, 0666)) == -1){
                perror("open ");
                close(socketfd);
                exit(-1);
        }
        //Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé précédemment.
        while((n=read(fichierfd,&DATA,4*TAILLE)) != 0){
                //printf("%s",DATA);
                n = write(socketfd, DATA, 4*TAILLE);
                if (n < 0)
                        perror("read ");
                /* Ne pas oublier de remettre à 0 le buffer */
                memset(DATA, 0, 4*TAILLE);
        }
    }else if (!strcmp(CMD, "DOWNLOAD")){
        printf("Passage download\n");
        /* Ecriture de la commande sur socket */
        strcat(CMD_COMPLET, CMD);
        strcat(CMD_COMPLET," ");
        strcat(CMD_COMPLET, NOM_EXTRAIT);
        n = write(socketfd, CMD_COMPLET, 4*TAILLE);
        if (n < 0)
                perror("write");
        strcat(NOM_FICHIER, REP_COURANT);
        strcat(NOM_FICHIER, NOM_EXTRAIT);
        //il crée alors dans son répertoire d'exécution un fichier vide portant ce nom.
        if((fichierfd = open(NOM_FICHIER, O_CREAT|O_WRONLY, 0666)) == -1){
                perror("open ");
                close(socketfd);
                exit(-1);
        }
        //Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé précédemment.
        while(1){
                n = read(socketfd, &DATA,4*TAILLE);
                printf("Impression data %d\n",n);
                if (n < 0)
                        perror("read ");
                if(n == 0)
                        break;
                if(write(fichierfd, DATA, 4*TAILLE)==-1)
                        perror("write ");
                /* Ne pas oublier d'effacer le buffer */
                memset(DATA, 0, 4*TAILLE);
        }
    }
  }
}
```

Socket et parallélisme

Service FTP en parallèle

```
#define TAILLE 256
struct sigaction action;
pthread_t pidThread;
struct sockaddr_in adr;
socklen_t adr_len = sizeof(struct sockaddr_in);
int connexion, n;
int *connexionToThread;
int socketfd, fichierfd;
char NOM_REPERTOIRE[TAILLE];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
/*
1 Service FTP en parallèle
Modifier le serveur réalisé lors de la séance Simulation d'un service FTP afin de gérer plusieurs clients en parallèle. On pourra les traiter soit par des processus fils, soit par des processus légers.
Exemple d'appel :
$PWD/bin/ftp_multi_serveur 2000 /tmp &
*/
void funcTraitement(void *arg){

	pthread_mutex_lock(&lock);
	int connect = (*(int*)arg);
	pthread_mutex_unlock(&lock);
	printf("connect : %d\n",connect);
	free(arg);
	char CMD_BRUT[TAILLE];
	char *CMD;
	char *NOM;
	char DATA[4*TAILLE];
	char DATALIST[4*TAILLE];
	char NOM_FICHIER[TAILLE];
	int tube[2];
	memset(CMD_BRUT,0,TAILLE);
	//Lorsqu'une connexion s'ouvre, il lit la première ligne envoyée et considère que c'est le nom d'un fichier
	n = read(connect,&CMD_BRUT,TAILLE);
	if (n < 0)
		perror("thread read ");
	CMD = strtok(CMD_BRUT, " ");
	printf("CMD : %s\n",CMD);
	NOM = strtok(NULL, " ");
	printf("NOM : %s\n",NOM);
	if(!strcmp(CMD,"LIST")){
		pipe(tube); //Création d'un tube pour communication (récupération du résultat de ls);
		if(fork() == 0){
			dup2(tube[1], STDOUT_FILENO);
			dup2(tube[1], STDERR_FILENO); //On place également l'erreur dans le flux de sortie;
			close(tube[0]);
			close(tube[1]);
			execlp("ls", "ls","-l", NOM_REPERTOIRE,NULL);
			perror("execlp");
			exit(2);

		}
		close(tube[1]);
		wait(NULL); //Attente du fils
		read(tube[0], DATALIST, 4*TAILLE);
		//On envoi la réponse au client maintenant
		n = write(connect, DATALIST, 4*TAILLE);
		if (n < 0)
			perror("write ");
	}else if(!strcmp(CMD,"UPLOAD")){
		strcat(NOM_FICHIER, NOM_REPERTOIRE);
		strcat(NOM_FICHIER, NOM);
		printf("NOM FINAL : %s",NOM_FICHIER);
		//il crée alors dans son répertoire d'exécution un fichier vide portant ce nom.
		if((fichierfd = open(NOM_FICHIER, O_CREAT|O_WRONLY, 0666)) == -1){
```

```
                perror("open ");
                close(connexion);
                exit(-1);
        }
        //Il lit ensuite les données transmises jusqu'à la fin de la connexion, et les recopie dans le fichier créé précédemment.
        while(1){
                n = read(connect, &DATA,4*TAILLE);
                printf("Impression data %d\n",n);
                if (n < 0)
                        perror("read ");
                if(n == 0)
                        break;
                if(write(fichierfd, DATA, 4*TAILLE)==-1)
                        perror("write ");
                /* Ne pas oublier d'effacer le buffer */
                memset(DATA, 0, 4*TAILLE);
        }
        printf("Fin d'écriture du fichier \n");
    }else if(!strcmp(CMD,"DOWNLOAD")){
        strcat(NOM_FICHIER, NOM_REPERTOIRE);
        strcat(NOM_FICHIER, NOM);
        /*Ouverture du fichier */
        fichierfd = open(NOM_FICHIER, O_RDONLY, 0666);
        /* Boucle de traitement */
        while((n=read(fichierfd,&DATA,4*TAILLE)) != 0){
                n = write(connect, DATA, 4*TAILLE);
                if (n < 0)
                        perror("write");
                /* Ne pas oublier de remettre à 0 le buffer */
                memset(DATA, 0, 4*TAILLE);
        }
        close(fichierfd);
    }
    close(connect);
    pthread_exit((void*)0);
}
void traitSignal(int sig){
    if(SIGSEGV == sig)
        printf("Signal fault reçu\n");
    else
        printf("Signal int ou autre reçu\n");

    perror("ERROR");
    close(socketfd);
    close(connexion);
}
int main(int argc, char *argv[]) {
    int MAXCLIENT = 10;

    if(argc <3){
        fprintf(stderr, "usage : %s <port> <repertory>",argv[0]);
        exit(-1);
    }
    /* Création de la socket */
    if((socketfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
        perror("Création socket problème ");
        exit(-1);
    }
    /* Préparation de l'adresse d'attachement */
    adr.sin_family = AF_INET;
    adr.sin_addr.s_addr = htonl(INADDR_ANY);
    adr.sin_port = htons(atoi(argv[1]));
    action.sa_handler = traitSignal;
    //sigaction(SIGCHLD, &action, NULL);
    //sigaction(SIGSEGV, &action, NULL);
    sigaction(SIGINT, &action, NULL);
    printf("Configuration signaux : OK\n");

    /* Attachement de la socket */
```

```
    if (bind(socketfd, (struct sockaddr *)&adr,adr_len)==-1){
        perror("Attachement impossible ");
        exit(-1);
    }

    /* ouverture du service */
    if(listen(socketfd, MAXCLIENT) == -1){
        perror("listen impossible ");
        exit(-1);
    }
    strcpy(NOM_REPERTOIRE, argv[2]);
    printf("NOM REPERTOIRE : %s\n",NOM_REPERTOIRE);
    /* Boucle de traitement */
    while(1){
        printf("socketfd: %d\n",socketfd);
        connexion = accept(socketfd, (struct sockaddr *)&adr, &adr_len);
        if (connexion == -1) {
            if(errno == EINTR) continue; //Intéruption d'appel, on continue
            else {      //Trop grave, on quitte.
                perror("accept");
                exit(-1);
            }
        }
        connexionToThread = malloc(sizeof(int));
        *connexionToThread = connexion;      //Pointeur vers variable i;
        printf("ConnexionTOThread : %d, connexion %d\n",*connexionToThread,connexion);
        pthread_create(&pidThread,NULL,(void*)funcTraitement,connexionToThread);
        printf("Lancement du Thread :%d\n",(int)pidThread);
    }
}
```

Journalisation de connexions

```
#define TAILLE 256
struct sigaction action;
pthread_t pidThread;
struct sockaddr_in adr;
socklen_t adr_len = sizeof(struct sockaddr_in);
int connexion, n;
int *connexionToThread;
int *socketfd, fichierfd, logfd;
char NOM_REPERTOIRE[TAILLE];
struct in_addr ipAddr;
char str[INET_ADDRSTRLEN+1];
fd_set readfds;
/*
2 Journalisation de connexions
Écrire un programme serveur, sans sous-processus ni Thread, qui prend sur la ligne de commande un nombre arbitraire de numéro de ports, et attend en parallèle un client sur chacun de ces ports. On utilisera bien sûr la fonction select. Il enregistre dans le fichier cx.log les adresses des clients successifs.
Le code du processus client vous est fourni en annexe. Il prend sur sa ligne de commande l'adresse du serveur et un numéro de port où il écoute.
Exemple d'appel :
$PWD/bin/journal_serveur 2820 2821 2822 2823 &
*/

int main(int argc, char *argv[]) {
	int MAXCLIENT = 10;
	int i = 0;
	int max = 0;
	FD_ZERO(&readfds);
	if(argc <2){
		fprintf(stderr, "usage : %s <port...> ",argv[0]);
		exit(-1);
	}
	/*Allocation de notre tableau de socket */
	socketfd = malloc(sizeof(int)*argc-1);
	/* Ouverture du fichier de log */
	if((logfd = open("cx.log", O_CREAT|O_RDWR, 0666)) == -1){
		perror("cx.log");
		exit(2);
	}
	for(i = 0; i< argc-1;i++){
		printf("Création du socket : %s\n",argv[i+1]);
		/* Création de la socket */
		if((socketfd[i] = socket(AF_INET,SOCK_STREAM,0)) == -1){
			perror("Création socket problème ");
			exit(-1);
		}
		printf("Socketfd : %d\n",(int)socketfd[i]);
		/* Préparation de l'adresse d'attachement */
		adr.sin_family = AF_INET;
		adr.sin_addr.s_addr = htonl(INADDR_ANY);
		adr.sin_port = htons(atoi(argv[i+1]));
		/* Attachement de la socket */
		if (bind(socketfd[i], (struct sockaddr *)&adr,adr_len)==-1){
			perror("Attachement impossible ");
			exit(-1);
		}
		/* ouverture du service */
		if(listen(socketfd[i], MAXCLIENT) == -1){
			perror("listen impossible ");
			exit(-1);
		}
		/*On ajout à l'ensemble des descripteurs */
		FD_SET(socketfd[i],&readfds);
		printf("-> fin d'initialisation du socket : %d\n",socketfd[i]);
		max = socketfd[i] > max ? socketfd[i] : max;
	}
```

```
    /* Boucle de traitement */
    while(1){

        /*Attente d'une lecture disponible sur l'ensemble */
        if(select(max + 1, &readfds, NULL, NULL, NULL) == -1)
        {
            perror("select");
            exit(errno);
        }

        /* On test l'ensemble des descripteurs pour savoir sur lequel répondre */
        for(i = 0;i<argc-1;i++){

            if(FD_ISSET(socketfd[i],&readfds)){

                printf("Connexion sur socketfd: %d\n",socketfd[i]);
                connexion = accept(socketfd[i], (struct sockaddr *)&adr, &adr_len);

                if (connexion == -1) {
                    if(errno == EINTR) continue; //Intéruption d'appel, on continue
                    else {      //Trop grave, on quitte.
                        perror("accept");
                        exit(-1);
                    }
                }

                ipAddr= adr.sin_addr;
                inet_ntop( AF_INET, &ipAddr, str, INET_ADDRSTRLEN );
                str[strlen(str)] = '\n';
                printf("Adresse : %s\n",str);
                write(logfd, str, strlen(str));
                close(connexion);
            }
        }

        FD_ZERO(&readfds);

        for(i = 0;i<argc-1;i++)
            FD_SET(socketfd[i],&readfds);
    }
}
```

Socket et multi-diffusion

Sonar

```
#define TAILLE 256
struct sigaction action;
pthread_t pidThread;
struct sockaddr_in adr;
socklen_t adr_len = sizeof(struct sockaddr_in);
int connexion, n;
int *connexionToThread;
int socketfd,socketfdEnvoi, fichierfd, logfd;
char NOM_REPERTOIRE[TAILLE];
struct in_addr ipAddr;
char str[INET_ADDRSTRLEN+1];
fd_set readfds;
/*
1 Sonar
On cherche à créer une application qui permet, à la manière d'un sonar, de repérer tous les nœuds disponibles sur le réseau local. Un programme sonar diffuse toutes les 3 secondes un message "PING" en mode Broadcast sur le port 9999, puis attend en réponse un message "PONG" des sites disponibles et affiche leur identité. Le sonar comporte deux Threads : un qui attend les réponses sur le port 9999, et un autre qui s'occupe de la diffusion. De leurs côtés, les nœuds disponibles exécutent le programme Ponger qui attend l'arrivée de messages sur le port de diffusion, et renvoie à l'émetteur de tout "PING" un message "PONG".

N.B : Pour obtenir l'adresse valide de Broadcast sur votre sous-réseau, utilisez la valeur INADDR_BROADCAST.
Exemple d'appel :
$PWD/bin/sonar
*/
int main(int argc, char *argv[]) {
	int MAXCLIENT = 10;
	int i = 0;
	int max = 0;
	int broadcastPermission = 1;
	int broadcastPort = 9999;//atoi(argv[2]);	/* Port pour broadcast */
	char *sendString = "PING";//argv[3];		/* Message */
	char *rcvString;
	rcvString = malloc(sizeof(char)*4);
	/* Création de la socket */
	if((socketfd = socket(AF_INET,SOCK_DGRAM,0)) == -1){
		perror("Création socket problème ");
		exit(-1);
	}
	/* Préparation de l'adresse d'attachement */
	adr.sin_family = AF_INET;
	adr.sin_addr.s_addr = htonl(INADDR_ANY);
	adr.sin_port = htons(broadcastPort);
	/* Attachement au port */
	if (bind(socketfd, (struct sockaddr *) &adr, sizeof(adr)) < 0)
		perror("bind");
	int adrlen = sizeof(adr);

	/* Boucle de traitement */
	while(1){
		read(socketfd, rcvString, strlen(sendString));
		printf("RECU : %s\n",rcvString);
		if(!strcmp(rcvString,"PING")){
			printf("ENVOI : PONG\n");
			if (sendto(socketfdEnvoi, "PONG", strlen("PONG"), 0, (struct sockaddr *) &adr, sizeof(adr)) != strlen("PING"))
			   perror("sendto");
		}
		sleep(1);
	}
}
```

Messagerie instantanée groupée

```
/*

2 Messagerie instantanée groupée

Réalisez un programme qui permet d'échanger des messages ligne par ligne avec d'autres processus en communiquant par Multicast. Le programme prend sur la ligne de commande :

   l'adresse IP Multicast où la conversation a lieu ;
   le numéro du port sur lequel la conversation a lieu
   le nom (ou pseudonyme) utilisé dans la conversation

Une fois lancé, le programme affiche tous les messages envoyés par d'autres utilisateurs et permet parallèlement d'envoyer des messages pour participer à la conversation. Il utilisera un Thread pour écrire et un autre pour lire.
Exemple d'appel :
$PWD/bin/mychat 225.0.0.10 2001 $USER

*/
#define TAILLE 256
struct sigaction action;
pthread_t pidEnvoi, pidEcoute;
struct sockaddr_in adr,adrEcoute;
socklen_t adr_len = sizeof(struct sockaddr_in);
int connexion, n;
int *connexionToThread;
int socketfdEnvoi, socketfdEcoute, fichierfd, logfd;
char NOM_REPERTOIRE[TAILLE];
struct in_addr ipAddr;
char str[INET_ADDRSTRLEN+1];
fd_set readfds;
char *rcvString;
int broadcastPort = 9999;//atoi(argv[2]);    /* Port pour broadcast */
char *sendString = "PING";//argv[3];            /* Message */
int broadcastPermission = 1;

void funcEnvoi(void * arg){
      printf("Création du thread\n");


      /* Création de la socket */
      if((socketfdEnvoi = socket(AF_INET,SOCK_DGRAM,0)) == -1){
            perror("Création socket problème ");
            exit(-1);
      }

      /* Set socket to allow broadcast */
      if (setsockopt(socketfdEnvoi, SOL_SOCKET, SO_BROADCAST, (void *) &broadcastPermission,
              sizeof(broadcastPermission)) < 0)
            perror("setsockopt");


      /* Préparation de l'adresse d'attachement */
      adr.sin_family = AF_INET;
      adr.sin_addr.s_addr = htonl(INADDR_BROADCAST);
      adr.sin_port = htons(broadcastPort);

      /* Boucle de traitement */
      while(1){
            if (sendto(socketfdEnvoi, sendString, strlen(sendString), 0, (struct sockaddr *) &adr, sizeof(adr)) != strlen("PING"))
               perror("sendto");
            sleep(1);
      }

}

void funcEcoute(void* arg){

      /* Création de la socket */
```

```
    if((socketfdEnvoi = socket(AF_INET,SOCK_DGRAM,0)) == -1){
        perror("Création socket problème ");
        exit(-1);
    }

    /* Set socket to allow broadcast */
    if (setsockopt(socketfdEnvoi, SOL_SOCKET, SO_BROADCAST, (void *) &broadcastPermission,
          sizeof(broadcastPermission)) < 0)
        perror("setsockopt");


    /* Préparation de l'adresse d'attachement */
    adr.sin_family = AF_INET;
    adr.sin_addr.s_addr = htonl(INADDR_BROADCAST);
    adr.sin_port = htons(broadcastPort);
    while(1){
        read(socketfdEcoute, &rcvString, strlen("PING"));
        printf("THREAD ECOUTE | Recu : %s\n",rcvString);
    }

}
int main(int argc, char *argv[]) {


    rcvString = malloc(sizeof(char)*4);
    pthread_create(&pidEnvoi, 0, (void*)funcEnvoi, (void*)0);
    pthread_create(&pidEcoute, 0, (void*)funcEcoute, (void*)0);

    pthread_join(pidEnvoi,(void*) 0);
    pthread_join(pidEcoute, (void*) 0);


}
```

Entrées-Sorties asynchrones

Asynchronisme avec notification

```
//1 Asynchronisme avec notification
//
//Ecrire un programme qui prend en argument un nom de fichier et une chaîne de caractères,
//crée un fichier vide à partir du nom donné en argument,
// puis écrit la chaîne de caractères dans le fichier avec aio_write,
// de sorte que la fin de cet appel asynchrone soit notifiée par un signal SIGRTMIN.
// Pendant l'écriture le programme crée un nouveau descripteur vers le même fichier,
// puis attend la fin de l'écriture pour aller lire le contenu du fichier et l'affiche avant de se terminer.

char buff[1024];

void real_time_handler(int sig_number, siginfo_t * info, void * arg __attribute__ ((unused)))
{
	printf("passage\n");
}

int main(int argc, char *argv[]) {
	if(argc < 3){
		perror("arg");
		exit(EXIT_FAILURE);
	}
	struct sigaction action;
	sigset_t mask;
	action.sa_sigaction = real_time_handler;
	action.sa_flags = 0;
	sigfillset(&action.sa_mask);
	sigfillset(&mask);
	sigaction(SIGRTMIN,&action,NULL);
	sigprocmask(SIG_BLOCK,&mask,NULL);
	int fd = open(argv[1], O_CREAT|O_WRONLY, 0666);
	struct aiocb *aio;
	aio = malloc(sizeof(struct aiocb));
	strcat(buff,argv[2]);
	printf("strlen : %d\n",strlen(buff));
	aio->aio_fildes = fd;
	aio->aio_buf = buff;
	aio->aio_nbytes = strlen(buff);
	aio->aio_offset = 0;
	aio->aio_reqprio = 0;
	aio->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
	aio->aio_sigevent.sigev_signo = SIGRTMIN;

	if (aio_write(aio) == -1) {
		printf(" Error at aio_write(): %s\n", strerror(errno));
		close(fd);
		exit(2);
	}

	sigdelset (&mask, SIGRTMIN);
	sigsuspend(&mask);

	printf("fin sig\n");
	int fd2 = open(argv[2], O_CREAT|O_WRONLY, 0666);

	read(fd2,buff,1024);
	printf("fin read : %s\n",buff);

}
```

Asynchronisme avec suspension

```
//2 Asynchronisme avec suspension

//Modifier le programme de l'exercice précédant pour que la fin de l'écriture ne soit plus notifiée par signal,
//,mais soit attendue par un appel à aio_suspend. Pendant l'écriture le programme crée un nouveau descripteur vers le même fichier,
//puis attend la fin de l'écriture pour aller lire, cette fois-ci de manière asynchrone (aio_read), le contenu du fichier et l'affiche avant de se terminer.

char buff[1024];


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

	if(argc < 3){
		perror("arg");
		exit(EXIT_FAILURE);
	}
	int fd = open(argv[1], O_CREAT|O_WRONLY, 0666);
	struct aiocb aio;
	const struct aiocb *clio[1];
	strcat(buff,argv[2]);
	printf("strlen : %d\n",strlen(buff));
	//buff[strlen(buff)] = '\n';
	aio.aio_fildes = fd;
	aio.aio_buf = buff;
	aio.aio_nbytes = strlen(buff);
	aio.aio_offset = 0;
	aio.aio_reqprio = 0;
	aio.aio_sigevent.sigev_notify = SIGEV_NONE;
	aio.aio_sigevent.sigev_signo = SIGRTMIN;

	if (aio_write(&aio) == -1) {
		printf(" Error at aio_write(): %s\n", strerror(errno));
		close(fd);
		exit(2);
	}

	clio[0] = &aio;
	if(aio_suspend(clio,1,NULL) == -1){
		perror("aio_suspend ");
		exit(-1);
	}
	printf("fin ecriture\n");
	int fd2 = open(argv[2], O_CREAT|O_WRONLY, 0666);


	read(fd2,buff,1024);
	printf("fin read : %s\n",buff);

}
```

Asynchronisme avec temporisation

```
int end = 0;
struct aiocb *aio;
char *file;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
/*
Reprenez à nouveau le premier programme en remplaçant la déclaration du signal par celle d'un temporisateur périodique, examinant
toutes les 50 nanosecondes si l'écriture est terminée. Si c'est le cas, il affiche le contenu du fichier puis se termine. */
char buff[1024];
void real_time_handler(int sig_number, siginfo_t * info, void * arg __attribute__ ((unused))){
	int err;
	if((err = aio_error (aio)) == EINPROGRESS){
		printf("THREAD %lu passage\n",pthread_self());
		pthread_exit(0);
	}else{
		pthread_mutex_lock(&mutex);
		int fd2 = open(file, O_CREAT|O_WRONLY, 0666);
		read(fd2,buff,1024);
		printf("fin read : %s\n",buff);
		end=1;
		pthread_mutex_unlock(&mutex);
		exit(EXIT_SUCCESS);
	}
}
int main(int argc, char *argv[]) {
	sigset_t mask;
	sigfillset(&mask);
	sigprocmask(SIG_SETMASK,&mask,NULL);
	if(argc < 3){
		perror("arg");
		exit(EXIT_FAILURE);
	}
	timer_t timerid;
	struct itimerspec value;
	value.it_value.tv_sec = 0;
	value.it_value.tv_nsec = 5;
	value.it_interval.tv_sec = 0;
	value.it_interval.tv_nsec = 5;
	file = malloc(sizeof(char)*strlen(argv[2]));
	strcpy(file,argv[2]);
	struct sigevent event;
	event.sigev_notify = SIGEV_THREAD;
	event.sigev_signo = SIGRTMIN;
	event.sigev_value.sival_int = 0; /*Correspond au fait que ce n'est pas terminé */
	event.sigev_value.sival_ptr = &timerid;
	event.sigev_notify_function = (void*)real_time_handler;
	event.sigev_notify_attributes = NULL;
	timer_create(CLOCK_REALTIME,&event,&timerid);
	perror("apres create");
	timer_settime(timerid,0,&value,NULL);
	perror("apres settimer");
	int fd = open(argv[1], O_CREAT|O_WRONLY, 0666);
	aio = malloc(sizeof(struct aiocb));
	strcat(buff,argv[2]);
	printf("strlen : %d\n",strlen(buff));
	aio->aio_fildes = fd;
	aio->aio_buf = buff;
	aio->aio_nbytes = strlen(buff);
	aio->aio_offset = 0;
	aio->aio_reqprio = 0;
	aio->aio_sigevent.sigev_value.sival_int = 1; /* Correspond au fait que c'est fini */
	if (aio_write(aio) == -1) {
		printf(" Error at aio_write(): %s\n", strerror(errno));
		close(fd);
		exit(2);
	}
	while(!end);
}
```

Remontée de valeurs asynchrone

```
//Reprenez l'exercice Remontée de valeurs par partage de mémoire pour que les valeurs aléatoires transmises par les fils soient échangées avec le père en passant par un fichier lu et écrit de manière asynchrone.
//En commentaire de votre programme, répondez également aux questions suivante :
//    est-il nécessaire de synchroniser le père avec ses fils pour que les lectures se fassent correctement ?
//    est-il nécessaire de synchroniser les fils entre eux pour que les écritures se fassent correctement ?
//Exemple d'appel :
//$PWD/bin/remonte_async 4
int main(int argc, char *argv[]){
int  n, i, total, res, temp;
int *pids;
n = (argc < 2) ? 0 : strtol(argv[1], NULL, 10);
if (n <= 0) {
  fprintf(stderr, "Usage: %s nombre\n", argv[0]);
  exit(EXIT_FAILURE);
}
//Une fois shm_open ftruncate mmap fait, la mémoire est alloué et segmenter correctement.
struct aiocb aio;
 int fd = open(argv[1], O_CREAT|O_WRONLY, 0666);
 //buff[strlen(buff)] = '\n';
 aio.aio_fildes = fd;
 aio.aio_buf = &temp;
 aio.aio_nbytes = sizeof(int);
pids = malloc(n * sizeof(int));
for(i=0; i<n; i++){
  int pid = fork();
  if (pid == -1){
    perror("fork");
    return -1;
  } else if (pid) {
    pids[i] = pid;
  } else {
    srand(time(NULL)*i);    /* Ecriture dans la ressource partagee */
     temp = (int) (10*(float)rand()/ RAND_MAX);
     aio.aio_offset = i*sizeof(int);
     aio.aio_reqprio = 0;
     aio.aio_sigevent.sigev_notify = SIGEV_NONE;
     aio.aio_sigevent.sigev_signo = SIGRTMIN;
     printf("Ecriture\n");
   if (aio_write(&aio) == -1) {
     printf(" Error at aio_write(): %s\n", strerror(errno));
     close(fd);
     exit(2);
   }
   exit(EXIT_SUCCESS);
  }
}
for(i=0; i<n; i++){
  int status;
  waitpid(pids[i], &status, 0);
}
total = 0;
FILE* file;
file = fopen(argv[1],"r");
for(i=0; i<n; i++){
  /* Lecture dans la ressource partagee */
  fread(&res, sizeof(int), 1, file);
  printf("%d, pid %d envoie %d\n", fd,pids[i], res);
  total += res;
}
free(pids); printf("total: %d\n", total);
return EXIT_SUCCESS;
}
```

Inverseur de contenu asynchrone

```
/*
5 Inverseur de contenu asynchrone
Écrire un programme qui lit un fichier par paquet de 10 caractères et les recopie dans un autre fichier en inversant l'ordre des
caractères (mais pas des paquets). La lecture est synchrone, mais la recopie est asynchrone pour chaque caractère. Vous utiliserez
lio_listio.
*/
int main(int argc,char *argv[]){
	struct stat		buf;
	int			fd1,fd2;
	char			buffer[10];
	char			*charac;
	int			k = 10;
	int			n,t=0 ;
	struct aiocb		aiocb[k];
	struct aiocb*		clio[k];
	struct sigevent		lio_sigev;
	if(argc != 3){
		printf("Il manque des arguments\n");
		exit(1);
	}
	printf("Récupération stat\n");
	if(stat(argv[1],&buf)!=0){ //Récupération des informations du fichier
		perror("Echec stat : ");
		exit(0);
	}
	printf("Vérification si le fichier est régulier\n");
	if(S_ISREG(buf.st_mode)){ //Vérification si le fichier est un fichier régulier
		printf("Le fichier est regulier\n");
	}else{
		printf("Le fichier n'est pas un fichier régulier, sortie..\n");
		exit(0);
	}
	printf("Fin de vérification\n");
	if((fd1=open(argv[1],O_RDONLY,buf.st_mode)) ==-1){ //Ouverture du premier fichier
		printf("Probleme lecture\n");
		printf("%s",strerror(errno));
	}
	if((fd2=open(argv[2],O_CREAT | O_RDWR | O_EXCL, S_IRUSR | S_IWUSR)) == -1){ //Ouvert* ou créat° du 2ème fic
		printf("Problème dans la création\n");
		printf("%s\n",strerror(errno));
	}
	lio_sigev.sigev_notify = SIGEV_NONE;
	int i = 0;
	//Recopie des fichiers
	while((n=read(fd1,&buffer,k)) > 0){
		for(i=0;i<n;i++){
			if(buffer[k-i-1] == 0 || buffer[k-i-1] == '\0')
				continue;
			charac = malloc(sizeof(char)+1);
			charac[0] = buffer[k-i-1];
			charac[1] = 0;
			aiocb[i].aio_fildes = fd2;
			aiocb[i].aio_buf = charac;
			aiocb[i].aio_nbytes = sizeof(char)+1;
			aiocb[i].aio_reqprio = 0;
			//printf("offset : %d\n",t*sizeof(char));
			aiocb[i].aio_offset = t*sizeof(char);
			aiocb[i].aio_sigevent.sigev_notify = SIGEV_NONE;
			aiocb[i].aio_lio_opcode = LIO_WRITE;
			clio[i] = &aiocb[i];
			t++;
		}
		if (lio_listio(LIO_WAIT, clio, n, &lio_sigev) < 0) {perror("lio_listio"); }
	}
	sleep(1);
	close(fd1); close(fd2); //Fermeture des fichiers
}
```

Signaux Temps Réel

Synchronisation par signaux temps réel

```
//1 Synchronisation par signaux temps réel
//
//Ecrire un programme qui crée une chaîne de N processus, N étant l'argument du programme. Chaque processus créé doit afficher son Pid ainsi que son ordre de création, mais l'ordre des affichages se fait dans l'ordre inverse des créations. Cet ordre sera imposé en utilisant des signaux POSIX 4. N.B : vous n'utiliserez pas de processus coordinateur pour effectuer cette synchronisation.
//Exemple d'appel :
//$PWD/bin/creation_tr 4
int *pid;
int i = 0 ;
void real_time_handler(int sig_number, siginfo_t * info, void * arg __attribute__ ((unused)))
{
	printf("PID : %d\n",getpid());
	union sigval value;
		value.sival_int = i;
		if(i == 0){
			sigqueue(getppid(),SIGRTMIN,value);
		}else
			sigqueue(pid[i-1],SIGRTMIN,value);
	exit(EXIT_SUCCESS);
}

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



		int n = 0 ;

	if(argc < 2){
		perror("arg");
		exit(EXIT_FAILURE);
	}
	n = atoi(argv[1]);
	pid = malloc(sizeof(int)*n);

	struct sigaction action;
	sigset_t mask;
	action.sa_sigaction = real_time_handler;
	action.sa_flags = 0;
	sigfillset(&action.sa_mask);
	sigfillset(&mask);
	sigaction(SIGRTMIN,&action,NULL);
	sigprocmask(SIG_BLOCK,&mask,NULL);
	for(i= 0;i<n;i++){
		if((pid[i] =fork()) == 0){
			sigdelset (&mask, SIGRTMIN);
				sigsuspend(&mask);
		}
	}

	union sigval value;
	value.sival_int = i;
	sigqueue(pid[i-1],SIGRTMIN,value);
	sigdelset (&mask, SIGRTMIN);
	sigsuspend(&mask);
}
```

Remontée de valeurs par signaux

```
/*
Reprenez à nouveau l'exercice Remontée de valeurs par partage de mémoire pour que les valeurs aléatoires transmises par les fils soient échangées avec le père en envoyant des signaux temps réel POSIX 4.
Exemple d'appel :
$PWD/bin/remonte_signal 4
*/
int received_signals[10];
int *received_signals_value;
int received_signals_count = 0;
void real_time_handler(int sig_number, siginfo_t * info,void * arg __attribute__ ((unused)))
{
  received_signals_value[received_signals_count] = info->si_value.sival_int;
  ++received_signals_count;
}
void send_real_time_signal(int sig_number, int value)
{
}
int main(int argc,char *argv[]){
  struct sigaction action;
  sigset_t set;
  int i =0, n=0;
  // Handler setup
  action.sa_sigaction = real_time_handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = SA_SIGINFO;
  if ((sigaction(SIGRTMIN, & action, NULL) < 0)) {
     perror("sigaction");
     exit(EXIT_FAILURE);
  }
  n = atoi(argv[1]);
  received_signals_value = malloc(n*sizeof(int));
  // Block all signals
  sigfillset(&set);
  sigprocmask(SIG_BLOCK, &set, NULL);
  for(i=0; i<n; i++){
     int pid = fork();
    if (pid == -1){
        perror("fork");
        return -1;
    } else if(pid == 0 ) {
        union sigval sig_value;
        srand(time(NULL)*i);
        /* Ecriture dans la ressource partagee */
        int temp = (int) (10*(float)rand()/ RAND_MAX);
   printf("PID : %d | Envoi du signal vers %d, value %d\n",getpid(),getppid(), temp);
        sig_value.sival_int = temp;
        if (sigqueue(getppid(), SIGRTMIN, sig_value) < 0) {
           perror("sigqueue");
           exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
    }
  }
  for(i=0; i<n; i++){
    wait(NULL);
  }
  sigfillset(&set); // Unblock all signals
  sigprocmask(SIG_UNBLOCK, &set, NULL)
  for (i = 0; i < n; ++i) {// Display results
     printf("SIGNAL RECU | Valeur : %d\n", received_signals_value[i]);
  }

  return EXIT_SUCCESS;
}
```