
**Barbier Frédéric GI02**
**Lourd Rodolphe GI02**

# EDITION COLLABORATIVE SOUS LINUX/UNIX

# SOMMAIRE

# 1. Cahier des charges

On souhaite réaliser un éditeur classique (pas d'interface graphique) qui permette l'édition collaborative, c'est-à-dire que plusieurs personnes peuvent accéder à un document simultanément. Cet éditeur ne prendra pas en charge le formatage de texte (gras, italique…) ; quatre fonctions sont attendues dans cet éditeur : créer un document, ouvrir un document, modifier/enregistrer un document et fermer un document.

Afin de pouvoir choisir une fonction à n'importe quel moment de l'édition d'un document, il est nécessaire de réserver une console pour un menu répertoriant les fonctions, et ce pour chaque utilisateur. Le menu se présentera sous la forme d'une liste de choix numérotés de un à quatre. Au lancement de l'éditeur, la console affiche le menu ; l'utilisateur a le choix entre la création et l'ouverture d'un document. Ensuite, une fois un document ouvert ou créé, l'utilisateur a la possibilité de le modifier ou de le fermer.

## *a. Création*

Lorsque l'option création est lancée, on demande un nom de fichier à l'utilisateur. Le programme vérifie notamment que le nom de fichier ne comporte de caractères interdits ; il vérifie aussi la longueur de ce nom. Si le nom est incorrect, on demande à l'utilisateur de saisir un autre nom. On note que le fichier est créé dans un répertoire fixe accessible à tous, par exemple //gi/lo41, c'est-à-dire le même répertoire utilisé pour l'emplacement de l'éditeur.

## *b. Ouverture*

Lorsque l'utilisateur appelle la fonction ouverture, l'éditeur va demander un nom de fichier à cet utilisateur ; on vérifie que ce nom de fichier existe dans le répertoire du programme. Si ce n'est pas le cas, le programme demande à l'utilisateur de saisir un nouveau nom de fichier valide. Quand le nom de fichier existe, l'éditeur ouvre ce document dans une nouvelle console réservée à l'affichage et au rafraîchissement du document ; si ce document est déjà ouvert, le programme le notifie à l'utilisateur sans le rouvrir.

## *c. Modification*

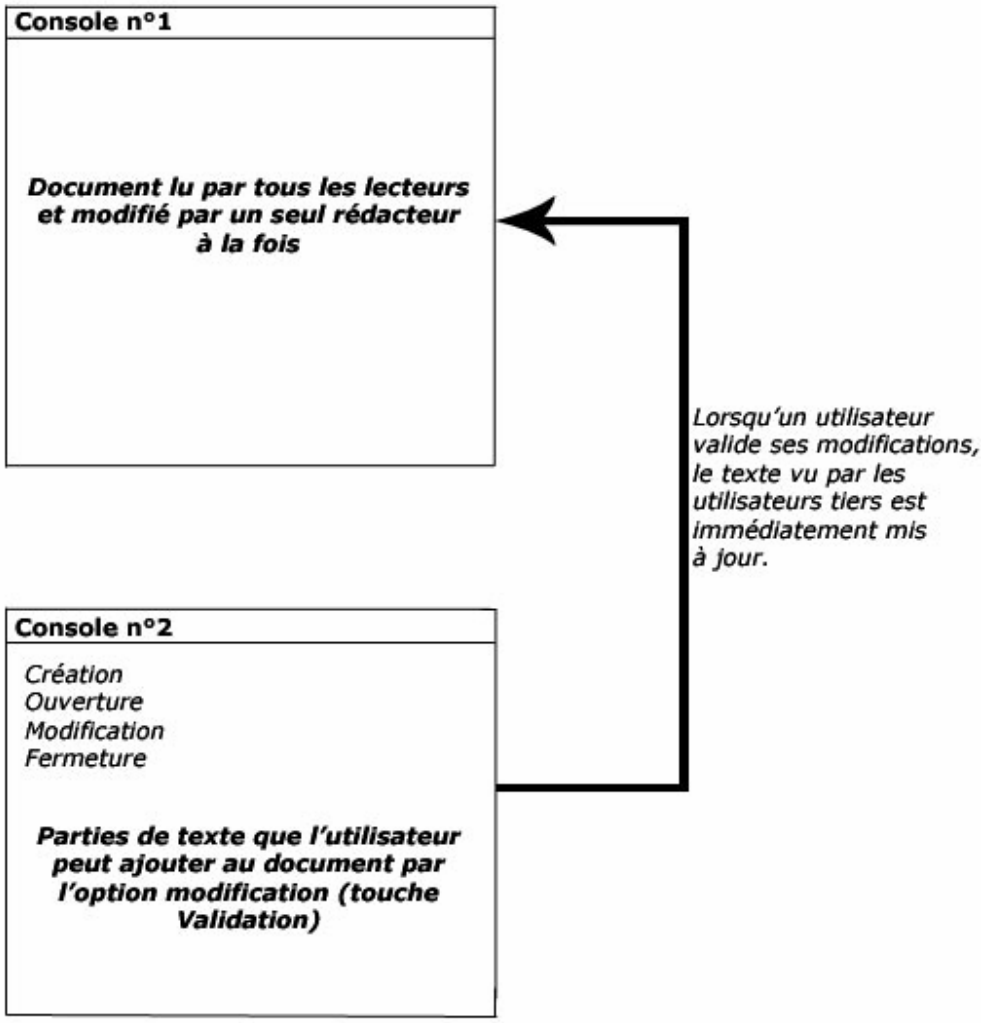
La fonction modification est assurée par la touche *validation* (*entrée*). Quand un utilisateur appuie sur cette touche, la ligne qu'il vient de taper est ajoutée au document ouvert dans sa console d'affichage ainsi que celle de chaque utilisateur. Un enregistrement est également effectué à cet instant.

### *d. Fermeture*

Lorsque l'utilisateur choisit l'option fermeture, seule la fenêtre d'affichage du document est fermée, l'autre console affichant le menu et la saisie restant ouverte, pour une autre ouverture de fichier éventuelle.

### *e. Schéma de l'interface*

L'interface très simple se présente de la manière suivante :




Chaque utilisateur ouvre deux consoles lorsqu'il lance le programme. La *console n°1* montre le document à l'ouverture ; la *console n°2* offre les commandes nécessaires telles que *création, ouverture, modification* et *fermeture*. Les modifications potentielles apportées au document sont d'abord rédigées dans la *console n°2* ; lorsque l'utilisateur décide des les valider, il choisit la commande *Modification* (touche

*Entrée*). A ce moment-là s'applique le *principe d'exclusion mutuelle* qui empêche un autre utilisateur de valider aussi ses modifications pendant le très court laps de temps dont la première modification a besoin.

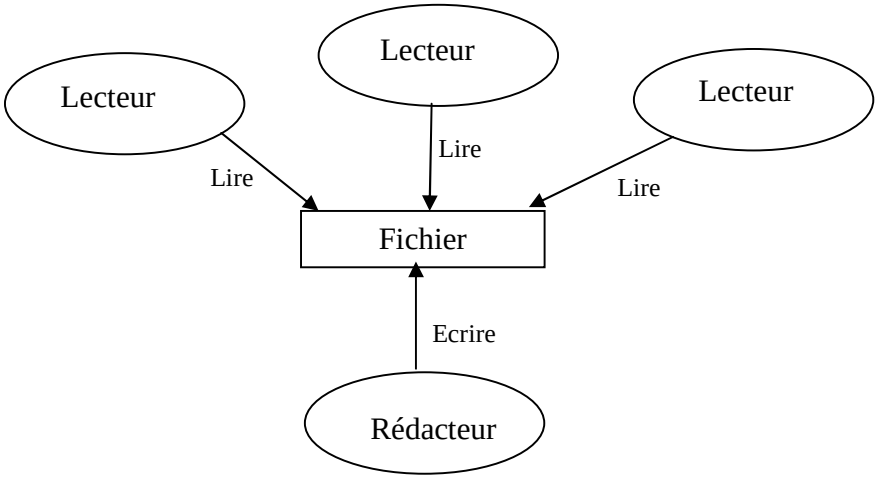
Une fois la modification effectuée, les autres utilisateurs voient leur document mis à jour en temps réel.

# 2. Spécifications

## *a. Synchronisation entre les processus lecteurs et rédacteurs*

Le fichier créé est une ressource commune à tous les processus utilisateurs, certains de ces processus étant considérés comme *lecteurs* (ceux voulant lire dans le fichier) et d'autres comme *rédacteurs* (ceux voulant écrire dans le fichier).

On obtient donc le modèle des Lecteurs-Rédacteurs :




*Principes*

- La lecture est possible, simultanément par plusieurs lecteurs, à condition qu'il n'y ait aucune écriture.
- L'écriture n'est possible que s'il n'y a ni lecture ni autre écriture en cours.

Pour synchroniser ces processus, nous devons appliquer une méthode d'exclusion mutuelle. Nous pensons que la synchronisation *Lecteurs-Rédacteurs* par sémaphores est la plus adaptée. L'algorithme d'une modification dans notre cas serait le suivant :

```
procedure modification (verrouillage, texte, document)
début
        tant que (verrouillage = vrai) faire
            …
        fintantque
        sinon verrouillage := vrai
                document := document + texte
                verrouillage := faux
        finsi
fin
```

Dans cette procédure, *verrouillage* est une variable booléenne globale qui indique si le document est bloqué en écriture ou non. Ce type de procédure est lancé chaque fois qu'un utilisateur valide sa modification ; ainsi on évite les modifications simultanées préjudiciables au bon fonctionnement de l'éditeur. La boucle *tant que* met simplement en attente l'utilisateur d'un déverrouillage du document en écriture.

## *b. Envoi de signaux pour la communication entre terminaux*

Pour rafraîchir les fenêtre d'affichage utilisateurs ou encore pour fermer sa fenêtre d'affichage à la fermeture du fichier, nous utiliserons l'envoi de signaux aux terminaux correspondants et donc, il nous faudra connaître le *pid* de chaque console d'affichage. Nous récupéreront donc ce *pid* à l'ouverture du fichier et, pour que chaque processus utilisateur puisse accéder à ces numéros, nous les stockerons dans un fichier. Nous devrons bien sûr mettre à jour ce fichier, c'est-à-dire qu'à chaque nouvelle ouverture de fichier, nous rajouterons un *pid* et à chaque fermeture de fichier, nous effacerons le *pid* correspondant.

Comme nous allons de nouveau utiliser un fichier avec des opérations de lecture écriture, nous devons bien sûr synchroniser les processus à l'aide de l'exclusion mutuelle décrite précédemment. Ainsi, nous utiliserons deux systèmes d'exclusion mutuelle similaires.

Pour conclure, nous devons préciser que ces idées de conception sont celles qui nous semblent les plus adaptées au vu de nos connaissances actuelles. Il est par conséquent possible que durant la phase de codage, des difficultés imprévues nous incitent à rechercher des solutions différentes qui seraient plus facilement réalisables.

# 3. Réalisation

Le programme s'appuie sur plusieurs sous-programmes : *affichage* s'occupe de tout ce qui concerne l'affichage de l'interface, notamment la gestion des fenêtres *xterm* ; *essai* est le programme lui-même qui fait donc appel à d'autres fonctions ; *sema* gère les sémaphores qui sont essentiels dans l'application, particulièrement dans la gestion de l'exclusion mutuelle.

## *a. Affichage.c*

```
#include <stdio.h>      /* librairies*/
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <wait.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "sema.h"       /*importation de sema.h pour les opérations sur sémaphores*/

void handler(int sig);
char fichaff[40];       /*variable contenant le nom de fichier à afficher*/
void affich(void);
int sema;               /*sémaphore*/

main(int argc, char *argv[]){
        umask(0000);            /*pour donner tous les droits aux fichiers créés par le programme*/
        int i;
        FILE *fp;
        struct sigaction action;
        struct tabpid{          /*structure utilisée pour stocker les pid de menus et d'affichage récupérés de pid2.dat*/
                int pidmenu[10];
                int pidaff[10];
                };
        struct tabpid tp;
        pid_t pid=getpid();             /*récupération du pid d'affichage*/
        printf("%d",pid);
        strcpy(fichaff,argv[1]);
        action.sa_handler=handler;  /*mise en place du gestionnaire de signaux*/
        sigaction(SIGUSR2,&action,0);
        sigaction(SIGUSR1,&action,0);
        initMutex(&sema);               /*initialisation du sémaphore*/
```

```
	P(sema);			/*entrée en section critique*/
	fp=fopen("pid2.dat","r");
	fread(&tp,sizeof(struct tabpid),1,fp);	/*récupération de la structure tabpid
contenue dans pid2.dat*/
	fclose(fp);
	i=0;
	while(tp.pidmenu[i]!=0)
		i++;
/*ajout des nouveaux pid (pid du processus d'affichage et du menu correspondant)
dans tp*/
	tp.pidmenu[i]=atoi(argv[2]);
	tp.pidaff[i]=(int)pid;
/*écrase le contenu de pid2.dat avec la structure mise à jour*/
	fp=fopen("pid2.dat","w");
	fwrite(&tp,sizeof(struct tabpid),1,fp);
	fclose(fp);
	V(sema);		/*sortie section critique*/
	affich();		/*affichage du fichier*/
	}

void handler(int sig){
	switch(sig){
		case SIGUSR1:{
			system("clear");
			affich();		/*mise à jour du contenu*/
			break;
			}
		case SIGUSR2:{
			system("exit");	/*cas 4 du menu*/
			break;
			}
		}
	}

void affich(void){
	char chaine[60];
	strcpy(chaine,"more ");
	strcat(chaine,fichaff);
	system(chaine);
	pause();
	}
```

## b. Essai.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
#include "sema.h"       /*importation de sema.h pour les opérations sur sémaphores*/
#define true 0
#define false 1

void menu(void);        /* déclaration des fonctions*/
void modifier(void);
void lecture(char textalire[]) ;
void creer_fichier(void);
int FichierExiste (char Nom[20]);
void DemanderNomFichier (char NomFichier[20]);
void wait(void);

/* variables globales pouvant être utilisées dans tout le programme*/
char strFile[20];       /*nom du fichier saisi par l'utilisateur*/
int sema;               /*sémaphore*/

int main(void){
        umask(0000);    /*pour donner les droits à tous les fichiers créés par le programme*/
        initMutex(&sema);       /*initialisation du sémaphore*/
        menu();         /*le main ne comporte qu'une seule procédure : menu()*/
        exit(0);
        }

void menu(void){
        struct tabpid{  /*structure utilisée pour stocker les pid de menus et d'affichage récupérés de pid2.dat*/
                int pidmenu[10];
                int pidaff[10];
                };
        struct tabpid tp;
        char rep[30],chaine1[100],erase[100]="";    /*rep contiendra la réponse de l'utilisateur concernant le choix du menu*/
        int i;
        int ouvert=false;
```

```
	FILE *fp;
	pid_t pid;
	char pidm[5];
	pid_t pidmen=getpid();		/*récupération du pid du menu*/
	//system("clear");
	printf("\t\t\tPROJET LO41\n\t\t\t BIENVENUE\n");
	do{
		printf("\n    ***menu***\n");		/*affichage du menu*/
		printf("\n1 : créer fichier\n2 : ouvrir \n3 : modifier\n4 : fermer fichier\n5 : quitter\n");
		printf("\nvotre choix: ");
		scanf("%s",&rep);
		if((atoi(rep)<1)||(atoi(rep)>5))	/*atoi : convertit la réponse de caractère à entier*/
			printf("erreur : entrez un chiffre entre 1 et 5\n");
		else{
			switch (atoi(rep)){	/* appel des différentes procédures/fonctions suivant la reponse*/
				case 1:{
					creer_fichier();
					break;
					}
				case 2:{
					if (ouvert==false){
						do{
							printf("Entrez le nom du fichier à ouvrir\n");
							scanf("%s",strFile);
							if (!FichierExiste(strFile))
								printf("Ce fichier n'existe pas\n");
							}while(!FichierExiste(strFile));
						strcpy(chaine1,erase);  /*vidage de la chaine*/
						strcat(chaine1,"xterm -e ./affichage.exe ");
						strcat(chaine1,strFile);
						sprintf(pidm,"%d",pidmen);
						strcat(chaine1," ");
						strcat(chaine1,pidm);   /*on envoie en paramètre le pid du menu*/
						strcat(chaine1," &");
						system(chaine1);
						ouvert=true;
						}
					else{
						printf("Un fichier est déjà ouvert;\nPour en ouvrir un autre, veuillez d'abord fermer ce fichier\n");
						}
					break;
```

```
                }
        case 3:{
                if (ouvert==true){
                        int i=0;
                        printf("modification du fichier\n");
                        P(sema);
                        modifier();
                        /*récupération de la structure contenant les pid
de menus et d'affichage de tous les utilisateurs*/
                        fp=fopen("pid2.dat","r");
                        fread(&tp,sizeof(struct tabpid),1,fp);
                        fclose(fp);
                        V(sema);
                        for (i=0;i<10;i++){
                                if (tp.pidaff[i]!=0)
                                        pid=tp.pidaff[i];
                                kill(pid,SIGUSR1);/*envoi du signal de mise
à jour à toutes les consoles d'affichage*/
                                }

                        }
                else{
                        printf("Il faut d'abord ouvrir un fichier\n");
                        }
                break;
                }
        case 4:{
                int i=0;
                if (ouvert==true){
                        P(sema);    /*rentre dans la section critique car il
ne faut pas que le contenu du fichier pid change entre la lecture du contenu et la
mise à jour*/
                        fp=fopen("pid2.dat","r");/*récupération des pid
contenus dans pid2.dat*/
                        fread(&tp,sizeof(struct tabpid),1,fp);
                        fclose(fp);
                        while (tp.pidmenu[i]!=(int)pidmen)
                                i++;
                        pid=tp.pidaff[i]; /*récupération du pid
d'affichage correspondant au bon menu pour la fermeture*/
                        do{   /*on décale les valeurs pour supprimer le
couple de pid correspondant au fichier ouvert*/
                                tp.pidmenu[i]=tp.pidmenu[i+1];
                                tp.pidaff[i]=tp.pidaff[i+1];
                                i++;
                                }while(i<9);
```

```
                                    tp.pidmenu[9]=0;
                                    tp.pidaff[9]=0;
                                    fp=fopen("pid2.dat","w");/*on écrase le contenu
de pid.dat avec la structure mise à jour*/
                                    fwrite(&tp,sizeof(struct tabpid),1,fp);
                                    fclose(fp);
                                    V(sema);
                                    kill(pid,SIGUSR2);
                                    printf("%d",pid);
                                    ouvert=false;
                                    }
                                else{
                                    printf("Aucun fichier à fermer\n");
                                    }
                                break;
                                }
                            case 5:{
                                if (ouvert==true){
                                    printf("Fermer d'abord le fichier encore ouvert
avant de quitter\n");                   }
                                else{
                                    printf("\nFIN DU PROGRAMME\n");
                                    fp=fopen("pid2.dat","r");
                                    fread(&tp,sizeof(struct tabpid),1,fp);
                                    fclose(fp);
                                    if (tp.pidmenu[0]==0)
                                        libereSem(sema);/* si plus aucun pid dans
pid.dat, libération du sémaphore*/
                                    }
                                break;
                                }
                            default:{
                                printf("entrer un chiffre entre 1 et 6");
                                /*message d'erreur*/
                                break;
                                }
                            }
                        }
                        wait();
                        system("clear");
                    }while ((atoi(rep)!=5)||(ouvert==true));
                }

void creer_fichier(void){       /*enregistre la liste sur un fichier physique*/
        int fd;
        DemanderNomFichier(strFile);
```

```
	P(sema);
	fd = open(strFile, O_CREAT | O_WRONLY|O_APPEND, S_IREAD +
S_IWRITE);
	if (fd < 0){
		printf("Erreur en créant le fichier %s.\n", strFile);
		}
	close(fd);
	V(sema);
	}

void DemanderNomFichier (char NomFichier[20]){
	int Existe=true;
	char rp[10];
	do{
		printf ("Entrez le nom du fichier à créer: ");
		scanf ("%s", NomFichier);
		if (Existe = FichierExiste (NomFichier)){
			printf ("Existe déjà\nVoulez vous créer un autre fichier? (o/n)\n");
			scanf("%s",&rp);
			}
		else
			printf("Création du fichier %s \n",NomFichier);
		}while ((Existe)&&(strcmp(rp,"o")==0));
	}

int FichierExiste (char Nom[20]){
	int Retour = true;
	FILE* Fichier = fopen (Nom, "rb");
	if (Fichier != NULL){
		fclose (Fichier);
		Retour = false;
	}
	else
		Retour = true;
	return Retour;
}

void modifier(void){
	int fd;
	char texte[300];
	int nNbCharWritten;
	lecture(texte);
	fd=open(strFile, O_CREAT|O_WRONLY|O_APPEND, S_IREAD + S_IWRITE);
	nNbCharWritten = write(fd, texte, strlen(texte));
	if (nNbCharWritten != strlen(texte)){
		printf("Erreur en écrivant dans le fichier %s.\n", strFile);
```

```
            }
        close(fd);
        }

void lecture(char textalire[]){       /* procédure qui permet de lire un texte de 300
caractères maxi*/
        int nbcar=0;
        int i, ch;
        strcpy(textalire,"");      /*initialise le texte à lire*/
        printf("\n***lecture du texte***\n");
        printf("(nbre maxi de caractere:300)\ntaper '@ et <<entree>>' pour
arreter:\n");
        for( i = 0; (i < 300) &&  ((ch = getchar()) != EOF) && (ch != '@'); i++ ){
                textalire[i] = (char)ch;
                nbcar++;
                }
        /* Termine la chaine avec le caractere nul*/
        textalire[i] = '\0';
        printf("Fin du texte\n");
        if (nbcar>=300){
                printf("\nle nombre maximum de caractere a ete atteint\n");
                getchar();
                }
        }

void wait(void){  /* procédure pour faire une pause dans l'exécution du
programme*/
        printf("\nAppuyez sur une touche pour continuer...\n");
        getchar();
        getchar();
        }
```

## c. Sema.c

```
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <stdio.h>

union semun{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf;
        }sem_union;

struct tabpid{      /* définition de la structure à double tableau tabpid */
        int pidmenu[10];
        int pidaff[10];
        };

static void erreurFin(const char* msg){
        perror(msg);
        exit(1);
        }

static void appelSem(int tabSemId, int op){
        struct sembuf sempar;
        const int nbop = 1;
        sempar.sem_num = 0;
        sempar.sem_op = op;
        sempar.sem_flg = 0;
        if (semop(tabSemId,&sempar,nbop) == -1) erreurFin("appelSem");
        }

/* primitives */
void P(int tabSemId){
        appelSem(tabSemId,-1);
        }
void V(int tabSemId){
        appelSem(tabSemId,1);
        }

/* création du sémaphore */
void initMutex(int* tabSemId){
        key_t cle;
        /* vérification que le sémaphore n'existe pas déjà ;
        s'il n'existe pas, on le crée et on l'initialise à 1 */
```

```
        if ((cle = ftok("fich_sema2",'0'))==(key_t)-1)
                erreurFin("initMutex: ftok");
        sem_union.val=1;        /* cas où le sémaphore existe déjà */
        if ((*tabSemId  = semget(cle, 1, IPC_CREAT | IPC_EXCL | 0600))== -1){
                printf("le sémaphore existe déjà\n");
                /* on accède au sémaphore */
                if ((*tabSemId  = semget(cle, 1, IPC_CREAT | 0600))== -1){
                        erreurFin("initSem : semget");
                        }
                else
                        printf("semaphore accede\n");
                }
        else{
        /*cas où le sémaphore n'existait pas encore = initialisation du sémaphore */
                if (semctl(*tabSemId, 0, SETVAL,sem_union) == -1){
                        erreurFin("initSem : semctl 1");
                }
                else{
                        int i;FILE *fp;
                        struct tabpid tp;
                        for(i=0;i<10;i++){
                                tp.pidmenu[i]=0;
                                tp.pidaff[i]=0;
                                }
                        fp=fopen("pid2.dat","w+");
                        fwrite(&tp,sizeof(struct tabpid),1,fp);
                        fclose(fp);
                        printf("semaphore initialise a 1\n");
                        }
        }

int getVal(int tabSemId){
        return(semctl(tabSemId,0,GETVAL,0));
        }

/* destruction du sémaphore */
void libereSem(int tabSemId){
         if (semctl(tabSemId,0,IPC_RMID,0)==-1)
        erreurFin("libereSem");
        }
```

# 4. Difficultés rencontrées

## *a. Communication entre les processus menu/affichage*

La fonction affichage a posé quelques difficultés. En effet, pour établir la communication entre le processus du menu et celui de sa fenêtre d'affichage, il fallait faire correspondre le *pid* du menu avec celui de *xterm* (la fenêtre d'affichage).

Le problème fut résolu par l'envoi du *pid* du menu en le mettant en argument de main dans la fonction *affichage*. Dans ce dernier, on met en relation les deux *pid* (menu et affichage) par une structure *tabpid* composée de deux tableaux (un pour les *pid* de menu, un pour les *pid* d'affichage). Cette structure est initialisée simultanément avec le sémaphore à 0 ; avec la fonction *fwrite*, elle est écrite dans le fichier *pid.dat*. Cette opération n'est par conséquent réalisée qu'une seule fois avec le premier utilisateur du document ; pour chaque ouverture supplémentaire, on utilise la fonction *fread* qui récupère la structure ; celle-ci est parcourue et le couple *pid* menu/affichage est écrit après ceux des utilisateurs précédents.

A chaque fois que la structure est modifiée, est réécrite entièrement en écrasant la précédente dans *pid.dat*. On procède de cette manière car lorsqu'un utilisateur ferme sa fenêtre d'affichage, le couple correspondant présent dans la structure doit être supprimé et tous les autres qui le suivent décalés. Pour cette raison il est plus simple de tout réécrire sauf le couple de *pid* à supprimer.

## *b. Utilisation des signaux pour l'affichage*

Une fois la précédente structure établie, on utilise deux types de signaux afin de mettre à jour ou de fermer la fenêtre d'affichage (*xterm*) :

- *SIGUSR1* pour le rafraîchissement de la fenêtre
- *SIGUSR2* pour la fermeture de la fenêtre

Pour le rafraîchissement, on récupère en boucle les *pid* d'affichage inclus dans le champ *pidaff* de la structure *tabpid* et d'envoyer le signal *SIGUSR1* à chaque *pid*. Pour la fermeture, on parcourt la structure à la recherche du *pid* qui correspond au fichier à fermer, c'est-à-dire le *pid* courant du menu ; lorsque le bon *pid* est trouvé dans le champ *pidmenu*, on envoie *SIGUSR2* au *pid* contenu à l'indice correspondant au champ *pidaff*.

## *c. Traitement des signaux (affichage.c)*

Dans la fonction *affichage*, on utilise des signaux afin de communiquer la mise à jour ou la fermeture d'un fichier. Seulement il faut que l'utilisateur ne puisse pas sortir anormalement du programme par les raccourcis *Ctrl-C* et *Ctrl-Z*. En effet, si cela se produit, un fichier qu'il a ouvert mais pas fermé risque de le rester, et même si dans un éditeur de texte cela ne pose pas de problème majeur, il peut en être tout autrement

dans le cas de plusieurs dizaines d'utilisateurs. Il faut donc empêcher l'utilisateur d'utiliser ces raccourcis. Le code :

```
action.sa_handler=handler;
sigaction(SIGUSR2,&action,0);
sigaction(SIGUSR1,&action,0);
initMutex(&sema);
```

met en œuvre le gestionnaire des signaux, les deux signaux de rafraîchissement/fermeture et initialise le sémaphore (*mutex*). Pour masquer les signaux dus à *Ctrl-C* ou *Ctrl-Z*, on utilise la section critique du sémaphore, c'est-à-dire un laps de temps durant lequel aucune action n'est possible concernant le sémaphore (pas de mise en attente ou de commencement de processus) ; ainsi, les raccourcis *Ctrl-C* et *Ctrl-Z* ne sont pas accessibles. Les entrée/sortie de la section critique sont réalisées par les instructions :

- *P(sema)* pour l'entrée
- *V(sema)* pour la sortie

## *d. Utilisation/implantation des sémaphores*

Le concept de sémaphore est issu de la signalisation ferroviaire ; il est utilisé pour contrôler l'accès à des tronçons de voie ferrée par des trains circulant les uns derrière les autres avec des ressources non partageables. Les sémaphores sont une amélioration de la relation d'*exclusion mutuelle* qui reste limitée pour décrire la synchronisation entre processus. Ils sont composés :

- d'un *marqueur* utilisé pour synchroniser plusieurs processus ; il agit comme une clé qu'un processus doit obtenir pour continuer son exécution
- d'une *file d'attente des tâches* qui sont éventuellement bloquées devant le sémaphore

Un sémaphore fonctionne comme un distributeur de jetons ; on fixe celui-ci à la création du sémaphore. Un jeton est non renouvelable et rendu après utilisation ; lorsqu'un jeton est acquis (devient *verrouillé*), le nombre de jetons décroît, lorsqu'un jeton est libéré (devient *déverrouillé*), le nombre de jetons croît. Naturellement, le nombre de jetons ne peut être négatif. Lorsqu'il n'y a plus de jetons disponibles, les processus qui arrivent restent en attente jusqu'à ce qu'un jeton se libère ; ainsi, une ressources à *n points d'accès* est gérée par un sémaphore à *n jetons*. Le distributeur de jetons est géré de la manière suivante :

- *P(s) = down(s)* : si un jeton est disponible, le processus l'obtient, sinon le jeton devient disponible
- *V(s) = up(s)* : si un processus est en attente, un seul obtient le jeton, sinon le jeton est rendu disponible
- C'est entre *P(s)* et *V(s)* que l'on accède à la ressource

L'implantation d'une telle structure étant déjà réalisée en Travaux Pratiques, elle ne posa pas trop de problèmes sous Unix. En revanche, elle ne fonctionnait pas sous Linux, système d'exploitation sous lequel la quasi-totalité du programme fut développée. En effet, les paramètres de la fonction *semctl* n'étaient pas pris en compte de la même manière, c'est pourquoi la fonction *union* fut utilisée pour rendre les instructions compatibles sous systèmes Unix et Linux :

```
m_union.val=1;        /* pour donner la valeur d'initialisation du sémaphore */
semctl(*tabSemId,0,SETVAL,sem_union);        /* pour l'initialiser */
```

# 5. Conclusion

Ce projet fut long dans sa phase de conception, lorsqu'il fallut décrire le cahier des charges. En effet nous n'avions qu'une vague idée de ce qu'il fallait réaliser, et une vision encore plus floue des concepts qui allaient être mis en œuvre. Ce n'est qu'au fur et à mesure de l'avancement du projet que le sujet se précisa, pour arriver au programme final.

Celui-ci marche parfaitement, néanmoins quelques améliorations peuvent être apportées. Tout d'abord, le nombre d'utilisateurs maximum est fixe (10 maximum) ; la solution consisterait en la création de tableaux dynamiques dans la structure tabpid plutôt que des statiques. Ensuite, dans le programme un seul sémaphore gère les exclusions mutuelles entre utilisateurs ; il serait judicieux d'en utiliser un second pour par exemple gérer d'une part les exclusions sur le fichier de travail et d'autre part une sur le fichier pid.dat (déjà réalisée). Enfin, un détail d'ordre pratique, actuellement les fichiers créés se font dans le répertoire courant du programme ; ils pourraient l'être dans un répertoire accessible par tous (par exemple /public) voire même dans un répertoire spécifié par l'utilisateur.