TP2 ET 3
SEMAPHORE ET MEMOIRE PARTAGEE

CS366 : Systèmes d'exploitations

Binôme :
Jessy GIACOMONI
Adrien ROGIER

Enseignant :
Stéphanie CHOLLET
Eric SIMON

Sommaire

Table des matières

Avant propos :

But du Projet

Ce projet a pour but de nous familiariser avec la gestion des processus, les sémaphores et que les mémoires partagées. Dans ce projet, nous devrons gérer la création et l'ordonnancement des processus entre père et fils, la gestion des accès mémoires et fichier, la protection des données.

Rappel des objectifs

L'objectif de ce TP est de réaliser une chaîne d'acquisition de données. Le résumé en schéma :

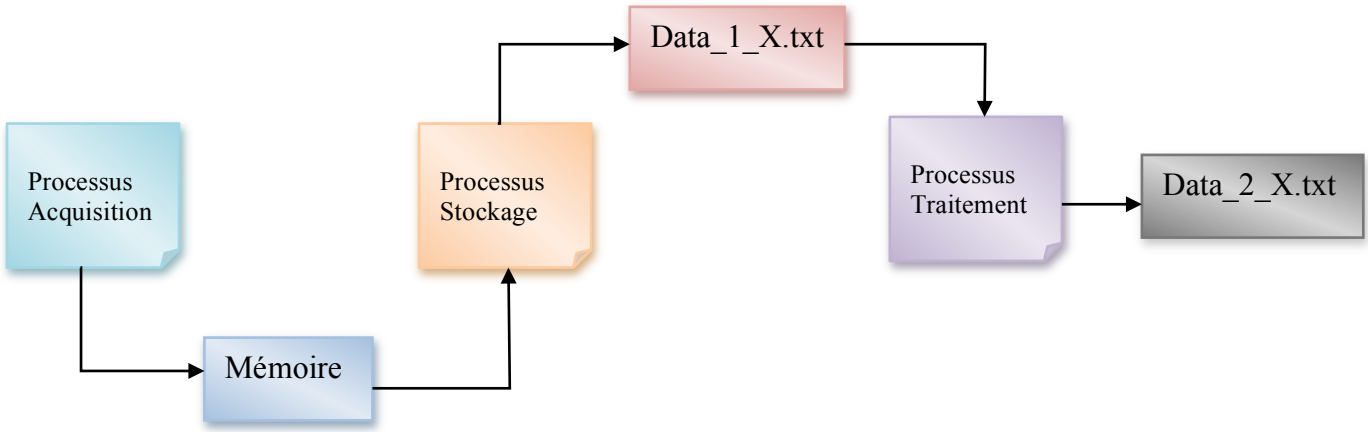

Figure 1 : Chaîne d'acquisition

Nous avons 4 processus à gérer dans cette chaîne d'acquisition :

- **Processus père :** Créer les 3 processus fils de la chaîne d'acquisition. Il veille à ce que chaque fils exécute bien son code.
- **Processus d'acquisition :** Génère des données aléatoires, puis stockent les valeurs dans une mémoire allouée par l'ordinateur.
- **Processus de stockage :** Ce processus récupère les valeurs stockées dans la mémoire alloué par le processus d'acquisition. Puis stockent ces valeurs dans un fichier texte *Data_1_X.txt*.

- **Processus de traitement :** Récupère les valeurs dans le fichier Data_1_X.txt, y effectuent un traitement, et stockent les résultats dans un fichier texte Data_1_X.txt.

Sémaphore

Dans ce projet, nous allons utiliser en tout 6 sémaphores :

- 1 sémaphore pour indiquer que la mémoire alloué est pleine.(initialisé à 0)
- 1 sémaphore pour indiquer que la mémoire alloué est vide.(initialisé à 1)
- 1 sémaphore d'exclusion mutuelle pour l'accès à la mémoire alloué en commun avec les processus acquisitions et stockage. (initialisé à 1)
- 1 sémaphore pour indiquer que le fichier de stockage entre les processus stockage et traitement est pleine.(initialisé à 0)
- 1 sémaphore pour indiquer que le fichier de stockage entre les processus stockage et traitement est vide.(initialisé à 1)
- 1 sémaphore d'exclusion mutuelle pour l'accès au fichier de data en commun avec les processus stockages et traitement.(initialisé à 1)

Contenu

Ce projet est composé de 8 fichiers (+2 avec le bonus):

Description du contenu

Makefile

Pour notre Makefile nous avons décidé d'aller plus loin en faisant un makefile générique c'est à dire qu'il soit utilisable pour plusieurs projets. Nous voulions au départ utiliser des générateurs de makefile avec configure.ac. Nous n'avons pas voulu perdre trop de temps sur cet outil donc nous avons abandonné l'idée.

Voici les grandes lignes pour expliquer notre Makefile.

SRC = $(wildcard *.c) : Il s'en sert pour récupérer tous les fichiers finissant par .c du répertoire dans lequel se situe le Makefile.
OBJS = $(SRC:.c=.o) : permet de préciser les nom de chaque fichier .o à partir du fichier .c.
> Le nom de la cible est « @ »
> Le nom de la première dépendance est « < »
> La liste des dépendances est « ^ »
%.o : %.c : règle générique pour la construction d'un fichier .o.

Option GCC :
o2 : Le compilateur tente de réduire la taille du code et le temps d'exécution tout en limitant le temps de compilation.
Wall : activer les messages d'avertissement

Un fichier main.c

Ce fichier contient le code du processus père. Celui-ci doit exécuter tous les processus fils c'est à dire acquisition, stockage et traitement. Nous verrons plus en détail chaque processus plus tard.

Algorithme

```
Fonction main(nbrSerie, delaiEntreSerie, nbrAcquisition,
delaiAcquisition)
     Initialisation (Sémaphore + Mémoire partagée)
     Création processus acquisition
     Création processus stockage
     Création processus traitement
     Attente de la fin de tous les processus
     Destruction de la mémoire partagée et des sémaphores
Fin Fonction
```

Tableau 1 : Algorithme processus père

Diagramme de séquence

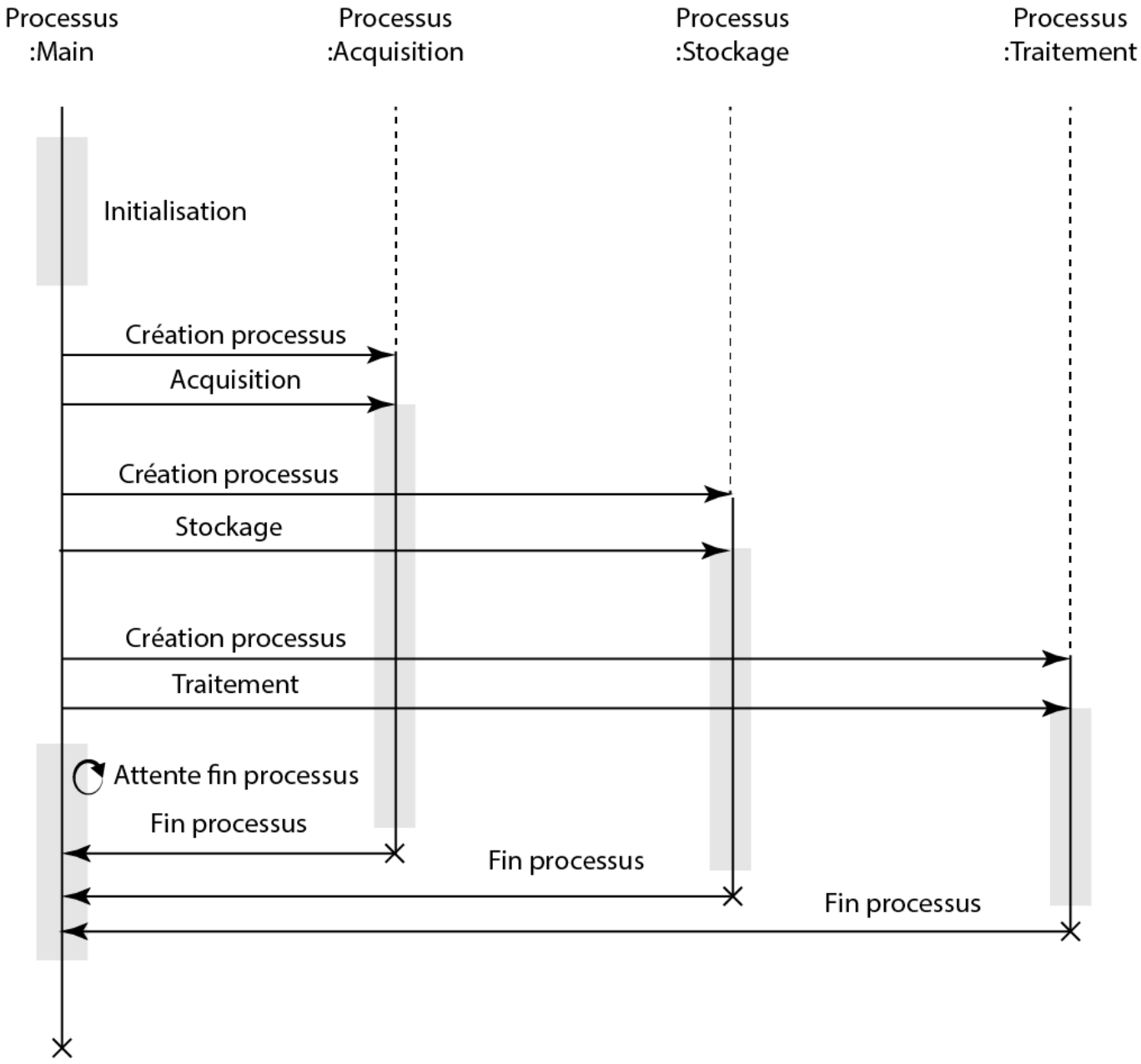

Figure 2 : Diagramme séquence processus père

Fichier acquisition.c/h :

L'acquisition des données est simulée : les données sont l'heures actuelles du système.
La fonction gettimeofday() permet de récupérer cette information par l'intermédiaire d'une structure de donnée timeval.

Remarque : L'algorithme et organigrammes proposé fonctionne à condition que les sémaphores soit initialisé aux bonnes valeurs dans le fichier main.

Algorithme

```
Fonction acquisition (nbrAcquisition, delaiEntreSerie,  nbrSerie,
delaiAcquisition, ptr_mem_partagee, mem_ID_Proc_Acquisition,
MUTEX_acquisition_stockage, semaphore_Proc_Acquisition_Stockage_Mem_plein,
semaphore_Proc_Acquisition_Stockage_Mem_vide)
      Tantque nbrSerie>nbrSerieEtabliefaire
            Prise semaphore_Proc_Acquisition_Stockage_Mem_vide
            Prise MUTEX_acquisition_stockage
             Section Critique
                  Réservation ptr_mem_partageeavecmem_ID_Proc_Acquisition
                  TantquenbrAcquisition>nbracquistionAcquisfaire
                        Ecriture des données dans ptr_mem_partagee
                        AttendredelaiAcquisition
                  FinTantque
                  Libération ptr_mem_partageeavecmem_ID_Proc_Acquisition
             Fin Section Critique
             Libération MUTEX_acquisition_stockage
             Libération semaphore_Proc_Acquisition_Stockage_Mem_plein
             Attendre delaiEntreSerie
      FinTantque
Fin Fonction
```

Tableau 2: Algorithme processus acquisition

Organigramme

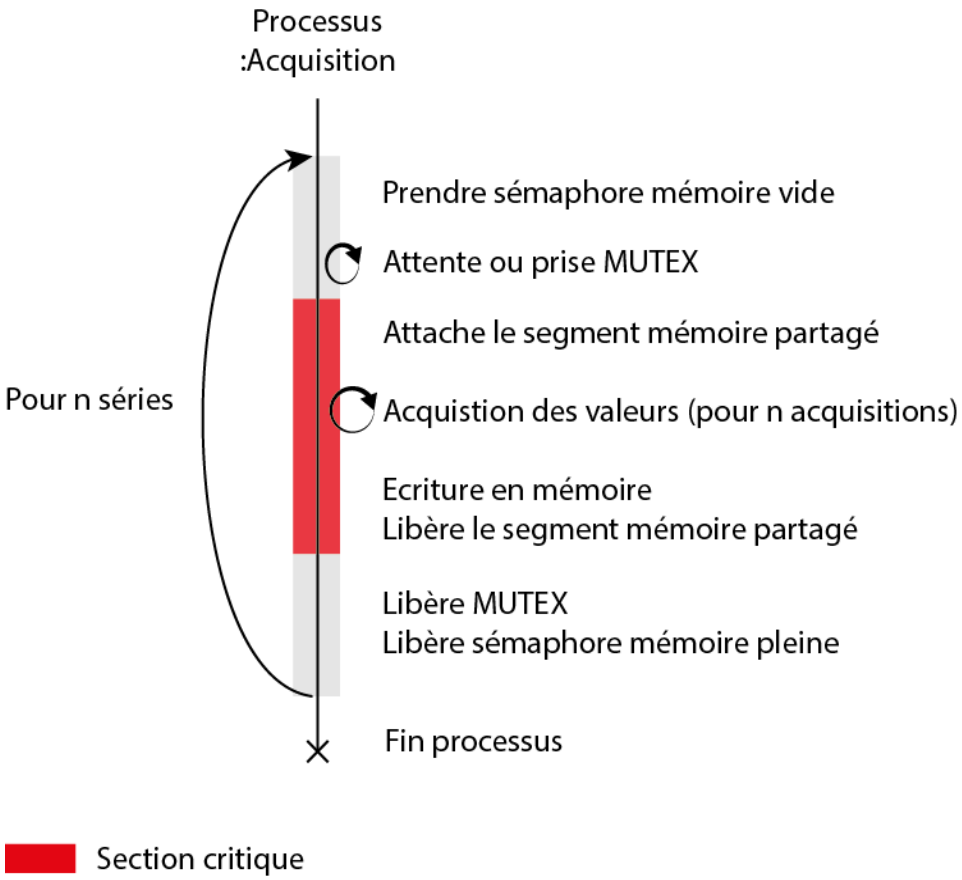

Figure 3 : Diagramme séquence processus acquisition

Un programme stockage.c/h :

DESCRIPTION :

Cette fonction prends les données que nous avons récupéré dans le processus précédent et ensuite copier ces données dans un fichier.

SPECIFICATION :

- La fonction stockage devra générée un fichier data_1_X.txt avec X correspondant au nombre de série.
- Dans le fichier data_1_X.txt on aura strictement une valeur par ligne.
- La fonction supprimera les valeurs du tableau que celle-ci à copier dans le fichier data_1_X.txt

Algorithme

```
Fonction stockage(nbrAcquisition, delaiEntreSerie, nbrSerie,
sem_P, sem_V, mem_ID_Proc_Stockage, ptr_mem_partagee,
MUTEX_acquisition_stockage, MUTEX_stockage_traitement,
semaphore_Proc_Acquisition_Stockage_Mem_plein,
semaphore_Proc_Acquisition_Stockage_Mem_vide,
semaphore_Proc_Stockage_Traitement_fichier_plein,
semaphore_Proc_Stockage_Traitement_fichier_vide)
Tantque nbrSerie > nbrSerieTraitée faire
    Prise du semaphore_Proc_Stockage_Traitement_fichier_vide
    Prise du semaphore_Proc_Acquisition_Stockage_Mem_plein
    Prise du MUTEX_acquisition_stockage
    Prise du MUTEX_stockage_traitement
     Section Critique
          Réservation segment mémoire partagée
          Lecture mémoire alloué
          Tantque nbrAcquisition > nbrAcquisitionTraité faire
               Ecriture dans le fichier texte
          FinTantque
          On libère le segment mémoire partagée
     Fin Section Critique
    On libère le MUTEX_stockage_traitement
    On libère le MUTEX_acquisition_stockage
    On libère le semaphore_Proc_Acquisition_Stockage_Mem_vide
    On libère le semaphore_Proc_Stockage_Traitement_fichier_plein
FinTantque
Fin Fonction
```

Tableau 3 : Algorithme processus stockage

Organigramme

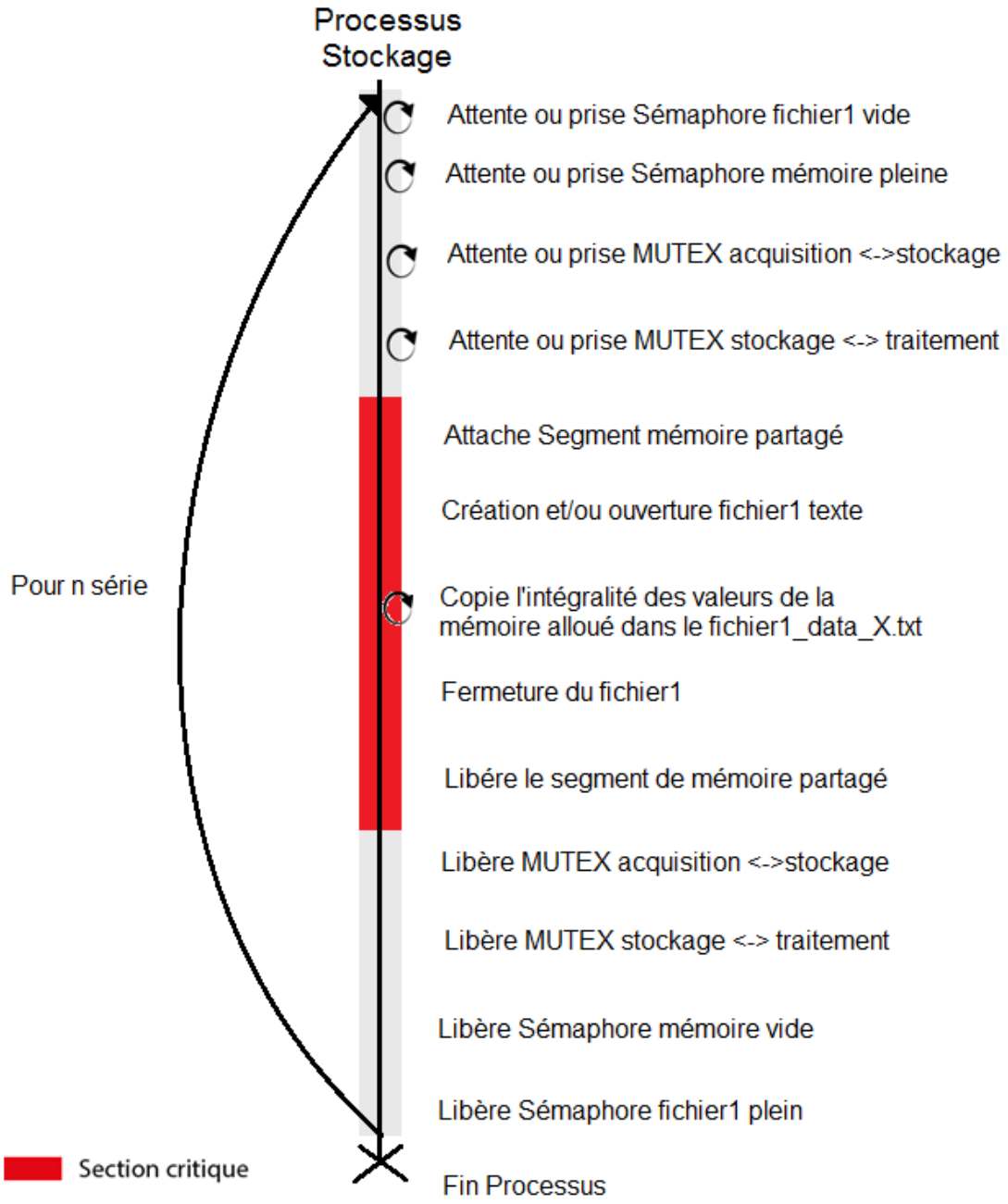

Figure 4 : Diagramme séquence processus stockage

Un programme traitement.c/h :

DESCRIPTION :

Cette fonction récupère les valeurs présentent dans le fichier data_1_X.txt. Puis traite les données du fichier et stocke le résultat dans un fichier data_2_X.txt. X étant le numéro de la série.

SPECIFICATION :

- La fonction traitement devra générée un fichier data_2_X.txt avec X correspondant au nombre de série.
- Dans le fichier data_1_X.txt on aura strictement une valeur par ligne.

Algorithme

```
Traitement(nbrSerie,sem_P,sem_V,
semaphore_Proc_Stockage_Traitement_fichier_plein,
semaphore_Proc_Stockage_Traitement_fichier_vide,
MUTEX_stockage_traitement)
Tantque nbrSerie > serieTraitée faire
        Attente ou prise semaphore_Proc_Stockage_Traitement_fichier_plein
        Prise MUTEX_stockage_traitement
             Section Critique
                    Pour Ligne dans fichier1 faire
                       Valeur <- Ligne
                       Si ligneSuivante dans fichier1 != vide alors
                          Valeur2 <- ligneSuivante
                          ValeurDiff <- Valeur2 - Valeur
                          Lignefichier2 <- ValeurDiff
                          Lignefichier2 <- LigneSuivanteFichier2
                       Sinon
                          Lignefichier2 <- Valeur
                       finSi
                    finPour
             Fin Section Critique
        Libère MUTEX_stockage_traitement
        Libère semaphore_Proc_Stockage_Traitement_fichier_vide
finTantque
Fin Traitement
```

Tableau 4 : Algorithme processus traitement

Organigramme

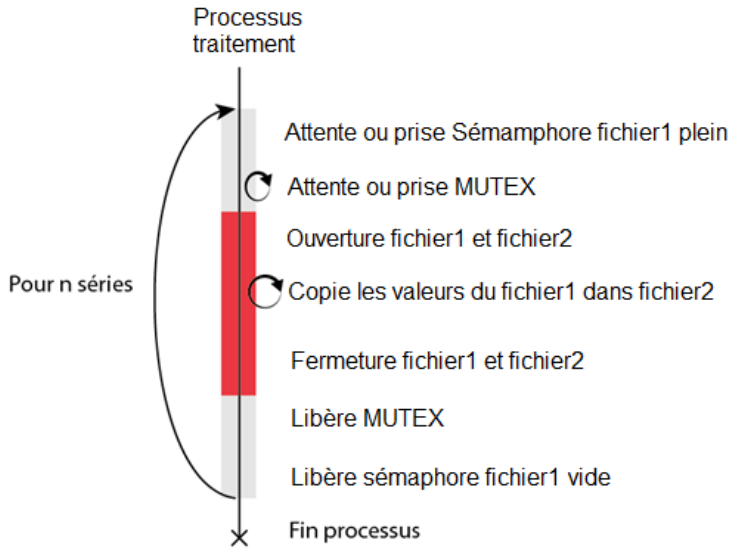

Figure 5 : Diagramme séquence processus traitement

Transition processus X et processus Y:

Les deux transitions présentes dans ce programme suivent le même type de transition. En effet, pour résoudre le problème de synchronisation entre nos trois processus nous avons suivis le schéma producteur consommateur. Celui-ci a l'avantage de synchroniser correctement nos deux processus. Ainsi pour pouvoir effectuer une transition entre deux processus (ici X et Y) nous avons besoin de 3 sémaphores. Les deux sémaphores vide et plein permettent la synchronisation, c'est à dire laisser la main au bon moment. Cependant le rôle MUTEX c'est de sécuriser la section critique. Ainsi nous avons fait un schéma permettant de comprendre la synchronisation avec trois sémaphores et deux processus.

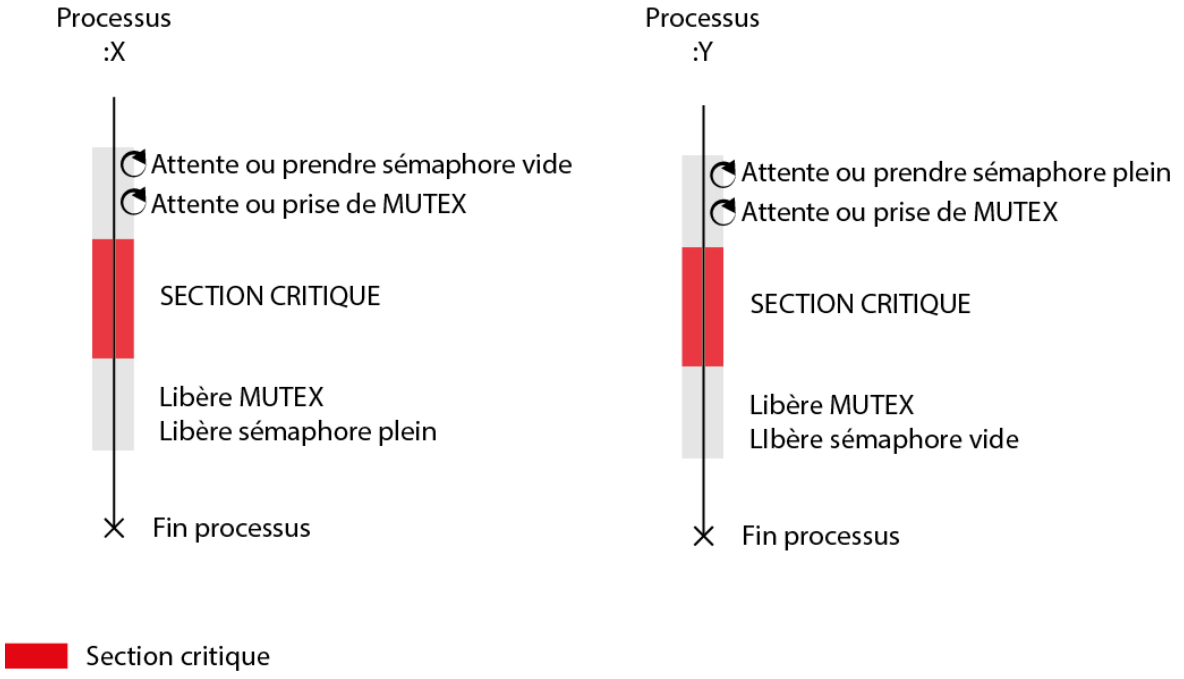

Figure 6 : Diagramme séquence de la transition entre les processus

Remarque : si le délai entre les séries est inférieur au temps de copie des données dans les fichiers alors nous aurons un problème de synchronisation. Cependant, dans notre cahier des charges le délai doit être en secondes. Donc ceci ce problème peut intervenir si la quantité d'acquisitions est importante.

Transition global entre les processus

Afin de comprendre la synchronisation entre tous les processus nous avons dessiné un diagramme de séquence entre chaque processus. De plus, nous avons mis par couleur les sémaphores. Ici on ne précise le rôle de la section critique car nous voulons montrer comment son synchronisé nos 3 processus.

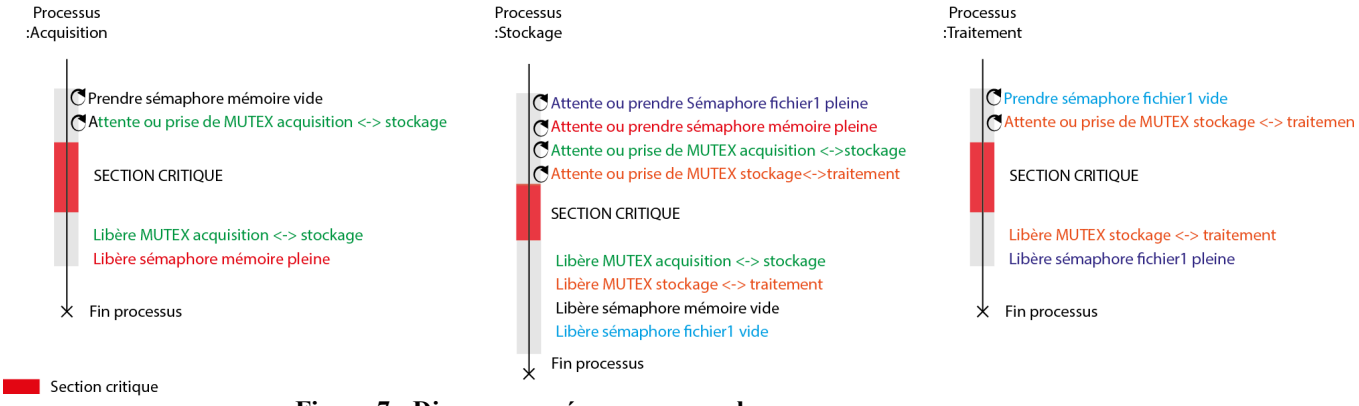

Figure 7 : Diagramme séquence pour chaque processus

Programme courbe.c/h

Fonction bonus. Cette fonction nécessite l'installation du logiciel Gnuplot (Et celui-ci est compatible avec le merveilleux système d'exploitation MAC OS ☺). Ce logiciel permet de tracer des courbes. Cette fonction s'intégrera dans la fonction traitement. Elle dessinera les courbes en fonction des valeurs de data_2_X.txt.

Nous avions plusieurs solutions pour exécuter GNUPLOT il y avait aussi execvp. Mais cependant nous n'avons pas trouvé la manière dont on peut envoyer chaque ligne. Nous avons ainsi pris connaissance d'une autre méthode qui se nomme popen. Celui-ci créé un processus fils (comme un fork) et invoque un shell. Ainsi nous pouvons envoyer nos commandes sans aucun souci.

Vous trouverez ci-contre l'algorithme pour résoudre le graphe.

A propos de l'appelle de cette fonction.

Nous appelons cette méthode dans traitement.c. En effet, lorsque nous avons terminé de traité tous les documents pour 'n' séries alors les deux précédents processus n'ont pu lieu d'être. Ainsi, lorsque nous avons finit notre traitement alors on peut appeler GNUPLOT pour tracer nos courbes avec chaque fichiers data_2_X.txt.

Algorithme

```
DessinerGraphe(nbrSerie)
    Tantque incrementalNumeroSerie < nbrSerie faire
       Récupérer tous les noms de fichiers de type data_2_X.txt
    finTantque

    Création d'un pipe pour GNUPLOT
    Enregistrer les commandes pour tracer le GRAPHE sur GNUPLOT
    Vider le tampon
    Attendre 10 secondes avant de terminer cette pipe
    Termine la pipe GNUPLOT

Fin DessinerGraphe
```

Communication avec les IPCs Système V

Dans cette partie nous allons nous s'intéresser au fonctionnement des IPCs en système avec le langage C. En effet, nous avons vu en cours comment fonctionner les IPCs mais nous n'avions pas mis en pratique ce système. Ainsi nous mettrons en parallèle nos connaissances sur le système V et le langage C.

Principes

- Les segments de mémoire partagée, qui sont accessibles simultanément par deux processus ou plus, avec éventuellement des restrictions telles que la lecture seule.

- Les sémaphores, qui permettent de synchroniser l'accès à des ressources partagées.

Dans tous les cas, ces outils de communication peuvent être partagés entre des processus n'ayant pas immédiatement d'ancêtre commun. Pour cela, les IPC introduisent le concept de clé.

Une ressource IPC partagée est accessible par l'intermédiaire d'un nombre entier servant d'identificateur (l'ensemble de sémaphores, identifiant du segment mémoire), qui est commun aux processus désirant l'utiliser.

Pour partager ce numéro d'identification, consiste à demander au système de créer lui-même une clé, fondée sur des références communes pour tous les processus. La clé est constituée en employant un nom de fichier et un identificateur de projet. De cette manière, tous les processus d'un ensemble donné pourront choisir de créer leur clé commune en utilisant le chemin d'accès du fichier exécutable de l'application principale.

Une clé est fournie par le système sous forme d'un objet de type key_t, défini dans<sys/type.h>.

Pour créer une nouvelle clé à partir d'un nom de fichier et d'un identificateur de projet, on emploie la fonction ftok( ). Déclarée ainsi dans <sys/ipc.h> :

```
key_tftok (char * nom_fichier, char projet);
```

La clé créée emploie une partie du numéro d'i-noeud du fichier indiqué, le numéro mineur du périphérique sur lequel il se trouve et la valeur transmise en second argument pour faire une clé sur 32 bits :

31...24	*23...16*	*15...0*
Numéro projet &OxFF	Mineur périphérique &OxFF	Numéro i-noeud&OxFFFF

La fonction ftok() ne garantit pas réellement l'unicité de la clé, car plusieurs liens matériels sur le même fichier renvoient le même numéro d'i-noeud. De plus, la restriction au numéro mineur de périphérique ainsi que l'utilisation seulement des 16 bits de poids faibles de 1'i-nœud rendent possible l'existence de fichiers différents renvoyant la même clé.

Ouverture de l'IPC

L'obtention de la ressource IPC se fait à l'aide de l'une des deux commandes shmget( ) et semget( ). Les détails d'appel seront précisés plus bas, mais ces fonctions demandent au système de créer éventuellement la ressource si elle n'existe pas, puis de renvoyer un numéro d'identification. Si la ressource existe déjà et si le processus appelant n'a pas les autorisations nécessaires pour y accéder, les routines échouent en renvoyant -1.

À partir de l'identifiant ainsi obtenu, il sera possible respectivement :

- D'attacher puis de détacher un segment de mémoire partagée dans l'espace d'adressage du processus avec shmat( ) ou shmdt( ) ;
- De lever de manière bloquante ou non un sémaphore, puis de le relâcher avec la fonction commune semop( ).

Contrôle et paramétrage

Les IPC proposent quelques options de paramétrage spécifiques au type de communication, ou générales. Pour cela, il existe deux fonctions, shmctl( ) et semctl( ), qui permettent de consulter des attributs regroupés dans des structures shmid_ds et semid_ds. Dans tous les cas, ces structures permettent l'accès à un objet de type struct ipcperm, défini ainsi dans <sys/ipc.h> :

Nom	Type	Signification
__key	key_t	Clé associée à la ressource IPC
__seq	unsigned short	Numéro de séquence, utilisé de manière interne par le système, à ne pas toucher
mode	unsigned short	Autorisations d'accès à la ressource, comme pour les permissions des fichiers
uid	uid_t	UID effectif de la ressource IPC
gid	gid_t	GID effectif de la ressource IPC
cuid	uid_t	UID du créateur de la ressource
cgid	gid_t	GID du créateur de la ressource

Les fonctions de contrôle permettent également de détruire une ressource IPC. En effet, un ensemble de sémaphores ou une zone de mémoire partagée restent présents dans le noyau même s'il n'y a plus de processus qui les utilisent. Ceci présente l'avantage d'une persistance des données entre deux lancements de la même application mais pose aussi

l'inconvénient d'une utilisation croissante de la mémoire du noyau sans libération automatique. Il est donc possible de demander explicitement la destruction d'une ressource IPC. Les processus en train de l'employer recevront une indication d'erreur lors de la tentative d'accès suivante.

Mémoire partagée

Le système de la mémoire partagée offert par les IPC Système V est le suivant:

- Une fonction shmget( ) permet à partir d'une clé key_t d'obtenir l'identifiant d'un segment de mémoire partagée existant ou d'en créer un au besoin.
- L'appel-système shmat( ) permet d'attacher le segment dans l'espace d'adressage du processus.
- La fonction shmdt( ) sert à détacher le segment si on ne l'utilise plus.
- Enfin, l'appel-système shmctl( ) permet de paramétrer ou de supprimer un segment partagé.

Les prototypes de ces routines sont déclarés dans <sys/shm.h> ainsi :

```
intshmget (key_tkey, inttaille, intattributs);
char * shmat (int identifiant, char * adresse, int attributs);
intshmdt (char * adresse);
intshmctl (int identifiant, intcommande,structshmid_ds * attributs);
```

L'appel-système shmget( ) fonctionne comme il suit, en employant la clé transmise en premier argument pour rechercher ou créer un bloc de mémoire partagée. Les attributs indiqués en dernière position comportent les 9 bits de poids faibles de l'autorisation d'accès, et éventuellement les constantes IPC_CREAT et IPC_EXCL.

Le second argument de cette routine est la taille du segment désiré, en octets. Cette taille sert lors de la création d'une nouvelle zone de mémoire partagée. La valeur indiquée est arrondie au multiple supérieur de la taille des pages mémoire sur le système (4 Ko sur un PC). Si la taille demandée lors de la création est inférieure à la valeur SHMMIN ou supérieure à SHMMAX, une erreur se produit. Pour accéder à une zone mémoire déjà existante, il faut demander une valeur inférieure ou égale à la taille effective du segment. On emploie généralement zéro dans ce cas, car le système ne réduit pas la taille de la projection d'un segment existant.

Une fois obtenu l'identifiant d'un segment partagé, on doit l'attacher dans l'espace mémoire du processus à l'aide de la fonction shmat(). On indique en second argument l'adresse désirée pour l'attachement. Si cette adresse est nulle, le noyau recherche un emplacement libre dans l'espace d'adressage du processus, y réalise la projection, et l'appel-système shmat() renvoie l'adresse du premier octet de la zone partagée. L'attachement peut

être réalisé en lecture seule si l'attribut SHM_RDONLY est passé en troisième argument de shmat(), sinon la projection est réalisée en lecture et écriture.

La fonction shmctl() permet, d'agir sur un segment partagé. La commande employée en seconde position peut être :

- IPC_STAT : pour remplir la structure shmidds que nous allons détailler ci-dessous.
- IPC_SET : pour modifier l'appartenance ou les autorisations d'accès au segment.
- IPC_RMID : pour supprimer le segment. Ce dernier est alors marqué comme «prêt pour la suppression », mais ne sera effectivement détruit qu'une fois qu'il aura été détaché par le dernier processus qui l'utilise. Cela signifie aussi que tant qu'un processus conserve le segment attaché, il est toujours possible de le lier à nouveau avec shmat(), même s'il a été marqué pour la destruction.
- SHM_LOCK : permet de verrouiller le segment en mémoire pour s'assurer qu'il ne sera pas envoyé sur le périphérique de swap. Cette opération réduisant la mémoire vive disponible pour les autres processus, elle est privilégiée et nécessite un UID nul ou la capacité CAP_IPC_LOCK.
- SHM_UNLOCK : permet symétriquement de déverrouiller une page de la mémoire, autorisant à nouveau son transfert en mémoire secondaire.

La structure shmidds contenant les paramètres associés au segment de mémoire partagée comprend notamment les membres suivants :

Nom	Type	Signification
shm_perm	structipc_perm	Autorisation d'accès au segment de mémoire
shm_segsz	size_t	Taille en octets du segment
shm_atime	time_t	Heure du dernier attachement
shm_dtie	time_t	Heure du dernier détachement
shm_ctime	time_t	Heure de la dernière modification des autorisations
shm_cpid	pid_t	PID du processus créateur du segment
shm_lpid	pid_t	PID du processus ayant réalisé la dernière intervention
shm_nattch	unsigned short	Nombre actuel d'attachements en mémoire

L'utilisation des segments de mémoire partagée est le mécanisme de communication entre processus le plus rapide, car il n'y a pas de copie des données transmises. On évite notamment le transfert des informations entre l'espace mémoire de l'utilisateur et l'espace mémoire du noyau, à la différence de msgsnd( ) sur les files de messages, ou même de write( )

lors de l'utilisation de sockets BSD. Ce procédé de communication est donc parfaitement adapté au partage de gros volumes de données entre processus distincts.

Il est indispensable, lors de l'accès à des ressources communes, de synchroniser les différents acteurs, pour éviter les interférences regrettables. Pour cela, on dispose d'un dernier mécanisme IPC servant à organiser l'utilisation des mémoires partagées : les sémaphores.

Sémaphores

Un sémaphore est dans sa forme la plus simple un drapeau qui peut être levé ou baissé. Il sert à contrôler l'accès à une ressource critique grâce à deux opérations :

- Avant l'accès, un processus attend que le drapeau soit levé, puis il le baisse.
- Après avoir utilisé la ressource protégée, le processus relève le drapeau, et le noyau réveille les autres processus bloqués dans l'opération précédente.

Le test qui intervient dans la première de ces opérations est atomiquement lié à la modification qui le suit. Ceci garantit qu'en aucun cas deux processus ne verront simultanément le drapeau baissé et se l'attribueront.

Les routines servant à manipuler les sémaphores sont semop( ), qui regroupe les opérations P_n( ) etV_n( ) et semctl( ) , qui permet entre autres de configurer ou de supprimer un ensemble de sémaphores.

Leurs prototypes sont déclarés dans <sys/sem.h> ainsi :

```
intsemget (key_tkey, int nombre, int attributs);
intsemop (int identifiant, structsembuf * operation, unsigned
nombre);
intsemctl (int identifiant, intnumero, int commande, union
semun attributs);
```

L'appel-système semget( ) fonctionne comme ses confrères msgget( ) et shmget( ),avec simplement en second argument le nombre de sémaphores dans l'ensemble. Cette valeur n'est prise en compte que lors de la création de la ressource, pas au moment de l'accès à un ensemble existant. Le troisième argument contiendra IPC_CREAT et les autorisations d'accès.

La routine semop( ) sert à la fois pour les opérations P_n( ) et V_n( ) sur de multiples sémaphores appartenant au jeu indiqué en premier argument. Chaque opération est décrite par une structure sembuf, définie dans <sys/sem.h> ainsi :

Nom	Type	Signification
sem_num	short int	Numéro du sémaphore concerné dans l'ensemble. La numérotation débute à zéro.

sem_op	short int	Valeur numérique correspondant à l'opération à réaliser
sem_flg	short int	Attributs pour l'opération.

L'opération effectuée est déterminée ainsi :

- Lorsque le champ sem_op d'une structure sembuf est strictement positif, le noyau incrémente le compteur interne associé au sémaphore de la valeur indiquée et réveilleles processus en attente.
  Quand sembuf.sem_op = n, avec $n > 0$, alors l'opération est $V_n()$.

- Lorsque le champ sem_op est strictement négatif, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit supérieur à sem_op, puis il décrémente le compteur de cette valeur avant de continuer l'exécution du processus.

  Quand sembuf.sem_op = n, avec $n < 0$, alors l'opération est $P_n( )$.

- Lorsque le champ sem_op est nul, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit nul, puis il continue l'exécution du programme. Cette fonctionnalité permet de synchroniser les processus.

Il existe deux options possibles pour le membre sem_flg :

- IPC_NOWAIT : l'opération ne sera pas bloquante, même si le champ sem_op est négatif ou nul, mais l'appel-système indiquera l'erreur EAGAIN dans errno si l'opération n'est pas réalisable.
- SEM_UNDO : pour être sûr que le sémaphore retrouvera un état correct même en cas d'arrêt intempestif du programme, le noyau va mémoriser l'opération inverse de celle qui a été réalisée et l'effectuera automatiquement à la fin du processus.

La routine semop( ) prend en second argument une table de structures sembuf. Le nombre d'éléments dans cette table est indiqué en dernière position. Le noyau garantit que les opérations seront atomiquement liées, ce qui signifie qu'elles seront toutes réalisées ou qu'aucune ne le sera. Il suffit qu'une seule opération avec sem_op négatif ou nul échoue avec l'attribut IPC_NOWAIT pour que toutes les modifications soient annulées.

Pour implémenter les fonctions P() et V() définies par Dijkstra, on peut donc employer un ensemble avec un seul sémaphore, qu'on manipulera ainsi :

```
P ()
{
     sem_num = 0;
     sem_op = -1;
     sem_flg= SEM_UNDO;
}
```

```
V ()
{
      sem_num = 0;
      sem_op = 1;
      sem_flg = SEM_UNDO;
}
```

L'option SEM_UNDO employée lors d'une opération permet au processus de s'assurer qu'encas de terminaison impromptue alors qu'il bloque un sémaphore le noyau en restituera l'état initial. Ceci est réalisé en utilisant un compteur par sémaphore et par processus quia demandé un accès à l'ensemble. Ce mécanisme est donc coûteux en mémoire. Le noyau modifie l'état de ce compteur à chaque opération sur le processus en y inscrivant l'opération inverse. Si par exemple le processus effectue une opération $P_n( )$, le noyau le bloque jusqu'à ce que le compteur du sémaphore soit supérieur à n, puis il diminue le compteur de cette valeur, et il augmente le compteur d'annulation du sémaphore pour ce processus de la valeur n. Lorsque le processus réalisera $V_n( )$ le noyau augmentera le compteur du sémaphore et réduira le compteur d'annulation.

Lorsque le processus se termine, le noyau ajoute le compteur d'annulation à celui du sémaphore. Si le processus a bien libéré le sémaphore, le compteur d'annulation est nul, et rien ne se passe. Si par contre le processus s'est terminé après avoir effectué $P_n( )$, mais sans avoir réalisé $V_n( )$, le compteur d'annulation vaut +n et le noyau libère ainsi automatiquement le sémaphore. Si le noyau doit décrémenter le compteur du sémaphore lors de la fin d'un processus. L'implémentation actuelle sous Linux consiste à diminuer immédiatement le compteur, mais à limiter ce dernier à zéro. D'autres systèmes peuvent préférer bloquer indéfiniment pour garantir l'annulation de n'importe quelle opération.

L'emploi des sémaphores doit autant que possible être restreint aux opérations P() et V() sur un seul sémaphore à la fois. On limiter également le compteur du sémaphore à la valeur 1. Si le processus risque de bloquer – ou d'être tué – durant la portion critique où il tient un sémaphore, on emploiera l'option SEM_UNDO. Bien sûr, si on utilise une fois cette option, on prendra la précaution de le faire à chaque opération sur le sémaphore.

La fonction semctl( ) permet de consulter ou de modifier le paramétrage d'un jeu de sémaphore, mais également de fixer l'état du compteur. Cette routine utilise traditionnellement en dernier argument une union définie ainsi :

```
unionsemun
{
      int valeur
      struct semid_ds * buffer
      unsigned short int * table
}
```

En fait, cette union n'est pas définie dans les fichiers d'en-tête système, elle doit être déclarée manuellement dans le programme utilisateur. En réalité, le prototype de semctl( ), vu par le compilateur, est en substance le suivant :

```
intsemctl (intidentifiant,intnumero, int commande, ...) ;
```

Pour garder une certaine homogénéité aux appels semctl( ), on préfère généralement regrouper les diverses possibilités dans une union, qui permet quand même une vérification minimale.

La structure semid_ds qui représente le paramétrage d'un jeu de sémaphore contient notamment les membres suivants :

Nom	Type	Signification
sem_perm	Structipc_perm	Autorisations d'accès à l'ensemble de sémaphores
semotime	time_t	Heure de la dernière opération semop( )
semctime	time_t	Heure de la dernière modification de sem_perm
sem_nsems	unsigned short	Nombre de sémaphores dans l'ensemble

Lorsqu'un ensemble de sémaphores est créé, les compteurs sont initialement vides. Aucun processus ne peut donc se les attribuer. Il faut donc leur donner une valeur initiale à l'aide de la commande SETVAL.