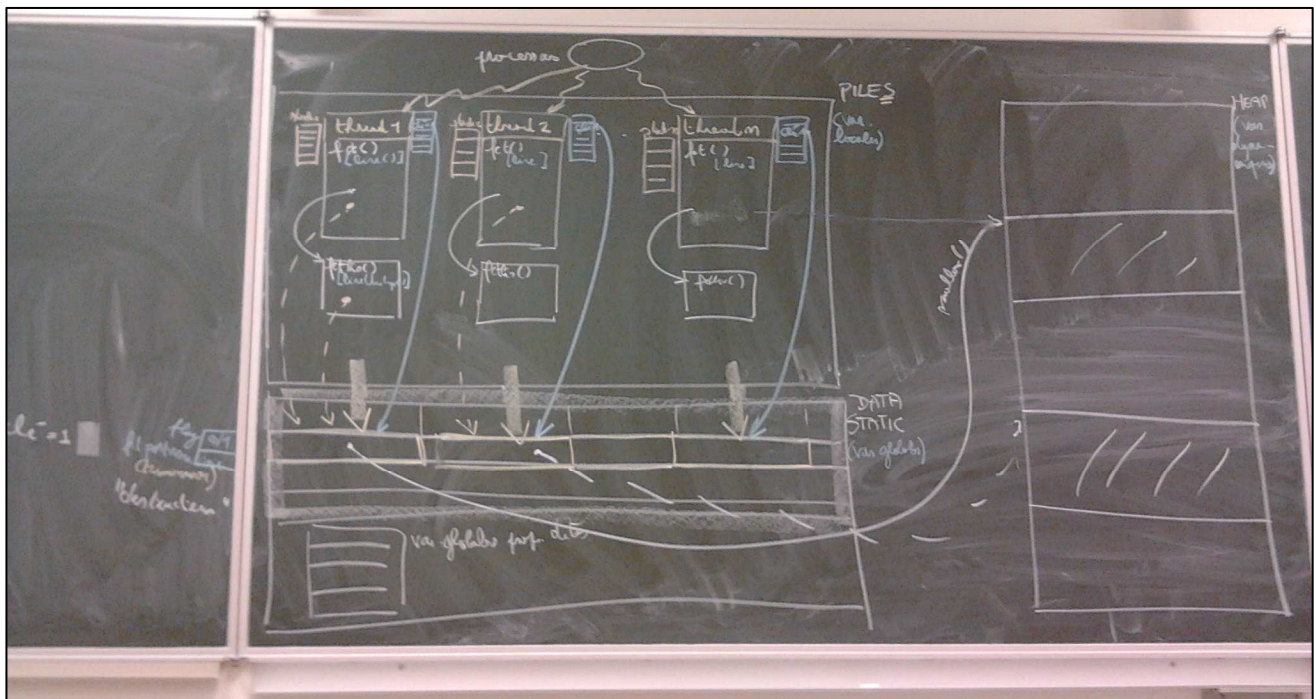
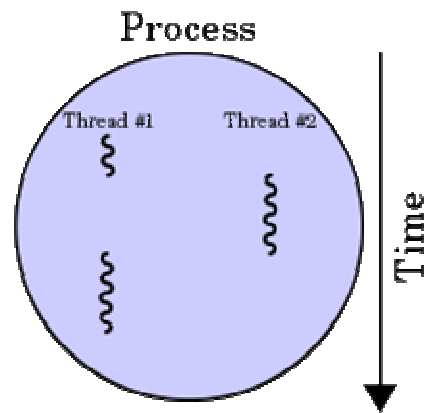


Les threads POSIX

UE: Développement orienté objets et multitâche
AA : Programmation de multitâche léger - Threads

Claude VILVENS

claude.vilvens@hepl.be



Sommaire

Introduction

I. Les fonctions de base des threads POSIX

1. Les sous-processus classiques	2
2. Le concept de processus léger	3
3. Quelques exemples typiques d'utilisation des threads	4
4. Différents modèles	5
5. Du code réentrant	6
6. La norme POSIX	6
7. Les 4 fonctions de base	
7.1 Un handle pour un thread	7
7.2 Créer un thread	8
7.3 Terminer et détruire un thread	9
7.4 Attendre la fin d'un thread	9
7.5 Attendre la fin d'un thread	10
8. Un exemple élémentaire de thread	
8.1 Une illustration des fonctions de base	10
8.2 En pratique : compilation et édition de liens	12

II. Quelques fonctions complémentaires

1. L'identité d'un thread	16
2. Les process status pour les threads	19
3. La mise en sommeil d'un thread	19
4. Le supermarché : version de base	22
5. Les fonctions de terminaison d'un thread	26
6. Le supermarché : passer à la caisse	27

III. L'asynchronisme dans les threads

1. Les threads et les signaux	30
2. Le supermarché : quand on ferme	34
3. L'arrêt programmé d'un thread	37
4. Le supermarché : il y a un voleur	39
5. Le supermarché : l'achat de quelques articles	45
6. Forcer un thread à libérer le processeur	50

IV. La synchronisation des threads

1. Un premier pas vers la synchronisation : l'attente de la fin d'un thread	54
2. Les politiques de scheduling et les priorités	61
3. Les mutex pour threads	64
4. Le supermarché : une section critique pour prendre un article	67
5. Différents types de mutex	77
6. Les variables de condition	
6.1 Le principe d'une variable de condition	78
6.2 Les fonctions d'opérations sur les variables de condition	79
6.3 Un programme utilisant une variable de condition	81
7. Le supermarché gère ses ruptures de stock	86

V. Des variables statiques spécifiques aux threads

1. Quelque part entre locales et globales	98
2. D'une fonction normale à une fonction de thread : compter les lignes	98
3. Des variables statiques mais pas tout à fait ...	103
4. Le principe de l'implémentation	104
5. Les primitives Posix pour variables spécifiques	
5.1 Obtenir une clé	105
5.2 Accéder aux données spécifiques par la clé	106
5.3 Retour au compteur de lignes	106
6. Les routines d'initialisation unique	111
7. L'allocation d'une zone mémoire spécifique	115

VI. Les modèles classiques

1. L'implémentation de divers modèles	119
2. Le modèle parallèle ou du travail en équipe	
2.1 Le principe du modèle	119
2.2 Une implémentation : les recherches parallèles dans des fichiers	119
2.3 Le schéma général du modèle parallèle	128
3. Le modèle du producteur et des consommateurs	
3.1 Le principe du modèle	129
3.2 Le schéma général du modèle producteur-consommateurs à la demande	129
3.3 Une implémentation : le traitement des noms de clients	131
3.4 Le schéma général du modèle producteur-consommateurs en pool de threads	138
4. Le modèle du pipeline	
4.1 Le principe du modèle	139
4.2 Une implémentation : le traitement des commandes de fabrication	139
4.3 Le schéma général du modèle pipeline	150

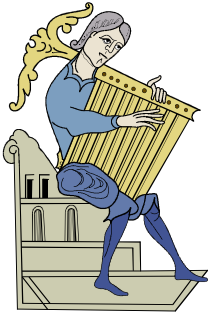
Annexe 1: Les symboles utilisés dans les schémas

Annexe 2: Les librairies sous UNIX

1. L'objectif : réutiliser	1
2. Les librairies statiques et les librairies dynamiques	1
3. Les conventions	2
4. L'exploration d'une librairie existante	3
5. Les librairies utilisées par un programme	4
6. La construction d'une librairie statique	5
7. L'utilisation d'une librairie statique	6
8. La construction d'une librairie dynamique	7
9. L'utilisation d'une librairie dynamique	9
10. L'utilisation d'une librairie dynamique (bis)	10

Ouvrages consultés

Introduction



On pourrait comparer la programmation multithreads avec une entreprise en pleine croissance.

Ainsi, au départ, le "patron" est seul : il mène à bien la tâche qu'il a entreprise, à moins qu'il ne la mette en attente ("sur la pile") pour en poursuivre une autre. Mais il se rend vite compte qu'il lui faut des collaborateurs - il en engage donc quelques-uns (`pthread_create`).

Au début, il leur confie une tâche et, après avoir un tour de l'entreprise, il attend qu'ils aient terminé (`pthread_join`). Après un certain temps, il leur donne plus d'autonomie (`pthread_detach`) et se contente de demander à être prévenu quand ils ont fini leur travail (`pthread_cond_wait`). Il arrive aussi que certains collaborateurs s'en aillent (`pthread_exit`) ou soient renvoyés (`pthread_cancel`).

Evidemment, les contingences de la vie de tous les jours ne doivent pas être oubliées : tout le monde ne peut pas accéder la machine à café en même temps (`pthread_mutex`), certains doivent réagir lorsque l'on sonne à la porte et d'autres pas (signaux et masques de signaux), certains traitent de sujets confidentiels dont l'accès leur est réservé (variables spécifiques), tous sont égaux mais certains le sont plus que d'autres (priorités et scheduling), etc.

Et puis, l'entreprise grandissant de plus en plus, il faut un gestionnaire des ressources humaines pour définir les responsabilités et les devoirs de chacun (modèles de threads).

Comme le montre cette métaphore, les threads (les Français disent "activités") sont donc les amis des programmeurs modernes, puisqu'ils leur permettent de répartir les tâches sur plusieurs unités de programmation exécutant leur code.

Bien sûr, les sous-processus classiques jouent déjà ce rôle. Cependant, les threads sont plus simples d'utilisation, notamment parce qu'ils communiquent simplement et qu'ils sont moins gourmands en ressources machine. En revenant à l'analogie de l'entreprise, les threads sont les collaborateurs locaux, tandis que les sous-processus sont les correspondants éloignés.

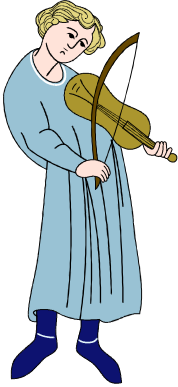
On trouve l'usage des threads dans de nombreux contextes, comme la programmation Windows ou Java. Pour ce qui nous concerne, nous ferons leur connaissance avec le langage C, qui a le mérite de nous faire mieux ressentir "ce qui se passe à l'intérieur".

Avec toutes les précautions d'usage en programmation système, je vous invite donc à me suivre dans cet univers impitoyable (celui qui crée des threads dans une boucle, au risque de "planter" la machine, est en général massacré par ses congénères !) mais si séduisant ...

Claude Vilvens

*Merci à mes estimés collègues Jean-Marc Wagner et Denys Mercenier
pour leurs remarques constructives !*

I. Les fonctions de base des threads POSIX



Les idées de la veille font les mœurs de demain..

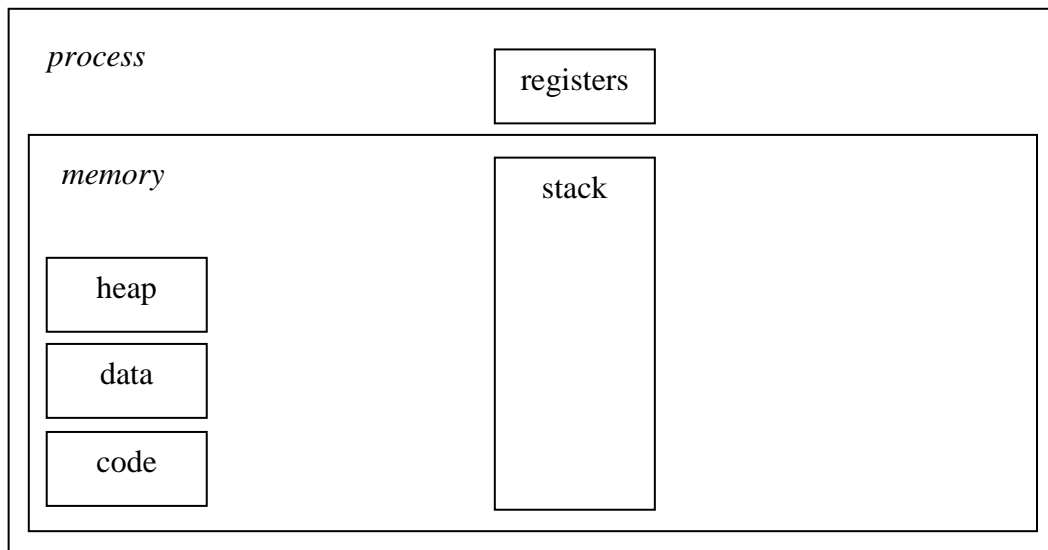
(A. France, Discours au banquet des étudiants)

1. Les sous-processus classiques

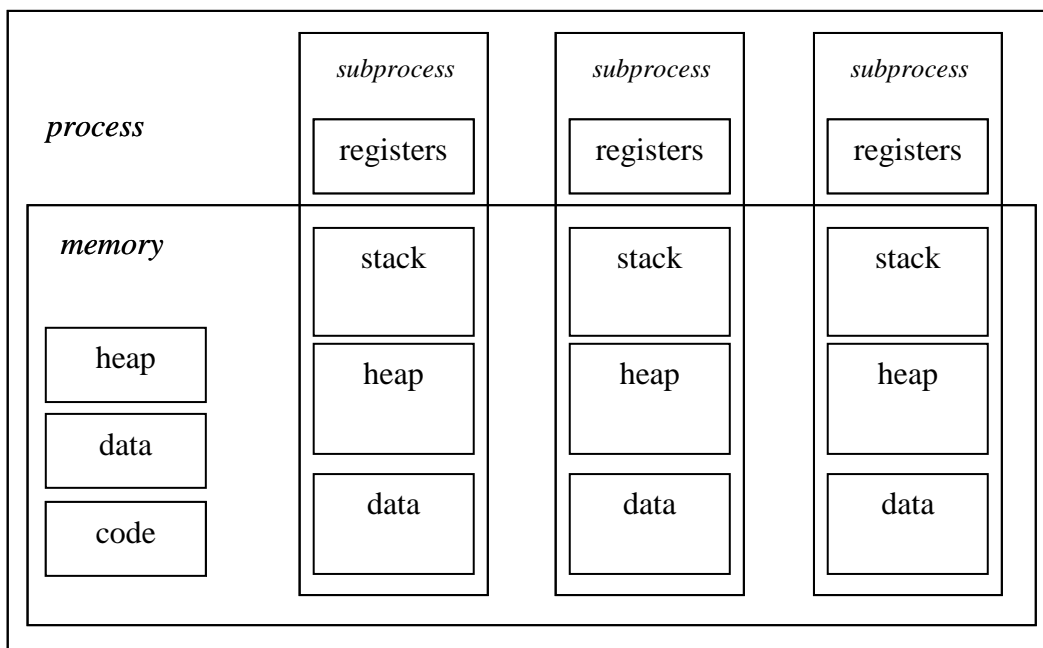
Un processus classique (dans l'acceptation des systèmes d'exploitations classiques) peut se caractériser, du point de vue de son exécution, par :

- ◆ ses valeurs des registres du processeur;
- ◆ son segment de données statiques;
- ◆ son segment de code;
- ◆ sa pile (stack);
- ◆ sa zone d'allocation dynamique (heap).

Schématiquement :



En particulier, un processus peut créer un ou plusieurs sous-process (ou processus fils). Le segment de données globales d'un processus-fils est une copie de celui de son père. Ce processus de duplication est coûteux pour le système d'exploitation.



2. Le concept de processus léger

L'idée de base est de **séparer** les ressources d'un processus des actions qu'il implémente sur celles-ci, permettant ainsi de lancer **en parallèle** plusieurs de ces actions. Il s'agit donc de distinguer

- ◆ le **processus** proprement dit, qui est **l'unité d'encapsulation des ressources utilisées**;
- ◆ les **activités** ou *threads of control* qui sont **les unités d'exécution sur cet ensemble**.
Puisqu'elles ne contiennent pas les données sur lesquelles elles agissent, ces unités d'exécutions sont appelées des "processus légers" (*lightweight process*).

Un processus comporte au minimum un thread, dit **thread principal** ou **thread initial** ou **thread 0**. Il lui est donc loisible de lancer d'autres threads. A l'image des processus, les threads peuvent se trouver dans l'un des 4 états suivants :

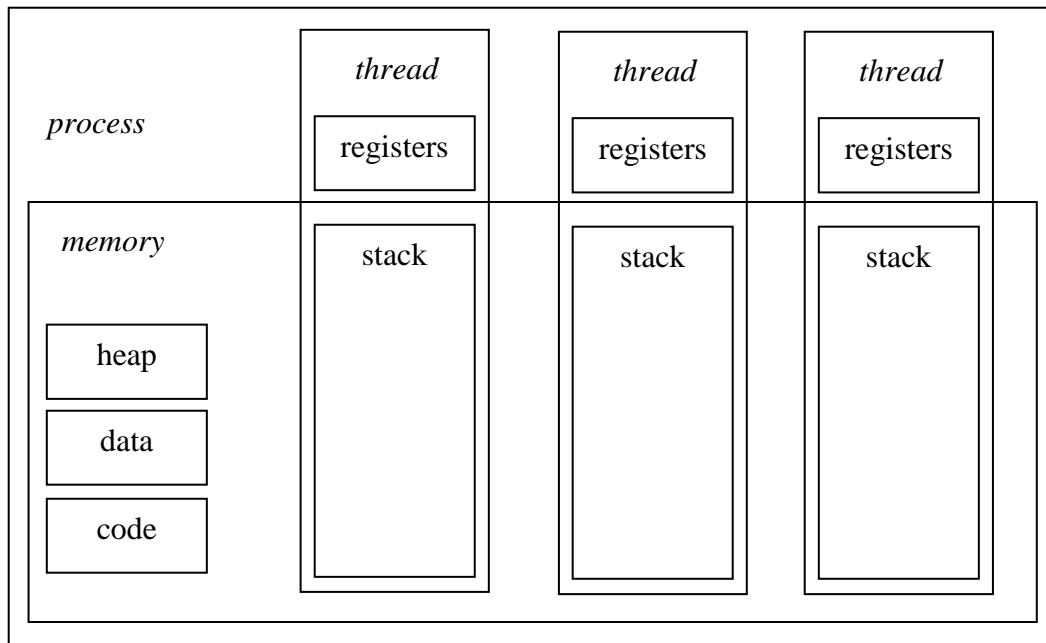
- ◆ actif ou en cours d'exécution [*running*] : il s'exécute en utilisant un processeur (ou le processeur si celui-ci est unique);
- ◆ prêt [*runnable* ou *ready*] : le thread attend de disposer d'un processeur pour s'exécuter;
- ◆ en attente [*waiting*] : le thread attend un événement ou une ressource autre qu'un processeur;
- ◆ terminé [*terminated*] : le thread a complètement exécuté sa tâche.

Tous les threads **ont en commun** :

- ◆ les données globales;
- ◆ le répertoire de travail;
- ◆ les propriétaires et groupes;
- ◆ les descripteurs de fichiers ouverts;
- ◆ l'émission des signaux et leur traitement au moyen d'un handler.

Chaque thread **possède en propre** :

- ◆ ses valeurs de registres;
- ◆ sa pile, donc les données locales (automatiques);
- ◆ sa prise en compte par le scheduler;
- ◆ une variable `errno`;
- ◆ son propre masque de signal.



Historiquement, il revient au monde UNIX la paternité de cette notion de thread.

On mesure donc la différence entre un thread et un sous-processus classique : le segment de données globales d'un processus-fils est une copie de celui de son père, alors que le thread partage ce segment avec son géniteur. Ce processus de duplication est d'ailleurs coûteux pour le système d'exploitation – beaucoup plus que la mise en place d'un thread, ce qui justifie encore une fois le terme de *lightweight process* utilisé pour désigner les threads. De ce fait, un processus et un sous-processus sont totalement indépendants, alors qu'un thread fils dépend de son père. Ainsi, la terminaison du thread initial du processus entraîne celle de tous les autres threads fils.

3. Quelques exemples typiques d'utilisation des threads

L'informatique moderne fait une large place à ces threads, comme le montrent les exemples suivants.

1) On sait qu'un **tableur** doit recalculer la feuille de calcul active à chaque fois qu'une cellule est modifiée. Si la feuille est complexe, cela peut prendre un temps important, préjudiciable à la souplesse d'utilisation du tableur – au point que l'on souhaiterait parfois inhiber la fonction de recalcul automatique ! Une solution permettant d'alléger le travail de l'utilisateur consiste à structurer l'application tableur en deux threads :

- ◆ un thread principal, qui gère le dialogue direct avec l'utilisateur, par exemple les saisies;
- ◆ un thread secondaire, de priorité moindre, qui s'occupe de recalculer le contenu des cellules modifiées.

Comme le thread de recalcul est moins prioritaire, il est préempté dès que l'utilisateur effectue une saisie (thread principal actif, thread secondaire arrêté) alors qu'il peut s'exprimer lorsque l'utilisateur n'encode rien (thread secondaire actif, thread principal arrêté).

2) D'une manière similaire, un **traitement de texte** doit gérer la repagination tout en permettant à l'utilisateur d'encoder son texte. Un thread secondaire de repagination jouera un rôle similaire au thread de recalcul de l'exemple du tableur.

3) Les **application graphiques** utilisant plusieurs animations simultanées font grande utilisation des threads. En fait, sans eux, elles ne fonctionneraient pas de manière satisfaisante ! On aura compris que chaque animation est prise en charge par un thread distinct (parfois même plusieurs); les divers threads ont cette fois la même priorité, sauf si l'on privilégie l'une des opérations.

4) Une tâche d'**impression** peut être confiée à un thread qui gérera cette opération de sortie réputée pour sa lenteur pendant que le thread principal, plus prioritaire, permettra de conserver l'usage de l'application.

5) La notion de thread est très utile dans la construction d'un **serveur réseau** multi-clients. En effet, il suffira de *créer un thread pour chaque requête d'un client*, ce qui permettra l'exécution en parallèle des traitements de ces requêtes. On conçoit sans peine, comme pour les sous-processus, qu'un tel mécanisme est plus efficace que la méthode classique qui consiste à placer les requêtes dans une file d'attente et à les traiter l'une après l'autre.

6) L'intérêt des threads est évident dans le cas de l'utilisation des **systèmes multiprocesseurs** (*of course* !).

4. Différents modèles

Les threads peuvent accomplir leur tâche selon différentes modèles :

- ◆ le modèle parallèle ou du travail en équipe (*work crew model* or *parallel model*): après une phase d'initialisation, les différents threads réalisent une sous-tâche de la tâche commune; la nécessité de méthodes de synchronisation est évidente;
- ◆ le modèle pipeline (*pipeline model*) : les threads forment une chaîne et chaque thread démarre lorsque le précédent a terminé, les résultats en sortie de l'un constituant les données en entrée de l'autre;
- ◆ le modèle producteur-consommateur ou maître-esclave (*producer-consumer model* or *boss-worker model*) : un thread assigne des tâches à d'autres threads, qui ignorent ce que font leurs collègues.

C'est en général à ce dernier modèle que l'on pense lorsque l'on parle de serveur "**multi-threads**".

5. Du code réentrant

Il convient de remarquer que, dans le cas du modèle multi-threads parallèle ou dans celui du producteur-consommateurs, plusieurs threads peuvent exécuter le même code simultanément. Un environnement de développement multithreads doit fournir du code qui supporte une telle exécution simultanée avec des points d'avancements différents : on parle encore dans ce cas de **code réentrant**. Ceci signifie notamment que certaines fonctions des bibliothèques standards du C doivent être remplacées par leur version réentrante.

Pour le reste, un thread se comporte de manière assez semblable à un sous-processus. Ainsi, il peut se trouver en différents états (en attente, prêt à être exécuté, en cours d'exécution, terminé). Bien sûr, on rencontrera aussi les problèmes bien connus de la synchronisation des diverses actions et des accès concurrents. Nous en reparlerons, car le mécanisme des threads propose des solutions spécifiques à ces problèmes.

6. La norme POSIX

POSIX (Portable Operating System Interface) est une famille de standards développés par IEEE (Institute for Electrical and Electronic Engineers) et adoptés par l'ISO (International Organization for Standardization). Le premier standard POSIX est apparu en 1988 (POSIX 1003.1) : il spécifiait les interfaces C pour un système UNIX.

Le mécanisme des threads, évidemment très bien adapté aux problèmes de temps réels et à l'utilisation de machines multiprocesseurs, trouve sa norme POSIX d'interface dans POSIX 1003.4a¹, si bien que l'on parle des *Posix threads* ou *P-threads*. L'objectif est de l'implémenter sur les systèmes ouverts. Les fonctions APIs correspondantes sont typiquement "unixiennes" : 0 ou -1 comme valeur de retour, code d'erreur dans la variable globale `errno`.

En particulier, les threads POSIX sont implémentés les machines Unix (essentiellement, au sein du Département informatique, les machines SunRay – il sera à l'occasion encore fait référence aux anciennes machines Copernic et Boole). Le standard respecté actuellement par ces machines est IEEE POSIX 1003.1c-1995 ou POSIX.1c. *Elle n'utilise pas la variable globale `errno`*; les valeurs qu'aurait du prendre cette variable sont en fait les **valeurs de retour** de la fonction correspondante. Cette façon de procéder s'explique facilement : `errno` est en effet, en principe, une variable globale, donc partagée par les différents threads et elle ne saurait donc être positionnée valablement par des threads fonctionnant en parallèle ! En réalité, afin que chaque thread dispose de sa propre variable `errno`, on a imaginé bien logiquement de la placer sur la pile de chaque thread, ce qui en assure la duplication souhaitée. Pour qu'il en soit ainsi, on trouve dans `errno.h`:

la définition d'`errno` pour les threads

```
...
extern int *_errno __((void));
extern int _Geterrno __((void));
extern int _Seterrno __((int));

#ifdef _REENTRANT
#define errno (*_errno())
#else
extern int errno;
#endif /* _REENTRANT */ ...
```

¹ les standards POSIX sont constamment mis à jour : on peut consulter <http://www.pasc.org/standing/sd11.html> (PASC pour Portable Applications Standards Committee).

Dans la version réentrante, chaque thread possède donc bien *une valeur locale de errno*, puisque donnée par la valeur de retour d'une fonction et pas par une variable externe !

Le format général des *fonctions de threads POSIX* est :

pthread[_<objet auquel la fonction s'applique>]_opération[_**np**]

le suffixe np particularisant les fonctions non portables. Les *noms de type* utilisés par la norme sont de la forme :

pthread[_<objet dont on spécifie le type>]_t

Remarques

- 1) On peut obtenir de l'aide sur les threads par (sous UNIX) **man pthread** et aussi (pour la programmation sous Sun Solaris) sur <http://docs.sun.com/app/docs/coll/40.10>.
- 2) Un thread peut posséder des données statiques qui lui sont propres: il existe en effet un ensemble de données statiques réservé aux threads; ces données sont réparties à la demande explicite entre les divers threads d'un processus. Nous en reparlerons.

7. Les 4 fonctions de base

7.1 Un handle pour un thread

Un thread est désigné par le système d'exploitation au moyen d'un *tid* (le cousin du *pid*). Dans la programmation C, un thread est repéré par une espèce de handle de type **pthread_t**. Ce type pthread_t peut être vu comme un simple int sur 64 bits ou un pointeur de structure (dont le détail ne nous intéresse pas trop ici parce que sa gestion est encapsulée dans les primitives POSIX). Digital Unix prend les définitions suivantes (dans le header **pthread.h**) selon l'environnement de travail :

le type pthread_t (pthread.h)

```
typedef struct __pthreadTeb_t
{
    ...
    __pthreadLongUInt_t    sequence;    /* Thread sequence number */
    ...
    __pthreadLongAddr_t    per_kt_area; /* Pointer to kernel context */
    __pthreadLongAddr_t    stack_base;  /* Current stack base */
    __pthreadLongUInt_t    stack_size;  /* Size of stack */
    ...
} pthreadTeb_t, *pthreadTeb_p;

# if defined (_PTHREAD_ALLOW_MIXED_PROTOS_) &&
    defined (__INITIAL_POINTER_SIZE)
typedef pthreadTeb_p    pthread_t;    /* Long pointer if possible */
# pragma __required_pointer_size __restore
# elif defined (_PTHREAD_ENV_ALPHA) && defined (_PTHREAD_ENV_VMS)
typedef uint64          pthread_t;    /* Force 64 bits anyway */
# else
typedef pthreadTeb_p pthread_t;    /* Pointers is pointers */
# endif
#endif
```

Les fonctions indispensables sont celles qui permettent de

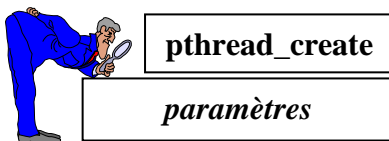
- ◆ créer un thread en lui indiquant la fonction qu'il doit exécuter (avec éventuellement son paramètre) [**pthread_create**];
- ◆ terminer un thread en terminant la fonction qu'il exécute par un appel de **pthread_exit** qui permet de renvoyer une valeur; ceci ne fait pas disparaître le thread !
- ◆ attendre qu'un thread se termine en récupérant sa valeur de retour [**pthread_join**];
- ◆ détruire définitivement un thread et ainsi récupérer ses ressources [**pthread_detach**]; le thread en cours d'exécution n'est pas arrêté – il ne sera détruit que lorsqu'il aura terminé sa tâche; on peut donc dire que cette fonction "marque pour la destruction" plutôt qu'elle ne détruit réellement – en fait, elle "détache" le thread, c'est-à-dire qu'elle le rend *indépendant et laisse sa terminaison libérer ses ressources automatiquement*.

7.2 Créer un thread

Cette primitive fondamentale se décline selon :

```
int pthread_create ( <handle du thread - pthread_t *>,
                    <attributs du thread - const pthread_attr_t *>,
                    <fonction exécutée par le thread – void * (*) (void *)>,
                    <paramètre de cette fonction – void *>);
```

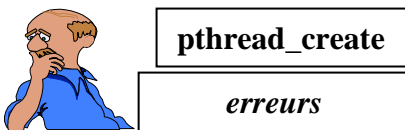
Cette fonction a donc pour tâche de fournir, en cas de succès, le handle attribué au thread créé.



Le deuxième paramètre est du type **pthread_attr_t** : il s'agit d'une structure permettant de fixer les caractéristiques du thread (priorité, taille initiale de la pile, le fait d'être un démon ou pas, etc). Selon la stricte norme POSIX, les attributs par défaut sont définis par la constante `pthread_attr_default`; pour les machines Sun et Compaq, NULL remplace cette constante et a le même effet.

Les deux derniers paramètres désignent respectivement la fonction à exécuter par le thread et le paramètre que l'on peut passer à cette fonction. On remarquera les pointeurs `void*` qui n'attendent que les castings permettant d'agir selon la volonté du programmeur ... A priori, la fonction exécutée par le thread a donc pour prototype :

```
void * f (void *)
```



valeur de retour		erreur
0		succès
#define EAGAIN	35	/* Operation would block */ : le nombre maximum de threads est atteint

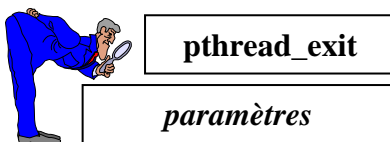
#define EINVAL	22	/* Invalid argument */ : le paramètre des attributs n'est pas valide
#define ENOMEM	12	/* Not enough core */ : devinez – ben oui : l'espace mémoire est insuffisant
#define EPERM	1	/* Not owner */ : "pas propriétaire" pour UNIX, "pas la permission" pour VMS ;-) ...

7.3 Terminer et détruire un thread

La fonction exécutée par le thread peut se terminer de manière "normale" : le retour de la fonction marque la fin du thread par la même occasion.

Sans précaution particulière, la fin du thread père marque également la fin de ses fils. Mais on peut aussi le terminer par :

```
void pthread_exit (<valeur de retour - void *>);
```



Ceci permet au thread qui se termine de fournir une valeur de retour, passée en paramètre et éventuellement testable par le thread père (voir pthread_join). On remarquera que c'est l'adresse de la valeur de retour qui est renvoyée, et non cette valeur seulement. Evidemment, s'il s'agit du thread principal, on a l'équivalent de exit(), si ce n'est que les threads fils peuvent continuer.

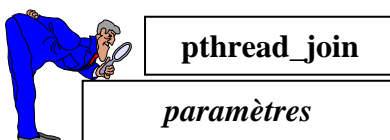
Nous verrons qu'il est aussi possible d'interrompre un thread depuis l'extérieur de celui-ci (pthread_cancel).

7.4 Attendre la fin d'un thread

Un thread qui a lancé un thread peut attendre la fin de celui-ci par :

```
int pthread_join    (<handle du thread - pthread_t>,
                    <adresse de la valeur retournée par le thread attendu – void *>);
```

Le thread qui appelle cette fonction suspend donc son exécution jusqu'à ce que le thread visé se soit terminé par pthread_exit². Le thread dont on attend la terminaison est encore appelé un "joinable thread" : son identificateur et sa valeur de retour sont conservées jusqu'à ce qu'un appel de pthread_join le concernant ait lieu.



Cette fonction récupère donc la valeur renvoyée par le thread en la plaçant dans une zone mémoire référencée par une double indirection. En pratique, cela signifie simplement que l'on peut renvoyer une donnée quelconque (void * en C).

² à moins qu'il ait été victime d'un pthread_cancel

**pthread_join****erreurs**

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre handle de thread n'est pas valide
#define ESRCH 3	/* No such process */ : le thread référencé n'existe pas (ou plus)
#define EDEADLK 11	/* Operation would cause deadlock */ : autrement dit, deux threads s'attendent l'un l'autre

Si plusieurs threads tentent de "joindre" le même thread, le résultat est imprévisible – donc, à éviter.

7.5 Détacher un thread

Il est possible de créer un **thread détaché** (detached thread) : lorsqu'un tel thread se termine, toutes ses ressources sont libérées (y compris sa valeur de retour) et un pthread_join le concernant n'est plus possible. Il faut bien remarquer que le détachement ne termine pas le thread, mais indique que l'on pourra récupérer ses ressources lorsqu'il sera détruit. La propriété de thread détaché est donnée à un thread par la fonction :

```
int pthread_detach (<handle du thread - thread_t>);
```

**pthread_detach****erreurs**

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre handle de thread n'est pas valide
#define ESRCH 3	/* No such process */ : le thread référencé n'existe pas (ou plus)

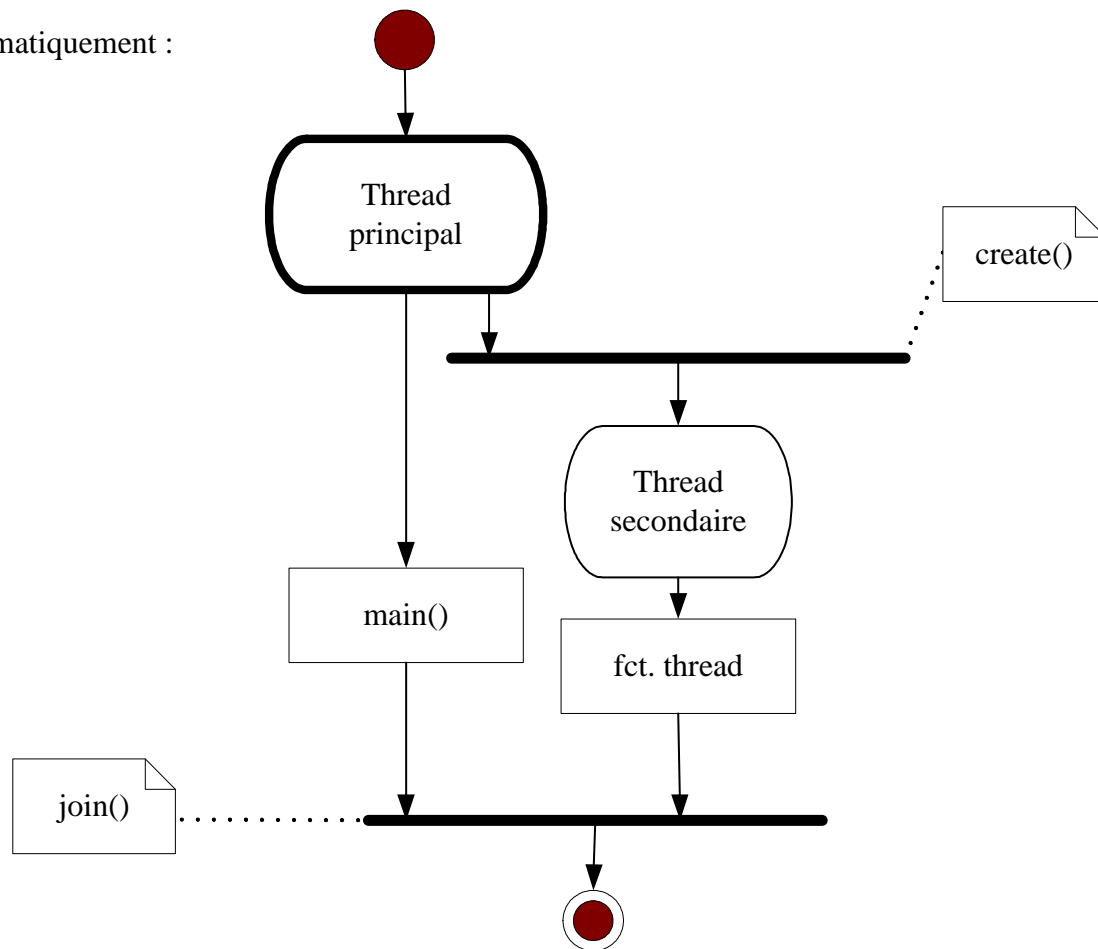
Une autre façon de dire la même chose est de considérer que *cette fonction demande que le thread soit détruit une fois son action terminée*. A remarquer que la primitive pthread_join détache également le thread une fois la synchronisation terminée.

8. Un exemple élémentaire de thread

8.1 Une illustration des fonctions de base

Afin de bien maîtriser ces quelques fonctions de base, voici pour le lecteur averse (de science) un exemple simple : le processus crée un thread qui exécute une fonction dont le seul mérite est de changer la valeur de retour par défaut (encore que ce ne soit pas une bonne idée de placer des entrées dans un thread puisque, si ils sont plusieurs, on ne sait pas quel thread est en cours d'exécution ☹). Le thread principal attend que son thread secondaire se termine avant de récupérer les ressources concédées.

Schématiquement :

**THREAD01.C**

```

/* THREAD01.C
Claude Vilvens
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *fctThread(int * param);

pthread_t threadHandle;

int main()
{
    int ret, *retThread;
    int paramEff = 5;

    puts("Thread principal démarre");
    ret = pthread_create(&threadHandle, NULL, (void (*)(void *))fctThread, &paramEff);
    puts("Thread secondaire lance !");
    puts("Attente de la fin du thread secondaire");
    ret = pthread_join(threadHandle, (void **)&retThread);
    printf("Valeur renvoyee par le thread secondaire = %d\n", *retThread);
    puts("Fin du thread principal");
}

```



```

void * fctThread (int * param)
{
    static int vr = 1007;

    puts("th> Debut de thread");
    printf("th> Parametre reçu = %d\n", *param);
    printf("th> Valeur renvoyee par default = %d\n", vr);
    printf("th> Nouvelle valeur : ");scanf("%d",&vr);
    printf("th> Valeur renvoyee = %d\n", vr);

    pthread_exit(&vr);
    return 0;
}

```

On remarquera les castings pour éviter les warnings donnés par certains compilateurs C, comme le compilateur C de Digital et le compilateur C++ de Digital.

8.2 En pratique : compilation et édition de liens

Le programme ci-dessus produira un exécutable si par l'habituel appel de cc si

- ◆ on fixe le niveau des librairies linkées au niveau réentrant (un must si on utilise des threads) [-mt];
- ◆ on force le link avec la librairie libpthread.so [-lpthread] :

```
sunray2v440.inpres.epl.prov-liege.be> cc -o thr thread01.c -mt -lpthread
```

Un résultat de l'exécution du programme donnera :

```

sunray2v440.inpres.epl.prov-liege.be> thr
Thread principal démarre
Thread secondaire lance !
Attente de la fin du thread secondaire
th> Debut de thread
th> Parametre reçu = 5
th> Valeur renvoyee par default = 1007
th> Nouvelle valeur : 654
th> Valeur renvoyee = 654
Valeur renvoyee par le thread secondaire = 654
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>

```

En ce qui concerne C++, l'appel de **CC** (cxx sur certaines machines UNIX) remplace, comme d'habitude, celui de **cc**, avec les mêmes options de compilation et de link.

Remarques

1) Si on détache le thread sans attendre la fin de celui-ci :

THREAD01-D.C

```
/* THREAD01-D.C
Claude Vilvens
*/
...

int main()
{
    int ret, paramEff = 5;

    puts("Thread principal démarre");
    ret = pthread_create(&threadHandle, NULL, (void*)(*)(void*)) fctThread, &paramEff);
    puts("Thread secondaire lance !");
    puts("Detachement du thread");
    ret = pthread_detach(threadHandle);
    puts("Fin du thread principal");
}

void * fctThread (int * param)
{
    static int vr = 1007;

    puts("th> Debut de thread");
    ...
    pthread_exit(&vr);
    return 0;
}
```

on obtient le résultat prévisible :

```
sunray2v440.inpres.epl.prov-liege.be> c
Thread principal démarre
Thread secondaire lance !
Detachement du thread
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

Autrement dit, le thread n'a pas le temps de parler, puisque son père s'est terminé !

2) La valeur de retour peut être récupérée dans une variable locale au lieu d'utiliser un pointeur :

THREAD01-R.C

```
/* THREAD01-R.C
Claude Vilvens
*/
...

int main()
{
    int ret, retThread;
    ...
    ret = pthread_create(&threadHandle, NULL, (void (*)(void*)) fctThread, &paramEff);
    ...
    ret = pthread_join(threadHandle, (void **)&retThread);
    ...
}

void * fctThread (int * param) {...}
```

3) On peut remarquer que, pour bien faire, il faut éviter de renvoyer l'adresse d'une variable locale. On peut donc penser à définir, dans la fonction exécutée par le thread, la variable de retour comme statique : c'est ce qui est fait dans l'exemple ci-dessus. On peut aussi penser à placer cette valeur de retour sur le heap. On peut ainsi, sans problème, imaginer ceci :

THREAD01-VR.C

```
/* THREAD01-VR.C
Claude Vilvens
*/
...

int main()
{
    int *retThread;
    ...
    ret = pthread_create(&threadHandle, NULL, (void (*)(void*)) fctThread, &paramEff);
    puts("Thread secondaire lance !");
    puts("Attente de la fin du thread secondaire");
    ret = pthread_join(threadHandle, (void **)&retThread);
    printf("Valeur renvoyee par le thread secondaire = %d\n", *retThread);
    free(retThread);
    puts("Fin du thread principal");
}

void * fctThread (int * param)
{
    int * avr = (int *)malloc(sizeof(int));
    *avr = 1007;

    puts("th> Debut de thread");
    printf("th> Parametre reçu = %d\n", *param);
    printf("th> Valeur renvoyee par default = %d\n", *avr);
```

```
printf("th> Nouvelle valeur : ");scanf("%d",&avr);  
printf("th> Valeur renvoyee = %d\n", *avr);  
  
pthread_exit(avr);  
return 0;  
}
```

que le handle de thread soit global ou local à main().

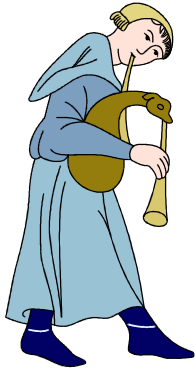
Evidemment, une manière simple d'occulter le problème est de renvoyer comme valeur le paramètre de la fonction du thread.



Les éléments de base concernant les threads étant acquis, plusieurs questions attaquent nos esprits surchauffés : comment "voir" les threads ? comment les "temporiser" ? et les signaux ?

Effectivement, voyons cela ...

II. Quelques fonctions complémentaires



Je suis un mensonge qui dit toujours la vérité.

(J. Cocteau, Opéra)

1. L'identité d'un thread

Il peut être utile, surtout en phase de mise au point, de savoir quel thread est en activité. Un thread s'identifie par :

- ♦ le pid de son processus père;
- ♦ son *tid* (Thread IDentifier);

soit par un label du type **pid.tid**. Ces deux éléments d'identification peuvent s'obtenir en utilisant les fonctions :

```
pid_t getpid (void);  
pthread_t pthread_self (void);
```

La première est une vieille connaissance (pour rappel, on trouve dans types.h un typedef int pid_t et le header unistd.h est nécessaire pour le prototype de getpid). La deuxième fonction fournit le handle du thread qui l'a appelée.



pthread_self

erreurs

Il n'y a pas de code d'erreurs. Si la spécification du thread n'est pas valide, le résultat est aléatoire.

On peut donc imaginer de rectifier le programme de fin de chapitre précédent par :

```
int main()  
  
    int ret, * retThread;  
    puts("Thread principal démarre");  
    printf("identite = %u.%u\n", getpid(), pthread_self());  
    ... puts("Fin du thread principal");  
}
```

```

void * fctThread (int * param)
{
    static int vr = 1007;

    puts("th> Debut de thread");
    printf("th> identite = %u.%u\n", getpid(), pthread_self());
    ...
    return 0;
}

```

Cela donne, sur une machine UNIX :

```

% thr
Thread principal démarre
identite = 20158.1072155384
Thread secondaire lance !
Attente de la fin du thread secondaire
th> Debut de thread
th> identite = 20158.1073795088
th> Parametre reçu = 5
th> Valeur renvoyée par défaut = 1007
th> Nouvelle valeur : 342
th> Valeur renvoyée = 342
Valeur renvoyée par le thread secondaire = 342
Fin du thread principal
%

```

En fait, la valeur fournie par `pthread_self()` est

- ◆ dans le cas des machines Sun, un **numéro de séquence** donné au thread à sa création; la norme donne l'assurance que ce numéro est unique pour tous les threads actifs; évidemment, ce numéro pourra être réutilisé une fois le thread supprimé;
- ◆ dans le cas d'autres machines Unix, un **pointeur sur la structure descriptive** du thread qui contient un champ correspondant à ce numéro de séquence; plutôt que de décortiquer la structure pointée, on peut alors utiliser la fonction non portable :

```

unsigned long pthread_getsequence_np((<handle du thread - thread_t>);

```

On peut donc modifier le programme de base comme suit :

THREAD01-NUM.C

```

/* THREAD01-NUM.C
Claude Vilvens
*/

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

void * fctThread(int * param);
pthread_t threadHandle;

```

```

int main()
{
    int ret, * rethread, paramEff = 5;
    unsigned long numSequence;
    puts("Thread principal démarre");
    printf("identite = %d.%u\n", getpid(), pthread_self());
    numSequence = pthread_self( );
    printf("Identite avec numero de sequence = %d.%u\n", getpid(), numSequence);
    ret = pthread_create(&threadHandle, NULL, (void*)(*)(void*)) fctThread, &paramEff);
    puts("Thread secondaire lance !"); puts("Attente de la fin du thread secondaire");
    ret = pthread_join(threadHandle, (void **)&retThread);
    printf("Valeur renvoyee par le thread secondaire = %d\n", *retThread);
    puts("Fin du thread principal");
}

void * fctThread (int * param)
{
    static int vr = 1007;
    unsigned long numSequence;
    puts("th> Debut de thread");
    numSequence = pthread_self( );
    printf("th> Identite avec numero de sequence = %d.%u\n", getpid(), numSequence);
    printf("th> identite = %d.%d\n", getpid(), pthread_self());
    printf("th> Parametre reçu = %d\n", *param);
    printf("th> Valeur renvoyee par default = %d\n", vr);
    printf("th> Nouvelle valeur : "); scanf("%d",&vr);
    printf("th> Valeur renvoyee = %d\n", vr);

    pthread_exit(&vr);
    return 0;
}

```

On obtient ainsi les numéros de séquence du thread principal et du thread secondaire :

```

sunray2v440.inpres.epl.prov-liege.be> c
Thread principal démarre
identite = 8820.3223038392
Identite avec numero de sequence = 26373.1
Thread secondaire lance !
Attente de la fin du thread secondaire
th> Debut de thread
th> Identite avec numero de sequence = 26373.4
th> identite = 8820.1073985920
th> Parametre reçu = 5
th> Valeur renvoyee par default = 1007
th> Nouvelle valeur : 21
th> Valeur renvoyee = 21
Valeur renvoyee par le thread secondaire = 21
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>

```

2. Les process status pour les threads

Sur certaines machines Unix, on peut vérifier l'existence des threads au moyen de la commande **ps** améliorée des qualificatifs **-m**, **-f** et **-l** (**ps -mlf** en abrégé). Si on lance cette commande depuis une 2ème session pendant que le programme **thread01-num.c** fonctionne dans l'autre session, on peut obtenir :

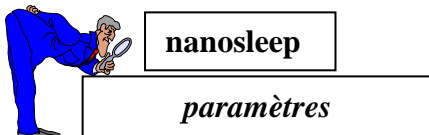
```
sunray2v440.inpres.epl.prov-liege.be> ps -m -f -l
  F S      UID  PID  PPID %CPU PRI  NI  RSS WCHAN    STARTED    D
80808001 S + 28   2405 11647 0.1  44   0  328K *      20:25:49    c
      S              0.1  44   0      tty
      S              0.0  44   0      nxmbloc
      S              0.0  44   0      nxmidle
80c08001 S   28   11647  125  0.0  44   0  256K pause    20:22:25    )
80c08001 S   28   22717 23071 0.0  44   0  256K pause    20:24:46    )
```

On peut constater que le process 2405, qui a lancé l'exécutable "c", comporte 3 threads, qui sont d'ailleurs tous en sommeil : **nxmbloc** désigne le thread père en attente; **nxmidle** désigne le thread fils en cours d'exécution, mais en attente d'une entrée gérée par le thread **tty** (pour rappel, **R**=runnable, **S**=sleep, **I**=Idle, **W**=swapped).

3. La mise en sommeil d'un thread

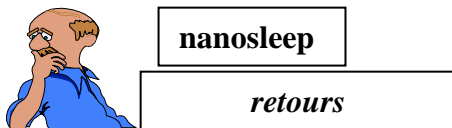
On peut mettre un thread en sommeil comme tout processus, mais par un "sleep" propre aux threads (car le sleep classique n'est pas supporté par les librairies multi-threads) : il s'agit de la fonction

```
int nanosleep (<intervalle de temps - const struct timespec *>,
               <temps restant - struct timespec *>);
```



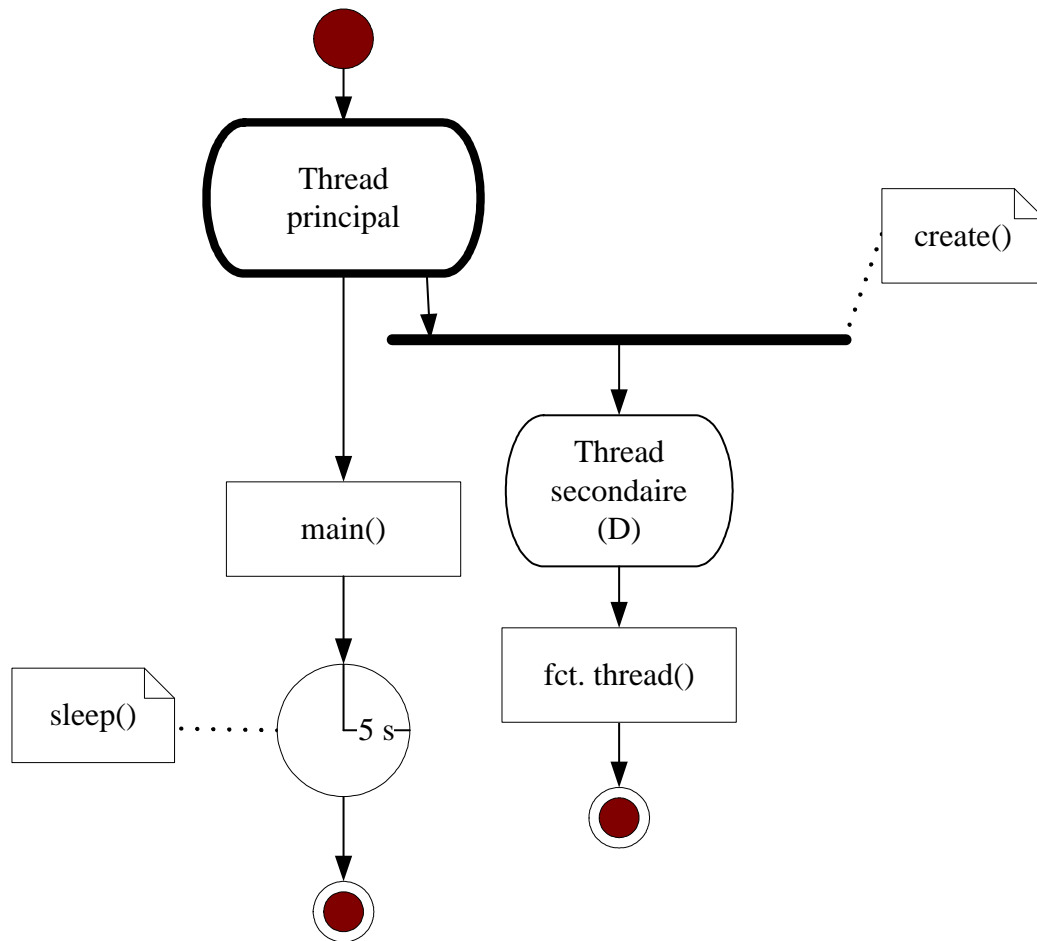
La structure de définition du temps est définie dans **pthread.h** par :

```
structure timespec
typedef struct timespec
{
    unsigned long    tv_sec;        /* seconds */
    long            tv_nsec;        /* nanoseconds */
} timespec_t;
```



<i>valeur de retour</i>	<i>erreur</i>
0	succès : le temps est écoulé
-1	le délai d'attente a été interrompu et errno prend la valeur EINTR et le deuxième paramètre reçoit la valeur du temps non écoulé

Dans l'exemple précédent, on "freine" le thread principal en le mettant en sommeil 5 secondes. Cela laisse le temps d'entrer une valeur :



THREAD01-D-S.C

```
/* THREAD01-D-S.C
```

```
Claude Vilvens
```

```
*/
```

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

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

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <time.h>
```

```
void * fctThread(int * /*pthread_addr_t*/ param);
```

```
pthread_t threadHandle;
```

```
int main()
```

```
{
```

```
    int ret;
```

```
    int paramEff = 5;
```

```
    struct timespec_t temps;
```

```
    puts("Thread principal démarre");
```

```
printf("identite = %d.%u\n", getpid(), pthread_self());
ret = pthread_create(&threadHandle, NULL, (void*)(*)(void*)) fctThread, &paramEff);
puts("Thread secondaire lance !");
puts("Detachement du thread");
ret = pthread_detach(threadHandle);

temps.tv_sec = 5; temps.tv_nsec = 0;
nanosleep(&temps, NULL);

puts("Fin du thread principal");
}

void * fctThread (int * param)
{
    ...
    printf("th> Nouvelle valeur : ");scanf("%d",&vr);
    ...
    pthread_exit(&vr);
    return 0;
}
```

La création de l'exécutable réclame en plus l'inclusion de la librairie de run-time : **-lrt**.
Ensuite, si l'on ne traîne pas à l'exécution :

```
sunray2v440.inpres.epl.prov-liege.be> c
Thread principal demarre
identite = 8938.3223038392
Thread secondaire lance !
Detachement du thread
th> Debut de thread
th> identite = 8938.1073985920
th> Parametre recu = 5
th> Valeur renvoyee par default = 1007
th> Nouvelle valeur : 25
th> Valeur renvoyee = 25
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

Par contre, si l'on tarde trop :

```
sunray2v440.inpres.epl.prov-liege.be> c
Thread principal demarre
identite = 8939.3223038392
Thread secondaire lance !
Detachement du thread
th> Debut de thread
th> identite = 8939.1073985920
th> Parametre recu = 5
th> Valeur renvoyee par default = 1007
th> Nouvelle valeur : Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

Le thread fils a été interrompu avec le père !

Remarque importante

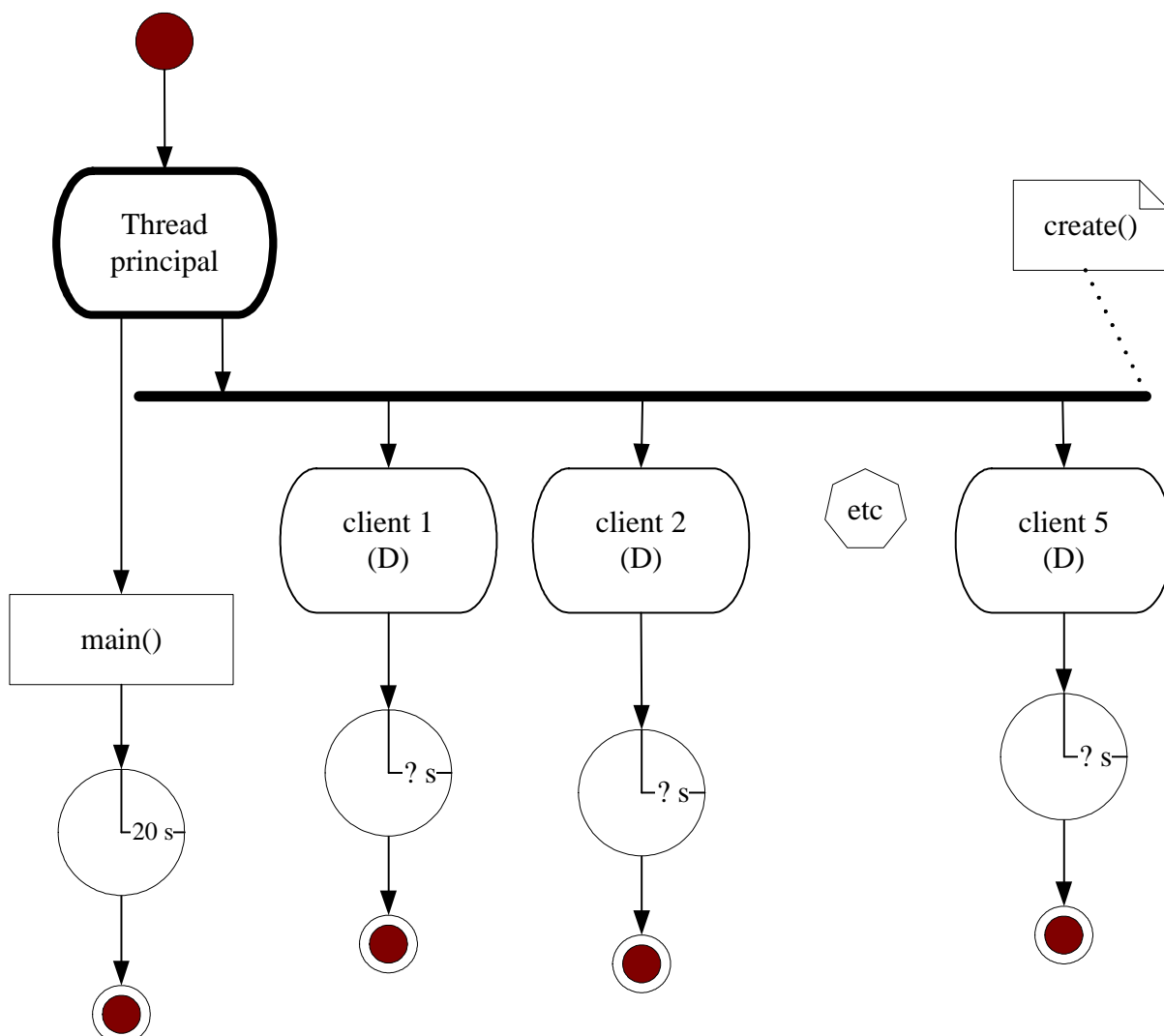
Il va de soi que jouer ainsi sur les temps peut relever du bricolage programmatique : dans les situations critiques, il faut impérativement utiliser les primitives de synchronisation (comme join, ainsi que vu plus haut, ou wait, comme nous le verrons)

4. Le supermarché : version de base

Afin d'illustrer nos propos de manière un peu plus ludique, nous allons développer progressivement un petit programme de simulation. La simulation porte sur un supermarché. Le thread principal est le **gérant** du magasin : il ouvre et ferme le magasin, s'occupe des stocks (avec l'éventuelle collaboration d'un thread manutentionnaire), fixe les prix, etc. Les **clients** sont représentés par des threads détachés. Chaque client entre dans le supermarché, s'empare d'un certain nombre d'articles disponibles, passe à la caisse et finalement sort.

Une version élémentaire et simpliste sera alors la suivante :

- ♦ de manière aléatoire, 5 clients entrent dans le magasin;
- ♦ ils y passent un temps aléatoire;
- ♦ lorsque le 5ème client est entré, le gestionnaire ferme le magasin au bout d'un certain temps (par exemple, 20s).



SUPERMARCH01.C

```
/* SUPERMARCH01.C
Claude Vilvens
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)

void * fctThreadClient(int * param);
char * getThreadIdentity();

pthread_t threadHandle;
int nbClientEntres = 0;
char ouvert=0;

int main()
{
    int ret, * retThread;
    char buf[80], rep, nouveauClient;

    /* Initialisation */
    puts("Thread principal démarre");
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;

    /* Création des clients */
    while (ouvert && nbClientEntres<=5)
    {
        nouveauClient = rand()%5 ==0 ;
        if (nouveauClient)
        {
            puts("Nouveau client ! Bonjour !");
            ret = pthread_create(&threadHandle, NULL,
                               (void (*)(void *))fctThreadClient,0);
            puts("Thread secondaire lance !");nbClientEntres++;
            puts("Detachement du thread");
            ret = pthread_detach(threadHandle);
        }
        if (nbClientEntres==5)
        {

```

```
        sleep(20);    // mmm ... pas idéal !
        ouvert=0;
    }
}

puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char buf[80];

    char * numThr = getThreadIdentity();

    if (ouvert)
    {
        temps.tv_sec = rand()/5000;
        temps.tv_nsec = 0;
        sprintf(buf, "Attente de %d secondes ...", temps.tv_sec);
        affThread(numThr, buf);
        nanosleep(&temps, NULL);
    }
    else affThread(numThr, "Le magasin est fermé - trop tard :-( ...");

    sprintf(buf, "Fin du thread client");affThread(numThr, buf);
    pthread_exit(&ouvert);
    return 0;
}

char * getThreadIdentity()
{
    unsigned long numSequence;
    char *buf = (char *)malloc(30);

    numSequence = pthread_self( );
    /* numSequence = pthread_getsequence_np( pthread_self( ) ); */
    sprintf(buf, "%d.%u", getpid(), numSequence);

    return buf;
}
```

On remarquera tout d'abord, même si cela ne constitue pas le point essentiel :

- ◆ la macro **affThread** permettant l'affichage d'une chaîne de caractère précédée d'un identifiant du thread de la forme **th_<pid>.<tid>**;
- ◆ la fonction **getThreadIdentity**() qui recherche le numéro de séquence d'un thread et fabrique l'identifiant évoqué ci-dessus.

Pour ce qui est du programme lui-même, on y trouve :

- ◆ l'ouverture du magasin;
- ◆ la création, à un rythme aléatoire, de 5 threads qui sont les clients; chaque thread s'attarde pendant un temps aléatoire;
- ◆ la temporisation du thread principal une fois le dernier client entré (ici, 20 secondes, ce qui permet dans la simulation à tous les threads clients de se terminer);
- ◆ la "communication" entre les threads par l'intermédiaire de la variable *ouvert* globale, donc partagée par tous les threads;
- ◆ la fermeture du magasin qui correspond, en fait, à la fin du thread principal.

Enfin, on peut voir que le générateur de nombres aléatoires n'est pas initialisé à chaque exécution : on obtient donc toujours la même séquence, ce qui est pratique pour étudier les variations de comportement en fonction de la valeur de certains paramètres. Si l'on désire un véritable effet aléatoire, on utilisera `srand()`.

Une exécution de ce programme sur sunray donne :

```
sunray2v440.inpres.epl.prov-liege.be> c
Thread principal démarre
th_9238.1> Thread principal démarre
Ouvrir le magasin ?O
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
th_9238.4> Attente de 2 secondes ...
th_9238.5> Attente de 2 secondes ...
th_9238.6> Attente de 6 secondes ...
th_9238.7> Attente de 3 secondes ...
th_9238.8> Attente de 2 secondes ...
th_9238.4> Fin du thread client
th_9238.5> Fin du thread client
th_9238.8> Fin du thread client
th_9238.7> Fin du thread client
th_9238.6> Fin du thread client
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

Evidemment, il faudrait que nos clients achètent quelque chose. Cela va venir ...

Remarque

Sur les machines Unix qui supportent les commandes utilisées, l'état des processus est, pendant l'exécution :

```
sunray2v440.inpres.epl.prov-liege.be> ps -mlf
  F S      UID  PID  PPID %CPU PRI  NI  RSS WCHAN  STARTED  D
80c08001 I    28   8722  8719  0.0  44   0 256K pause    10:37:01  )
80808001 S +   28   9150  8722  0.0  44   0 448K *      11:37:34  c
      S                0.0  44   0      nxmidle
      S                0.0  44   0      nxmbloc
      S                0.0  44   0      usleep
80c08001 S    28   8786  8783  0.0  44   0 256K pause    10:46:48  )
sunray2v440.inpres.epl.prov-liege.be>
```

5. Les fonctions de terminaison d'un thread

Il est possible de spécifier, pour chaque thread, une fonction à exécuter au moment où ce thread se termine; en d'autres lieux, on parlerait de "destructeur" ou de "postroutine". En fait, il est même possible d'empiler plusieurs fonctions qui seront appelées dans l'ordre inverse de leur définition (comme il convient à une pile digne de ce nom).

L'empilement d'un appel à une fonction de terminaison se réalise au moyen de la fonction :

```
int pthread_cleanup_push (<fonction exécutée à la fin - void (*) (void *)>,
                          <paramètre de cette fonction - void *arg>);
```

Il n'y a pas de valeur de retour prévue ... et pour cause, puisqu'il s'agit d'une macro !

L'appel de la fonction en fin d'exécution du thread n'est pas automatique (dans ce cas, d'ailleurs, quel serait l'intérêt de pouvoir en définir plusieurs ?), il doit être provoqué par l'appel de la fonction de dépilement :

```
int pthread_cleanup_pop (<flag d'exécution - int>);
```



pthread_cleanup_pop

paramètre

Si le paramètre est différent de 0, la fonction est alors effectivement exécutée immédiatement. Sinon, elle ne l'est pas.

En fait, sous Digital Unix, ces fonctions pthread_cleanup_? n'en sont pas vraiment : ce sont en fait des macros, qui ne simulent pas à un appel mais dont l'expansion génère du code de début et de fin entre lesquels le code effectif de terminaison doit prendre place. En quelque sortes, ces macros définissent des "accolades" particulières. La fonction à appeler en terminaison de thread a le prototype suivant :

```
void fctThreadFin (void *param) ;
```

tandis que la fonction à exécuter par le thread comportera les appels des macros :

```
void * fctThread (int * param)
{
    <déclarations> ...

    pthread_cleanup_push(fctThreadClientFin,<paramètre de la fonction de fin>);

    <code de la fonction de fin> ...

    pthread_cleanup_pop(1);
    pthread_exit(&<valeur de fin>);
    return 0;
}
```

La macro `pthread_cleanup_pop()` recevant un argument non nul, l'appel est exécuté juste avant que le thread se termine.

6. Le supermarché : passer à la caisse

Reprenons notre supermarché en faisant passer nos clients à la caisse lorsque leur activité au sein du magasin se termine. Nous allons donc spécifier une routine de terminaison pour chaque thread client, routine qui annoncera que le client paie ses achats. Cela donne :

SUPERMARCH02.C

```
/* SUPERMARCH02.C
Claude Vilvens
*/
...

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
...
int vFin = 99;

int main()
{
    int ret, * retThread;
    char buf[80], rep, nouveauClient;

    /* Initialisation */
    ...
    /* Création des clients */
    ...
    puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char buf[80];
    char * numThr = getThreadIdentity();
```



```

pthread_cleanup_push(fctThreadClientFin,0);

if (ouvert)
{
    temps.tv_sec = rand()/5000;
    temps.tv_nsec = 0;
    sprintf(buf, "Attente de %d secondes ...", temps.tv_sec);
    affThread(numThr, buf);
    nanosleep(&temps, NULL);
}
else affThread(numThr, "Le magasin est fermé - trop tard :-( ...");
sprintf(buf, "Fin du thread client");affThread(numThr, buf);

pthread_cleanup_pop(1);
pthread_exit(&vFin);
return 0;
}

void fctThreadClientFin (void *p)
{
    char buf[80];
    char * numThr = getThreadIdentity();
    sprintf(buf, "... je passe à la caisse ...");
    affThread(numThr, buf);
}

char * getThreadIdentity() { ... }

```

Un exemple d'exécution sur Sunray sera :

```

sunray2v440.inpres.epl.prov-liege.be> c
h_31453.1> Thread principal démarre
Ouvrir le magasin ?O
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
th_31453.2> Attente de 2 secondes ...

```

```
th_31453.3> Attente de 2 secondes ...
th_31453.4> Attente de 6 secondes ...
th_31453.5> Attente de 3 secondes ...
th_31453.6> Attente de 2 secondes ...
th_31453.2> Fin du thread client
th_31453.2> ... je passe à la caisse ...
th_31453.3> Fin du thread client
th_31453.3> ... je passe à la caisse ...
th_31453.6> Fin du thread client
th_31453.6> ... je passe à la caisse ...
th_31453.5> Fin du thread client
th_31453.5> ... je passe à la caisse ...
th_31453.4> Fin du thread client
th_31453.4> ... je passe à la caisse ...
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

Bien sûr, tous les clients ont eu le temps de passer à la caisse à la condition que le temps de sommeil du thread principal soit suffisamment long (par exemple, 20 secondes). Si ce dernier est trop court, *certaines threads n'auront pas le temps d'exécuter leur fonction terminale* avant la fin du thread principal et donc de l'ensemble du processus.

Remarque

Sur les machines Compaq, le programme ci-dessus doit être généré avec une option – **lexc** complémentaire :

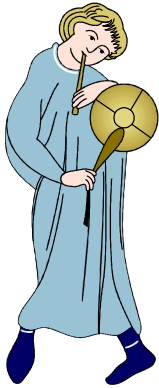
```
%>cc -o SuperMarche03.c c -lpthread -lexc
```



La vie de nos threads semble couler comme un long fleuve tranquille. On frémit à l'avance à l'idée de ce qui pourrait se passer si des signaux ou, plus généralement, des événements asynchrones bousculaient cette quiétude.

Pourtant, il va bien falloir se poser la question ...

III. L'asynchronisme dans les threads



*Ah ! Qu'en termes galants ces choses-là
sont mises !*

(Molière, Le Misanthrope)

1. Les threads et les signaux

Chaque thread a la possibilité de déterminer son propre comportement vis-à-vis des différents signaux. Ainsi, tout thread

- ◆ **possède son propre masque de signaux**, lequel est initialisé au moyen du masque du thread créateur; libre ensuite au thread fils de modifier ce masque (par **sigprocmask()**);
- ◆ **peut installer un handler de traitement** d'un signal particulier (par **sigaction()**), handler utilisable par tous les autres threads.

Par conséquent, la délivrance synchrone d'un signal émis par le noyau qui vient de détecter une exception dans l'exécution d'une instruction (par exemple, SIGINT ou SIGPIPE) se fera au thread concerné. Par contre, et c'est logique, l'émission d'un signal asynchrone (par exemple, un kill() à un processus d'identificateur précisé) est répercutée chez tous les threads du processus – on ne sait donc pas quel thread exécute le handler.

Le petit exemple suivant nous donnera une idée des possibilités. Un processus, armé pour traité SIGINT, lance deux threads. L'un d'entre eux est également armé pour traiter SIGINT, l'autre, au contraire, bloque ce signal .

SIGNALTHREAD01.C

```
/* SIGNALTHREAD01.C
Claude Vilvens
*/

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

#define affThread(num, msg) printf("th_%s> %s\n", num, msg);

void * fctThread(int * param);
void handlerSignal (int sig);
void handlerThreadSignal(int sig);
char * getThreadIdentity();
```

```
pthread_t threadHandle1, threadHandle2;
struct sigaction sigAct;

int main()
{
    int ret;
    int paramEff1 = 1, paramEff2 = 2;

    affThread(getThreadIdentity(), "Thread principal démarre");
    sigemptyset(&sigAct.sa_mask);
    sigAct.sa_handler = handlerSignal;
    sigaction(SIGINT, &sigAct, NULL);

    printf("lancement du thread %d\n", paramEff1);
    ret = pthread_create(&threadHandle1, NULL, (void*)(*)(void*)) fctThread,
        &paramEff1);
    puts("Thread secondaire 1 lance !");
    printf("lancement du thread %d\n", paramEff2);
    ret = pthread_create(&threadHandle2, NULL, (void*)(*)(void*)) fctThread,
        &paramEff2);
    puts("Thread secondaire 2 lance !");

    for (;;)          /* pour éviter la terminaison */
}

void * fctThread (int * param)
{
    int vr = (int) (*param);
    sigset_t mask;
    char buf[80];
    char * numThr = getThreadIdentity();

    affThread(numThr, ". démarrage du thread");
    sprintf(buf, "Parametre recu = %d", vr);
    affThread(numThr, buf);

    if (vr == 1)
    {
        #if 1
            affThread(numThr, "Je capterai le signal SIGINT");
            sigAct.sa_handler=handlerThreadSignal;
            sigaction(SIGINT, &sigAct, NULL);
        #endif
        for(;;);
    }
    else
    {
        affThread(numThr, "Je masque le signal SIGINT");
        sigemptyset(&mask);
        sigaddset(&mask, SIGINT);
    }
}
```

```

        sigprocmask(SIG_SETMASK, &mask, NULL);
        while(1);
    }
    pthread_exit(&vr);
    return 0;
}

void handlerSignal (int sig)
{
    char * numThr = getThreadIdentity();
    affThread(numThr, "handlerSignal pour main a reçu SIGINT");
}

void handlerThreadSignal(int sig)
{
    char * numThr = getThreadIdentity();
    affThread(numThr, "handlerSignal pour thread a reçu SIGINT");
}

char * getThreadIdentity() {...}

```

Si on a armé sur SIGINT dans main et dans le thread :

```

th_24163.1> Thread principal démarre
lancement du thread 1
Thread secondaire 1 lance !
lancement du thread 2
Thread secondaire 2 lance !
th_24163.2> . démarrage du thread
th_24163.2> Parametre reçu = 1
th_24163.2> Je capterai le signal SIGINT
th_24163.3> . démarrage du thread
th_24163.3> Parametre reçu = 2
th_24163.3> Je masque le signal SIGINT
th_24163.1> handlerSignal pour thread a reçu SIGINT
th_24163.1> handlerSignal pour thread a reçu SIGINT
th_24163.2> handlerSignal pour thread a reçu SIGINT
th_24163.1> handlerSignal pour thread a reçu SIGINT
th_24163.2> handlerSignal pour thread a reçu SIGINT
th_24163.1> handlerSignal pour thread a reçu SIGINT
th_24163.2> handlerSignal pour thread a reçu SIGINT
th_24163.1> handlerSignal pour thread a reçu SIGINT
th_24163.2> handlerSignal pour thread a reçu SIGINT
th_24163.2> handlerSignal pour thread a reçu SIGINT
Killed
sunray2v440.inpres.epl.prov-liege.be>

```

C'est évidemment logique : le traitement de SIGINT, au niveau du processus, a été finalement confié à handlerThreadSignal (que la variable sigaction soit globale ou locale à

chaque thread) : le processus, bien que comportant plusieurs threads, reste unique ! Mais c'est tantôt le thread principal, tantôt le thread secondaire qui capte le signal. Si on avait seulement armé le thread principal sur SIGINT, on aurait obtenu le traitement par handlerSignal(), toujours tantôt par main(), tantôt par le thread secondaire :

```
sunray2v440.inpres.epl.prov-liege.be> c
th_15868.1> Thread principal démarre
lancement du thread 1
Thread secondaire 1 lance !
lancement du thread 2
Thread secondaire 2 lance !
th_15868.2> . démarrage du thread
th_15868.2> Parametre reçu = 1
th_15868.3> . démarrage du thread
th_15868.3> Parametre reçu = 2
th_15868.3> Je masque le signal SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.2> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
th_15868.2> handlerSignal pour main a reçu SIGINT
th_15868.1> handlerSignal pour main a reçu SIGINT
Killed
sunray2v440.inpres.epl.prov-liege.be>
```

Et si l'on avait seulement armé le thread secondaire sur SIGINT, les choses auraient été semblables, mais avec le handler du thread. Tout ceci pour bien rappeler que *les threads font partie du même processus* ...

Remarque

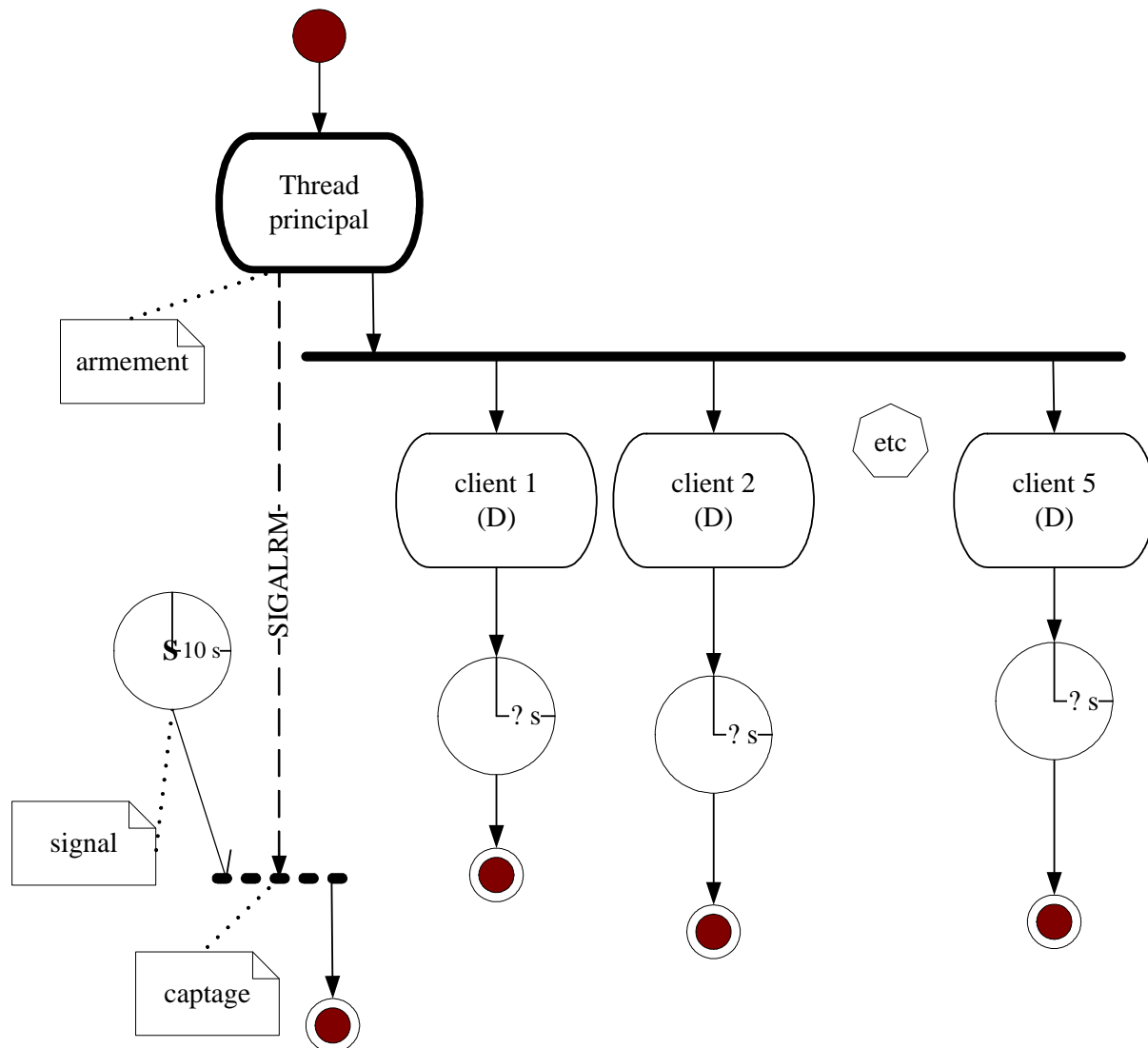
Deux primitives particulières aux threads ont été définies pour ce qui concerne les signaux :

- ◆ **int pthread_sigmask** (<type de modification pour l'ensemble des signaux masqués – int>
 <masque à définir - const sigset_t * >,
 <valeur actuelle du masque - sigset_t * >);
- comme sa cousine sigprocmask(), cette primitive ne peut bloquer les signaux SIGKILL et SIGSTOP.
- ◆ **int pthread_kill** (<handle du thread destinataire - pthread_t>,
 <signal – int>);
- cette primitive délivre un signal au thread spécifié en argument.

2. Le supermarché : quand on ferme

Dans notre exemple de supermarché, nous avons fixé le temps de fermeture au moyen d'un sleep lancé à partir du moment où 5 clients sont entrés. Nous allons à présent voir les choses différemment en utilisant le signal SIGALRM qui nous permettra de fixer un temps d'ouverture bien déterminé (par exemple 10 s) à partir du moment de l'ouverture.

Schématiquement :



SUPERMARCH03.C

```
/* SUPERMARCH03.C
Claude Vilvens
*/
```

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

```
#define affThread(num, msg) printf("th_%s> %s\n", num, msg)

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
char * getThreadIdentity();
void handlerSignalAlarme(int sig);

pthread_t threadHandle;
struct sigaction sigAct;

int nbClientEntres = 0;
char ouvert=0;
int vFin = 99;

int main()
{
    int ret, * retThread;
    char buf[80], rep, nouveauClient;

    /* Initialisation */
    puts("Thread principal démarre");
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;

    sigAct.sa_handler = handlerSignalAlarme;
    sigaction(SIGALRM, &sigAct, NULL);
    alarm(10);

    /* Création des clients */
    while (ouvert && nbClientEntres<=5)
    {
        nouveauClient = rand()%5 ==0 ;
        if (nouveauClient)
        {
            puts("Nouveau client ! Bonjour !");
            ret = pthread_create(&threadHandle, NULL,
                (void (*)(void *))fctThreadClient,0);
            puts("Thread secondaire lance !");nbClientEntres++;
            puts("Detachement du thread");
            ret = pthread_detach(threadHandle);
        }
    }
    while(1);
}
```



```

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char buf[80];
    char * numThr = getThreadIdentity();

    pthread_cleanup_push(fctThreadClientFin,0);

    if (ouvert)
    {
        temps.tv_sec = rand()/5000; temps.tv_nsec = 0;
        sprintf(buf, "Attente de %d secondes ...", temps.tv_sec);affThread(numThr, buf);
        nanosleep(&temps, NULL);
    }
    else affThread(numThr, "Le magasin est fermé - trop tard :-( ...");
    sprintf(buf, "Fin du thread client");affThread(numThr, buf);

    pthread_cleanup_pop(1);
    pthread_exit(&vFin);

    return 0;
}

void fctThreadClientFin (void *p) { ... }

void handlerSignalAlarme(int sig)
{
    puts("!!-!! Notre magasin ferme ses portes !!-!!");ouvert=0;
    pthread_exit(0);
    return;
}

char * getThreadIdentity() { ... }

```

Un exemple d'exécution peut être :

```

sunray2v440.inpres.epl.prov-liege.be> c
Thread principal démarre
th_27235.1> Thread principal démarre
Ouvrir le magasin ?O
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
Nouveau client ! Bonjour !
Thread secondaire lance !
Detachement du thread
th_27235.4> Attente de 6 secondes ...

```

```

th_27235.7> Attente de 3 secondes ...
th_27235.6> Attente de 5 secondes ...
th_27235.5> Fin du thread client
th_27235.5> ... je passe à la caisse ...
!-!! Notre magasin ferme ses portes !-!!
th_27235.6> Fin du thread client
th_27235.6> ... je passe à la caisse ...
th_27235.4> Fin du thread client
th_27235.4> ... je passe à la caisse ...
sunray2v440.inpres.epl.prov-liege.be>

```

On peut remarquer que :

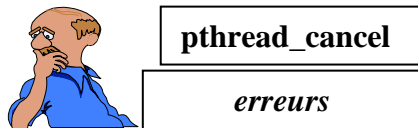
- ◆ le temps d'ouverture impartie est trop court pour permettre la création des 5 clients (maximum prévu);
- ◆ les fonctions de terminaisons sont appelées pour tous les threads créés.

3. L'arrêt programmé d'un thread

Il se fait que la norme POSIX a aussi prévu la fonction :

```
int pthread_cancel (<handle du thread - pthread_t>);
```

dont la fonction est bien claire : envoyer une demande d'arrêt au thread concerné. Il est donc demandé à celui-ci de se terminer "*aussi vite que possible*", notion qui varie selon sa configuration (voir ci-dessous). En fait, envoyer une requête de terminaison à un thread ne garantit pas qu'il va obtempérer, ni même recevoir le demande !



valeur de retour		erreur
0		succès
#define EINVAL	22	/* Invalid argument */ : le paramètre handle de thread n'est pas valide
#define ESRCH	3	/* No such process */ : le thread référencé n'existe pas (ou plus)

Comme on l'a dit, la manière de prendre en compte une telle demande dépend de la manière dont le thread a été configuré vis-à-vis de ce type d'événement. Cette configuration est déterminée par deux indicateurs positionnés par deux fonctions.

1) La première positionne un **indicateur général** qui, lorsqu'il est inhibé, reporte la prise en compte des requêtes reçues au moment où cet indicateur est à nouveau activé. Par défaut, cet indicateur est positionné. La syntaxe de la fonction est, sous Digital UNIX :

```
int pthread_setcancelstate (<état donné à l'indicateur - int>,
                           <ancien état de l'indicateur> - int *>);
```

l'état étant exprimé en utilisant les constantes :

```
#define PTHREAD_CANCEL_DISABLE    0
#define PTHREAD_CANCEL_ENABLE    1
```



pthread_setcancelstate

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre état n'est pas valide, c'est-à-dire qu'il n'est ni PTHREAD_CANCEL_ENABLE ni PTHREAD_CANCEL_DISABLE

Remarque

La fonction équivalente de la norme POSIX 1003.4a est :

```
int pthread_setcancel ( <état donné à l'indicateur – pthread_t>);
```

les valeurs de l'état étant CANCEL_ON et CANCEL_OFF.

2) La deuxième fonction, qui ne présente d'intérêt que si l'indicateur général est positionné, positionne un deuxième indicateur d'asynchronisme. Sous Digital UNIX :

```
int pthread_setcanceltype ( <état donné à l'indicateur – int>,
                           <ancien état de l'indicateur> - int *>);
```

en utilisant les constantes :

```
#define PTHREAD_CANCEL_DEFERRED    0
#define PTHREAD_CANCEL_ASYNCHRONOUS 1
```

Si il est activé, une requête d'abandon est prise en compte immédiatement. Sinon, la demande sera pris en compte au premier point de contrôle rencontré, lequel est mis en place par la fonction :

```
void pthread_testcancel (void);
```

La demande est, de toute manière, prise en compte lors des appels de fonctions de synchronisation comme pthread_join() ou pthread_cond_wait()³.



pthread_setcanceltype

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès

³ voir plus loin (synchronisation des threads)

#define EINVAL	22	/* Invalid argument */ : le paramètre état n'est pas valide, c'est-à-dire qu'il n'est ni PTHREAD_CANCEL_DEFERRED ni PTHREAD_CANCEL_ASYNCHRONOUS
----------------	----	---

Remarque

A nouveau, il existe une fonction équivalente à `setcanceltype()` pour la norme POSIX 1003.4a est :

`int pthread_setasynccancel (<état donné à l'indicateur – pthread_t>);`

les valeurs de l'état étant toujours `CANCEL_ON` et `CANCEL_OFF`.

4. Le supermarché : il y a un voleur

Imaginons qu'un client du supermarché soit en fait un voleur. Lorsque l'on s'en rend compte, le gestionnaire du magasin jette le voleur dehors – après l'avoir fait passer à la caisse (et encore, c'est parce qu'il est bon !).

Dans la simulation, lorsqu'un thread est détecté comme un voleur, on lance un signal `SIGUSR1` au thread principal, qui va arrêter les sombres agissements du sinistre personnage.

SUPERMARCH04.C

```
/* SUPERMARCH04.C
Claude Vilvens
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define N_MAX_CLIENTS 5

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
char * getThreadIdentity();
void handlerSignalAlarme(int sig);
void handlerVoleur (int sig);

pthread_t threadHandle;
pthread_t threadHandleVoleur;
struct sigaction sigAct;

int nbClientEntres = 0;
char ouvert=0;
int vFin = 99;

int main()
{
    int ret, * retThread, aleat;
    char buf[80], rep, nouveauClient;

    /* Initialisation */
    puts("Thread principal démarre");
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;

    sigAct.sa_handler = handlerSignalAlarme;
    sigaction(SIGALRM, &sigAct, NULL);
    sigemptyset(&sigAct.sa_mask);
    sigAct.sa_handler = handlerVoleur;
    if ( (ret=sigaction(SIGUSR1, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction sur SIGUSR1");
    alarm(25);
```

```
/* Création des clients */
while (ouvert && nbClientEntres<N_MAX_CLIENTS)
{
    aleat = rand();
    nouveauClient = aleat%5==0 ;
    if (nouveauClient)
    {
        nbClientEntres++;
        printf("%d. Nouveau client ! Bonjour !", nbClientEntres);
        ret = pthread_create(&threadHandle, NULL,
            (void (*)(void *))fctThreadClient,0);
        puts("Thread secondaire lance !");
        if (aleat%20==0)
        {
            threadHandleVoleur = threadHandle;
            kill (getpid(), SIGUSR1);
        }
        else
        {
            puts("Detachement du thread");
            ret = pthread_detach(threadHandle);
        }
    }
}
while(1);
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    int ancEtat;
    char buf[80];
    char * numThr = getThreadIdentity();

    pthread_cleanup_push(fctThreadClientFin,0);

    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");

    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");
#ifdef 0
    pthread_testcancel();
#endif

    if (ouvert)
    {
        temps.tv_sec = rand()/5000;
        temps.tv_nsec = 0;
        sprintf(buf, "Attente de %d secondes ...", temps.tv_sec);
    }
}
```

```

        affThread(numThr, buf);
        nanosleep(&temps, NULL);
    }
    else affThread(numThr, "Le magasin est fermé - trop tard :-( ...");
    sprintf(buf, "Fin du thread client");affThread(numThr, buf);

    pthread_cleanup_pop(1);
    pthread_exit(&vFin);

    return 0;
}

void fctThreadClientFin (void *p) { ... }
void handlerSignalAlarme(int sig) { ... }

void handlerVoleur (int sig)
{
    int ret;
    puts(" !!!!!!!!!!! Au voleur !!!!!!!!!!!");
    if ( ret=pthread_cancel(threadHandleVoleur) )
        affThread(getThreadIdentity(), "erreur de pthread_cancel");
}

char * getThreadIdentity() { ... }

```

Une première exécution sur une machine Unix relativement lente (ici, Boole), sans l'appel de `test_cancel()` donne (attention au décalage entre le numéro de client et le numéro de séquence du thread associé) :

```

Thread principal démarre
th_22630.1> Thread principal démarre
Ouvrir le magasin ?O
1. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
2. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
3. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!
4. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
5. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
th_22630.2> Attente de 2 secondes ...
th_22630.3> Attente de 2 secondes ...
th_22630.4> Attente de 6 secondes ...
th_22630.4> ... je passe à la caisse ...
th_22630.5> Attente de 3 secondes ...
th_22630.6> Attente de 2 secondes ...
th_22630.2> Fin du thread client
th_22630.2> ... je passe à la caisse ...

```



```
th_22630.3> Fin du thread client
th_22630.3> ... je passe à la caisse ...
th_22630.6> Fin du thread client
th_22630.6> ... je passe à la caisse ...
th_22630.5> Fin du thread client
th_22630.5> ... je passe à la caisse ...
!-!-! Notre magasin ferme ses portes !-!-!
boole.inpres.epl.prov-liege.be>
```

Sur une machine Unix relativement plus rapide (ici, Copernic), toujours sans `testcancel()` :

```
copernic.inpres.epl.prov-liege.be> c
Thread principal démarre
th_13759.1> Thread principal démarre
Ouvrir le magasin ?O
1. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
2. Nouveau client ! Bonjour !
th_13759.2> Attente de 1 secondes ...
th_13759.3> Attente de 3 secondes ...
Thread secondaire lance !
Detachment du thread
3. Nouveau client ! Bonjour !Thread secondaire lance !
th_13759.4> Attente de 6 secondes ...
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!
4. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
5. Nouveau client ! Bonjour !Thread secondaire lance !
Detachment du thread
th_13759.4> ... je passe à la caisse ...
th_13759.5> Attente de 3 secondes ...
th_13759.6> Attente de 5 secondes ...
th_13759.2> Fin du thread client
th_13759.2> ... je passe à la caisse ...
th_13759.3> Fin du thread client
th_13759.3> ... je passe à la caisse ...
th_13759.5> Fin du thread client
th_13759.5> ... je passe à la caisse ...
th_13759.6> Fin du thread client
th_13759.6> ... je passe à la caisse ...
!-!-! Notre magasin ferme ses portes !-!-!
copernic.inpres.epl.prov-liege.be>
```

Dans les deux cas, on peut constater que le thread voleur (n° 4) n'a pas le temps de se terminer normalement : on l'envoie immédiatement à la caisse puisque, interrompu, il exécute immédiatement sa fonction de terminaison.

Sur copernic, le thread voleur n'a même pas eu le temps de se mettre en attente. On peut forcer cela dans tous les cas si nous reprenons l'exemple sur Boole avec un `test_cancel()` :

```
boole.inpres.epl.prov-liege.be> c
Thread principal démarre
th_22770.1> Thread principal démarre
Ouvrir le magasin ?O
1. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
2. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
3. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!
4. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
5. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
th_22770.2> Attente de 2 secondes ...
th_22770.3> Attente de 2 secondes ...
th_22770.4> ... je passe à la caisse ...
th_22770.5> Attente de 6 secondes ...
th_22770.6> Attente de 3 secondes ...
th_22770.2> Fin du thread client
th_22770.2> ... je passe à la caisse ...
th_22770.3> Fin du thread client
th_22770.3> ... je passe à la caisse ...
th_22770.6> Fin du thread client
th_22770.6> ... je passe à la caisse ...
th_22770.5> Fin du thread client
th_22770.5> ... je passe à la caisse ...
!-!-! Notre magasin ferme ses portes !-!-!
boole.inpres.epl.prov-liege.be>
```

5. Le supermarché : l'achat de quelques articles

Ce n'est pas tout de se promener dans le supermarché, il faut encore y faire ses achats. Nous allons donc imaginer que le stock du magasin est représenté au moyen de structures du type :

```
struct article
{
    int num;
    char libelle[30];
    double prixUnitaire;
    int quStock;
};
```

Bien sûr, le stock devrait être un fichier. Mais, pour alléger et simplifier, mettons que ce stock sera simplement représenté dans un tableau mémoire. Chaque client va donc se promener dans le magasin et acheter un nombre aléatoire d'articles. Pour l'instant, ils seront tous différents et on n'achètera qu'un seul article de type donné. Un petit tableau local contiendra les numéros des articles déjà achetés. On supposera aussi que le stock est suffisant. Toutes ces hypothèses simplificatrices seront levées au chapitre suivant.

Avec de telles activités, nos threads clients peuvent passer un temps plus long dans le magasin. Par conséquent, il se peut très bien que lorsque l'heure de la fermeture sonne certains clients soient encore occupés ... Dans un premier temps (mais nous ferons mieux après), l'ensemble du processus sera arrêté violemment (c'est-à-dire par un SIGKILL), après un ultime délai pour éventuellement terminer les achats.

Notre programme évoluera donc de la manière suivante (le chocolat est avarié et ne peut être vendu ;-)) :

SUPERMARCHE05.C

```

/* SUPERMARCHE05.C
Claude Vilvens */

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

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define N_MAX_CLIENTS 6
#define random(n) (rand()%(n)+1)

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
void handlerSignalAlarme(int sig);
void handlerVoleur (int sig);
char * getThreadIdentity();
char exists(int n, int tab[], int dim);

pthread_t threadHandle;
pthread_t threadHandleVoleur;
struct sigaction sigAct;
/* ----- */
/* Pour les articles du magasin */
struct article
{
    int num;
    char libelle[30];
    double prixUnitaire;
    int quStock;
};

#define NB_ART_DIFF 6
struct article magasins[] =
{
    { 1, "chocolat", 39, 2300},
    { 2, "whisky", 599, 50},
    { 3, "cereales", 67, 2600},
    { 4, "chips", 20, 5000},
    { 5, "preservatif", 20, 4000},
    { 6, "herbes aromatiques", 67, 3450}
};

```

```
/* ----- */
int nbClientEntres = 0;
char ouvert=0;

int main()
{
    int ret, * retThread;
    char buf[80], rep, nouveauClient;

    /* Initialisation */
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;
    sigAct.sa_handler = handlerSignalAlarme;
    sigaction(SIGALRM, &sigAct, NULL);
    sigemptyset(&sigAct.sa_mask);
    sigAct.sa_handler = handlerVoleur;
    if ( (ret=sigaction(SIGUSR1, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction sur SIGUSR1");
    alarm(25);

    /* Création des clients */
    while (ouvert && nbClientEntres<N_MAX_CLIENTS)
    {
        int aleat;
        aleat = rand();
        nouveauClient = aleat%5==0 ;
        if (nouveauClient)
        {
            printf("%d. Nouveau client ! Bonjour !", nbClientEntres++);
            ret = pthread_create(&threadHandle, NULL,
                               (void (*)(void *))fctThreadClient,0);
            puts("Thread secondaire lance !");
            if (aleat%20==0)
            {
                threadHandleVoleur = threadHandle; kill (getpid(), SIGUSR1);
            }
            else
            {
                puts("Detachement du thread"); ret=pthread_detach(threadHandle);
            }
        }
        if (nbClientEntres == N_MAX_CLIENTS)
            puts(" == C'était le dernier client ==");
    }
    while (ouvert);
}
```

```
    puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char buf[80];
    int nbArt, i=0, ancEtat;
    int numAch[NB_ART_DIFF];

    char * numThr = getThreadIdentity();
    char fini;
    int nEssai = 0;

    pthread_cleanup_push(fctThreadClientFin,0);

    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");
    pthread_testcancel();

    if (ouvert)
    {
        do
        {
            nbArt = random(NB_ART_DIFF);
        }
        while (nbArt<=0);
        sprintf(buf, "... je compte acheter %d articles", nbArt);
        affThread(numThr, buf);
        fini=0;
        do
        {
            int na = random(NB_ART_DIFF-1);
            if (!exists(na, numAch, NB_ART_DIFF))
            {
                numAch[i]=na;
                i++;
                sprintf(buf, "J'achete 1 %s", magasins[na].libelle);
                affThread(numThr, buf);
            }
            nEssai++;
            if (nEssai%20==0) sched_yield();
                /* Sinon la recherche peut durer ... */
            fini = i==nbArt || ouvert==0;
        }
        while (!fini);
        sprintf(buf, "J'ai fini mon tour"); affThread(numThr, buf);
        temps.tv_sec = rand()/5000;
```

```
        temps.tv_nsec = 0;
        sprintf(buf, "Attente a la caisse de %d secondes ...",
                temps.tv_sec);affThread(numThr, buf);
        nanosleep(&temps, NULL);
    }
    else affThread(numThr, "Le magasin est ferme - trop tard :-(");

    sprintf(buf, "Fin du thread client - il va sortir");
    affThread(numThr, buf);
    pthread_exit(&ouvert);

    pthread_cleanup_pop(0);

    return 0;
}

void fctThreadClientFin (void *p) {...}

void handlerSignalAlarme(int sig)
{
    timespec_t temps;
    char buf[80];
    char * numThr = getThreadIdentity();
    int i;

    puts("!!-!! Notre magasin ferme ses portes !!-!!");
    ouvert=0;
    temps.tv_sec = 10;
    temps.tv_nsec = 0;
    sprintf(buf, "!!-! Nous vous laissons %d secondes ...", temps.tv_sec);
    affThread(numThr, buf);
    nanosleep(&temps, NULL);
    kill(getpid(), SIGKILL);
    return;
}

void handlerVoleur (int sig)
{
    int ret;
    char buf[80];
    unsigned long numSequence;
    char * numThr = getThreadIdentity();

    numSequence = threadHandleVoleur;
    puts(" !!!!!!!!!!! Au voleur !!!!!!!!!!!");
    sprintf(buf, "!!-! Le thread : %u : Dehors !!!", numSequence);
    affThread(numThr, buf);
    if ( ret=pthread_cancel(threadHandleVoleur) )
        affThread(getThreadIdentity(), "erreur de pthread_cancel");
}
```

```
char * getThreadIdentity() { ... }

char exists(int n, int tab[], int dim)
{
    char trouve=0;
    int i;
    for (i=0; !trouve && i<dim; i++) trouve = tab[i]==n;
    return trouve;
}
```

Pour que la simulation prenne en compte les cas limites, on a sciemment programmé la possibilité de vouloir choisir 6 articles différents, alors qu'en fait seulement 5 sont disponibles (on ne vend pas de chocolat avarié). De plus, le temps d'ouverture a été choisi suffisamment court pour qu'il se puisse très bien qu'un client soit encore occupé au moment de la fermeture, même s'il n'achète que 5 articles différents au plus. Le thread client qui correspond, par malheur, à l'un de ces cas risque donc de ne pas se terminer de sitôt ... à moins qu'il ne teste périodiquement la variable globale ouvert qui lui apprendra que le magasin ferme ! Pour assurer que les choses se passent bien ainsi, nous allons obliger un tel thread à périodiquement passer la main.

6. Forcer un thread à libérer le processeur

En fait, tous les threads du programme précédent sont de même priorité. Il existe bien sûr une politique de scheduling, mais on peut forcer un thread à passer la main en lui faisant appeler la fonction :

```
void sched_yield(void);
```

Le résultat est qu'un thread de priorité supérieure ou égale recevra alors la main. Bien sûr, si un tel thread n'existe pas, le thread initial conserve la main et poursuit son exécution. En pratique, on utilise cette fonction pour tenter d'optimiser les performances. Dans notre exemple, c'est essentiellement pour laisser le thread principal mettre la variable ouvert à 0.

Sur une machine lente comme boole, on obtient une exécution du type :

```
sunray2v440.inpres.epl.prov-liege.be> c
th_30589.1> Thread principal démarre
Ouvrir le magasin ?O
0. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
1. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
2. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!! Au voleur !!!!!!!!
th_30589.1> !-! Le thread : 4 : Dehors !!!
3. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
4. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
5. Nouveau client ! Bonjour !Thread secondaire lance !
```

Detachement du thread

== C'était le dernier client ==

th_30589.2> ... je compte acheter 5 articles

th_30589.2> J'achete 1 cereales

th_30589.2> J'achete 1 herbes aromatiques

th_30589.2> J'achete 1 chips

th_30589.2> J'achete 1 preservatif

le thread est préempté
– mais il faut encore
acheter un article !

th_30589.3> ... je compte acheter 1 articles

th_30589.3> J'achete 1 preservatif

th_30589.3> J'ai fini mon tour

th_30589.3> Attente a la caisse de 0 secondes ...

th_30589.4> ... je passe à la caisse ...

th_30589.5> ... je compte acheter 5 articles

th_30589.5> J'achete 1 preservatif

th_30589.5> J'achete 1 chips

th_30589.5> J'achete 1 whisky

th_30589.5> J'achete 1 herbes aromatiques

th_30589.5> J'achete 1 cereales

th_30589.5> J'ai fini mon tour

th_30589.5> Attente a la caisse de 0 secondes ...

th_30589.6> ... je compte acheter 5 articles

th_30589.6> J'achete 1 herbes aromatiques

th_30589.6> J'achete 1 preservatif

th_30589.6> J'achete 1 chips

th_30589.6> J'achete 1 cereales

th_30589.6> J'achete 1 whisky

th_30589.6> J'ai fini mon tour

th_30589.6> Attente a la caisse de 2 secondes ...

th_30589.7> ... je compte acheter 1 articles

th_30589.7> J'achete 1 herbes aromatiques

th_30589.7> J'ai fini mon tour

th_30589.7> Attente a la caisse de 1 secondes ...

th_30589.3> Fin du thread client - il va sortir

th_30589.3> ... je passe à la caisse ...

th_30589.5> Fin du thread client - il va sortir

th_30589.5> ... je passe à la caisse ...

th_30589.7> Fin du thread client - il va sortir

th_30589.7> ... je passe à la caisse ...

th_30589.6> Fin du thread client - il va sortir

th_30589.6> ... je passe à la caisse ...

!-!-! Notre magasin ferme ses portes !-!-!

th_30589.1> !-! Nous vous laissons 10 secondes ...

th_30589.2> J'ai fini mon tour

bien obligé !

th_30589.2> Attente a la caisse de 6 secondes ...

th_30589.2> Fin du thread client - il va sortir

th_30589.2> ... je passe à la caisse ...

Killed

sunray2v440.inpres.epl.prov-liege.be>

Sur copernic ou sunray, le même programme donne :

```
copernic.inpres.epl.prov-liege.be> c
th_27034.1> Thread principal démarre
Ouvrir le magasin ?O
0. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
1. Nouveau client ! Bonjour !th_27034.2> ... je compte acheter 4 articles
th_27034.2> J'achete 1 cereales
th_27034.2> J'achete 1 herbes aromatiques
th_27034.2> J'achete 1 chips
th_27034.2> J'achete 1 whisky
th_27034.2> J'ai fini mon tour
th_27034.2> Attente a la caisse de 6 secondes ...
th_27034.3> ... je compte acheter 1 articles
th_27034.3> J'achete 1 preservatif
th_27034.3> J'ai fini mon tour
th_27034.3> Attente a la caisse de 5 secondes ...
Thread secondaire lance !
Detachement du thread
2. Nouveau client ! Bonjour !Thread secondaire lance !
Detachement du thread
3. Nouveau client ! Bonjour !th_27034.4> ... je compte acheter 4 articles
Thread secondaire lance !
th_27034.5> ... je compte acheter 3 articles
th_27034.5> J'achete 1 herbes aromatiques
th_27034.5> J'achete 1 whisky
th_27034.5> J'achete 1 chips
th_27034.5> J'ai fini mon tour
th_27034.5> Attente a la caisse de 4 secondes ...
th_27034.4> J'achete 1 chips
th_27034.4> J'achete 1 herbes aromatiques
th_27034.4> J'achete 1 preservatif
th_27034.4> J'achete 1 cereales
th_27034.4> J'ai fini mon tour
Detachement du thread
4. Nouveau client ! Bonjour !th_27034.4> Attente a la caisse de 5 secondes ...
th_27034.6> ... je compte acheter 4 articles
th_27034.6> J'achete 1 chips
th_27034.6> J'achete 1 cereales
th_27034.6> J'achete 1 whisky
th_27034.6> J'achete 1 preservatif
th_27034.6> J'ai fini mon tour
th_27034.6> Attente a la caisse de 5 secondes ...
Thread secondaire lance !
Detachement du thread
5. Nouveau client ! Bonjour !Thread secondaire lance !
th_27034.7> ... je compte acheter 5 articles
!!!!!!! Au voleur !!!!!!!!
th_27034.1> !-! Le thread : 7 : Dehors !!!
```

```
== C'était le dernier client ==  
th_27034.7> J'achete 1 preservatif  
th_27034.7> J'achete 1 chips  
th_27034.7> J'achete 1 whisky  
th_27034.7> J'achete 1 herbes aromatiques  
th_27034.7> J'achete 1 cereales  
th_27034.7> J'ai fini mon tour  
th_27034.7> Attente a la caisse de 0 secondes ...  
th_27034.7> ... je passe à la caisse ...  
th_27034.5> Fin du thread client - il va sortir  
th_27034.5> ... je passe à la caisse ...  
th_27034.3> Fin du thread client - il va sortir  
th_27034.3> ... je passe à la caisse ...  
th_27034.4> Fin du thread client - il va sortir  
th_27034.4> ... je passe à la caisse ...  
th_27034.6> Fin du thread client - il va sortir  
th_27034.6> ... je passe à la caisse ...  
th_27034.2> Fin du thread client - il va sortir  
th_27034.2> ... je passe à la caisse ...  
!-!-! Notre magasin ferme ses portes !-!-!  
th_27034.1> !-! Nous vous laissons 10 secondes ...  
Killed  
copernic.inpres.epl.prov-liege.be>
```

Ici, l'exécution a été très rapide, plus chaotique, et aucun client n'a vu trop grand, si bien que tous les clients ont terminé leurs achats avant la fermeture. Ce qui nous indique que :

Il est extrêmement dangereux et irraisonné de baser l'exactitude d'un programme multithread sur des arguments de temps et/ou de scheduling : la synchronisation des événements doit donc être également programmée.



Parole d'évangile, de veda, du ou des prophète(s), du Guide, du Sage, etc. Nous allons donc l'appliquer.

IV. La synchronisation des threads



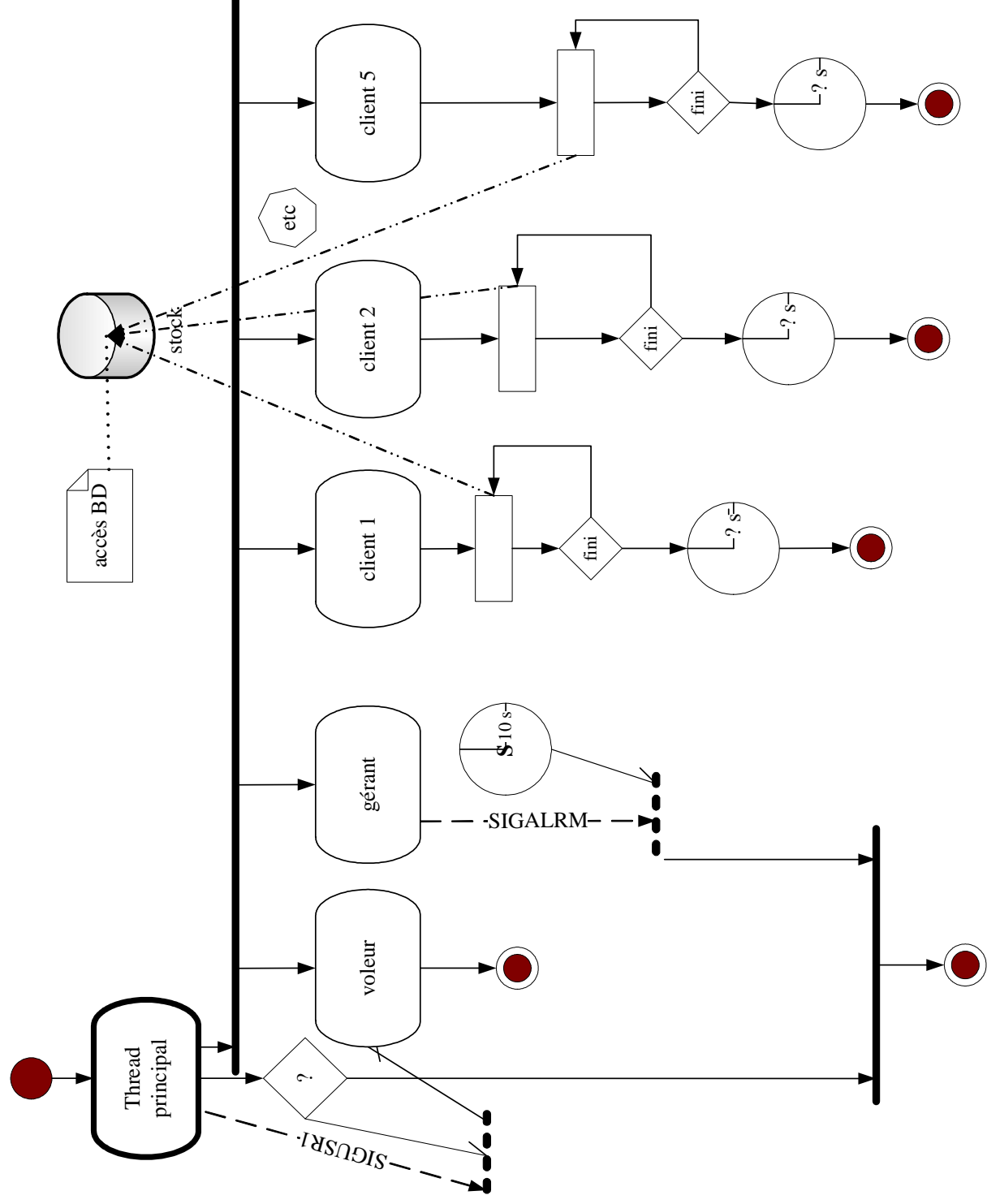
*Hâtez-vous lentement, et sans perdre courage
Vingt fois sur le métier remettez votre ouvrage.
Polissez-le sans cesse et le repolissez.*

(N. Boileau, L'Art poétique)

1. Un premier pas vers la synchronisation : l'attente de la fin d'un thread

Le gérant de notre supermarché n'a pas une identité réellement propre, puisqu'il correspond au thread principal. Nous allons donc le faire correspondre à un thread particulier. Ce sera ce thread qui devra réagir au signal SIGALRM, les autres threads (donc les clients) ignorant ce signal (leur masque de signaux sera configuré pour qu'il en soit ainsi). Le point crucial, dans cette nouvelle structure de notre programme, est que le thread principal attendra que le thread du gérant du supermarché se termine, marquant la fin l'application. Cette attente se programmera au moyen de la fonction `pthread_join`, déjà évoquée. Les thread clients ne seront plus détachés (un `join()` sur ces threads clients pourrait donc éventuellement être envisagé).

Le schéma suivant récapitule tout cela. Comme on l'a déjà dit, l'accès au stock devrait, pour bien faire, être un accès à une base de données véritable ...



Notre programme avec un gérant à l'identité reconnue sera :

SUPERMARCHE06.C

```
/* SUPERMARCHE06.C
Claude Vilvens
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sched.h>

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define N_MAX_CLIENTS 6
#define random(n) (rand()%(n)+1)

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
void handlerSignalAlarme(int sig);
void handlerVoleur (int sig);
void * fctThreadGerant (int * param);
char * getThreadIdentity();
char exists(int n, int tab[], int dim);

pthread_t threadHandle;
pthread_t threadHandleVoleur;
pthread_t threadHandleGerant;
struct sigaction sigAct;

/* ----- */
/* Pour les articles du magasin */
...
/* ----- */
int nbClientEntres = 0;
char ouvert=0;
int *retThread;

int main()
{
    int ret;
    char * buf = (char *)malloc(80), rep, nouveauClient;

    /* Initialisation */
    ...

    sigAct.sa_handler = handlerVoleur;
    if ( (ret=sigaction(SIGUSR1, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction sur SIGUSR1");
```

```
/* Création du thread gérant */
ret = pthread_create(&threadHandleGerant, NULL,
                    (void (*)(void*))fctThreadGerant,0);
puts("Thread gerant lance !");
/* Création des clients */
while (ouvert && nbClientEntres<N_MAX_CLIENTS)  { ... }

pthread_join(threadHandleGerant, (void **)&retThread);
printf("Valeur renvoyee par le thread gerant = %d\n", *retThread);

puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char *buf = (char*)malloc(80);
    int nbArt, i=0, ancEtat;
    int numAch[NB_ART_DIFF];

    char * numThr = getThreadIdentity();
    char fini;
    int nEssai = 0;
    struct sigaction sigAct;
    sigset_t mask;

    pthread_cleanup_push(fctThreadClientFin,0);

    /* Les threads clients ne sont pas concernés par SIGALRM */
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_SETMASK, &mask, NULL);

    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &ancEtat))
        puts("Erreur de setcanceltype");

    pthread_testcancel();
    if (ouvert)
    {
        ...
        if (ouvert) sprintf(buf, "J'ai fini mon tour");
        else sprintf(buf, "Tant pis ! Je n'ai plus le temps");
        affTthread(numThr, buf);
        ...
    }
    else affTthread(numThr, "Le magasin est ferme - trop tard :-(");
}
```

```
    sprintf(buf, "Fin du thread client - il va sortir");
    affThread(numThr, buf);

    pthread_cleanup_pop(1);
    pthread_exit(0);

    return 0;
}

void fctThreadClientFin (void *p) {...}

void handlerVoleur (int sig) {...}

void * fctThreadGerant (int * param)
{
    struct sigaction sigAct;
    timespec_t temps;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    int ancEtat;

    sprintf(buf, "!! Le gerant est dans le magasin");
    affThread(numThr, buf);
    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");

    sigAct.sa_handler = handlerSignalAlarme;
    sigaction(SIGALRM, &sigAct, NULL);
    alarm(25);
    sprintf(buf, "!! Le gerant a enclenche son chrono");
    affThread(numThr, buf);

    while(ouvert);           /* il joue ? ... */
    return 0;
}

void handlerSignalAlarme(int sig)
{
    timespec_t temps;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    int i;

    sprintf(buf, "!!-!! Notre magasin ferme ses portes !!-!!");
    affThread(numThr, buf);
    ouvert=0;

    temps.tv_sec = 10;
```

```

    temps.tv_nsec = 0;
    sprintf(buf, "!! Nous vous laissons %d secondes ...", temps.tv_sec);
    affThread(numThr, buf);
    nanosleep(&temps, NULL);

    sprintf(buf, " Le gerant est parti !!!");
    affThread(numThr, buf);
    pthread_exit(&ouvert);
    return;
}

char * getThreadIdentity() { ... }

char exists(int n, int tab[], int dim) { ... }

```

Une exécution du programme sur sunray donne :

```

th_16049.1> Thread principal démarre
Ouvrir le magasin ?0. Nouveau client ! Bonjour !
Thread secondaire lance !
1. Nouveau client ! Bonjour !
Thread secondaire lance !
2. Nouveau client ! Bonjour !
Thread secondaire lance !
th_16049.4> ... je compte acheter 6 articles
th_16049.4> J'achete 1 preservatif
th_16049.4> J'achete 1 herbes aromatiques
th_16049.4> J'achete 1 chips
th_16049.4> J'achete 1 cereales
th_16049.4> J'achete 1 whisky
!!!!!!! Au voleur !!!!!!!!
th_16049.1> !-! Le thread : 6 : Dehors !!!
3. Nouveau client ! Bonjour !
Thread secondaire lance !
4. Nouveau client ! Bonjour !
Thread secondaire lance !
!!!!!!! Au voleur !!!!!!!!
th_16049.1> !-! Le thread : 8 : Dehors !!!
5. Nouveau client ! Bonjour !
Thread secondaire lance !
!!!!!!! Au voleur !!!!!!!!
th_16049.1> !-! Le thread : 9 : Dehors !!!
== C'était le dernier client ==
Thread gerant lance !
th_16049.5> ... je compte acheter 5 articles
th_16049.5> J'achete 1 chips
th_16049.5> J'achete 1 preservatif
th_16049.5> J'achete 1 herbes aromatiques
th_16049.5> J'achete 1 whisky

```



```
th_16049.5> J'achete 1 cereales
th_16049.5> J'ai fini mon tour
th_16049.5> Attente a la caisse de 2 secondes ...
th_16049.5> Fin du thread client - il va sortir
th_16049.5> ... je passe à la caisse ...
th_16049.6> ... je passe à la caisse ...
th_16049.7> ... je compte acheter 4 articles
th_16049.7> J'achete 1 preservatif
th_16049.7> J'achete 1 cereales
th_16049.7> J'achete 1 chips
th_16049.7> J'achete 1 herbes aromatiques
th_16049.7> J'ai fini mon tour
th_16049.7> Attente a la caisse de 1 secondes ...
th_16049.7> Fin du thread client - il va sortir
th_16049.7> ... je passe à la caisse ...
th_16049.8> ... je passe à la caisse ...
th_16049.9> ... je passe à la caisse ...
th_16049.10> !! Le gerant est dans le magasin
th_16049.10> !! Le gerant a enclenche son chrono
th_16049.10> !!-!! Notre magasin ferme ses portes !!-!!
th_16049.10> !! Nous vous laissons 10 secondes ...
th_16049.4> Tant pis ! Je n'ai plus le temps
th_16049.4> Attente a la caisse de 3 secondes ...
th_16049.4> Fin du thread client - il va sortir
th_16049.4> ... je passe à la caisse ...
th_16049.10> Le gerant est parti !!!
Valeur renvoyee par le thread gerant = 0
Fin du thread principal
```

Il convient de bien remarquer que les threads clients ont bloqué le signal SIGALRM. S'il n'en était pas ainsi, l'un d'entre eux pourrait exécuter le handler *handlerSignalAlarme()*, ne pas exécuter sa fonction de terminaison tandis que le gérant attendrait la fin du client : ce qui conduirait à un blocage ...

2. Les politiques de scheduling et les priorités

Comme le gérant est seul maître à bord, nous pouvons envisager de lui donner la priorité maximale.

1) Ceci peut se faire en définissant un attribut pour le thread qui spécifiera cette priorité. On utilise pour cela la fonction

```
int pthread_attr_setschedparam (<attribut de thread - pthread_attr_t *>,
                                <paramètres de scheduling - const struct sched_param*>);
```



pthread_attr_setschedparam

paramètres

Le deuxième paramètre est une structure, définie dans pthread.h et qui a pour rôle de définir les propriétés d'un thread, dont la priorité. Dans la version la plus simple :

struct sched_param (pthread.h)

```
typedef struct sched_param
{
    int sched_priority;
} sched_param_t;
```

et il suffit donc d'affecter la valeur souhaitée au champ sched_priority. Les niveaux de priorité minimal et maximal sont spécifiés par des constantes définies dans pthread.h :

```
#if (defined _PTHREAD_ENV_UNIX) && (defined _OSF_SOURCE)
# define PRI_FIFO_MIN 14
# define PRI_FIFO_MAX SCHED_PRIO_RT_MAX
# define PRI_RR_MIN 14
# define PRI_RR_MAX SCHED_PRIO_RT_MAX
# define PRI_OTHER_MIN 14
# define PRI_OTHER_MAX SCHED_PRIO_RT_MAX
...
#elif defined _PTHREAD_ENV_WIN32
# define PRI_RR_MIN THREAD_PRIORITY_ABOVE_NORMAL
...
```

Elles dépendent manifestement de la politique de scheduling suivie (voir plus loin).



pthread_attr_setschedparam

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : la valeur spécifiée par le paramètre n'est pas valide

#define ENOTSUP	81	/* Not supported */ : la valeur du paramètre est valide mais non supportée par le système hôte
-----------------	----	--

2) Mais pour définir cette priorité, il faut d'abord définir une politique de scheduling (encore, bien sûr, qu'il existe une politique par défaut). Cette politique se définit en construisant un attribut de thread par :

```
int pthread_attr_setschedpolicy (<attribut de thread - pthread_attr_t *>,
                                <politique de scheduling – int>);
```



pthread_attr_setschedpolicy

paramètres

Le deuxième paramètre est un entier qui peut prendre les valeurs symboliques définies dans pthread.h :

```
#ifndef _PTHREAD_ENV_UNIX
# define SCHED_FIFO 1          /* Digital UNIX sched.h defines */
# define SCHED_RR 2           /* these constants already */
# define SCHED_OTHER 3
# define SCHED_FG_NP SCHED_OTHER /* "Foreground" (Timeshare) */
# define SCHED_BG_NP (SCHED_OTHER+1) /* "Background" */
# define SCHED_LFI_NP (SCHED_OTHER+2) /* "Low FIFO" (background FIFO) */
# define SCHED_LRR_NP (SCHED_OTHER+3) /* "Low RR" (background RR) */
#endif
```

Les valeurs les plus courantes et les plus intéressantes sont :

◆ SCHED_FIFO

Le thread ayant la plus haute priorité reçoit la main jusqu'à ce qu'il soit bloqué. Un thread du groupe des threads les plus prioritaires est alors activé jusqu'à ce qu'il se bloque à son tour. Si un thread avec une priorité supérieure devient prêt à être exécuté, il préempte le thread actif et prend la main.

◆ SCHED_RR (Round_Robin)

Le thread ayant la plus haute priorité reçoit la main jusqu'à ce qu'il soit bloqué mais les autres threads ayant cette même haute priorité reçoivent une part de temps processeurs (time-slice). Si un thread avec une priorité supérieure devient prêt à être exécuté, il préempte le thread actif et prend la main.

◆ SCHED_FG_NP (ou SCHED_OTHER),

C'est la valeur par défaut. Tous les threads, quelle que soit leur priorité, reçoivent une part de temps. Mais les threads de plus haute priorité reçoivent une plus grande part de temps que les autres. Aucun thread ne se voit donc refuser l'exécution alors que c'est possible avec les deux politiques précédentes.

◆ SCHED_BG_NP (Background)

Elle est analogue à la politique par défaut, mais les threads en tâche de fond reçoivent moins de temps.

**pthread_attr_setschedpolicy****erreurs**

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : la valeur spécifiée par le paramètre ne correspond pas à une police valide

3) Selon la politique retenue, on dispose alors des symboles définissant les niveaux extrêmes de priorité. Citons :

SCHED_FIFO : PRI_FIFO_MIN et PRI_FIFO_MAX
SCHED_RR : PRI_RR_MIN et PRI_RR_MAX
SCHED_OTHER : PRI_OTHER_MIN et PRI_OTHER_MAX

4) Pour définir la politique de scheduling, il faut encore au préalable renoncer à celle qui a été donnée par défaut au thread au moment de sa création; en effet, un thread hérite à priori de la politique du thread qui le crée. Pour qu'il n'en soit pas ainsi, et permettre ainsi une définition propre, il faut appeler la primitive :

```
int pthread_attr_setinheritsched (<attribut de thread - pthread_attr_t *>,
                                  <flag d'héritage int>);
```

**pthread_attr_setinheritsched****paramètres**

Le deuxième paramètre est un entier qui peut prendre les valeurs symboliques définies dans pthread.h :

```
#define PTHREAD_INHERIT_SCHED 0
#define PTHREAD_EXPLICIT_SCHED 1
```

C'est évidemment la deuxième valeur qu'il faut spécifier pour préciser que le thread n'hérite pas de la politique de son créateur.

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : la valeur spécifiée par l'un ou les deux paramètre(s) n'est pas valide
#define ENOTSUP 81	/* Not supported */ : la valeur d'un paramètre est valide mais non supportée par le système hôte

5) En résumé, si l'on désire confectionner un attribut de thread avec une priorité maximale pour un scheduling de type Round-Robin, on programmera :

```
pthread_attr_t attrThrPrincipal;
struct sched_param schedP;

pthread_attr_setinheritsched(&attrThrPrincipal, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attrThrPrincipal, SCHED_RR);

schedP.sched_priority = PRI_RR_MAX;
pthread_attr_setschedparam(&attrThrPrincipal, &schedP);

ret = pthread_create(&threadHandleGerant, &attrThrPrincipal,
    (void (*)(void *))fctThreadGerant,0);
```

Remarque

Insistons bien sur le fait que :

Il ne peut être question d'user de changements de priorité de certains thread pour prétendre assurer une exclusion mutuelle à des données partagées. Il suffit d'ailleurs de penser à ce qui se passerait si la machine disposait de plusieurs processeurs : les threads pourraient très bien être exécutés simultanément, sans que leur priorité doive être évaluée ...

Et puisque nous parlons d'exclusion mutuelle ...

3. Les mutex pour threads

La possibilité d'accès concurrents des différents threads clients est devenu évidente depuis qu'ils achètent effectivement des articles du magasin : on conçoit bien que le fait de prendre une ou plusieurs unités d'un article donné doit se faire de manière atomique, sans quoi l'on risque de vendre le même article deux fois. De plus, le thread responsable du stock (que nous inventerons pour l'occasion) doit pouvoir refournir ce stock de manière atomique également.

La norme POSIX définit des mutex spécialement conçus pour les threads et permettant, bien sûr, de réaliser des sections critiques. Ces mutex sont des objets du type **pthread_mutex_t**, qui est en fait une structure contenant les informations sur le mutex considéré. On trouve dans pthread.h :

```
structure pthread_mutex_t
typedef volatile struct __pthread_mutex_t {
    unsigned int    lock;          /* LOCK, SLOW, TYPE, REFCNT */
    unsigned int    valid;         /* Validation info */
    __pthreadLongString_t name;    /* Name of mutex */
    unsigned int    arg;           /* printf argument for name */
    unsigned int    depth;         /* Recursive lock depth */
    unsigned long    sequence;     /* Mutex sequence number */
    unsigned long    owner;        /* Current owner (if known) */
    __pthreadLongAddr_p block;    /* Pointer to blocking struct */
} pthread_mutex_t;
```

A un tel mutex peut correspondre des attributs, qui sont fixés au moyen d'un ensemble d'attributs du type `pthread_mutexattr_t`; on peut utiliser pour ce faire la fonction `pthread_mutexattr_settype_np()`. Le mutex est créé, avec les caractéristiques désirées, au moyen de la fonction :

```
int pthread_mutex_init (<le mutex proprement dit - pthread_mutex_t *>,
                        <les attributs du mutex - const pthread_mutexattr_t *>);
```



pthread_mutex_init

paramètres

Pour un mutex usant de la configuration par défaut, il suffit d'utiliser `NULL` pour le deuxième paramètre. La norme récente propose pour ce faire la constante `pthread_mutexattr_default`.



pthread_mutex_init

erreurs

<i>valeur de retour</i>		<i>erreur</i>
0		succès
#define EAGAIN	35	/* Operation would block */ : le nombre maximum de mutex est atteint
#define EINVAL	22	/* Invalid argument */ : le paramètre mutex n'est pas valide
#define ENOMEM	12	/* Not enough core */ : comme d'habitude : l'espace mémoire est insuffisant
#define EPERM	1	/* Not owner */ : "pas propriétaire" pour UNIX, "pas la permission" pour d'autres OS ;-) ...
#define EBUSY	16	/* Mount device busy */ - ce mutex a déjà été initialisé

Les opérations P et V (c'est-à-dire de prise et de libération) sont réalisées au moyen des deux fonctions suivantes :

```
int pthread_mutex_lock (<le mutex à verrouiller si possible - pthread_mutex_t *>);
```

Cette tentative est **bloquante** : l'activité attend l'opération V correspondante.



pthread_mutex_lock

erreurs

<i>valeur de retour</i>		<i>erreur</i>
0		succès
#define EINVAL	22	/* Invalid argument */ : le paramètre mutex n'est pas valide
#define EDEADLK	11	/* Operation would cause deadlock */ A votre avis ? – ben oui : c'est un deadlock !

L'opération V est provoquée par :

```
int pthread_mutex_unlock (<le mutex à libérer - pthread_mutex_t *>);
```



pthread_mutex_unlock

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre mutex n'est pas valide
#define EPERM 1	/* Not owner */ : pas propriétaire ...

On peut programmer une opération de tentative de prise de sémaphore *non bloquante* en utilisant la fonction :

```
int pthread_mutex_trylock (<le mutex que l'on tente de prendre - pthread_mutex_t *>);
```



pthread_mutex_trylock

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre mutex n'est pas valide
#define EBUSY 16	/* Mount device busy */ - ce mutex a déjà été pris – on ne l'a donc pas pris

Enfin, on peut libérer les ressources allouées à un mutex par :

```
int pthread_mutex_destroy (<le mutex à supprimer - pthread_mutex_t *>);
```



pthread_mutex_destroy

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre mutex n'est pas valide
#define EBUSY 16	/* Mount device busy */ - ce mutex est en cours d'utilisation

Remarque

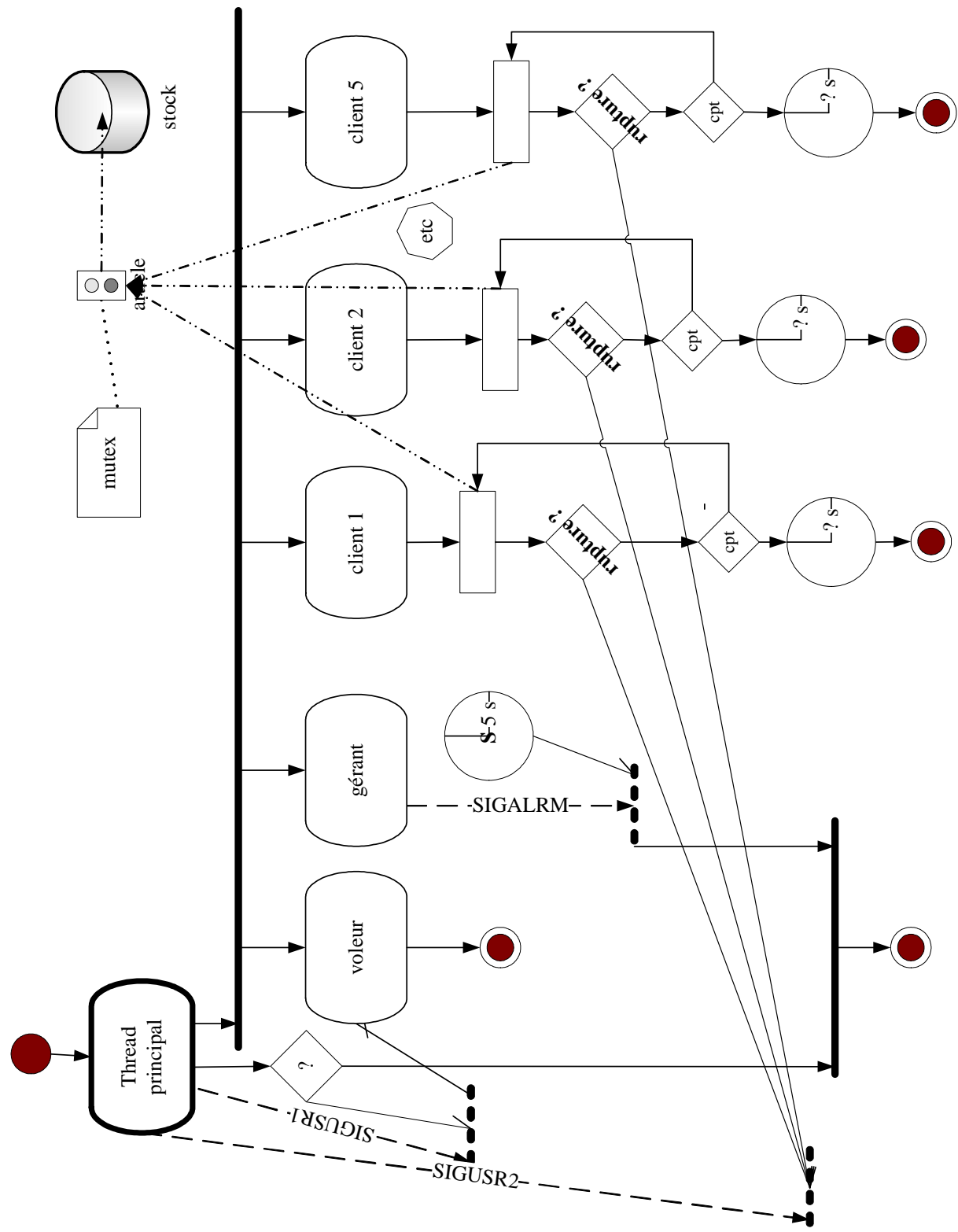
Un mutex statique peut s'initialiser en utilisant une macro PTHREAD_MUTEX_INITIALIZER. Cela peut s'écrire :

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
...
pthread_mutex_unlock(&mutex);
```

4. Le supermarché : une section critique pour prendre un article

Nous allons donc devenir plus réaliste et considérer que le nombre d'articles disponibles n'est pas infini. En clair, nous allons décrémenter le stock au fur et à mesure des achats. Bien sûr, il faut que l'opération de mise à jour du stock constitue avec l'achat une opération atomique; en clair, *il faudra installer une section critique autour de ces opérations*. De plus, une rupture de stock sera détectée (provoquant l'émission du signal SIGUSR2) – pour l'instant, cela ne provoquera aucune réaction du personnel – mais nous y reviendrons (voir paragraphes suivants). Le stock de whisky a été fortement réduit pour provoquer une telle rupture. Tiens, le chocolat avarié a été remplacé et donc est redevenu disponible ;-) ...

Schématiquement :



SUPERMARCHE07.C

```
/* SUPERMARCHE07.C
Claude Vilvens */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sched.h>

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define N_MAX_CLIENTS 6
#define random(n) (rand()%(n))

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
void handlerSignalAlarme(int sig);
void handlerVoleur (int sig);
void handlerRuptureStock (int sig);
void * fctThreadGerant (int * param);
char * getThreadIdentity();
char exists(int n, int tab[], int dim);

pthread_t threadHandle;
pthread_t threadHandleVoleur;
pthread_t threadHandleGerant;
struct sigaction sigAct;
pthread_mutex_t mutexArticle;

/* Pour les articles du magasin */
struct article
{
    int num;
    char libelle[30];
    double prixUnitaire;
    int quStock;
};

#define NB_ART_DIFF 6
#define NB_ART_MAX 10
struct article magasins[] =
{
    { 1, "chocolat", 39, 2300},
    { 2, "whisky", 599, 2},
    { 3, "cereales", 67, 2600},
    { 4, "chips", 20, 5000},
    { 5, "preservatif", 2, 4000},
    { 6, "herbes aromatiques", 67, 3450}
};
```

```
/* ----- */

int nbClientEntres = 0;
char ouvert=0;
int *retThread;

int main()
{
    int ret;
    char * buf = (char *)malloc(80), rep, nouveauClient;
    pthread_attr_t attrThrPrincipal;
    struct sched_param schedP;

    /* Initialisation */
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;

    pthread_mutex_init(&mutexArticle, NULL);
    sigAct.sa_handler = handlerVoleur;
    if ( (ret=sigaction(SIGUSR1, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction sur SIGUSR1");
    sigAct.sa_handler = handlerRuptureStock;
    if ( (ret=sigaction(SIGUSR2, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction sur SIGUSR2");

    /* Création des clients */
    while (ouvert && nbClientEntres<N_MAX_CLIENTS)
    { ... }

    ret = pthread_create(&threadHandleGerant, NULL,(void (*)(void *))fctThreadGerant,0);
    puts("Thread gerant lance !");
    pthread_join(threadHandleGerant, (void **)&retThread);
    printf("Valeur renvoyee par le thread gerant = %d\n", *retThread);

    puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char *buf = (char*)malloc(80);
    int nbArt, i=0, ancEtat;
    int numAch[NB_ART_DIFF];
    float totalAPayer=0;
```

```
char * numThr = getThreadIdentity();
char fini;
int nEssai = 0;
struct sigaction sigAct;
sigset_t mask;

pthread_cleanup_push(fctThreadClientFin,0);

sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
sigprocmask(SIG_SETMASK, &mask, NULL);
...
pthread_testcancel();
if (ouvert)
{
    do
    {
        nbArt = random(NB_ART_DIFF)+1;
    }
    while (nbArt<=0);
    sprintf(buf,"... je compte acheter %d articles differents", nbArt);
    affThread(numThr, buf);
    fini=0;
    do
    {
        int na = random(NB_ART_DIFF);
        if (!exists(na, numAch, NB_ART_DIFF))
        {
            int nba = random(NB_ART_MAX)+1;
            int nbaReel;
            pthread_mutex_lock(&mutexArticle);
            /* pthread_lock_global_np(); */
            if (magasins[na].quStock >= nba)
            {
                numAch[i]=na; i++;
                nbaReel = nba;
                magasins[na].quStock-=nba;
            }
            else
            {
                nbaReel = nba - magasins[na].quStock;
                if (nbaReel<=0) nbaReel=0;
                else
                {
                    numAch[i]=na; i++;
                }
                magasins[na].quStock = 0;
                kill(getpid(), SIGUSR2); /* rupture de stock */
            }
        }
        pthread_mutex_unlock(&mutexArticle);
    }
}
```

```

        /* pthread_unlock_global_np(); */
        if (nbaReel > 0)
        {
            sprintf(buf, "J'achete %d %s", nbaReel, magasins[na].libelle);
            affThread(numThr, buf);
            totalAPayer += nbaReel*magasins[na].prixUnitaire;
        }
    }
    nEssai++;
    pthread_testcancel();
    fini = i==nbArt || ouvert==0;
}
while (!fini);
if (ouvert) sprintf(buf, "J'ai fini mon tour");
else sprintf(buf, "Tant pis ! Je n'ai plus le temps");
affThread(numThr, buf);
temps.tv_sec = rand()/5000; temps.tv_nsec = 0;

sprintf(buf, "Attente a la caisse de %d secondes ...",
        temps.tv_sec); affThread(numThr, buf);
nanosleep(&temps, NULL);
}
else affThread(numThr, "Le magasin est ferme – trop tard :-(");

sprintf(buf, "Fin du thread client - il va sortir");
affThread(numThr, buf);

pthread_cleanup_pop(1);
pthread_exit(0);
return 0;
}

void fctThreadClientFin (void *p) { ... }

void handlerVoleur (int sig) { ...

void handlerRuptureStock (int sig)
{
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    sprintf(buf, ":-) Rupture de stock");
    affThread(numThr, buf);
}

void * fctThreadGerant (int * param)
{
    struct sigaction sigAct;
    timespec_t temps;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();

```

```

    int ret, ancEtat;

    sprintf(buf, "!! Le gerant est dans le magasin");
    affThread(numThr, buf);

    sigAct.sa_handler = handlerSignalAlarme;
    sigaction(SIGALRM, &sigAct, NULL);
    alarm(25);
    sprintf(buf, "!! Le gerant a enclenche son chrono");
    affThread(numThr, buf);

    while(ouvert); /* il joue ... */
    return 0;
}

void handlerSignalAlarme(int sig)
{
    timespec_t temps;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    int i;

    sprintf(buf, "!!-!! Notre magasin ferme ses portes !!-!!");
    affThread(numThr, buf);
    ouvert=0;

    temps.tv_sec = 10; temps.tv_nsec = 0;
    sprintf(buf, "!!-! Nous vous laissons %d secondes ...", temps.tv_sec);
    affThread(numThr, buf);
    nanosleep(&temps, NULL);

    sprintf(buf, " Le gerant est parti !!!");
    affThread(numThr, buf);
    pthread_exit(&ouvert);
    return;
}

char * getThreadIdentity() { ... }

char exists(int n, int tab[], int dim) { ... }

```

Sur boole :

```

boole.inpres.epl.prov-liege.be> s
th_18953.1> Thread principal démarre
Ouvrir le magasin ?O
0. Nouveau client ! Bonjour !Thread secondaire lance !
1. Nouveau client ! Bonjour !Thread secondaire lance !
2. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!

```

```
th_18953.1> !-! Le thread : 4 : Dehors !!!
3. Nouveau client ! Bonjour !Thread secondaire lance !
4. Nouveau client ! Bonjour !Thread secondaire lance !
5. Nouveau client ! Bonjour !Thread secondaire lance !
== C'était le dernier client ==
Thread gerant lance !
th_18953.2> ... je compte acheter 5 articles differents
th_18953.2> J'achete 5 cereales
th_18953.2> J'achete 5 herbes aromatiques
th_18953.2> J'achete 2 preservatif
th_18953.2> J'achete 2 whisky
th_18953.2> J'achete 9 chocolat
th_18953.2> J'ai fini mon tour
th_18953.2> Attente a la caisse de 3 secondes ...
th_18953.3> ... je compte acheter 4 articles differents
th_18953.3> J'achete 8 chips
th_18953.3> J'achete 6 cereales
th_18953.3> J'achete 3 herbes aromatiques
th_18953.3> J'achete 4 preservatif
th_18953.3> J'ai fini mon tour
th_18953.3> Attente a la caisse de 1 secondes ...
th_18953.4> ... je compte acheter 5 articles differents
th_18953.4> :-) Rupture de stock
th_18953.4> J'achete 8 whisky
th_18953.4> J'achete 10 preservatif
th_18953.4> J'achete 10 herbes aromatiques
th_18953.4> J'achete 7 cereales
th_18953.4> J'achete 7 chocolat
th_18953.4> J'ai fini mon tour
th_18953.4> Attente a la caisse de 4 secondes ...
th_18953.4> ... je passe à la caisse ...
th_18953.5> ... je compte acheter 2 articles differents
th_18953.5> J'achete 3 herbes aromatiques
th_18953.5> :-) Rupture de stock
th_18953.5> J'achete 7 whisky
th_18953.5> J'ai fini mon tour
th_18953.5> Attente a la caisse de 1 secondes ...
th_18953.6> ... je compte acheter 2 articles differents
th_18953.6> :-) Rupture de stock
th_18953.6> J'achete 2 whisky
th_18953.6> J'achete 1 herbes aromatiques
th_18953.6> J'ai fini mon tour
th_18953.6> Attente a la caisse de 4 secondes ...
th_18953.7> ... je compte acheter 4 articles differents
th_18953.7> :-) Rupture de stock
th_18953.7> J'achete 4 whisky
th_18953.7> J'achete 10 cereales
th_18953.7> J'achete 5 chips
th_18953.7> J'achete 7 herbes aromatiques
th_18953.7> J'ai fini mon tour
```

```
th_18953.7> Attente a la caisse de 1 secondes ...
th_18953.8> !-! Le gerant est dans le magasin
th_18953.8> !-! Le gerant a enclenche son chrono
th_18953.3> Fin du thread client - il va sortir
th_18953.3> ... je passe à la caisse ...
th_18953.5> Fin du thread client - il va sortir
th_18953.5> ... je passe à la caisse ...
th_18953.7> Fin du thread client - il va sortir
th_18953.7> ... je passe à la caisse ...
th_18953.2> Fin du thread client - il va sortir
th_18953.2> ... je passe à la caisse ...
th_18953.6> Fin du thread client - il va sortir
th_18953.6> ... je passe à la caisse ...
th_18953.8> !-!-! Notre magasin ferme ses portes !-!-!
th_18953.8> !-! Nous vous laissons 10 secondes ...
th_18953.8> Le gerant est parti !!!
Valeur renvoyee par le thread gerant = 0
Fin du thread principal
boole.inpres.epl.prov-liege.be>
```

On remarquera que le traitement du signal SIGUSR2 est assuré par un thread quelconque (celui qui a la main). Sur sunray, on obtient pour le même programme :

```
sunray2v440.inpres.epl.prov-liege.be>cc -o thr7 SuperMarche07.c -mt -lpthread -lrt
sunray2v440.inpres.epl.prov-liege.be> thr7
th_17271.1> Thread principal demarre
Ouvrir le magasin ?O
0. Nouveau client ! Bonjour !
Thread secondaire lance !
1. Nouveau client ! Bonjour !
Thread secondaire lance !
2. Nouveau client ! Bonjour !
th_17271.5> ... je compte acheter 6 articles differents
th_17271.6> ... je compte acheter 4 articles differents
th_17271.6> J'achete 7 chips
th_17271.6> J'achete 9 cereales
th_17271.6> J'achete 2 herbes aromatiques
th_17271.6> J'achete 7 preservatif
th_17271.6> J'ai fini mon tour
th_17271.6> Attente a la caisse de 6 secondes ...
Thread secondaire lance !
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!
th_17271.1> !-! Le thread : 6 : Dehors !!!
3. Nouveau client ! Bonjour !
th_17271.6> ... je passe à la caisse ...
th_17271.5> J'achete 10 herbes aromatiques
th_17271.5> J'achete 3 cereales
th_17271.5> J'achete 4 preservatif
th_17271.5> J'achete 7 chips
th_17271.7> ... je compte acheter 4 articles differents
```

Thread secondaire lance !
th_17271.4> ... je compte acheter 4 articles differents
4. Nouveau client ! Bonjour !
th_17271.5> :-) Rupture de stock
th_17271.5> J'achete 5 whisky
th_17271.8> ... je compte acheter 6 articles differents
th_17271.8> J'achete 5 chips
th_17271.8> J'achete 5 preservatif
th_17271.8> :-) Rupture de stock
th_17271.8> J'achete 2 whisky
th_17271.8> J'achete 4 cereales
th_17271.8> J'achete 5 herbes aromatiques
th_17271.4> J'achete 4 preservatif
th_17271.4> J'achete 3 chips
th_17271.4> J'achete 8 herbes aromatiques
th_17271.4> J'achete 10 cereales
th_17271.4> J'ai fini mon tour
th_17271.4> Attente a la caisse de 2 secondes ...
th_17271.4> Fin du thread client - il va sortir
th_17271.4> ... je passe à la caisse ...
Thread secondaire lance !
!!!!!!!!!!!! Au voleur !!!!!!!!!!!!!
th_17271.1> !-! Le thread : 8 : Dehors !!!
5. Nouveau client ! Bonjour !
th_17271.8> ... je passe à la caisse ...
th_17271.7> J'achete 3 herbes aromatiques
th_17271.7> J'achete 7 cereales
Thread secondaire lance !
== C'était le dernier client ==
th_17271.9> ... je compte acheter 5 articles differents
th_17271.7> :-) Rupture de stock
Thread gerant lance !
th_17271.10> !-! Le gerant est dans le magasin
th_17271.9> J'achete 2 chips
th_17271.10> !-! Armement sur le SIGUSR2
th_17271.9> :-) Rupture de stock
th_17271.10> !-! Le gerant a enclenche son chrono
th_17271.7> J'achete 4 whisky
th_17271.7> J'achete 10 preservatif
th_17271.7> J'ai fini mon tour
th_17271.7> Attente a la caisse de 4 secondes ...
th_17271.7> Fin du thread client - il va sortir
th_17271.7> ... je passe à la caisse ...
th_17271.9> J'achete 9 whisky
th_17271.9> J'achete 7 preservatif
th_17271.9> J'achete 6 herbes aromatiques
th_17271.9> J'achete 8 cereales
th_17271.9> J'ai fini mon tour
th_17271.9> Attente a la caisse de 0 secondes ...
th_17271.9> Fin du thread client - il va sortir

```
th_17271.9> ... je passe à la caisse ...
th_17271.10> !-!! Notre magasin ferme ses portes !-!!
th_17271.10> !-! Nous vous laissons 10 secondes ...
th_17271.5> Tant pis ! Je n'ai plus le temps
th_17271.5> Attente a la caisse de 2 secondes ...
th_17271.5> Fin du thread client - il va sortir
th_17271.5> ... je passe à la caisse ...
th_17271.10> Le gerant est parti !!!
Valeur renvoyee par le thread gerant = 0
Fin du thread principal
```

5. Différents types de mutex

Par défaut, les mutex ainsi utilisés sont ce que l'on appelle des "mutex rapides", ce qui correspond au type représenté par la constante

```
#define PTHREAD_MUTEX_NORMAL 0
```

Ils sont effectivement efficaces et sont caractérisés par le fait qu'un tel mutex peut être verrouillé par un seul thread et, normalement, au maximum une fois. Donc, si un thread verrouille un tel mutex qu'il a déjà verrouillé une fois, il y aura deadlock.

Il existe deux autres types de mutex :

◆ les mutex rékursifs

Un tel mutex peut être verrouillé plusieurs fois par le même thread sans qu'un deadlock se produise : en quelque sorte, le thread possède le mutex. Les verrouillages sont comptabilisés; il faudra donc autant d'appels de unlock() qu'il y a eu de lock() avant que le mutex redevienne disponible pour un autre thread.

Ce genre de mutex trouve son utilité dans le cas de figure où le thread, qui a déjà verrouillé l'accès à une ressource, appelle une autre fonction qui, elle aussi, a été programmée avec un accès verrouillé.

La constante correspondant à ce type de mutex est :

```
#define PTHREAD_MUTEX_RECURSIVE 1
```

◆ les mutex non rékursifs

Ce type de mutex correspond à un mutex rapide mais qui produit une erreur au lieu d'un deadlock lorsqu'un thread tente de verrouiller le mutex déjà verrouillé. A la différence des mutex rapides, il est donc ici possible de tester l'erreur de deadlock. Mais le procédé rend ces mutex lents; on les remplace donc par des mutex rapides passée la phase de mise au point.

La constante correspondant à ce type de mutex est :

```
#define PTHREAD_MUTEX_ERRORCHECK 2
```

Le type d'un mutex se fixe dans ses attributs au moyen de la fonction :

```
int pthread_mutexattr_settype (<attributs du mutex - pthread_mutexattr_t *>,
                               <type du mutex – int>);
```



pthread_mutexattr_settype

erreurs

valeur de retour	erreur
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre attribut n'est pas valide
#define ESRCH 3	/* No such process */ - le mutex visé n'existe pas

De plus, sur les machines Digital Unix mais pas sur les machines Sun Solaris, il existe d'office un *mutex global*, par ailleurs récursif, mis en place par les librairies multi-threads. On le manipule en utilisant les fonctions :

```
int pthread_lock_global_np (void);
int pthread_unlock_global_np (void);
```

Ce mutex est intéressant pour utiliser des fonctions de librairies qui ne sont pas multithreads. De telles fonctions sont en effet susceptibles de ne pas supporter la réentrance. L'utilisation du mutex global évite le problème.

6. Les variables de condition

Comment mettre un thread en sommeil, mais d'un œil seulement puisqu'il devrait être réveillé si un événement se produit ?

6.1 Le principe d'une variable de condition

Ce concept complète celui de *mutex* en apportant la possibilité de *surveiller, de manière atomique, la valeur d'une variable qui reflète l'occurrence ou non d'un certain événement*. La surveillance se terminera lorsque la variable de condition passe à l'état correspondant à la réalisation de l'événement attendu.

Prenons un exemple. Supposons qu'un programme reçoive une liste de noms de clients, noms séparés par un tiret (un peu à la mode des CGI des serveurs WEB). Le programme a pour tâche de rechercher pour chaque nom le coefficient de réduction qui a été consenti à ce client – pour simplifier, on supposera que ces informations sont dans un tableau (en pratique, ce sera au minimum dans un fichier). Afin d'accélérer les choses, chaque nom de client sera traité par un thread. *Le traitement global sera terminé quand tous les noms auront été traités*. Pour savoir quand il en est ainsi, on va évidemment gérer un compteur qui sera :

- ◆ incrémenté quand un thread s'attaque à un nom de client;
- ◆ décrémenté quand le thread a terminé.

Il est évidemment indispensable que l'accès au compteur soit surveillé par un *mutex* qui assurera l'atomicité des opérations ++ et --. L'événement auquel sera associé une *variable de condition* sera le fait que le compteur d'activités retombe à zéro. On peut donc constater que nous avons besoin d'un *couple variable de condition-mutex*.

Pour réaliser un petit programme implémentant cet exemple, il nous faut savoir comment manier les variables de condition.

6.2 Les fonctions d'opérations sur les variables de condition

Les variables de condition sont des objets du type **pthread_cond_t**, qui est en fait une structure contenant les informations sur la variable considérée. On trouve dans pthread.h :

structure pthread_cond_t

```
typedef volatile struct __pthread_cond_t
{
    unsigned int state;          /* EVENT, SLOW, REFCNT */
    unsigned int valid;         /* Validation info */
    __pthreadLongString_t name; /* Name of condition variable */
    unsigned int arg;           /* printf argument for name */
    unsigned long sequence;     /* Condition variable seq # */
    __pthreadLongAddr_p block;  /* Pointer to blocking struct */
} pthread_cond_t;
```

Comme pour les mutex, à une telle variable peut correspondre des attributs, qui sont fixés au moyen d'un ensemble d'attributs du type pthread_condattr_t; on peut utiliser pour ce faire la fonction pthread_condattr_init(). La variable est créée, avec les caractéristiques désirées, au moyen de la fonction :

```
int pthread_cond_init (<la variable de condition proprement dite - pthread_cond_t *>,
                     <les attributs de la variable - const pthread_condattr_t *>);
```



pthread_cond_init

paramètres

Pour un mutex usant de la configuration par défaut, il suffit d'utiliser NULL pour le deuxième paramètre. La norme récente propose pour ce faire la constante pthread_condattr_default.



pthread_cond_init

erreurs

valeur de retour		erreur
0		succès
#define EAGAIN	35	/* Operation would block */ : le nombre maximum de variables-condition est atteint
#define EINVAL	22	/* Invalid argument */ : le paramètre attribut n'est pas valide
#define ENOMEM	12	/* Not enough core */ : comme d'habitude : l'espace mémoire est insuffisant
#define EPERM	1	/* Not owner */ : "pas propriétaire" pour UNIX
#define EBUSY	16	/* Mount device busy */ - cette variable a déjà été initialisée

Une fois que le mutex associé à la variable de condition a pu être verrouillé, la mise en attente sur celle-ci se programme en utilisant la fonction :

```
int pthread_cond_wait (    <la variable de condition – pthread_cond_t *>,
                          <le mutex verrouillé associé- pthread_mutex_t *>);
```

Cette fonction a pour effet que :

- ◆ le mutex associé est **déverrouillé** (heureusement d'ailleurs, sans quoi il serait impossible de modifier ailleurs la variable condition dont l'accès est précisément réglementé par ce mutex);
- ◆ le thread est **bloqué**;
- ◆ lorsque la condition est "signalée", autrement dit lorsqu'une demande de réveil sur la variable de condition est demandée par pthread_cond_signal(), le mutex est acquis par le thread qui reprend son exécution; son premier travail sera de vérifier si l'événement attendu s'est effectivement passé ou pas.

Le paradigme classique du blocage sur une variable de condition est donc :

```
pthread_mutex_lock(&<mutex associé à la variable de condition>);
while (<condition sur la(les) variable(s) reflétant l'événement attendu>)
    pthread_cond_wait( &<variable condition associée à la variable événement>,
                      &<mutex associé à la variable de condition>);
pthread_mutex_unlock(&<mutex associé à la variable de condition>);
```



pthread_cond_wait

erreurs

<i>valeur de retour</i>		<i>erreur</i>
0		succès
#define EINVAL	22	/* Invalid argument */ : le paramètre condition ou le paramètre mutex n'est pas valide
#define ENOMEM	12	/* Not enough core */ : l'espace mémoire est insuffisant pour bloquer de l'espace de manière continue

On peut demander le réveil d'une activité en attente sur la condition par :

```
int pthread_cond_signal (<la variable de condition - pthread_cond_t *>);
```

Le paradigme classique du réveil d'un thread bloqué sur une variable de condition est alors :

```
pthread_mutex_lock(&<mutex associé à la variable de condition>);
<modification de la variable reflétant l'événement attendu>
pthread_mutex_unlock(&<mutex associé à la variable de condition>);
pthread_cond_signal(&<variable condition associée à la variable de condition>);
```

Une seule activité en attente sera effectivement réveillée pour évaluation de la variable condition. Si aucun thread ne surveille cette condition à ce moment, le signal est tout simplement perdu.



pthread_cond_signal

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre condition n'est pas valide

Une variable de condition devenue inutile est détruite par :

```
int pthread_cond_destroy (<la variable à supprimer - pthread_cond_t *>);
```



pthread_cond_destroy

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre condition n'est pas valide
#define EBUSY 16	/* Mount device busy */ : la variable condition fait l'objet d'un autre wait

6.3 Un programme utilisant une variable de condition

Nous pouvons à présent programmer notre programme chercheur de coefficient de réduction. Il s'attend à recevoir une requête constituée des noms des clients cherchés, soit par exemple :

Excellent-Superbe-Sublime-Divin-

Une fois cette chaîne décomposée, un thread sera créé pour chaque nom, avec pour mission de rechercher le nom en question et de fournir la réponse comportant le coefficient de réduction. Histoire de tout mélanger, chaque thread est, tout à fait artificiellement, ralenti avec une mise en sommeil aléatoire.

VARCONDI01.C

```
/* VARCONDI01.C
- Claude Vilvens -
*/

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

```

#include <unistd.h>
#include <signal.h>
#include <string.h> /* pour memcpy */

#define MAXSTRING 1000 /* Longueur des messages */
#define LONG_MAX_NOM 40 /* Longueur maximale des noms */
#define NBRE_MAX_NOMS_CLIENTS 12 /* Nombre maximum de noms de clients */
#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define random(n) (rand()%(n))

struct refClient
{
    char nom[LONG_MAX_NOM];
    float coeffRed;
};

struct refClient fichierClients[] =
{
    {"Excellent", 0.95},
    {"Genial", 0.34},
    {"Tres bon", 0.75},
    {"Superbe", 0.87},
    {"Divin", 0.21},
    {"Inaccessible", 0.45},
    {"Ecoeurant", 0.67},
    {"Un maitre", 0.78},
    {"Extraordinaire", 0.89}
};

const int tailleFichierClients = 9;
int ret;

pthread_t threadHandle, threadPrincipal;
pthread_mutex_t mutexNbreActivites;
pthread_cond_t condNbreActivites;

int nbreNomsClients; /* Ce qui caractérise l'événement */

void * fctThread(void * param);
char * getThreadIdentity();

int main(int argc, char **argv)
{
    int i, cptNomsClients;
    int * retThread;
    char msgClient [MAXSTRING];
    int vr;
    char buf[100];
    char *ptr, *ptrCliTr;
    char ptrCli[40];
    char * nomsClientsRech[NBRE_MAX_NOMS_CLIENTS], *token;

```

```
/* 1. Initialisations */
    puts("Thread principal serveur démarre");
    pthread_mutex_init(&mutexNbreActivites, NULL);
    pthread_cond_init(&condNbreActivites, NULL);

/* 2. Lecture du msgClient */
    if (argc < 2)
    {
        puts("Usage : scan message_a_analyser"); exit(0);
    }
    strcpy(msgClient, argv[1]);
    printf("msgClient = %s\n", msgClient);

/* 3. Decoupe du message client */
    puts("----- Analyse de msgClient");

    cptNomsClients = 0;
    token = strtok(msgClient, "-");
    while (token != NULL)
    {
        nomsClientsRech[cptNomsClients] = malloc(strlen(token)+1);
        strcpy(nomsClientsRech[cptNomsClients], token);
        cptNomsClients++;
        if (cptNomsClients >= NBRE_MAX_NOMS_CLIENTS) break;
        token = strtok((char *)NULL, "-");
    }
    puts("Fin scan msg client");

/* 4. Lancement des threads */
    nbreNomsClients = cptNomsClients;
    for (i=0; i<cptNomsClients; i++)
    {
        puts(nomsClientsRech[i]);
        ret = pthread_create(&threadHandle, NULL, fctThread,
            (void*)nomsClientsRech[i]);
        puts("Thread secondaire lance !");
        ret = pthread_detach(threadHandle);
        puts("Marquage pour effacement du thread secondaire");
    }
    pthread_mutex_lock(&mutexNbreActivites);
    while (nbreNomsClients)
        pthread_cond_wait(&condNbreActivites, &mutexNbreActivites);
    pthread_mutex_unlock(&mutexNbreActivites);

    puts("Fin du thread principal");
    return 0;
}

/* ----- */
```



```

void * fctThread (void *param)
{
    char * nomCli = (char *)param, *buf = (char*)malloc(100);
    int vr = pthread_self(), trouve, i;
    timespec_t temps;
    char * numThr = getThreadIdentity();

    /* 1. Temporisation */
    temps.tv_sec = random(15)+1;
    temps.tv_nsec = 0;
    sprintf(buf, "!! Temps de reflexion de %d secondes ...", temps.tv_sec);
    affThread(numThr, buf);
    nanosleep(&temps, NULL);

    /* 2. Recherche du nom du client */
    trouve = 0;
    for (i=0; i<tailleFichierClients && !trouve; i++)
        trouve = strcmp(nomCli, fichierClients[i].nom) == 0;
    if (trouve)
    {
        sprintf(buf, "%s trouve en %d -> C.R.= %5.3f",
                nomCli, i-1, fichierClients[i-1].coeffRed);
        affThread(numThr, buf);
    }
    else
    {
        sprintf(buf, "%s inconnu", nomCli);
        affThread(numThr, buf);
    }

    /* 3. Signal de fin de traitement */
    pthread_mutex_lock(&mutexNbreActivites);
    nbreNomsClients--;
    pthread_mutex_unlock(&mutexNbreActivites);
    pthread_cond_signal(&condNbreActivites);
    affThread(numThr, "--fin du thread--");

    pthread_exit(&vr);
    return &vr;
}

char * getThreadIdentity() { ... }

```

Sur boole, l'exécution du programme donne :

```

boole.INPRES.EPL.PROV-LIEGE.BE> scan Excellent-Superbe-Sublime-Divin-
Thread principal serveur demarre
msgClient = Excellent-Superbe-Sublime-Divin-
----- Analyse de msgClient
Fin scan msg client

```

```
Excellent
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Superbe
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Sublime
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Divin
Thread secondaire lance !
Marquage pour effacement du thread secondaire
th_7588.2> !-! Temps de reflexion de 9 secondes ...
th_7588.3> !-! Temps de reflexion de 14 secondes ...
th_7588.4> !-! Temps de reflexion de 4 secondes ...
th_7588.5> !-! Temps de reflexion de 11 secondes ...
th_7588.4> Sublime inconnu
th_7588.4> --fin du thread--
th_7588.2> Excellent trouve en 0 -> C.R.= 0.950
th_7588.2> --fin du thread--
th_7588.5> Divin trouve en 4 -> C.R.= 0.210
th_7588.5> --fin du thread--
th_7588.3> Superbe trouve en 3 -> C.R.= 0.870
th_7588.3> --fin du thread--
Fin du thread principal
boole.INPRES.EPL.PROV-LIEGE.BE>
```

Sur sunray, rien de vraiment différent :

```
sunray2v440.inpres.epl.prov-liege.be> scan beau-Excellent-Inaccessible-UnMaitre
Thread principal serveur démarre
msgClient = beau-Excellent-Inaccessible-UnMaitre
----- Analyse de msgClient
Fin scan msg client
beau
Thread secondaire lance !
Excellent
Thread secondaire lance !
Inaccessible
Thread secondaire lance !
UnMaitre
Thread secondaire lance !
th_10756.4> !-! Temps de reflexion de 9 secondes ...
th_10756.5> !-! Temps de reflexion de 14 secondes ...
th_10756.6> !-! Temps de reflexion de 4 secondes ...
th_10756.7> !-! Temps de reflexion de 11 secondes ...
th_10756.6> Inaccessible trouve en 5 -> C.R.= 0.450
th_10756.6> --fin du thread--
th_10756.4> beau inconnu
th_10756.4> --fin du thread--
```

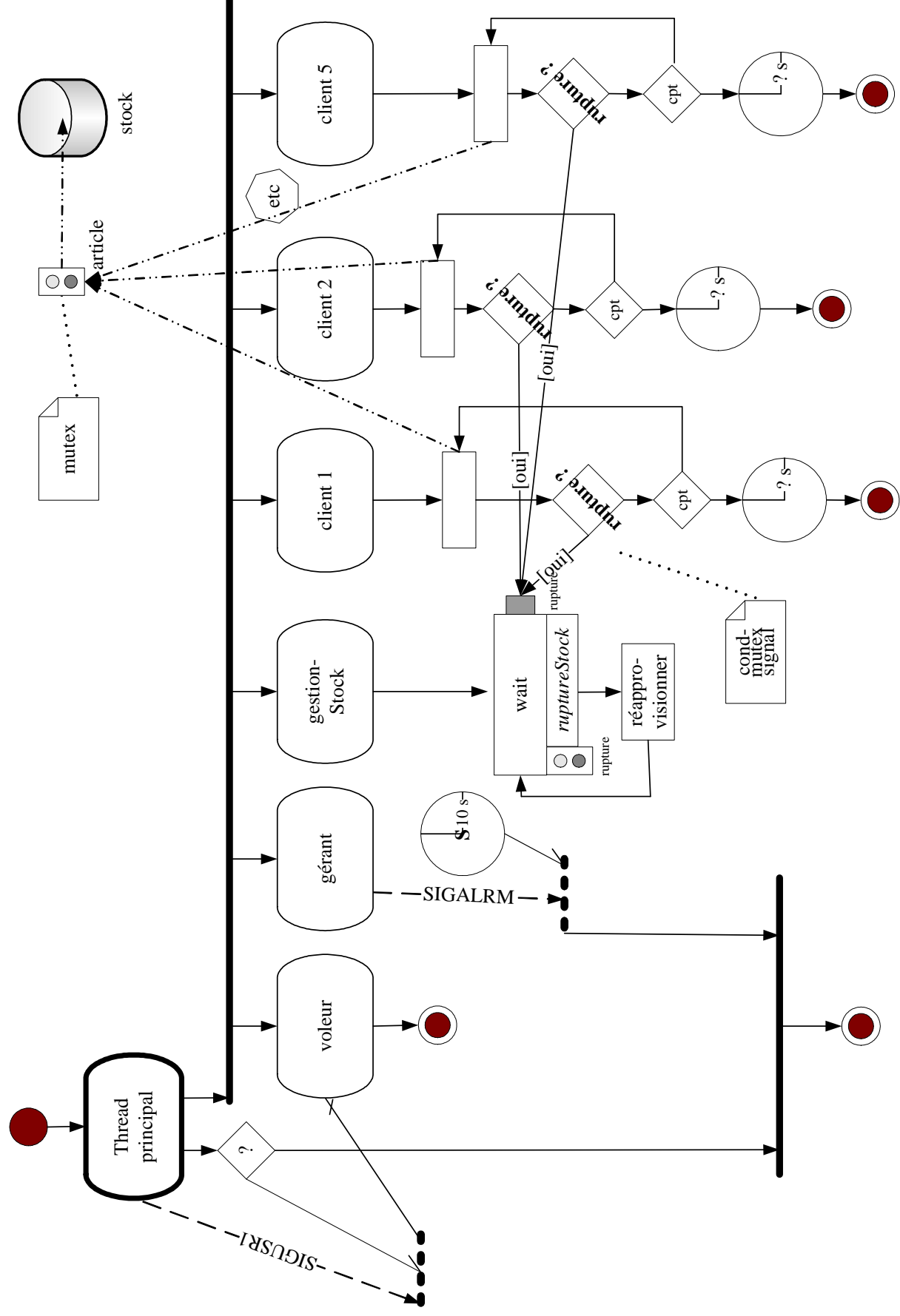
```
th_10756.7> UnMaitre inconnu
th_10756.7> --fin du thread--
th_10756.5> Excellent trouve en 0 -> C.R.= 0.950
th_10756.5> --fin du thread--
Fin du thread principal
sunray2v440.inpres.epl.prov-liege.be>
```

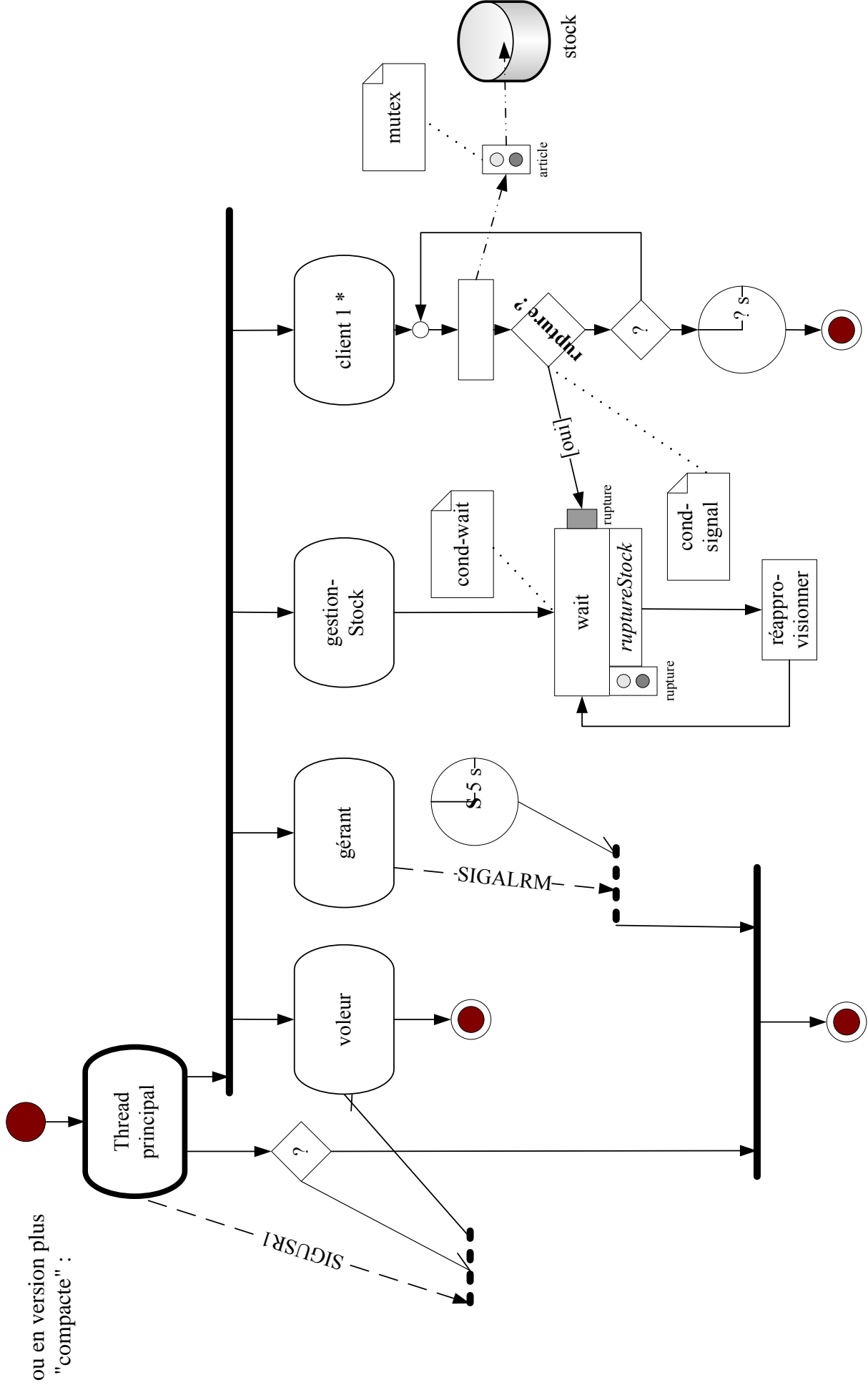
7. Le supermarché gère ses ruptures de stock

Cette fois, les clients vont réellement prendre les articles au risque de provoquer une rupture de stock. Un thread particulier (disons que c'est le magasinier) va se mettre en attente d'une rupture de stock pour y remédier. Pour ce faire, il va utiliser :

- ◆ une variable *ruptureStock* comptant le nombre de ruptures de stock - en fait, de la manière dont nous allons procéder, ce sera plutôt un flag indiquant qu'une rupture de stock s'est produite et que l'on est en train d'y remédier;
- ◆ une variable de condition *condRupture*;
- ◆ un mutex associé *mutexRupture*.

Schématiquement :





Lorsqu'une rupture de stock est en cours de traitement, un client laisse le magasinier faire son travail de réassortiment - en clair, il ne fait plus rien et une autre variable de condition serait sans doute à envisager (mais restons calmes). Ensuite, il serait temps de montrer que les clients paient (pour l'instant, tout semble gratuit ☹ ...) – une variable *totalAPayer* sera utilisée à cet effet. Enfin, afin de tester les ruptures de stock, le stock de whisky et celui de chips a été ramené à une valeur ridicule (pour une nation civilisée ;-) ...).

SUPERMARCHE08.C

```
/* SUPERMARCHE08.C
Claude Vilvens */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sched.h>

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define N_MAX_CLIENTS 6
#define random(n) (rand()%(n))

void * fctThreadClient(int * param);
void fctThreadClientFin (void *p);
void handlerSignalAlarme(int sig);
void handlerVoleur (int sig);
void handlerRuptureStock (int sig);
void * fctThreadGerant (int * param);
void * fctThreadGestionStock (int * param);
char * getThreadIdentity();
char exists(int n, int tab[], int dim);
void sleepThread(int ns);

pthread_t threadHandle;
pthread_t threadHandleVoleur;
pthread_t threadHandleGerant;
pthread_t threadHandleGestionStock;
struct sigaction sigAct;
pthread_mutex_t mutexArticle;
pthread_mutex_t mutexRupture;
pthread_cond_t condRupture;

/* Pour les articles du magasin */
struct article
{
    int num;
    char libelle[30];
    double prixUnitaire;
    int quStock;
};
```

```
#define NB_ART_DIFF 6
#define NB_ART_MAX 10

struct article magasins[] =
{
    { 1, "chocolat", 39, 2300},
    { 2, "whisky", 599, 2},
    { 3, "cereales", 67, 2600},
    { 4, "chips", 20, 9},
    { 5, "preservatif", 2, 4000},
    { 6, "herbes aromatiques", 67, 3450}
};

char ruptureStock = 0;
/* ----- */

int nbClientEntres = 0;
char ouvert=0;
int *retThread;

int main()
{
    int ret;
    char * buf = (char *)malloc(80), rep, nouveauClient;
    pthread_attr_t attrThrPrincipal;
    struct sched_param schedP;

    /* Initialisation */
    affThread(getThreadIdentity(), "Thread principal démarre");
    do
    {
        printf("Ouvrir le magasin ?");gets(buf);rep=buf[0];
    }
    while (rep!='O');
    ouvert=1;

    ret = pthread_create(&threadHandleGerant, NULL, (void (*)(void *))fctThreadGerant,0);
    puts("Thread gerant lance !");
    ret = pthread_create(&threadHandleGestionStock, NULL,
        (void (*)(void *))fctThreadGestionStock,0);
    puts("Thread de gestion du stock lance !");

    pthread_mutex_init(&mutexArticle, NULL);
    pthread_mutex_init(&mutexRupture, NULL);
    pthread_cond_init(&condRupture, NULL);
    ...

    /* Création des clients */
```

```

while (ouvert && nbClientEntres<N_MAX_CLIENTS)
{
    int aleat;
    aleat = rand();
    nouveauClient = aleat%5==0 ;
    if (nouveauClient)
    {
        printf("%d. Nouveau client ! Bonjour !", nbClientEntres++);
        ret = pthread_create(&threadHandle, NULL,
            (void (*)(void *))fctThreadClient,0);
        puts("Thread secondaire lance !");
        if (aleat%20==0)
        {
            threadHandleVoleur = threadHandle; kill (getpid(), SIGUSR1);
        }
    }
    if (nbClientEntres == N_MAX_CLIENTS) puts(" == C'était le dernier client ==");
}

pthread_join(threadHandleGerant, (void **)&retThread);
printf("Valeur renvoyee par le thread gerant = %d\n", *retThread);

puts("Fin du thread principal");
}

void * fctThreadClient (int * param)
{
    timespec_t temps;
    char *buf = (char*)malloc(80);
    int nbArt, i=0, ancEtat;
    int numAch[NB_ART_DIFF];
    int totalAPayer=0;

    char * numThr = getThreadIdentity();
    char fini;
    int nEssai = 0;
    struct sigaction sigAct;
    sigset_t mask;
    ...
    pthread_testcancel();
    if (ouvert)
    {
        do
        {
            nbArt = random(NB_ART_DIFF)+1;
        }
        while (nbArt<=0);
        sprintf(buf,"... je compte acheter %d articles differents", nbArt);
        affThread(numThr, buf);
        fini=0;
    }
}

```



```

do
{
    int na = random(NB_ART_DIFF);
    if (!exists(na, numAch, NB_ART_DIFF))
    {
        int nba = random(NB_ART_MAX)+1;
        int nbaReel;
        pthread_mutex_lock(&mutexArticle);
        if (magasins[na].quStock >= nba)
        {
            numAch[i]=na; i++; nbaReel = nba;
            magasins[na].quStock-=nba;
        }
        else
        {
            if (ruptureStock) continue;
            nbaReel = nba - magasins[na].quStock;
            if (nbaReel<=0) nbaReel=0;
            else
            {
                numAch[i]=na; i++;
            }
            magasins[na].quStock = 0;
            pthread_mutex_lock(&mutexRupture);
            ruptureStock++;
            pthread_mutex_unlock(&mutexRupture);
            pthread_cond_signal(&condRupture);
        }
        pthread_mutex_unlock(&mutexArticle);
        sprintf(buf, "J'achete %d %s", nbaReel, magasins[na].libelle);
        affThread(numThr, buf);
        totalAPayer += nbaReel*(magasins[na].prixUnitaire);
    }
    nEssai++;
    fini = i==nbArt || ouvert==0;
}
while (!fini && ouvert);
if (ouvert) sprintf(buf, "J'ai fini mon tour");
else sprintf(buf, "Tant pis ! Je n'ai plus le temps");
affThread(numThr, buf);
temps.tv_sec = rand()/5000;
temps.tv_nsec = 0;
sprintf(buf, "Attente a la caisse de %d secondes ...", temps.tv_sec);
affThread(numThr, buf);
sprintf(buf, "Je paie a la caisse %d BEF", totalAPayer);
affThread(numThr, buf);
nanosleep(&temps, NULL);
}
else affThread(numThr, "Le magasin est ferme – trop tard :-(");

```

```
    sprintf(buf, "Fin du thread client – il va sortir"); affThread(numThr, buf);

    pthread_exit(0);

    pthread_cleanup_pop(0);

    return 0;
}

void fctThreadClientFin (void *p) {...}

void handlerVoleur (int sig) {...}

void handlerRuptureStock (int sig)
{
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    sprintf(buf, ":-) Rupture de stock");
    affThread(numThr, buf);
}

void * fctThreadGerant (int * param)
{
    struct sigaction sigAct;
    timespec_t temps;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();
    int ret, ancEtat;

    sprintf(buf, "!! Le gerant est dans le magasin");
    affThread(numThr, buf);

    ...
    while(ouvert);          /* il joue ... */
    if (!ouvert) pthread_exit(0);
    return 0;
}

void handlerSignalAlarme(int sig) {...}

void * fctThreadGestionStock (int * param)
{
    int i;
    char *buf = (char *)malloc(80);
    char * numThr = getThreadIdentity();

    while(ouvert)
    {
        pthread_mutex_lock(&mutexRupture);
        while (!ruptureStock)
            pthread_cond_wait(&condRupture, &mutexRupture);
    }
}
```

```

    affThread(numThr, ":-( !! Une rupture de stock detectee");
    for (i=0; i<NB_ART_DIFF; i++)
        if (magasins[i].quStock<=0)
        {
            int nouv = random(10)+20;
            pthread_mutex_lock(&mutexArticle);
            sprintf(buf,":-) !! Je replace %d %s", nouv, magasins[i].libelle);
            sleepThread(5);
            affThread(numThr, buf);
            magasins[i].quStock+=nouv;
            ruptureStock--;
            pthread_mutex_unlock(&mutexArticle);
            sprintf(buf,":-| ruptureStock=%d", ruptureStock);
            affThread(numThr, buf);
        }
        pthread_mutex_unlock(&mutexRupture);
    }
    pthread_exit(0);
    return 0;
}

char * getThreadIdentity() { ... }

char exists(int n, int tab[], int dim){ ... }

void sleepThread(int ns)
{
    timespec_t temps;
    temps.tv_sec = ns; temps.tv_nsec = 0;
    nanosleep(&temps, NULL);
}

```

Observons ce qui se passe lorsque l'on exécute ce programme sur boole. Les décalages ne sont pas réels : ils ont seulement pour but de mettre en correspondance les opérations effectuées par le même client. On pourra ainsi constater qu'ils se terminent tous normalement. Le thread de gestion de stock devra intervenir deux fois.

```

boole.inpres.epl.prov-liege.be> c
th_23151.1> Thread principal démarre
Ouvrir le magasin ?O
Thread gerant lance !
Thread de gestion du stock lance !
0. Nouveau client ! Bonjour !Thread secondaire lance !
1. Nouveau client ! Bonjour !Thread secondaire lance !
2. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!! Au voleur !!!!!!!!
th_23151.1> !-! Le thread : 6 : Dehors !!!
3. Nouveau client ! Bonjour !Thread secondaire lance !
4. Nouveau client ! Bonjour !Thread secondaire lance !
5. Nouveau client ! Bonjour !Thread secondaire lance !

```

== C'était le dernier client ==

th_23151.2> !! Le gerant est dans le magasin

th_23151.2> !! Le gerant a enclenche son chrono

th_23151.4> ... je compte acheter 5 articles differents

th_23151.4> J'achete 5 cereales

th_23151.4> J'achete 5 herbes aromatiques

th_23151.4> J'achete 2 preservatif

th_23151.4> J'achete 2 whisky

th_23151.4> J'achete 4 chips

th_23151.4> J'ai fini mon tour

th_23151.4> Attente a la caisse de 1 secondes ...

th_23151.4> Je paie a la caisse 1952 BEF

th_23151.5> ... je compte acheter 3 articles differents

th_23151.5> J'achete 4 chips

th_23151.5> J'achete 5 cereales

th_23151.5> J'achete 4 preservatif

th_23151.5> J'ai fini mon tour

th_23151.5> Attente a la caisse de 1 secondes ...

th_23151.5> Je paie a la caisse 423 BEF

th_23151.6> ... je passe à la caisse ...

th_23151.7> ... je compte acheter 5 articles differents

th_23151.7> J'achete 8 whisky

th_23151.3> **:- (!! Une rupture de stock detectee**

th_23151.8> ... je compte acheter 2 articles differents

th_23151.9> ... je compte acheter 2 articles differents

th_23151.4> Fin du thread client - il va sortir

th_23151.4> ... je passe à la caisse ...

th_23151.5> Fin du thread client - il va sortir

th_23151.5> ... je passe à la caisse ...

th_23151.3> **:-) !! Je replace 21 whisky**

th_23151.3> **:-| ruptureStock=0**

th_23151.7> J'achete 10 cereales

th_23151.7> J'achete 2 chips

th_23151.9> J'achete 7 preservatif

th_23151.3> **:- (!! Une rupture de stock detectee**

th_23151.3> **:-) !! Je replace 26 chips**

th_23151.3> **:-| ruptureStock=0**

th_23151.7> J'achete 8 preservatif

th_23151.7> J'achete 1 herbes aromatiques

th_23151.7> J'ai fini mon tour

th_23151.7> Attente a la caisse de 5 secondes ...

th_23151.7> Je paie a la caisse 5585 BEF

th_23151.8> J'achete 2 whisky

th_23151.8> J'achete 4 cereales

th_23151.8> J'ai fini mon tour

th_23151.8> Attente a la caisse de 3 secondes ...

th_23151.8> Je paie a la caisse 1466 BEF

th_23151.9> J'achete 10 chips

th_23151.9> J'ai fini mon tour

th_23151.9> Attente a la caisse de 2 secondes ...

```
th_23151.9> Je paie a la caisse 214 BEF
th_23151.9> Fin du thread client - il va sortir
th_23151.9> ... je passe à la caisse ...
th_23151.8> Fin du thread client - il va sortir
th_23151.8> ... je passe à la caisse ...
th_23151.7> Fin du thread client - il va sortir
th_23151.7> ... je passe à la caisse ...
th_23151.2> !-! Notre magasin ferme ses portes !-!
th_23151.2> !-! Nous vous laissons 10 secondes ...
th_23151.2> Le gerant est parti !!!
Valeur renvoyee par le thread gerant = 0
Fin du thread principal
boole.inpres.epl.prov-liege.be>
```

L'exécution du même programme sur copernic ou sunray n'apporte rien de plus. Le lecteur méfiant pourrait cependant objecter que tout se passe bien parce qu'aucun client n'est interrompu dans sa quête par une fermeture prématurée du magasin. Réduisons donc le temps d'ouverture du magasin à 10 secondes. Sur copernic, cela donne :

```
sunray2v440.inpres.epl.prov-liege.be> thr8
th_10595.1> Thread principal démarre
Ouvrir le magasin ? O
Thread gerant lance !
th_10595.4> !-! Le gerant est dans le magasin
Thread de gestion du stock lance !
th_10595.4> !-! Le gerant a enclenché son chrono
0. Nouveau client ! Bonjour !Thread secondaire lance !
1. Nouveau client ! Bonjour !Thread secondaire lance !
2. Nouveau client ! Bonjour !Thread secondaire lance !
!!!!!!! Au voleur !!!!!!!!
th_10595.1> !-! Le thread : 8 : Dehors !!!
3. Nouveau client ! Bonjour !Thread secondaire lance !
4. Nouveau client ! Bonjour !Thread secondaire lance !
5. Nouveau client ! Bonjour !Thread secondaire lance !
== C'était le dernier client ==
th_10595.6> ... je compte acheter 5 articles différents
th_10595.6> J'achète 5 céréales
th_10595.6> J'achète 5 herbes aromatiques
th_10595.6> J'achète 2 conserves
th_10595.6> J'achète 2 whisky
th_10595.6> J'achète 4 chips
th_10595.6> J'ai fini mon tour
th_10595.6> Attente à la caisse de 1 secondes ...
th_10595.6> Je paie à la caisse 1952 BEF
th_10595.6> Fin du thread client - il va sortir
th_10595.6> ... je passe à la caisse ...
th_10595.7> ... je compte acheter 3 articles différents
th_10595.7> J'achète 4 chips
th_10595.7> J'achète 5 céréales
th_10595.7> J'achète 4 conserves
```

```
th_10595.7> J'ai fini mon tour
th_10595.7> Attente a la caisse de 1 secondes ...
th_10595.7> Je paie a la caisse 423 BEF
th_10595.7> Fin du thread client - il va sortir
th_10595.7> ... je passe à la caisse ...
    th_10595.8> ... je passe à la caisse ...
        th_10595.9> ... je compte acheter 5 articles differents
        th_10595.9> J'achete 8 whisky
th_10595.4> !-!! Notre magasin ferme ses portes !-!!
th_10595.4> !-! Nous vous laissons 10 secondes ...
    th_10595.9> Tant pis ! Je n'ai plus le temps
    th_10595.9> Attente a la caisse de 0 secondes ...
    th_10595.9> Je paie a la caisse 4792 BEF
    th_10595.9> Fin du thread client - il va sortir
    th_10595.9> ... je passe à la caisse ...
        th_10595.10> Le magasin est ferme - trop tard :-(
        th_10595.10> Fin du thread client - il va sortir
        th_10595.10> ... je passe à la caisse ...
            th_10595.11> Le magasin est ferme - trop tard :-(
            th_10595.11> Fin du thread client - il va sortir
            th_10595.11> ... je passe à la caisse ...
th_10595.5> :-( !! Une rupture de stock detectee
th_10595.5> :-) !! Je replace 28 whisky
th_10595.5> :-| ruptureStock=0
th_10595.4> Le gerant est parti !!!
Valeur renvoyee par le thread gerant = 0
Fin du thread principal
```

Tout est bien qui finit bien ...



A première vue, les threads vivent quelque peu comme les membres d'une communauté monastique : ils n'ont pas grand' chose en propre (quelques outils) et partagent tout ce qui est important. Mouais ... peut-être ...

V. Des variables statiques spécifiques aux threads



Croyez-le, si voulez; si ne voulez, allez y voir.

(F. Rabelais, Le Quart Livre)

1. Quelque part entre locales et globales

Nous savons déjà que les threads possèdent en propre leurs données **locales** automatiques, créées sur la pile. Evidemment, ceci limite leur durée de vie, si bien qu'une donnée locale à la fonction principale du thread (celle qui est précisée dans l'instruction de création `pthread_create()`) n'est pas utilisable par une autre fonction appelée par le thread.

Les threads peuvent évidemment user de données **globales**, variables statiques se trouvant dans le segment de données. Cependant, de telles données ne leur sont pas propres : elles sont au contraire à la portée de tous les threads. Ce qui conduit fréquemment à des erreurs d'exécution difficiles à diagnostiquer : elles apparaissent lorsque l'on passe d'une version mono-thread à une version multi-thread en affectant aux threads un morceau de code qui était anciennement exécuté par le seul processus ou plutôt par un seul thread ...

Pour cette raison, la norme Posix prévoit la création et l'utilisation de données que nous qualifierons de **spécifiques** : il s'agit de variables accessibles par toutes les fonctions utilisées par un thread (en ce sens, elles sont un peu globales) mais qui restent propres à chaque thread (en ce sens, elles sont plutôt locales). Comme ces variables sont en fait créées dans la mémoire statique, il a fallu prévoir un mécanisme particulier pour garantir que les différents threads accèdent bien à des variables distinctes.

2. D'une fonction normale à une fonction de thread : compter les lignes

Pour fixer les idées, considérons le programme suivant qui lit un fichier de texte. Il utilise une fonction `lireUneLigne()` fort simple, dont le rôle est de lire une ligne du fichier. Le fichier, `stockInput300`, est ici composé de lignes structurées en

numéro d'article + quantité en stock + date de la dernière livraison :

```
boole.inpres.epl.prov-liege.be> cat stockInput300
123 678 03/10/98
333 675 25/20/99
221 734 02/11/99
boole.inpres.epl.prov-liege.be>
```

Le programme lui-même s'écrit :

SPECIFICTHREAD01.C

```
/* specificThread01.c
Claude Vilvens */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MAX 18

void erreur (int l, char * msg, int numErr);
int lireUneLigne(int h);

void main()
{
    int hFile, r;

    hFile = open ("stockInput300", O_RDONLY );
    if (hFile == -1)
    { erreur (__LINE__, "Erreur ouverture du fichier", errno); exit(0); }
    while (r=lireUneLigne(hFile));
    close(hFile);
}

int lireUneLigne(int h)
{
    static int cptLectures = 0;
    int ret;
    char buf[MAX+1];

    ret=read(h, buf, sizeof (buf));
    if (ret==0) return 0;
    else if (ret==-1)
    {
        erreur (__LINE__, "Erreur de lecture dans le fichier", errno); exit(0);
    }
    buf[MAX]=0;
    printf ("On a lu : %s\n", buf);
    cptLectures++; printf("** Compteur = %d **\n", cptLectures);
    return ret;
}

void erreur (int l, char * msg, int numErr)
{
    printf("--Erreur ligne %d : %s (%d)\n", l, msg, numErr);
}
```


Le point crucial est la présence de la variable statique `cptLectures`, évidemment chargée de comptabiliser le nombre de lignes lues. Tel quel, le programme fonctionne sans problème :

```
sunray2v440.inpres.epl.prov-liege.be> c
On a lu : 123 678 03/10/98
** Compteur = 1 **
On a lu : 333 675 25/20/99
** Compteur = 2 **
On a lu : 221 734 02/11/99
** Compteur = 3 **
sunray2v440.inpres.epl.prov-liege.be>
```

Ainsi qu'évoqué dans l'introduction, le problème apparaît lorsque, pour avoir la possibilité de lire deux fichiers en même temps (disons *stockInput300* et *stockInput600*), on déplace le code de lecture dans une fonction *lire()* exécutée par un thread :

SPECIFICTHREAD02.C

```
/* specificThread02.c
Claude Vilvens
*/
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MAX 18

void *lire(void * h);
int lireUneLigne (int h);
void erreur (int l, char * msg, int numErr);

pthread_t hThr1, hThr2;

void main()
{
    int hFile1, hFile2;
    int rt, *retThread1, *retThread2;

    printf("Debut du thread principal %u\n", pthread_self());
    hFile1 = open ("stockInput300", O_RDONLY );
    if (hFile1 == -1)
    {
        erreur (__LINE__, "Erreur ouverture du fichier", errno); exit(1);
    }
    rt=pthread_create(&hThr1, NULL, lire, (void *)hFile1);
    if (rt) printf("Erreur a la creation du thread : %d\n", rt);
```

```

hFile2 = open ("stockInput600", O_RDONLY );
if (hFile2 == -1)
{
    erreur (__LINE__, "Erreur ouverture du fichier", errno); exit(1);
}
pthread_create(&hThr2, NULL, lire, (void *)hFile2);

pthread_join(hThr1,(void **) &retThread1);
printf("Valeur de retour du thread 1 = %d\n", *retThread1);
pthread_join(hThr2,(void **) &retThread2);
printf("Valeur de retour du thread 2 = %d\n", *retThread2);
close (hFile1); close(hFile2);
}

void *lire(void * h)
{
    static int vr;
    int ha = (int)h;

    printf("Debut du thread %u\n", pthread_self());
    while ( (vr = lireUneLigne(ha)) >0)
        printf("-- vr dans thread %u = %d\n", pthread_self(), vr);
    printf("Fin du thread %u\n", pthread_self());
    pthread_exit(&vr);
}

int lireUneLigne (int h)
{
    char buf[MAX+1];
    int ret;
    static int cptLectures;
    timespec_t temps;

    printf("*** Compteur initial dans %u = %d **\n", pthread_self(), cptLectures);
    temps.tv_sec = rand()/5000;
    temps.tv_nsec = 0;
    printf("Attente de %d secondes ...\n", temps.tv_sec);
    nanosleep(&temps, NULL);
    ret=read(h, buf, sizeof (buf));
    if (ret==-1)
        erreur (__LINE__, "Erreur de lecture dans le fichier", errno);
    else if (ret>0)
    {
        buf[MAX]=0;
        printf ("On a lu : %s\n", buf);
        cptLectures++;
    }
    printf("*** Compteur modifie dans %u = %d **\n", pthread_self(), cptLectures);
    return ret;
}

```

```
void erreur (int l, char * msg, int numErr)
{
    printf("--Erreur ligne %d : %s (%d)\n");
}
```

Pour faire court, on n'a pas spécialement soigné les castings dans les appels des primitives de threads. Des mises en sommeil aléatoires, comme d'habitude, permettent de donner un aspect chaotique à notre essai. Mais le problème est bien sûr ailleurs : **la variable statique *cptLectures* est commune aux deux threads**, si bien que le nombre de lignes lues pour chacun des deux fichiers s'est additionné ! Ainsi, si le deuxième fichier *stockInput600* contient :

```
sunray2v440.inpres.epl.prov-liege.be> cat stockInput600
983 899 23/04/00
954 843 03/08/99
943 855 20/12/00
911 833 31/12/99
932 344 23/83/23
992 565 02/04/88
sunray2v440.inpres.epl.prov-liege.be>
```

le résultat de l'exécution du programme est :

```
sunray2v440.inpres.epl.prov-liege.be> c
Debut du thread principal 3223038392
Debut du thread 1073985920
** Compteur initial dans 1073985920 = 0 **
Attente de 3 secondes ...
Debut du thread 1074051456
** Compteur initial dans 1074051456 = 0 **
Attente de 1 secondes ...
On a lu : 983 899 23/04/00
** Compteur modifie dans 1074051456 = 1 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 1 **
Attente de 2 secondes ...
On a lu : 123 678 03/10/98
** Compteur modifie dans 1073985920 = 2 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 2 **
Attente de 3 secondes ...
On a lu : 954 843 03/08/99
** Compteur modifie dans 1074051456 = 3 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 3 **
Attente de 6 secondes ...
On a lu : 333 675 25/20/99
** Compteur modifie dans 1073985920 = 4 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 4 **
```

```
Attente de 1 secondes ...
On a lu : 221 734 02/11/99
** Compteur modifie dans 1073985920 = 5 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 5 **
Attente de 4 secondes ...
On a lu : 943 855 20/12/00
** Compteur modifie dans 1074051456 = 6 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 6 **
Attente de 1 secondes ...
On a lu : 911 833 31/12/99
** Compteur modifie dans 1074051456 = 7 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 7 **
Attente de 3 secondes ...
** Compteur modifie dans 1073985920 = 7 **
Fin du thread 1073985920
Valeur de retour du thread 1 = 0
On a lu : 932 344 23/83/23
** Compteur modifie dans 1074051456 = 8 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 8 **
Attente de 0 secondes ...
On a lu : 992 565 02/04/88
** Compteur modifie dans 1074051456 = 9 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 9 **
Attente de 0 secondes ...
** Compteur modifie dans 1074051456 = 9 **
Fin du thread 1074051456
Valeur de retour du thread 2 = 0
sunray2v440.inpres.epl.prov-liege.be>
```

La variable statique compteur est bien la même pour tous les threads ! Ce n'est évidemment pas ce que l'on souhaite et c'est ici que la possibilité de créer des variables statiques propres à chaque thread va prouver son utilité.

3. Des variables statiques mais pas tout à fait ...

Pour cette raison, l'interface POSIX a prévu la possibilité de créer et de gérer des données statiques, donc permanentes, mais privées aux threads. On parle encore de "**variables spécifiques aux threads**" [*thread-specific data*]. En les utilisant, il devient possible de convertir une fonction existante fonctionnant au sein d'un processus en une fonction pouvant fonctionner correctement au sein d'un thread [*function thread-safe*].

Le principe global est simple : le système réserve pour un processus un ensemble de données statiques à ventiler entre les divers threads de ce processus. Schématiquement, on peut considérer que ces données sont structurées selon un tableau à deux dimensions, dont les colonnes sont indexées par le numéro des threads tandis que les indices des lignes sont

appelés des *clés*. L'idée est alors qu'un thread quelconque pourra toujours accéder à une donnée de "sa colonne".

	thread 0	thread 1	...	thread nt
clé = 0				
clé = 1				
...				
clé = nc-1				

4. Le principe de l'implémentation

La zone de mémoire statique réservée par le système à l'usage des threads peut donc être vue comme une zone contiguë de mémoire gérée par un certain nombre de structures appelées des clés [*keys*]; soit *nc* ce nombre de structures. Chaque structure comporte

- ♦ un flag indiquant si cette zone mémoire est utilisée;
- ♦ un pointeur sur une fonction de terminaison (que l'on peut franchement appeler le destructeur¹).

clé[0]	flag
	pDestructeur
clé[1]	flag
	pDestructeur
...	...
clé[nc-1]	flag
	pDestructeur

Pour chaque thread, le système mémorise un certain nombre d'informations (un peu à l'image de ce qui se passe pour un processus, mais, rappelons-le, en plus simple), dont notamment un tableau de pointeurs associés aux clés. Le rôle de ces pointeurs est de *désigner la donnée effectivement mémorisée par l'intermédiaire de la clé*. Donc, pour chaque processus, on peut imaginer que les informations correspondantes ont la forme :

thread 0	thread 1
informations diverses	informations diverses
pointeur-clé[0]	pointeur-clé[0]
pointeur-clé[1]	pointeur-clé[1]
pointeur-clé[nc-1]	pointeur-clé[nc-1]

Initialement, ces pointeurs sont nuls. Reste à voir comment on peut leur faire désigner une zone statique à usage réservé au thread considéré et, aussi, comment utiliser de telles données.

¹ on n'échappe jamais vraiment aux objets ;-) ...

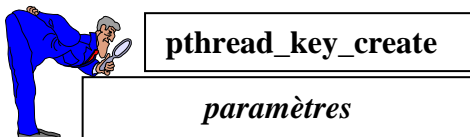
5. Les primitives Posix pour variables spécifiques

5.1 Obtenir une clé

Pour utiliser une telle zone statique dans un thread, il faut d'abord en demander la réservation, ainsi qu'obtenir une clé d'accès à cette zone. La primitive qui réalise cela est :

```
int pthread_key_create(  <clé - pthread_key_t *>,
                        <fonction - void (*)(void *)>);
```

qui renvoie 0 en cas de succès.



Le premier paramètre est la clé obtenue; c'est une sorte d'index dans le tableau des pointeurs. Elle est de type `pthread_key_t`, qui dissimule en fait (dans `pthread.h`) :

```
#ifndef _PTHREAD_ENV_UNIX
typedef unsigned int  pthread_key_t;
#endif
```

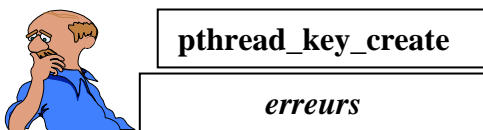
Cette clé est connue de tous les threads, qui pourront donc l'utiliser chacun, mais pour accéder chacun à une zone mémoire distincte. Autrement dit,

chaque thread lie une donnée différente à la clé commune.

La zone effectivement associée à chaque thread est pour l'instant inexistante : le pointeur correspondant est nul. Ce pointeur est cependant réservé et l'espace mémoire qu'il représente, en fait, peut être utilisé, par exemple, comme un simple entier.

Il faut remarquer que cette création de clé ne doit être effectuée qu'une seule fois, sous peine d'erreur.

Le deuxième paramètre permet de définir une fonction de terminaison dont le rôle est de libérer les ressources associées à la clé – on peut donc franchement parler de destructeur au sens de la P.O.O.



valeur de retour		erreur
0		succès
#define EAGAIN	35	/* Operation would block */ : il n'y a plus de ressources disponibles ou le nombre maximum de clés par thread (PTHREAD_KEYS_MAX) est dépassé.
#define ENOMEM	12	/* Not enough core */ : comme d'habitude : l'espace mémoire est insuffisant

Il est toujours possible de récupérer une clé par

```
int pthread_key_delete( <clé - pthread_key_t >);
```



pthread_key_delete

erreurs

<i>valeur de retour</i>		<i>erreur</i>
0		succès
#define EINVAL	22	/* Invalid argument */ : le paramètre clé n'est pas valide

5.2 Accéder aux données spécifiques par la clé

Un thread accède à sa donnée spécifique associée à la clé passée comme paramètre en utilisant la primitive :

```
void *pthread_getspecific( <clé - pthread_key_t >);
```

Cette fonction renvoie le pointeur vers la zone, ou NULL si cette zone n'existe pas encore. A l'inverse, un thread modifie sa donnée spécifique associée à une clé donnée par :

```
int pthread_setspecific( <clé - pthread_key_t>,
                        <pointeur vers la zone - const void *>);
```

Cette fonction renvoie 0 en cas de succès.



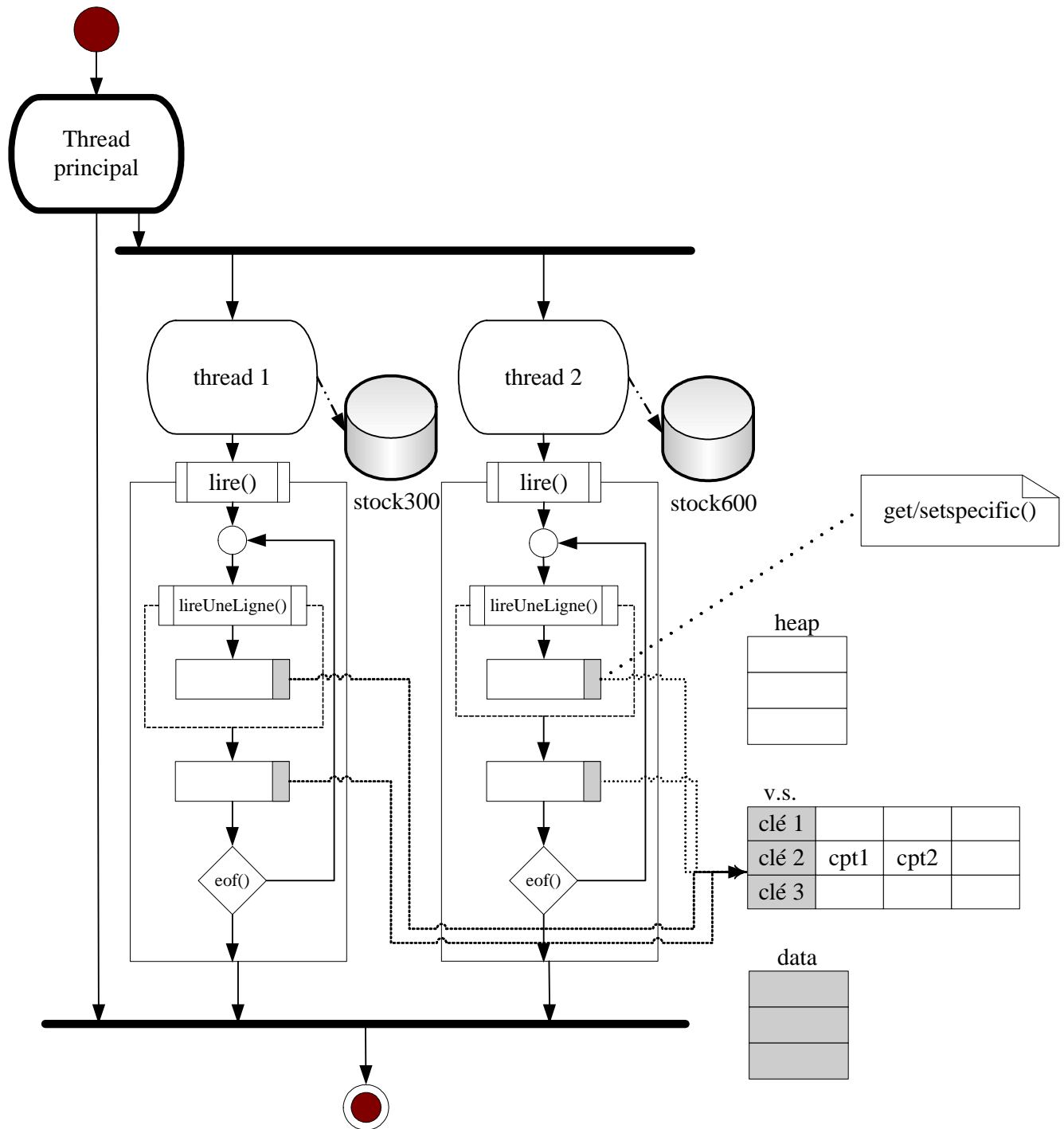
pthread_setspecific

erreurs

<i>valeur de retour</i>		<i>erreur</i>
0		succès
#define EINVAL	22	/* Invalid argument */ : le paramètre clé n'est pas valide
#define ENOMEM	12	/* Not enough core */ : l'espace mémoire associé à la clé est insuffisant

5.3 Retour au compteur de lignes

Nous allons reprendre notre compteur de lignes en utilisant l'espace mémoire obtenu par pthread_key_create(). L'espace mémoire obtenu ne sera pas utilisé comme un pointeur, mais comme un simple entier.



SPECIFICTHREAD03.C

```
/* specificThread03.c
Claude Vilvens
*/
```

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



```
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MAX 18

void *lire(void * h);
int lireUneLigne (int h);
void erreur (int l, char * msg, int numErr);

pthread_t hThr1, hThr2;
pthread_key_t cle;

void main()
{
    int hFile1, hFile2;
    int rt, *retThread1, *retThread2;

    printf("Debut du thread principal %u\n", pthread_self());
    pthread_key_create(&cle, NULL);

    hFile1 = open ("stockInput300", O_RDONLY );
    if (hFile1 == -1) { ... }
    rt=pthread_create(&hThr1, NULL, lire, (void *)hFile1);
    if (rt) printf("Erreur a la creation du threead : %d\n", rt);
    hFile2 = open ("stockInput600", O_RDONLY );
    if (hFile2 == -1) { ... }
    pthread_create(&hThr2, NULL, lire, (void *)hFile2);

    pthread_join(hThr1,(void **) &retThread1);
    printf("Valeur de retour du thread 1 = %d\n", *retThread1);
    pthread_join(hThr2,(void **) &retThread2);
    printf("Valeur de retour du thread 2 = %d\n", *retThread2);
    close (hFile1); close(hFile2);
}

void *lire(void * h)
{
    static int vr;
    int ha = (int)h, cptFin;

    printf("Debut du thread %u\n", pthread_self());
    while ( (vr = lireUneLigne(ha)) >0)
        printf("-- vr dans thread %u = %d\n", pthread_self(), vr);
    cptFin = (int) pthread_getspecific(cle);
    printf("***** Compteur final dans %u = %d **\n", pthread_self(), cptFin);
    pthread_exit(&vr);
    return 0;
}
```

```

int lireUneLigne (int h)
{
    char buf[MAX+1];
    int ret;
    int cptLectures, test;
    timespec_t temps;

    cptLectures = (int) pthread_getspecific(cle);
    printf("** Compteur initial dans %u = %d **\n", pthread_self(), cptLectures);
    temps.tv_sec = rand()/5000;
    temps.tv_nsec = 0;
    printf("Attente de %d secondes ...\n", temps.tv_sec);
    nanosleep(&temps, NULL);
    ret=read(h, buf, sizeof (buf));
    if (ret==-1) erreur (__LINE__, "Erreur de lecture dans le fichier", errno);
    else if (ret>0)
    {
        buf[MAX]=0;
        printf ("On a lu : %s\n", buf);
        cptLectures++;
        if (pthread_setspecific(cle, (void *)cptLectures))
        {
            puts("!!!! Erreur de setspecific");
        }
    }
    test = (int) pthread_getspecific(cle);
    printf("** Compteur modifie dans %u = %d **\n", pthread_self(), test);
    return ret;
}

void erreur (int l, char * msg, int numErr)
{
    printf("--Erreur ligne %d : %s (%d)\n", l, msg, numErr);
}

```

Le résultat est tout à fait convaincant :

```

sunray2v440.inpres.epl.prov-liege.be> CC -o c specificThread03.c -mt -lpthread -lrt
...warnings ...
sunray.inpres.epl.prov-liege.be> c
Debut du thread principal 3223038392
Debut du thread 1073985920
** Compteur initial dans 1073985920 = 0 **
Attente de 3 secondes ...
Debut du thread 1074051456
** Compteur initial dans 1074051456 = 0 **
Attente de 1 secondes ...
On a lu : 983 899 23/04/00
** Compteur modifie dans 1074051456 = 1 **
-- vr dans thread 1074051456 = 19

```

```
** Compteur initial dans 1074051456 = 1 **
Attente de 2 secondes ...
On a lu : 123 678 03/10/98
** Compteur modifie dans 1073985920 = 1 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 1 **
Attente de 3 secondes ...
On a lu : 954 843 03/08/99
** Compteur modifie dans 1074051456 = 2 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 2 **
Attente de 6 secondes ...
On a lu : 333 675 25/20/99
** Compteur modifie dans 1073985920 = 2 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 2 **
Attente de 1 secondes ...
On a lu : 221 734 02/11/99
** Compteur modifie dans 1073985920 = 3 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 3 **
Attente de 4 secondes ...
On a lu : 943 855 20/12/00
** Compteur modifie dans 1074051456 = 3 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 3 **
Attente de 1 secondes ...
On a lu : 911 833 31/12/99
** Compteur modifie dans 1074051456 = 4 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 4 **
Attente de 3 secondes ...
** Compteur modifie dans 1073985920 = 3 **
***** Compteur final dans 1073985920 = 3 **
Valeur de retour du thread 1 = 0
On a lu : 932 344 23/83/23
** Compteur modifie dans 1074051456 = 5 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 5 **
Attente de 0 secondes ...
On a lu : 992 565 02/04/88
** Compteur modifie dans 1074051456 = 6 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 6 **
Attente de 0 secondes ...
** Compteur modifie dans 1074051456 = 6 **
***** Compteur final dans 1074051456 = 6 **
Valeur de retour du thread 2 = 0
sunray2v440.inpres.epl.prov-liege.be>
```

6. Les routines d'initialisation unique

La création d'une clé, comme d'ailleurs celle d'un mutex ou d'une variable de condition, est une opération qui doit être réalisée une seule fois¹. Pour assurer cela, on peut placer un tel code d'initialisation dans une routine exécutée par la primitive :

```
int pthread_once( <structure d'enregistrement de l'initialisation - pthread_once_t *>,
                  <fonction d'initialisation - void (*)(void)>);
```

Chaque fonction d'initialisation est associée à une structure *pthread_once_t* qui lui est particulière. Le premier thread qui invoquera *pthread_once()* provoquera effectivement l'exécution de la routine d'initialisation. Les autres pourront invoquer *pthread_once()* sans provoquer d'effet (plus exactement, l'exécution de la routine d'initialisation est bloquée pour ces threads jusqu'à la fin du thread qui a initialisé).



pthread_once

paramètres

Pour ce qui est du premier paramètre, il s'agit d'une structure qui permet l'enregistrement de l'exécution de la routine d'initialisation. Dans *pthread.h*, elle est définie par :

```
structure pthread_once_t
#ifdef _PTHREAD_ENV_UNIX
typedef volatile struct __pthread_once_t
{
    long    _Pfield(state);
    long    _Pfield(reserved)[10];
} pthread_once_t;
#endif
```

Pour l'initialiser, le plus portable est d'utiliser la macro **PTHREAD_ONCE_INIT** définie dans *pthread.h* :

```
#define PTHREAD_ONCE_UNINIT    0
#define PTHREAD_ONCE_INITING   1
#define PTHREAD_ONCE_INITED    2
#define PTHREAD_ONCE_INIT      {PTHREAD_ONCE_UNINIT}
```



pthread_once

erreurs

<i>valeur de retour</i>	<i>erreur</i>
0	succès
#define EINVAL 22	/* Invalid argument */ : le paramètre clé n'est pas valide

¹ en C++, on parle encore de "singleton"

Le programme suivant a placé la création de la clé dans une routine d'initialisation exécutée une seule fois :

SPECIFICTHREAD04.C

```
/* specificThread04.c
Claude Vilvens
*/

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

#define MAX 18

void *lire(void * h);
int lireUneLigne (int h);
void erreur (int l, char * msg, int numErr);
void initCle();

pthread_t hThr1, hThr2;
pthread_key_t cle;
pthread_once_t controleur = PTHREAD_ONCE_INIT;

void main()
{
    int hFile1, hFile2;
    int rt, *retThread1, *retThread2;
    ...
}

void *lire(void * h)
{
    static int vr;
    int ha = (int)h, cptFin;

    printf("Debut du thread %u\n", pthread_self());
    pthread_once(&controleur, initCle);
    while ( (vr = lireUneLigne(ha)) > 0)
        printf("-- vr dans thread %u = %d\n", pthread_self(), vr);
    cptFin = (int) pthread_getspecific(cle);
    printf("***** Compteur final dans %u = %d **\n", pthread_self(), cptFin);
    pthread_exit(&vr);
    return 0;
}

int lireUneLigne (int h)
{
    char buf[MAX+1];
    int ret;
```

```

int cptLectures, test;
timespec_t temps;

cptLectures = (int) pthread_getspecific(cle);
printf("*** Compteur initial dans %u = %d **\n", pthread_self(), cptLectures);
temps.tv_sec = rand()/5000;
temps.tv_nsec = 0;
printf("Attente de %d secondes ...\n", temps.tv_sec);
nanosleep(&temps, NULL);
ret=read(h, buf, sizeof (buf));
if (ret==-1) erreur (__LINE__, "Erreur de lecture dans le fichier", errno);
else if (ret>0)
{
    buf[MAX]=0;
    printf ("On a lu : %s\n", buf);
    cptLectures++;
    if (pthread_setspecific(cle, (void *)cptLectures))
    {
        puts("!!!! Erreur de setspecific");
    }
}
test = (int) pthread_getspecific(cle);
printf("*** Compteur modifie dans %u = %d **\n", pthread_self(), test);
return ret;
}

void initCle()
{
    puts("=== initialisation d'une cle ===");
    pthread_key_create(&cle, NULL);
}

void erreur (int l, char * msg, int numErr) { ...}

```

Ce qui donne :

```

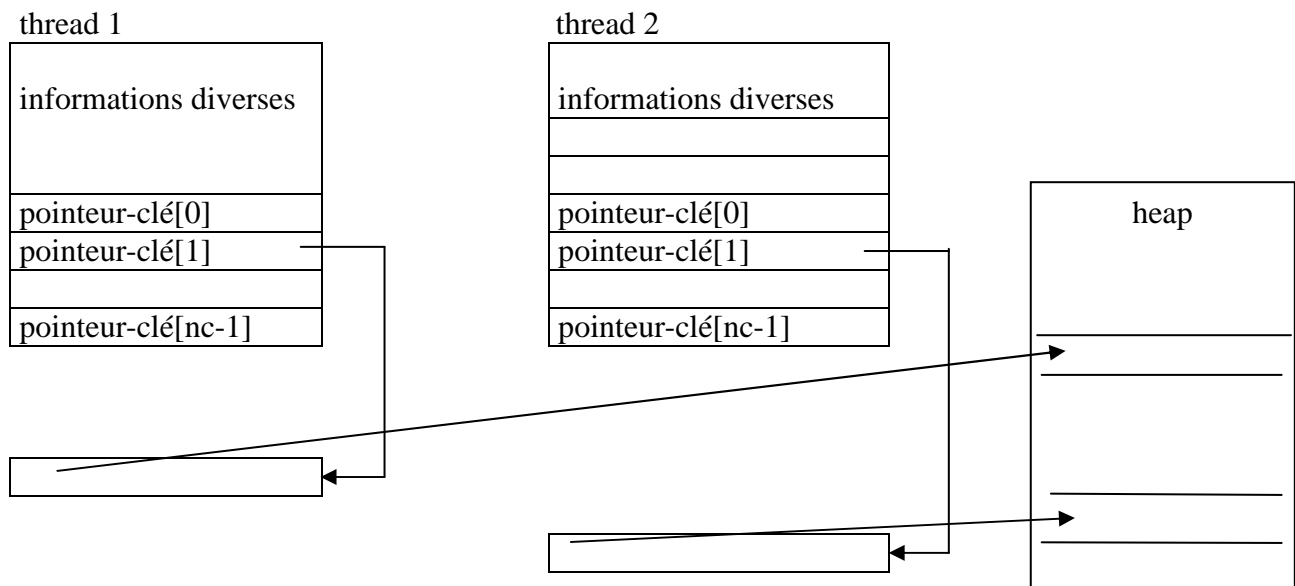
sunray2v440.inpres.epl.prov-liege.be> c
Debut du thread principal 3223038392
Debut du thread 1073985920
=== initialisation d'une cle ===
** Compteur initial dans 1073985920 = 0 **
Attente de 3 secondes ...
Debut du thread 1074051456
** Compteur initial dans 1074051456 = 0 **
Attente de 1 secondes ...
On a lu : 983 899 23/04/00
** Compteur modifie dans 1074051456 = 1 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 1 **

```

```
Attente de 2 secondes ...
On a lu : 123 678 03/10/98
** Compteur modifie dans 1073985920 = 1 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 1 **
Attente de 3 secondes ...
On a lu : 954 843 03/08/99
** Compteur modifie dans 1074051456 = 2 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 2 **
Attente de 6 secondes ...
On a lu : 333 675 25/20/99
** Compteur modifie dans 1073985920 = 2 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 2 **
Attente de 1 secondes ...
On a lu : 221 734 02/11/99
** Compteur modifie dans 1073985920 = 3 **
-- vr dans thread 1073985920 = 19
** Compteur initial dans 1073985920 = 3 **
Attente de 4 secondes ...
On a lu : 943 855 20/12/00
** Compteur modifie dans 1074051456 = 3 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 3 **
Attente de 1 secondes ...
On a lu : 911 833 31/12/99
** Compteur modifie dans 1074051456 = 4 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 4 **
Attente de 3 secondes ...
** Compteur modifie dans 1073985920 = 3 **
***** Compteur final dans 1073985920 = 3 **
On a lu : 932 344 23/83/23
** Compteur modifie dans 1074051456 = 5 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 5 **
Attente de 0 secondes ...
On a lu : 992 565 02/04/88
** Compteur modifie dans 1074051456 = 6 **
-- vr dans thread 1074051456 = 19
** Compteur initial dans 1074051456 = 6 **
Attente de 0 secondes ...
** Compteur modifie dans 1074051456 = 6 **
***** Compteur final dans 1074051456 = 6 **
Valeur de retour du thread 2 = 0
Valeur de retour du thread 1 = 0
sunray2v440.inpres.epl.prov-liege.be>
```

7. L'allocation d'une zone mémoire spécifique

Pour l'instant, nos pointeurs vers une zone de données ont été simplement utilisés comme des espaces mémoire. Nous allons à présent réellement *les faire pointer sur une zone mémoire propre à chaque thread*. Pour savoir si l'allocation a déjà eu lieu, il nous suffira de tester ce que nous fournit un appel à `pthread_getspecific()`. Si le résultat est `NULL`, nous pourrons allouer une zone désignée par le pointeur spécifique. Supposons que la clé obtenue soit `clé[1]`. Alors, après l'allocation pratiquée par chaque thread, la situation est la suivante :



A la fin de l'utilisation, la fonction destructeur, précisée dans la création de la clé, permettra de récupérer l'espace alloué. La zone allouée sert ici à contenir le dernier chiffre des numéros d'article (3ème colonne).

SPECIFICTHREAD05.C

```
/* specificThread05.c
Claude Vilvens
*/

#include <stdio.h>
#include <errno.h>
#include <pthread.h>
...

#define MAX 18
#define TAILLE_MSG 100

void *lire(void * h);
int lireUneLigne (int h, int i, char *s);
void erreur (int l, char * msg, int numErr);
void initCle();
void destructeur (void *p);

pthread_t hThr1, hThr2;
pthread_key_t cle;
pthread_once_t controleur = PTHREAD_ONCE_INIT;
```



```
void main()
{
    int hFile1, hFile2;
    int rt, *retThread1, *retThread2;
    ...
}

void *lire(void * h)
{
    static int vr;
    int ha = (int)h, cptFin, index=0, j;
    char * pSpec;

    printf("Debut du thread %u\n", pthread_self());
    pthread_once(&controleur, initCle);
    if ( (pSpec = pthread_getspecific(cle)) == NULL)
    {
        pSpec = (char *)malloc(TAILLE_MSG);
        pthread_setspecific(cle, pSpec);
    }
    while ( (vr = lireUneLigne(ha, index++, pSpec)) > 0)
        printf("-- vr dans thread %u = %d\n", pthread_self(), vr);
    pSpec[index]=0;
    printf("-- pSpec dans thread %u = %s\n", pthread_self(), pSpec);

    pthread_exit(&vr);
    return 0;
}

int lireUneLigne (int h, int i, char *s)
{
    char buf[MAX+1];
    int ret, j;
    int cptLectures, test;
    timespec_t temps;

    temps.tv_sec = rand()/5000; temps.tv_nsec = 0;
    printf("Attente de %d secondes ...\n", temps.tv_sec);
    nanosleep(&temps, NULL);
    ret=read(h, buf, sizeof (buf));
    if (ret==-1) erreur (__LINE__, "Erreur de lecture dans le fichier", errno);
    else if (ret>0)
    {
        buf[ret]=0;
        printf ("On a lu : %s\n", buf);
    }
    s[i] = buf[2];

    return ret;
}
```

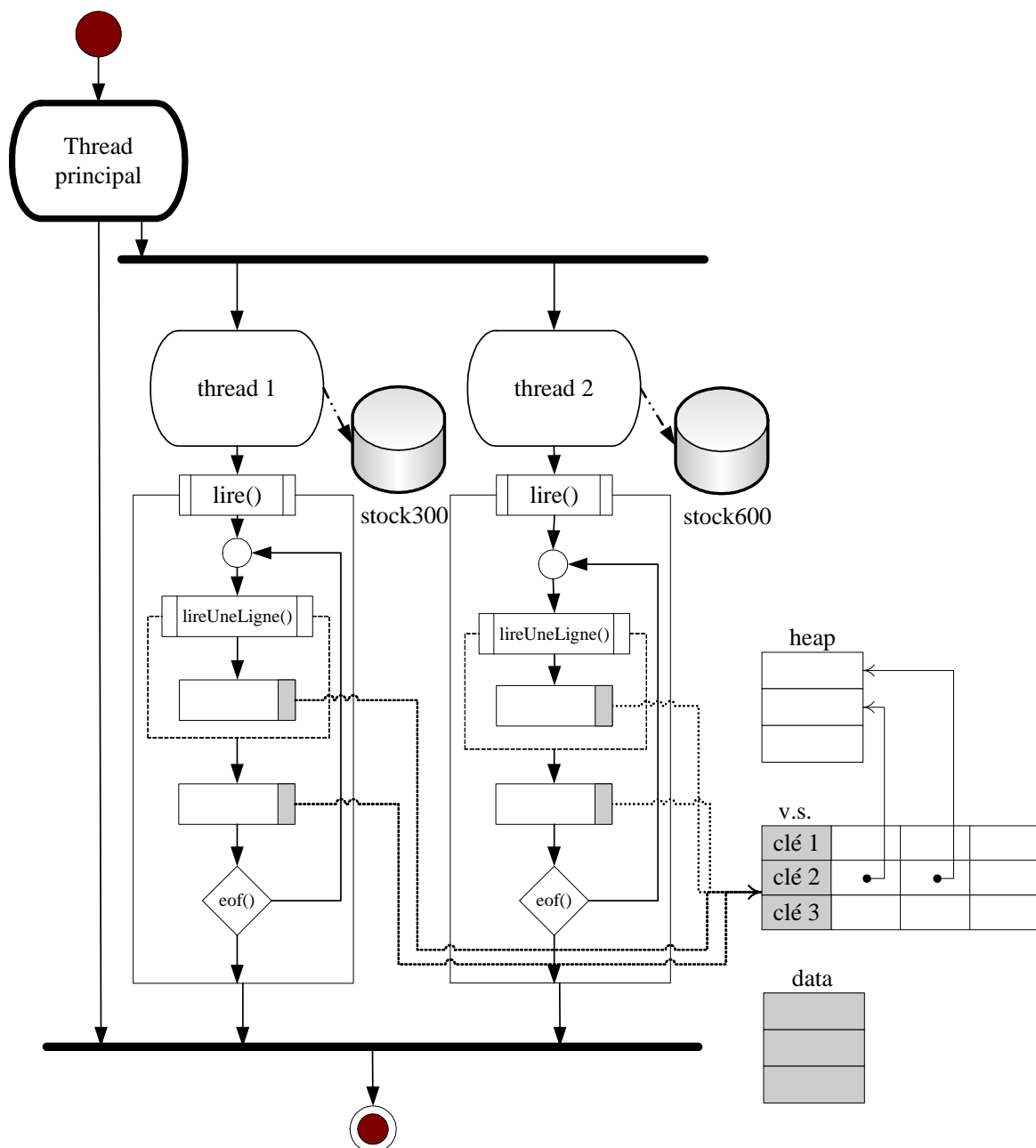
```

void initCle()
{
    puts("=== initialisation d'une cle ===");
    pthread_key_create(&cle, destructeur);
}

void destructeur (void *p)
{
    puts("=== liberation d'une zone specifique ===");
    free(p);
}

void erreur (int l, char * msg, int numErr) { ... }

```



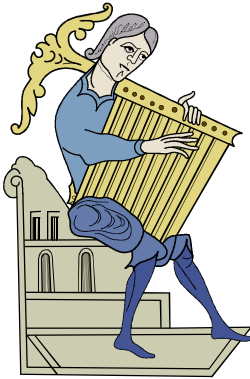
Ce qui donne :

```
sunray2v440.inpres.epl.prov-liege.be> c
Debut du thread principal 3223038392
Debut du thread 1073985920
=== initialisation d'une cle ===
Attente de 3 secondes ...
Debut du thread 1074051456
Attente de 1 secondes ...
On a lu : 983 899 23/04/00
-- vr dans thread 1074051456 = 19
Attente de 2 secondes ...
On a lu : 123 678 03/10/98
-- vr dans thread 1073985920 = 19
Attente de 3 secondes ...
On a lu : 954 843 03/08/99
-- vr dans thread 1074051456 = 19
Attente de 6 secondes ...
On a lu : 333 675 25/20/99
-- vr dans thread 1073985920 = 19
Attente de 1 secondes ...
On a lu : 221 734 02/11/99
-- vr dans thread 1073985920 = 19
Attente de 4 secondes ...
On a lu : 943 855 20/12/00
-- vr dans thread 1074051456 = 19
Attente de 1 secondes ...
On a lu : 911 833 31/12/99
-- vr dans thread 1074051456 = 19
Attente de 3 secondes ...
-- pSpec dans thread 1073985920 = 331
=== liberation d'une zone specifique ===
Valeur de retour du thread 1 = 0
On a lu : 932 344 23/83/23
-- vr dans thread 1074051456 = 19
Attente de 0 secondes ...
On a lu : 992 565 02/04/88
-- vr dans thread 1074051456 = 19
Attente de 0 secondes ...
-- pSpec dans thread 1074051456 = 343122
=== liberation d'une zone specifique ===
Valeur de retour du thread 2 = 0
sunray2v440.inpres.epl.prov-liege.be
```

Quand on récapitule tout ce que nous savons à présent sur les threads Posix, il paraît évident qu'ils ne sont pas si élémentaires qu'il y paraît au premier abord. Terminons donc cet exposé en prenant un peu de recul ...



VI. Les modèles classiques



Il est besoin de temporiser, nous ne pouvons pas toujours être les plus forts.

(E. de la Boétie, Discours de la servitude volontaire)

1. L'implémentation de divers modèles

Les chapitres précédents nous ont permis d'assimiler les diverses techniques utilisées dans la programmation des threads. Nous allons terminer le présent exposé par quelques exemples d'implémentation des modèles logiques classiques des threads. Ces modèles logiques sont en quelque sorte des canevas selon lesquels différents threads peuvent collaborer : qui lance qui, qui attend qui, qui réveille qui, etc. Les véritables applications implémentent le plus souvent l'un de ces modèles.

2. Le modèle parallèle ou du travail en équipe

2.1 Le principe du modèle

Dans ce modèle,

- ◆ une tâche complexe est fragmentée en sous-tâches (similaires ou différentes);
- ◆ chacune de celles-ci est confiée à un thread;
- ◆ le programme se poursuit lorsque tous les threads responsables d'une sous-tâche ont terminé.

En pratique, un thread distributeur de tâches réalise la fragmentation et lance les différents threads. Il se synchronise ensuite sur la fin de ces threads avant de passer à la suite du programme.

2.2 Une implémentation : les recherches parallèles dans des fichiers

Un bon exemple est celui où l'on recherche des informations sur une même clé de recherche mais dans des fichiers (ou des tables) différents : on créera autant de threads qu'il y a de fichiers à introspecter.

Supposons donc disposer de trois fichiers (ici, il s'agit de fichiers textes pour la facilité) qui comportent, pour les mêmes hypothétiques clients, respectivement la localité du domicile (fichier clients.data), le revenu mensuel net (fichier clients.fisc) et le nombre de prescriptions médicales par trimestre (fichier clients.sante).

clients.data		clients.fisc		clients.sante	
Excellent	Oupeye	Excellent	122300	Excellent	12
Genial	Huy	Genial	67300	Genial	2
TresBon	Remicourt	TresBon	45900	TresBon	26
Superbe	Antheit	Superbe	39678	Superbe	3
Divin	Hermee	Divin	75623	Divin	10
Inaccessible	Verviers	Inaccessible	66000	Inaccessible	35
Ecoeurant	Neupre	Ecoeurant	56323	Ecoeurant	9
UnMaitre	Vise	UnMaitre	55600	UnMaitre	13
Extraordinaire	Eupen	Extraordinaire	62670	Extraordinaire	7

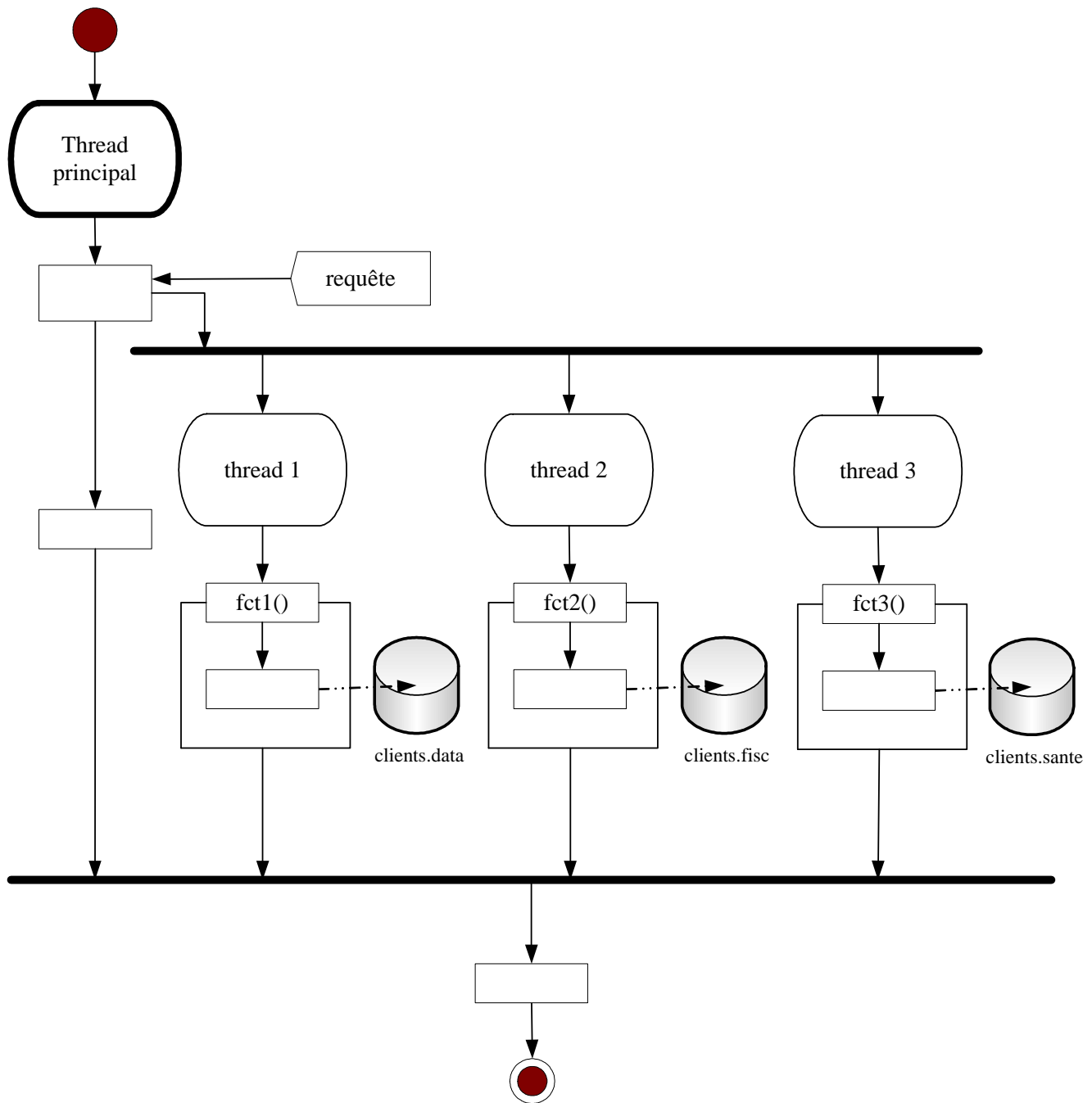
Comme dans un chapitre précédent, les noms de clients faisant l'objet de la recherche seront passés sur la ligne de commande, séparés par un "-". Pour chaque nom, trois threads seront lancés (un par fichier); ils recevront comme paramètre une structure contenant le nom du client à chercher, le nom du fichier à utiliser pour cette recherche et la position du nom dans la liste :

```
struct critereRech
{
    int numeroOrdre;
    char nomFich[20];
    char nomRech[20];
};
```

Il est implicitement supposé que le renseignement cherché est celui qui suit le nom (on pourrait compliquer pour être plus réaliste, mais ceci est sans intérêt ici). Les résultats de la recherche seront stockés dans un tableau de structure du type :

```
struct resultatRech
{
    char nomClient[20];
    int codeErreur;
    char localite[30];
    int revenuMensuel;
    int nbrePrescriptions;
};
```

le champ codeErreur permettant de détecter le cas d'un client non trouvé. Le thread principal attendra que les trois threads aient terminé leur travail pour un nom donné avant de passer au nom suivant. Schématiquement :



Le programme est en fait relativement simple à écrire. Il convient juste de remarquer l'utilisation pour le découpage de la chaîne de caractères des noms, de la fonction réentrante :

```
char *strtok_r(<chaîne à analyser - char *>,
               <chaîne de sdélimiteurs - const char *>,
               <adresse du pointeur sur le point de départ de la prochaine recherche - char**>);
```

en lieu et place de l'habituelle fonction strtok(). Le troisième paramètre sert à mémoriser le point de départ de la recherche suivante. On remarquera donc ici le souci de remplacer une fonction usuelle par sa version réentrante.

MODELEPARALLELE01.C

```
/* MODELEPARALLELE01.C
- Claude Vilvens -
*/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <unistd.h>
#include <string.h> /* pour memcpy */

#define MAXSTRING 1000 /* Longueur des messages */
#define LONG_MAX_NOM 40 /* Longueur maximale des noms */
#define NBRE_MAX_NOMS_CLIENTS 15 /* Nombre maximum de noms de clients */
#define NB_MAX_THR_TRAITEMENT 3 /* Nombre max de threads clients */
#define NB_FICHIERS 3

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define random(n) (rand()%(n))

int ret;

pthread_t threadHandle[NB_MAX_THR_TRAITEMENT];

void * fctThread(void * param);
char * getThreadIdentity();
void sleepThread(int ns);
char * trim(char *s);
void insereRenseignement(char * r, int n, int c);

struct critereRech
{
    int numeroOrdre;
    char nomFich[20];
    char nomRech[20];
} sCritereRech[NB_FICHIERS];

struct resultatRech
{
    char nomClient[20];
    int codeErreur;
    char localite[30];
    int revenuMensuel;
    int nbrePrescriptions;
} sResultatRecherche[NBRE_MAX_NOMS_CLIENTS];

#define R_CLI_OK 99
#define R_MAUVAIS_NUM_ORDRE 100
#define R_CLIENT_NON_TROUVE 101
```

```
int cpt;

int main(int argc, char **argv)
{
    int i;
    int * retThread;
    char msgClient [MAXSTRING];
    int vr;

    char buf[100];

    char *token, * savept=NULL;

    /* 1. Initialisations */
    puts("Thread principal serveur démarre");
    strcpy(sCritereRech[0].nomFich,"clients.data");
    strcpy(sCritereRech[1].nomFich,"clients.fisc");
    strcpy(sCritereRech[2].nomFich,"clients.sante");

    /* 2. Lecture du msgClient */
    if (argc < 2)
    {
        puts("Usage : scan message_a_analyser"); exit(0);
    }
    strcpy(msgClient, argv[1]);
    printf("msgClient = %s\n", msgClient);

    /* 3. Decoupe du message client */
    puts("----- Analyse de msgClient");

    cpt = 0;

    #if 0
    token = strtok(msgClient, "-"); /* strtok n'est pas supporté en multithread */
    #endif

    token = strtok_r(msgClient, "-", (char **)&savept);
    while (token != NULL)
    {
        for (i=0; i<NB_MAX_THR_TRAITEMENT; i++)
        {
            strcpy(sCritereRech[i].nomRech, token);
            sCritereRech[i].numeroOrdre = i;
            ret = pthread_create(&threadHandle[i], NULL, fctThread,
                                (void*)&sCritereRech[i]);
            printf("Thread secondaire %d lance !\n", i);
        }
        puts("-- Attente de la fin des threads paralleles");
    }
}
```



```

        for (i=0; i<NB_MAX_THR_TRAITEMENT; i++)
        {
            pthread_join(threadHandle[i], 0);
        }
        puts("-- Detachement des threads paralleles");

        cpt++;

        token = strtok_r((char *)NULL, "-", (char **)&savept);
    }
    puts("Fin scan msg client");

/* 4. Affichage des resultats */
    puts("*** Resultats des recherches ***");

    for (i=0; i<cpt; i++)
        if (sResultatRecherche[i].codeErreur == R_CLI_OK)
            printf("%s : %s - sal.net=%d et nb.prescr.=%d\n",
                sResultatRecherche[i].nomClient,
                sResultatRecherche[i].localite,
                sResultatRecherche[i].revenuMensuel,
                sResultatRecherche[i].nbrePrescriptions);
        else
            if (sResultatRecherche[i].codeErreur == R_CLIENT_NON_TROUVE)
                printf("%s non trouve !!!\n",
                    sResultatRecherche[i].nomClient);
    puts("*** Fin du thread principal ***");
    return 0;
}

/* ----- */
void * fctThread (void *param)
{
    char * nomCli = (char*)malloc(20),
        * nomFich = (char *)malloc(20),
        * ligneLue = (char *)malloc(100),
        * buf = (char*)malloc(100),
        * cherche, * encore,
        * renseignement = (char *)malloc(30);

    struct critereRech *cr = (struct critereRech *)param;
    int trouve, i, vr=0, num;
    int temps;
    FILE *f;
    char * numThr = getThreadIdentity();

    strcpy(nomCli, cr->nomRech);
    strcpy(nomFich, cr->nomFich);
    num = cr->numeroOrdre;

```

```

    sprintf(buf, "Je m'occupe de %s ...", nomCli);
    affThread(numThr, buf);
    f=fopen(nomFich, "r");
    sprintf(buf, "J'ai ouvert le fichier %s ...", nomFich);
    affThread(numThr, buf);
    strcpy(sResultatRecherche[cpt].nomClient, nomCli);

/* 1. Temporisation */
    temps = random(10)+1;
    sprintf(buf, "!-! Temps de reflexion de %d secondes ...", temps);
    affThread(numThr, buf);
    sleepThread(temps);

/* 2. Recherche du nom du client */
    trouve = 0;
    encore = fgets(ligneLue, 100, f);
    for (i=0; encore && !trouve; i++)
    {
        cherche=trim(strtok_r(ligneLue, " "));
        trouve = strcmp(nomCli, cherche) == 0;
        if (!trouve) encore = fgets(ligneLue, 100, f);
        else
        {
            cherche = strtok_r((char*)NULL, " ");
            sprintf(buf, "%s trouve en %d", nomCli, i);
            affThread(numThr, buf);
            strcpy(renseignement, cherche);
            sprintf(buf, "le renseignement = %s", renseignement);
            affThread(numThr, buf);
            sResultatRecherche[cpt].codeErreur=R_CLI_OK;
            insereRenseignement(renseignement, num, cpt );
        }
    }
    if (!trouve)
    {
        sprintf(buf, "%s inconnu", nomCli);
        affThread(numThr, buf);
        sResultatRecherche[cpt].codeErreur=R_CLIENT_NON_TROUVE;
    }
    fclose(f);
    free(nomCli); free(nomFich); free(ligneLue); free(buf);

    pthread_exit(&vr);
    return (void *)vr;
}

char * getThreadIdentity(){ ...}

void sleepThread(int ns) { ...}

```

```

char * trim(char *s)
{
    char *tr=s+strlen(s);
    while (*tr==' ' || *tr==0 || *tr=='\t')
    {
        *tr=0;
        tr--;
    }
    return s;
}

void insereRenseignement(char * r, int n, int c)
{
    switch(n)
    {
        case 0: strcpy(sResultatRecherche[c].localite, r); break;
        case 1: sResultatRecherche[c].revenuMensuel = atoi(r);break;
        case 2: sResultatRecherche[c].nbrePrescriptions = atoi(r);break;
        default: sResultatRecherche[c].codeErreur = R_MAUVAIS_NUM_ORDRE ;
                break;
    }
}

```

Il importe évidemment que chaque thread dispose de sa propre structure FILE, sans quoi tous les threads, sauf le premier, trouveront le pointeur de lecture déjà à la fin du fichier !

Une exécution de ce programme sur boole donne :

```

boole.inpres.epl.prov-liege.be>m UnMaitre-Excellent-Superbe-Sublime-Divin-
* Thread principal serveur demarre *
msgClient = UnMaitre-Excellent-Superbe-Sublime-Divin-
----- Analyse de msgClient

Thread secondaire 0 lance !
Thread secondaire 1 lance !
Thread secondaire 2 lance !
-- Attente de la fin des threads paralleles
th_32045.2> Je m'occupe de UnMaitre ...
th_32045.2> J'ai ouvert le fichier clients.data ...
th_32045.2> !-! Temps de reflexion de 9 secondes ...
th_32045.3> Je m'occupe de UnMaitre ...
th_32045.3> J'ai ouvert le fichier clients.fisc ...
th_32045.3> !-! Temps de reflexion de 9 secondes ...
th_32045.4> Je m'occupe de UnMaitre ...
th_32045.4> J'ai ouvert le fichier clients.sante ...
th_32045.4> !-! Temps de reflexion de 4 secondes ...
th_32045.4> UnMaitre trouve en 7
th_32045.4> le renseignement = 13
th_32045.2> UnMaitre trouve en 7
th_32045.2> le renseignement = Vise

```

```
th_32045.3> UnMaitre trouve en 7
th_32045.3> le renseignement = 55600
-- Detachement des threads paralleles

Thread secondaire 0 lance !
Thread secondaire 1 lance !
Thread secondaire 2 lance !
-- Attente de la fin des threads paralleles
th_32045.5> Je m'occupe de Excellent ...
th_32045.5> J'ai ouvert le fichier clients.data ...
th_32045.5> !-! Temps de reflexion de 6 secondes ...
th_32045.6> Je m'occupe de Excellent ...
th_32045.6> J'ai ouvert le fichier clients.fisc ...
th_32045.6> !-! Temps de reflexion de 2 secondes ...
th_32045.7> Je m'occupe de Excellent ...
th_32045.7> J'ai ouvert le fichier clients.sante ...
th_32045.7> !-! Temps de reflexion de 8 secondes ...
th_32045.6> Excellent trouve en 0
th_32045.6> le renseignement = 122300
th_32045.5> Excellent trouve en 0
th_32045.5> le renseignement = Oupeye
th_32045.7> Excellent trouve en 0
th_32045.7> le renseignement = 12
-- Detachement des threads paralleles

....

Thread secondaire 0 lance !
Thread secondaire 1 lance !
Thread secondaire 2 lance !
-- Attente de la fin des threads paralleles
th_32045.11> Je m'occupe de Sublime ...
th_32045.11> J'ai ouvert le fichier clients.data ...
th_32045.11> !-! Temps de reflexion de 7 secondes ...
th_32045.12> Je m'occupe de Sublime ...
th_32045.12> J'ai ouvert le fichier clients.fisc ...
th_32045.12> !-! Temps de reflexion de 10 secondes ...
th_32045.13> Je m'occupe de Sublime ...
th_32045.13> J'ai ouvert le fichier clients.sante ...
th_32045.13> !-! Temps de reflexion de 8 secondes ...
th_32045.11> Sublime inconnu
th_32045.13> Sublime inconnu
th_32045.12> Sublime inconnu
-- Detachement des threads paralleles

...

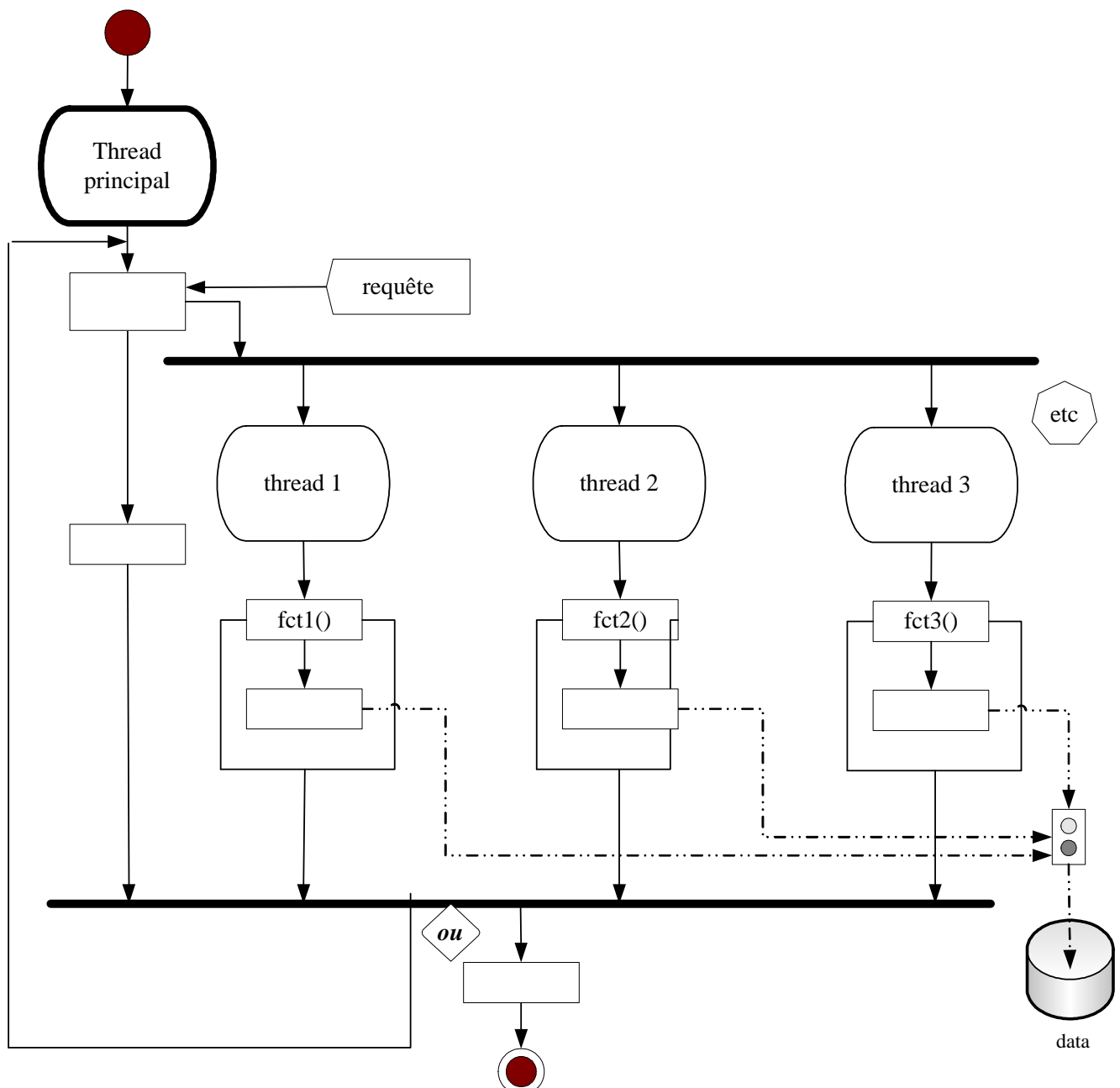
Fin scan msg client
*** Resultats des recherches ***
UnMaitre :   Visé - sal.net=55600 et nb.prescr.=13
```

Excellent : Oupeye - sal.net=122300 et nb.prescr.=12
Superbe : Antheit - sal.net=39678 et nb.prescr.=3
Sublime non trouve !!!
Divin : Hermee - sal.net=75623 et nb.prescr.=10
** Fin du thread principal **
boole.inpres.epl.prov-liege.be>

Une exécution sur Copernic n'apporte aucun élément supplémentaire.

2.3 Le schéma général du modèle parallèle

On peut résumer de manière générale le modèle parallèle selon le schéma suivant :



3. Le modèle du producteur et des consommateurs

3.1 Le principe du modèle

C'est probablement le modèle le plus courant, puisqu'il s'utilise presque naturellement dans une architecture client-serveur. Les principes généraux sont les suivants.

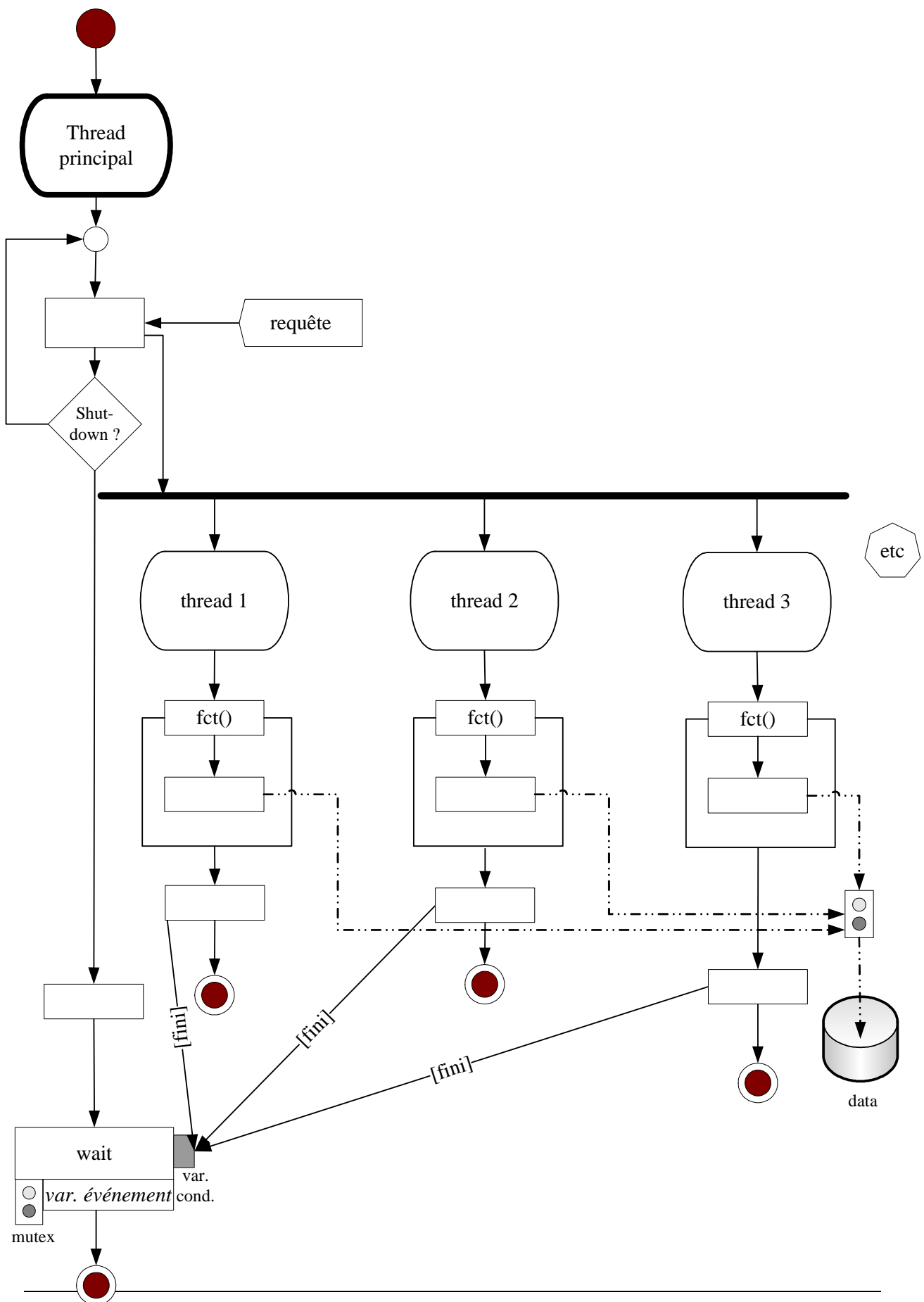
Un thread particulier (souvent le thread principal ou un thread créé à cet effet) reçoit des **requêtes** de divers clients par un moyen quelconque (par exemple, par terminal, par réseau, par ligne de commande, par fichier batch, ...). Chaque requête est prise en charge par un thread qui

- ◆ peut être **créé dynamiquement**, c'est-à-dire **à la demande** : le thread principal crée un thread à chaque requête reçue; ce thread client traite la requête puis se termine. Evidemment, les threads clients doivent être détachés pour que le thread principal puisse traiter les requêtes dès qu'elles lui parviennent. On peut encore remarquer que l'on ne connaît pas le nombre de threads effectivement actifs à un moment donné.
- ◆ fait partie d'un **groupe ("pool") de threads préalablement créés** au démarrage de l'application serveur et qui sont en attente de se voir affecter le traitement d'une requête. Cette fois, il ne peut y avoir le moindre danger d'une inflation du nombre de threads.

Que l'on utilise la première ou la deuxième alternative, le thread principal fait bien office de **producteur de tâches** tandis que les threads sont des **consommateurs de ces tâches**. Chacun des threads consommateurs ignore ce que font les autres; ils dépendent tous du serveur maître qui les a créés : ils en sont donc les esclaves. Cependant, les différents threads exécutent tous obligatoirement le même code, d'où l'importance de disposer de code réentrant.

3.2 Le schéma général du modèle producteur-consommateurs à la demande

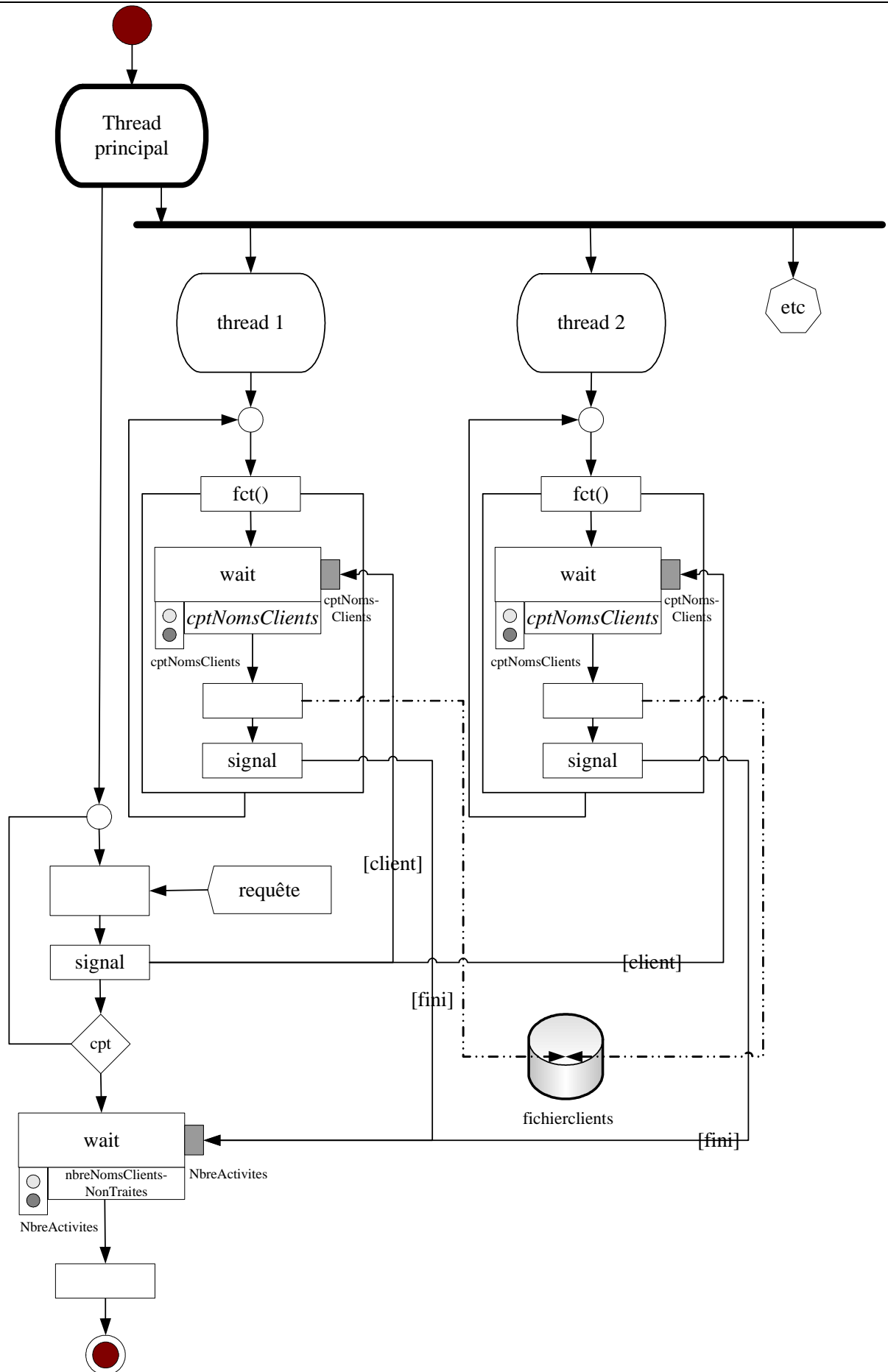
Nous avons déjà illustré la première stratégie dans le programme varcondi01.c du chapitre précédent, qui permettait de mettre en évidence l'utilisation d'une variable de condition. Le modèle général est, dans sa version basique, le suivant :



3.3 Une implémentation : le traitement des noms de clients

Nous allons donc plutôt implémenter la deuxième méthode, avec des threads prêts à l'emploi, mais en conservant le même contexte. Pour rappel, le programme reçoit une liste de noms de clients, noms séparés par un tiret, et a pour tâche de rechercher pour chaque nom le coefficient de réduction qui a été consenti à ce client – pour simplifier, on supposera toujours que ces informations sont dans un tableau (en pratique, ce sera au minimum dans un fichier ou plus probablement dans une base de données atteinte par du SQL intégré).

Afin d'accélérer les choses, chaque nom de client sera traité par un thread. Le traitement global sera terminé quand tous les noms auront été traités. Un thread est avisé qu'un nom est disponible parce qu'il est réveillé dans son attente sur une variable de condition *condCptNomsClients* qui lui permet de savoir quand la variable *cptNomsClients* devient différente de 0; la variable globale *indiceCourant* lui permet de savoir à quel nom le traitement global est parvenu. Le thread principal, une fois le découpage de la chaîne des noms terminée, est en attente sur la variable de condition *condNbreActivites* qui lui permet d'attendre que *nbreNomsClientsNonTraites* tombe à 0.



MODELEPRODUCCONSOM01.C

```

/* MODELEPRODUCCONSOM01.C
- Claude Vilvens -
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h> /* pour memcpy */

#define MAXSTRING 1000 /* Longueur des messages */
#define LONG_MAX_NOM 40 /* Longueur maximale des noms */
#define NBRE_MAX_NOMS_CLIENTS 15 /* Nombre maximum de noms de clients */
#define NB_MAX_THR_CLIENTS 5 /* Nombre max de threads clients */

#define affThread(num, msg) printf("th_%s> %s\n", num, msg)
#define random(n) (rand()%(n))

struct refClient
{
    char nom[LONG_MAX_NOM];
    float coeffRed;
};

struct refClient fichierClients[] =
    {
        {"Excellent", 0.95},
        {"Genial", 0.34},
        {"Tres bon", 0.75},
        {"Superbe", 0.87},
        {"Divin", 0.21},
        {"Inaccessible", 0.45},
        {"Ecoeurant", 0.67},
        {"Un maitre", 0.78},
        {"Extraordinaire", 0.89}
    };

const int tailleFichierClients = 9;
int indiceCourant=0;
char * nomsClientsRech[NBRE_MAX_NOMS_CLIENTS];
int ret;

pthread_t threadHandle[NB_MAX_THR_CLIENTS];

pthread_mutex_t mutexNbreActivites;
pthread_mutex_t mutexCptNomsClients;
pthread_cond_t condNbreActivites;
pthread_cond_t condCptNomsClients;

int nbreNomsClientsNonTraites; /* Ce qui caractérise l'événement */

```

```
int cptNomsClients=0;

void * fctThread(void * param);
char * getThreadIdentity();
void sleepThread(int ns);

int main(int argc, char **argv)
{
    int i, cpt;
    int * retThread;
    char msgClient [MAXSTRING];
    int vr;
    char buf[100];
    char *ptr, *ptrCliTr;
    char ptrCli[40];
    char *token;

    /* 1. Initialisations */
    puts("Thread principal serveur démarre");

    pthread_mutex_init(&mutexNbreActivites, NULL);
    pthread_mutex_init(&mutexCptNomsClients, NULL);
    pthread_cond_init(&condNbreActivites, NULL);
    pthread_cond_init(&condCptNomsClients, NULL);

    /* 2. Lecture du msgClient */
    if (argc < 2)
    {
        puts("Usage : scan message_a_analyser");
        exit(0);
    }
    strcpy(msgClient, argv[1]);
    printf("msgClient = %s\n", msgClient);

    /* 3. Lancement des threads */
    for (i=0; i<NB_MAX_THR_CLIENTS; i++)
    {
        ret = pthread_create(&threadHandle[i], NULL, fctThread, (void*)i);
        printf("Thread secondaire %d lance !\n", i);
        ret = pthread_detach(threadHandle[i]);
    }

    /* 4. Decoupe du message client */
    puts("----- Analyse de msgClient");

    cpt = 0;
    token = strtok(msgClient, "-");
    while (token != NULL)
    {
        nomsClientsRech[cpt] = malloc(strlen(token)+1);
```

```

strcpy(nomsClientsRech[cpt], token);
cpt++;

pthread_mutex_lock(&mutexCptNomsClients);
cptNomsClients++;
pthread_mutex_unlock(&mutexCptNomsClients);
pthread_cond_signal(&condCptNomsClients);

if (cptNomsClients >= NBRE_MAX_NOMS_CLIENTS) break;
token = strtok((char *)NULL, "-");
}
puts("Fin scan msg client");
pthread_mutex_lock(&mutexCptNomsClients);
nbreNomsClientsNonTraites = cptNomsClients;
pthread_mutex_unlock(&mutexCptNomsClients);

/* 5. Attente que tous les noms soient traites */
puts("Attente de la fin du traitement de tous les noms ...");
printf("nbreNomsClientsNonTraites = %d\n", nbreNomsClientsNonTraites);
pthread_mutex_lock(&mutexNbreActivites);
while (nbreNomsClientsNonTraites)
    pthread_cond_wait(&condNbreActivites, &mutexNbreActivites);
pthread_mutex_unlock(&mutexNbreActivites);

sleepThread(cpt*3);
/* for (i=0; i<NB_MAX_THR_CLIENTS; i++)
{
    ret = pthread_detach(threadHandle[i]);
    printf("Thread secondaire %d arrete !\n", i);
}*/
puts("*** Fin du thread principal ***");
return 0;
}

/* ----- */
void * fctThread (void *param)
{
    char * nomCli, *buf = (char*)malloc(100);
    int vr = (int)(param), trouve, i, iCliTraite;
    int temps;
    char * numThr = getThreadIdentity();

    while (1)
    {
/* 1. Attente d'un nom a traiter */
pthread_mutex_lock(&mutexCptNomsClients);
while (!cptNomsClients)
    pthread_cond_wait(&condCptNomsClients, &mutexCptNomsClients);
iCliTraite = indiceCourant;
nomCli = (char *)malloc(strlen(nomsClientsRech[iCliTraite])+1);

```

```

        strcpy(nomCli, nomsClientsRech[iCliTraite]);
        cptNomsClients--; indiceCourant++;
        pthread_mutex_unlock(&mutexCptNomsClients);
        sprintf(buf, "Je m'occupe de %s (%d) ...", nomCli, iCliTraite);
        affThread(numThr, buf);

/* 2. Temporisation */
        temps = random(10)+1; sleepThread(temps);
        sprintf(buf, "!! Temps de reflexion de %d secondes ...", temps);
        affThread(numThr, buf);

/* 3. Recherche du nom du client */
        trouve = 0;
        for (i=0; i<tailleFichierClients && !trouve; i++)
            trouve = strcmp(nomCli, fichierClients[i].nom) == 0;
        if (trouve)
        {
            sprintf(buf, "%s trouve en %d -> C.R.= %5.3f",
                    nomCli, i-1, fichierClients[i-1].coeffRed);
            affThread(numThr, buf);
        }
        else
        {
            sprintf(buf, "%s inconnu", nomCli);
            affThread(numThr, buf);
        }
        sleepThread(5);

/* 4. Signal de fin de traitement */
        pthread_mutex_lock(&mutexNbreActivites);
        if (nbreNomsClientsNonTraites>0) nbreNomsClientsNonTraites--;
        sprintf(buf, "nbreNomsClientsNonTraites tombe à %d",
                nbreNomsClientsNonTraites);
        affThread(numThr, buf);
        pthread_mutex_unlock(&mutexNbreActivites);
        pthread_cond_signal(&condNbreActivites);
    }

    return (void *)vr;
}

char * getThreadIdentity() { ... }
void sleepThread(int ns) { ... }

```

Un exemple d'exécution sur boole donne :

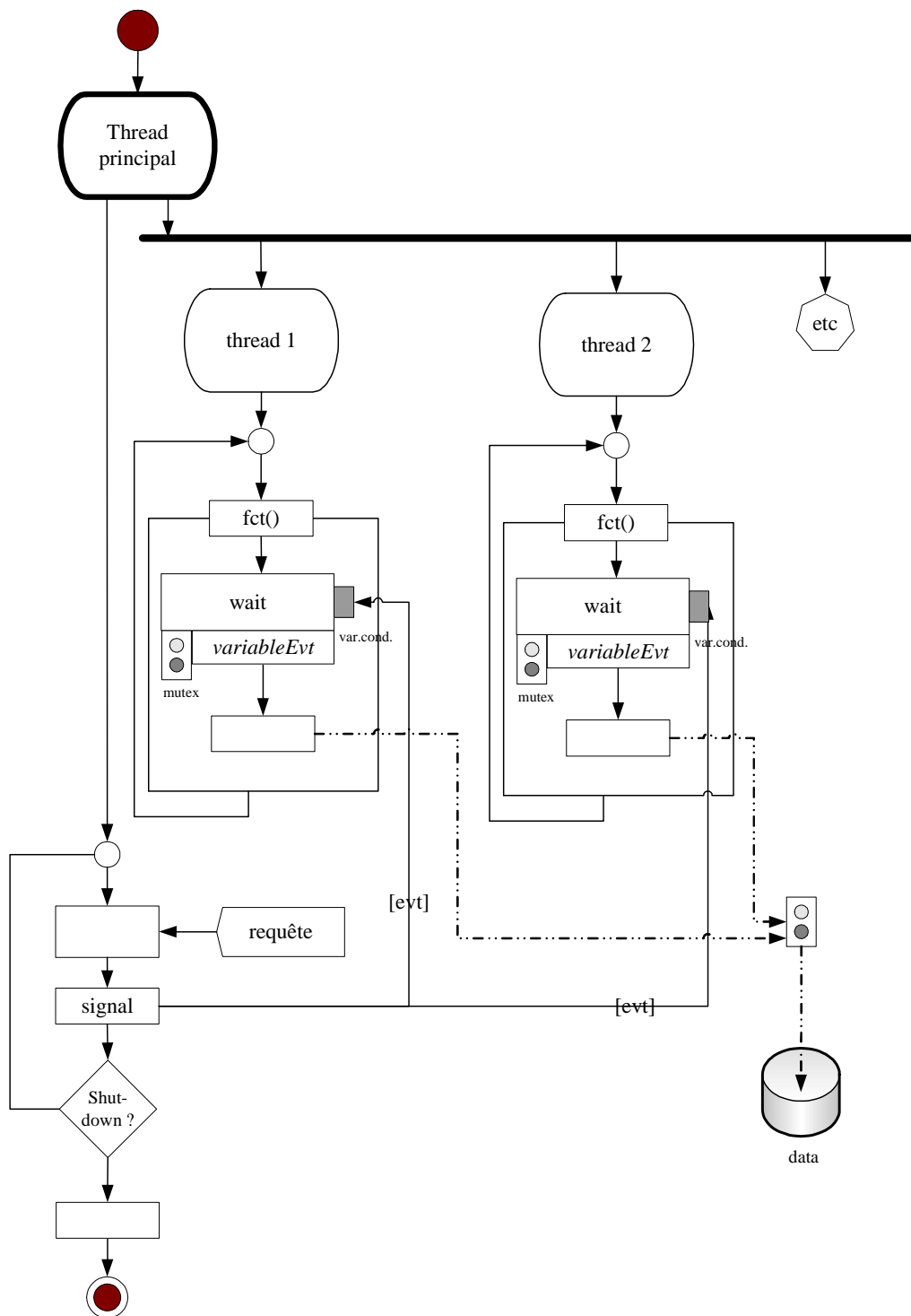
```

boole.inpres.epl.prov-liege.be> m Excellent-Superbe-Sublime-Divin-Infect- Extraordinaire-
Ecoeurant-Mazette-
* Thread principal serveur demarre *

```

```
msgClient = Excellent-Superbe-Sublime-Divin-Infect-Extraordinaire-Ecoeurant-Mazette-
Thread secondaire 0 lance !
Thread secondaire 1 lance !
Thread secondaire 2 lance !
Thread secondaire 3 lance !
Thread secondaire 4 lance !
----- Analyse de msgClient
Fin scan msg client
Attente de la fin du traitement de tous les noms ...
nbreNomsClientsNonTraites = 7
th_75119.5> Je m'occupe de Divin (3) ...
th_75119.3> Je m'occupe de Superbe (1) ...
th_75119.4> Je m'occupe de Sublime (2) ...
th_75119.6> Je m'occupe de Infect (4) ...
th_75119.2> Je m'occupe de Excellent (0) ...
th_75119.2> !-! Temps de reflexion de 2 secondes ...
th_75119.2> Excellent trouve en 0 -> C.R.= 0.950
th_75119.4> !-! Temps de reflexion de 4 secondes ...
th_75119.4> Sublime inconnu
th_75119.6> !-! Temps de reflexion de 6 secondes ...
th_75119.6> Infect inconnu
th_75119.2> nbreNomsClientsNonTraites tombe à 6
th_75119.2> Je m'occupe de Extraordinaire (5) ...
th_75119.5> !-! Temps de reflexion de 9 secondes ...
th_75119.5> Divin trouve en 4 -> C.R.= 0.210
th_75119.4> nbreNomsClientsNonTraites tombe à 5
th_75119.4> Je m'occupe de Ecoeurant (6) ...
th_75119.3> !-! Temps de reflexion de 9 secondes ...
th_75119.3> Superbe trouve en 3 -> C.R.= 0.870
th_75119.4> !-! Temps de reflexion de 1 secondes ...
th_75119.4> Ecoeurant trouve en 6 -> C.R.= 0.670
th_75119.6> nbreNomsClientsNonTraites tombe à 4
th_75119.6> Je m'occupe de Mazette (7) ...
th_75119.5> nbreNomsClientsNonTraites tombe à 3
th_75119.3> nbreNomsClientsNonTraites tombe à 2
th_75119.2> !-! Temps de reflexion de 8 secondes ...
th_75119.2> Extraordinaire trouve en 8 -> C.R.= 0.890
th_75119.4> nbreNomsClientsNonTraites tombe à 1
th_75119.2> nbreNomsClientsNonTraites tombe à 0
th_75119.6> !-! Temps de reflexion de 10 secondes ...
th_75119.6> Mazette inconnu
th_75119.6> nbreNomsClientsNonTraites tombe à 0
Thread secondaire 0 arrete !
Thread secondaire 1 arrete !
Thread secondaire 2 arrete !
Thread secondaire 3 arrete !
Thread secondaire 4 arrete !
** Fin du thread principal **
boole.inpres.epl.prov-liege.be>
```

3.4 Le schéma général du modèle producteur-consommateurs en pool de threads



4. Le modèle du pipeline

4.1 Le principe du modèle

C'est typiquement le modèle utilisé dans les grands programmes de calcul, où l'ampleur de la tâche réclame un éclatement des opérations et un certain travail simultané pour espérer obtenir les résultats dans un délai acceptable. Les principes généraux sont les suivants.

Les threads fonctionnent simultanément selon le principe d'une *chaîne de montage* : ils travaillent sur des données à différents stades de traitement. Donc :

- ◆ un premier thread TH_P produit des données ou les reçoit par une voie de communication quelconque; les données sont placées au fur et à mesure dans une structure de données SD1 (un vecteur, une file, éventuellement un fichier);
- ◆ un deuxième thread TH_2 surveille cette structure de données SD1; dès qu'une donnée y est placée, il s'en empare pour la traiter puis, ce traitement effectué, place le résultat dans une autre structure de données SD2;
- ◆ celle-ci est sous la surveillance d'un troisième thread TH_3; dès qu'une donnée y apparaît, il la traite et place le résultat dans une nouvelle structure de données SD3;
- ◆ et ainsi de suite jusqu'à voir arriver des données dans une structure SDn :
- ◆ un dernier thread TH_R prend la donnée suivante dans SDn, effectue éventuellement un dernier traitement puis fournit le résultat final.

En résumé,

- ◆ le premier thread fournit les données;
- ◆ le dernier thread récupère les résultats;
- ◆ les threads intermédiaires trouvent leurs données dans une structure de données se trouvant *en amont* et placent leurs résultats dans une structure de données se trouvant *en aval*.

Dans ce genre de modèle, tout l'art consiste évidemment à découper judicieusement l'énorme tâche globale en sous-tâches assurant un maximum de fluidité dans la chaîne de traitement.

4.2 Une implémentation : le traitement des commandes de fabrication

Nous allons illustrer cette stratégie dans un programme traitant des commandes et fabricant à la demande ce qui est commandé.

Un premier thread, le thread de **Commande**, permet d'introduire les éléments d'une commande. Cette commande est placée dans une structure de données, soit ici un tableau de structures appelé *listeCommandes*, protégée par un mutex *mutexCommande*. Il reste alors à prévenir le thread qui suit dans la chaîne au moyen de la variable de condition *condCommande*.

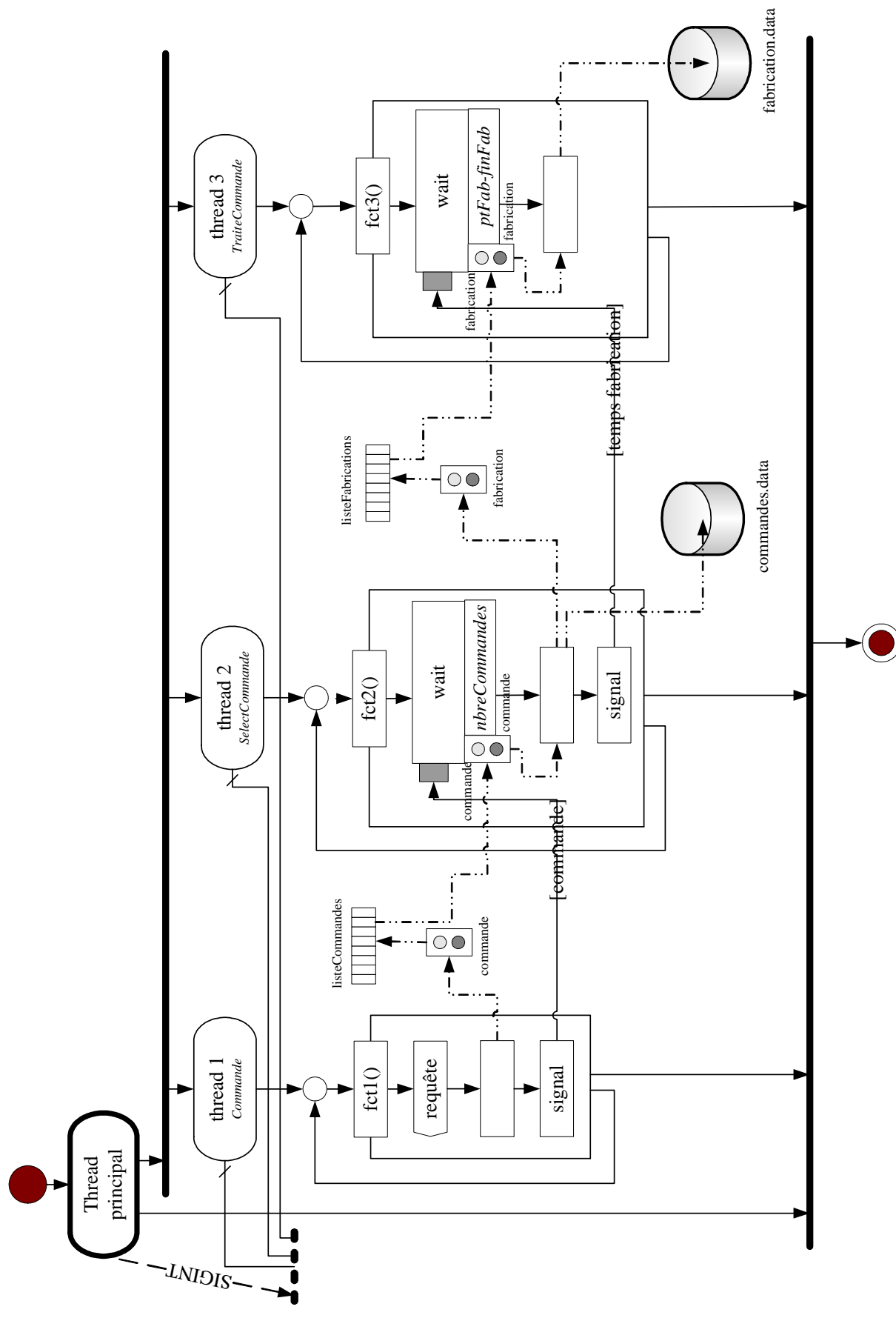
Ce second thread, le thread **SelectCommande**, est donc en attente sur *condCommande*. Une fois activé, il écrit la commande dans un fichier *commandes.data*. Il recherche ensuite le temps de fabrication correspondant à l'article dans le tableau *listeTemps* et ajuste éventuellement les délais. Il insère alors l'ordre de fabrication dans le tableau

listeFabrications protégé par le mutex *mutexFabrication*. C'est alors l'instant de prévenir le troisième thread qui attend sur la variable de condition *condFabrication*.

Ce troisième et dernier thread, le thread **TraiteCommande**, est donc débloqué. Il surveille la fabrication (en dormant ;-)) et écrit le résultat dans le fichier *fabrications.data*.

On peut encore remarquer que tous les threads écrivent dans un fichier de log *commandes.log*, protégé des accès concurrents par le mutex *mutexLog*.

Donc :



MODELEPIPELINE01.C

```
/* MODELEPIPELINE01.C
Claude Vilvens
*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

#define NB_MAX_COMMANDES 50
#define random(n) (rand()%(n))

void * fctThreadCommande(int * param);
void * fctThreadSelectCommande (int * param);
void * fctThreadTraiteCommande (int * param);
void handlerInt (int sig);
void sleepThread(int ns);
void ecritLog( char * buf);

pthread_t threadHandleCommande;
pthread_t threadHandleSelectCommande;
pthread_t threadHandleTraiteCommande;
struct sigaction sigAct;
pthread_mutex_t mutexCommande;
pthread_mutex_t mutexFabrication;
pthread_mutex_t mutexLog;
pthread_cond_t condCommande;
pthread_cond_t condFabrication;

/* ----- */
/* Pour les articles commandes */
struct commande
{
    int num;
    char client[30];
    int quantite;
    int delai;
    char enFabrication;
};

struct commande listeCommandes[NB_MAX_COMMANDES];
int ptCom = 0, finCom = 0;
struct commande acCours;
void insereCommande (struct commande c);

struct commande listeFabrications[NB_MAX_COMMANDES];
int ptFab = 0, finFab = 0;
```

```
void insereFabrication (struct commande c);
/* ----- */
/* Pour les temps de fabrication */
struct temps
{
    int numDeb, numFin;
    int temps;
};

struct temps listeTemps[] =
{
    { 1, 89, 2},
    { 90, 155, 4},
    { 156, 237, 10},
    { 238, 400, 1},
    { 401, 440, 2},
    { 441, 678, 10},
    { 679, 800, 3},
    { 900, 10000, 7}
};

int getTempsFabrication (int num);
/* ----- */
int nbreCommandes = 0;
int *retThread;
int ret1;
FILE * fLog;

int main()
{
    char * buf = (char *)malloc(80), rep, nouveauClient;
    int ret;

    /* Initialisation */
    puts("Thread principal démarre");
    fLog=fopen("commandes.log","w");
    retThread = (int *)malloc(sizeof(int));
    pthread_mutex_init(&mutexCommande, NULL);
    pthread_mutex_init(&mutexFabrication, NULL);
    pthread_mutex_init(&mutexLog, NULL);
    pthread_cond_init(&condCommande, NULL);
    pthread_cond_init(&condFabrication, NULL);

    ret = pthread_create(&threadHandleCommande, NULL,
        (void (*)(void *))fctThreadCommande,0);
    puts("Thread commandes lance !");
    ret = pthread_create(&threadHandleSelectCommande, NULL,
        (void (*)(void *))fctThreadSelectCommande,0);
    puts("Thread de selection des commandes lance !");
    ret = pthread_create(&threadHandleTraiteCommande, NULL,
```

```

        (void (*)(void *))fctThreadTraiteCommande,0);
puts("Thread de traitement des commandes lance !");

sigAct.sa_handler = handlerInt;
if ( (ret=sigaction(SIGINT, &sigAct, 0)) == -1)
    perror("\nErreur de sigaction sur SIGINT");

pthread_join(threadHandleCommande, (void **)&retThread);
pthread_join(threadHandleSelectCommande, (void **)&retThread);
pthread_join(threadHandleTraiteCommande, (void **)&retThread);
puts("*** Fin du thread principal ***");
}

void *fctThreadCommande (int * param)
{
    char *buf = (char*)malloc(80);
    struct commande ac;

    int ancEtat;
    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");

    do
    {
        printf("Nouvel article a commander ...\n");
        pthread_testcancel();
        printf("Numero : ");gets(buf); ac.num=atoi(buf);
        printf("Nom client : "); gets(ac.client);
        printf("Quantite : "); scanf("%d",&ac.quantite);
        printf("Delai : ");scanf("%d", &ac.delai);
        ac.enFabrication = 0;
        fflush(stdin);
        sleepThread(random(5)+1);

        pthread_mutex_lock(&mutexCommande);
        insereCommande(ac);
        nbreCommandes++;
        pthread_mutex_unlock(&mutexCommande);
        pthread_cond_signal(&condCommande);

        sprintf(buf, "Commande> article : %d\n", ac.num); ecritLog(buf);
    }
    while (1);

    /*ret1 = 10;
    pthread_exit(&ret1);*/
    return 0;
}

```

```

void handlerInt (int sig)
{
    puts("--- Arret total ---");
    puts("- Arret de Commande -");
    pthread_cancel(threadHandleCommande);
    puts("- Arret de SelectCommande -");
    pthread_cancel(threadHandleSelectCommande);
    puts("- Arret de TraiteCommande -");
    pthread_cancel(threadHandleTraiteCommande);
}

void * fctThreadSelectCommande (int * param)
{
    FILE * f;
    char *buf = (char *)malloc(80);
    int tempsTrouve, attente;

    int ancEtat;
    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");

    f=fopen("commandes.data","w");
    do
    {
        pthread_mutex_lock(&mutexCommande);
        while (nbreCommandes<=0)
            pthread_cond_wait(&condCommande, &mutexCommande);
        acCours = listeCommandes[ptCom];
        ptCom++;if (ptCom== NB_MAX_COMMANDES) ptCom=0;
        nbreCommandes--;
        pthread_mutex_unlock(&mutexCommande);
        if (acCours.enFabrication==0)
        {
            acCours.enFabrication=1;
            fwrite(&acCours,sizeof(struct commande), 1, f);
            fflush(f);
            attente=random(5)+1;
            sprintf(buf, "Select> temps attente : %d\n", attente);
            ecritLog(buf);
            sleepThread(attente);
            tempsTrouve = getTempsFabrication(acCours.num);
            if (acCours.delai < tempsTrouve) acCours.delai = tempsTrouve;
            pthread_mutex_lock(&mutexFabrication);
            insereFabrication(acCours);
            pthread_mutex_unlock(&mutexFabrication);
            pthread_cond_signal(&condFabrication);

            sprintf(buf, "Select> article : %d\n", acCours.num);

```

```

        ecritLog(buf);
    }
    fflush(fLog);
}
while (1);

pthread_exit(0);
return 0;
}

void * fctThreadTraiteCommande (int * param)
{
    FILE * f;
    char *buf = (char *)malloc(80);
    int attente;

    int ancEtat;
    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancEtat))
        puts("Erreur de setcancelstate");
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ancEtat))
        puts("Erreur de setcanceltype");

    f=fopen("fabrications.data","w");
    do
    {
        pthread_mutex_lock(&mutexFabrication);
        while (ptFab==finFab)
            pthread_cond_wait(&condFabrication, &mutexFabrication);
        acCours = listeCommandes[ptFab];
        ptFab++;if (ptFab== NB_MAX_COMMANDES) ptFab=0;
        pthread_mutex_unlock(&mutexFabrication);
        fwrite(&acCours,sizeof(struct commande), 1, f);
        fflush(f);
        attente = acCours.delai*acCours.quantite;
        sprintf(buf, "Traite> temps attente : %d\n", attente);
        ecritLog(buf);
        sleepThread(acCours.delai*acCours.quantite);
        sprintf(buf, "Traite> article : %d\n", acCours.num);
        ecritLog(buf);
    }
    while (1);

    pthread_exit(0);
    return 0;
}

void sleepThread(int ns) {...}

```

```
void insereCommande (struct commande c)
{
    listeCommandes[finCom]=c;
    finCom++; if (finCom== NB_MAX_COMMANDES) finCom=0;
}

int getTempsFabrication (int num)
{
    char trouve=0;
    int i=0;
    while (1)
    {
        if (num >=listeTemps[i].numDeb && num < listeTemps[i].numFin)
            return listeTemps[i].temps;
        i++;
        if (i==7) return listeTemps[i].temps;
    }
}

void insereFabrication (struct commande c)
{
    listeFabrications[finFab]=c;
    finFab++; if (finFab== NB_MAX_COMMANDES) finFab=0;
}

void ecritLog( char * buf)
{
    time_t t;
    struct tm* pt;
    char * tbuf = (char *)malloc(100);


    pthread_mutex_lock(&mutexLog);
    t = time(0);
    pt = gmtime(&t);
    sprintf(tbuf,"%d:%d:%d]: %s",pt->tm_hour, pt->tm_min, pt->tm_sec, buf);
    fputs(tbuf, fLog);
    fflush(fLog);
    pthread_mutex_unlock(&mutexLog);
}
```

Une exécution sur copernic donne :

```
copernic.inpres.epl.prov-liege.be> m
* Thread principal démarre *
Thread commandes lance !
Thread de selection des commandes lance !
Thread de traitement des commandes lance !
Nouvel article a commander ...
```



```
Numero : 345
Nom client : VIL
Quantite : 10
Delai : 3
Nouvel article a commander ...
Numero : 765
Nom client : BAST
Quantite : 20
Delai : 2
Nouvel article a commander ...
Numero : 987
Nom client : DEL
Quantite : 30
Delai : 1
Nouvel article a commander ...
Numero : 287
Nom client : CLER
Quantite : 25
Delai : 2
Nouvel article a commander ...
Numero : --- Arret total ---
- Arret de Commande -
- Arret de SelectCommande -
- Arret de TraiteCommande -
Nom client : --- Arret total ---
- Arret de Commande -
- Arret de SelectCommande -
- Arret de TraiteCommande -
Quantite : --- Arret total ---
- Arret de Commande -
- Arret de SelectCommande -
- Arret de TraiteCommande -
Delai : --- Arret total ---
- Arret de Commande -
- Arret de SelectCommande -
- Arret de TraiteCommande -
** Fin du thread principal **
copernic.inpres.epl.prov-liege.be>
```



un méchant CTRL-C

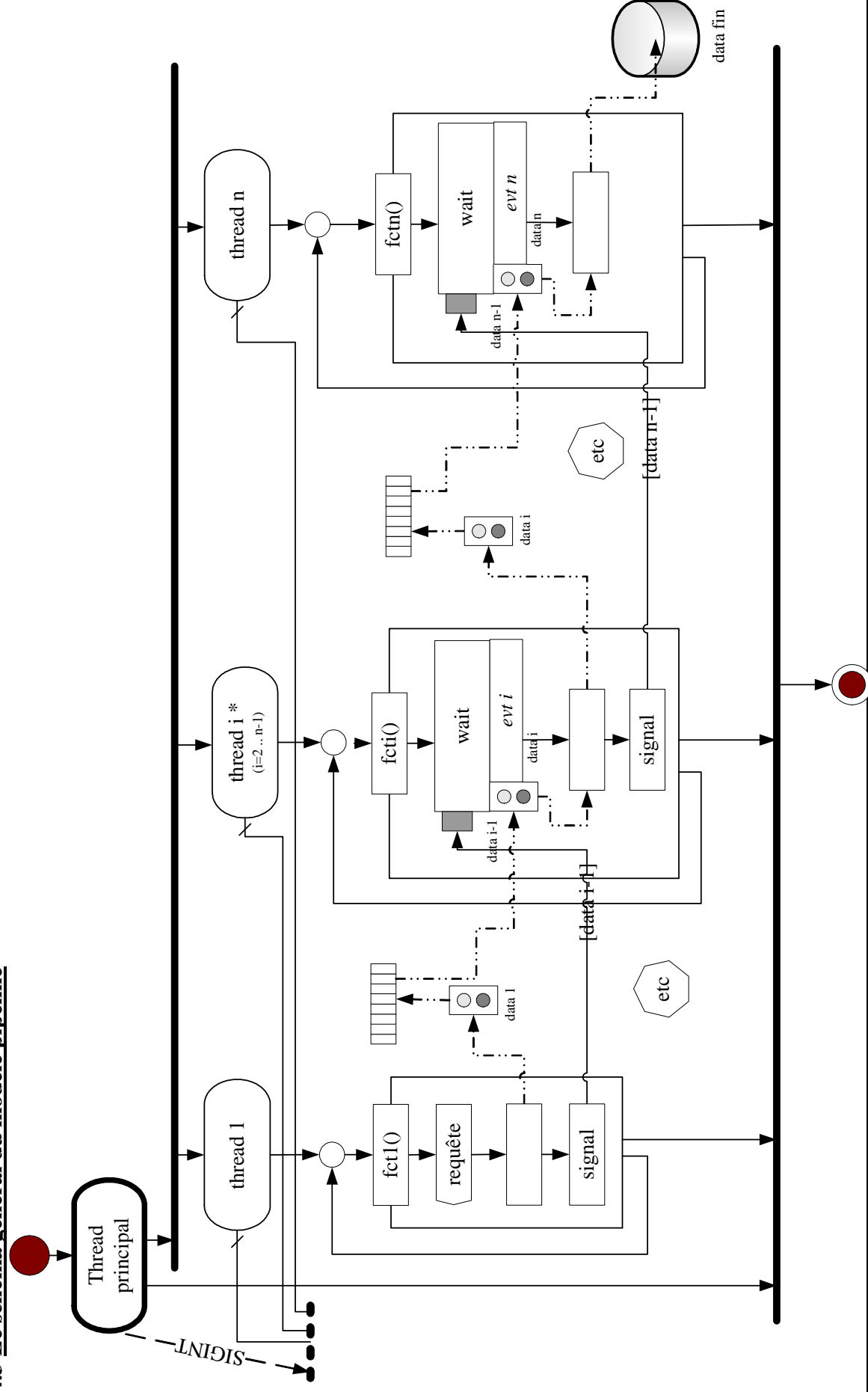
On peut constater que les fichiers résultats existent :

```
copernic.inpres.epl.prov-liege.be> ls -al commandes.data
-rw----- 1 vilvens prof    192 Dec 6 14:32 commandes.data
copernic.inpres.epl.prov-liege.be> ls -al fabrications.data
-rw----- 1 vilvens prof    192 Dec 6 14:34 fabrications.data
copernic.inpres.epl.prov-liege.be>
```

tandis que le fichier log reflète les activités des threads :

```
copernic.inpres.epl.prov-liege.be> cat commandes.log
[13:32:10]: Commande> article : 345
[13:32:10]: Select> temps attente : 4
[13:32:14]: Select> article : 345
[13:32:14]: Traite> temps attente : 30
[13:32:26]: Commande> article : 765
[13:32:26]: Select> temps attente : 1
[13:32:27]: Select> article : 765
[13:32:38]: Commande> article : 987
[13:32:38]: Select> temps attente : 3
[13:32:41]: Select> article : 987
[13:32:44]: Traite> article : 987
[13:32:44]: Traite> temps attente : 40
[13:32:49]: Commande> article : 287
[13:32:49]: Select> temps attente : 5
[13:32:54]: Select> article : 287
[13:33:24]: Traite> article : 287
[13:33:24]: Traite> temps attente : 30
[13:33:54]: Traite> article : 987
[13:33:54]: Traite> temps attente : 50
[13:34:44]: Traite> article : 287
copernic.inpres.epl.prov-liege.be>
```

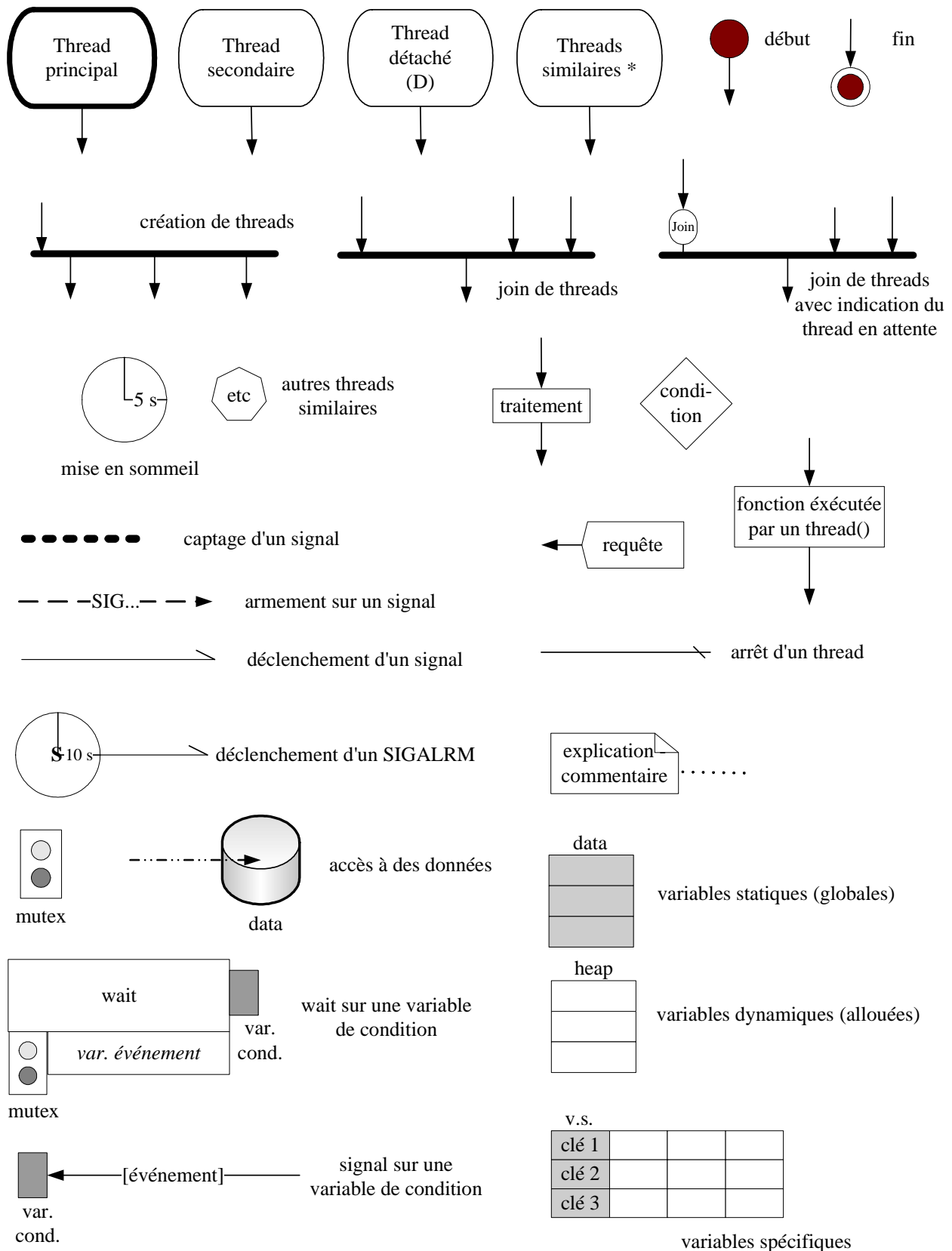
4.3 Le schéma général du modèle pipeline



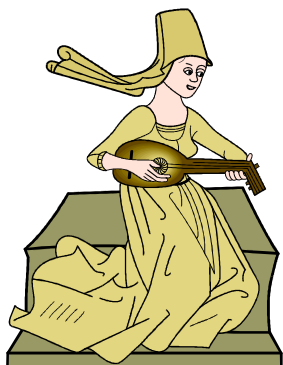


Nous en resterons là dans notre promenade au pays des threads POSIX. Mais il ne tient qu'à nous de les retrouver dans d'autres environnements, par exemple en programmation Windows ou en programmation Java. Mais dans ce cas, les reconnâitrons-nous ?

Annexe 1 : Les symboles utilisés dans les schémas



Annexe 2 : Les librairies sous UNIX



*Je veux bien faire la belle,
mais pas dormir au bois
Je veux bien être Reine,
mais pas l'ombre du Roi*

(Zazie, A ma place)

1. L'objectif : réutiliser

Les librairies¹ constituent l'un des éléments fondamentaux de l'arsenal des outils de programmation. Elles contribuent à la réalisation d'un objectif omniprésent dans les démarches de la programmation : *la réutilisation, au sein d'une nouvelle application, de code déjà écrit, déjà testé, déjà mis au point et déjà compilé dans un autre contexte*. Tous les systèmes d'exploitation dignes de ce nom proposent cette notion. UNIX les possède donc aussi, offrant les librairies "système" (comme celle des *threads* ...) et la possibilité de créer ses propres librairies.

Dans sa forme la plus simple, on pourrait considérer qu'une librairie se présente sous la forme du résultat d'une compilation, soit un fichier objet d'extension .o. Mais la notion est plus large : une librairie est en fait un fichier qui contient un ou plusieurs fichiers objets, ces derniers étant appelés des **modules** ou des **membres** de la librairie. Ces modules sont mémorisés selon un certain format, avec une *table* associant les noms symboliques de la librairie (par exemple, le nom d'une fonction) au module dans lequel le symbole est effectivement défini. Ceci permet de retrouver rapidement la définition d'un symbole quelconque lors du linkage (pardon, je voulais dire "l'édition de liens") de la librairie avec les autres fichiers objets de l'application.

2. Les librairies statiques et les librairies dynamiques

En fait, UNIX (tout comme Windows d'ailleurs) distingue deux types de librairies.

Une librairie est dite **statique** si son code objet est lié à celui de l'application qui l'utilise (autrement dit, physiquement copié dans celui-ci) pour fournir un exécutable.

Cet exécutable est "auto-suffisant" : il peut fonctionner sur n'importe quelle machine UNIX compatible. On peut cependant remarquer que si plusieurs applications utilisent la même librairie statique, chaque exécutable contiendra le code de la librairie. De plus, si la librairie est modifiée ultérieurement, il faudra relinker toutes les applications utilisatrices avec la nouvelle version.

Une librairie est par contre dite **dynamique** si le code de ses fonctions est lié et chargé seulement au moment de l'exécution de l'application qui l'utilise (au "runtime").

¹ comme on sait, "librairie" est ici une traduction malheureuse du mot anglais "library" – il faudrait dire "bibliothèque" ou alors utiliser "bookshop" ;-) ...

Les librairies dynamiques présentent un certain nombre d'avantages :

- ◆ leur code n'est pas lié statiquement avec celui de chacune des applications qui les utilisent; ceci implique que l'on occupera inévitablement ***moins d'espace disque*** puisque le code des applications sera moins grand.
- ◆ leur code est donc en fait ***partagé*** par les applications utilisatrices – c'est pour cette raison que le monde UNIX parle encore de "librairies partagées" [*shared library*]. Elles ne sont donc ***chargées en mémoire qu'une seule fois*** (lors de la première utilisation); ceci implique que l'on occupera inévitablement moins d'espace mémoire.
- ◆ la production d'une ***nouvelle version*** de la librairie n'implique pas le relinkage des applications utilisatrices : il suffit que cette nouvelle version soit installée à la place de l'ancienne.

Evidemment, le (pseudo-)exécutable n'est plus "auto-suffisant" : il ne peut fonctionner que sur une machine UNIX sur laquelle la librairie partagée a été installée¹ ...

3. Les conventions

Pour faciliter la gestion des librairies, un certain nombre de conventions ont été établies :

- ◆ tous les noms de librairies débutent par le préfixe "**lib**";
- ◆ les librairies statiques se trouvent dans des fichiers d'extension **.a** [*archive*] – il s'agit en fait de fichiers créés au moyen de l'utilitaire *ar* (nous y reviendrons);
- ◆ les librairies dynamiques se trouvent dans des fichiers d'extension **.so** [*shared object*].

En fait, la désignation des librairies dynamiques est un peu plus compliquée à cause des numéros de version. Un fichier .so est en fait un lien vers un fichier dont le nom complet suit le format :

<nom librairie>.<n° principal de version>.<n° secondaire de version>.<n° de patch>
--

Par exemple, si nous nous intéressons à la librairie dynamique des threads sur la machine Copernic :

```
copernic> ls -l /usr/shlib/libpthread.so
```

```
| lrwxr-xr-x 1 root system 25 Feb 12 2001 /usr/shlib/libpthread.so -> ../shlib/libpthread.so
```

donc il s'agit d'un lien vers /shlib/libpthread.so qui lui est un vrai fichier :

```
| -rw-r--r-- 1 bin bin 590688 Apr 5 2000 libpthread.so
```

Bien sûr, un nouveau numéro principal signifie que la nouvelle version est incompatible avec l'ancienne, un nouveau numéro secondaire signifie que la nouvelle version

¹ l'application avec librairie statique est un voyageur qui emmène sa tente et ses vivres, l'application avec librairie dynamique va à l'hôtel ...

est cette fois compatible tandis qu'un nouveau numéro de patch indique que quelques bugs ont été corrigés.

En réalité, le link dynamique n'est réalisé ni sur le nom raccourci (.so), ni sur le nom complet (par exemple, .so.2.1.8), mais sur un nom de forme intermédiaire, le **soname**. Ce soname est *le nom de la librairie complété du numéro principal de version* (pour notre exemple, .so.2). Faut-il le préciser ? Des liens symboliques relient les trois formes de nom ...

Remarque

Pour la version archive statique, on obtient pour ls -l :

```
| lrwxr-xr-x 1 root system 23 Feb 12 2001 /usr/lib/libpthread.a -> ../ccs/lib/libpthread.a
```

4. L'exploration d'une librairie existante

Avant de penser à construire ses propres librairies, on peut se demander comment utiliser les librairies existantes. Bien sûr, chaque librairie correspond à un header d'extension **.h** qui fournit la liste des symboles définis et des prototypes des fonctions fournies. Il est cependant possible d'introspecter un fichier libraire au moyen de la commande :

nm [options] <nom de fichier>

Une option intéressante est -mangled_name_only

copernic> **nm /usr/shlib/libpthread.so**

Ce qui donne (en abrégé) :

Name	Value	Type	Size
.bss	0004396974564896	B	0000000000000000
.data	0004396974538752	D	0000000000000000
.fini	0004395905101200	F	0000000000000000
.init	0004395905101120	I	0000000000000000
.lit4	0004396974558560	G	0000000000000000
.lit8	0004396974558560	G	0000000000000000
.pdata	0004395904889952	P	0000000000000000
.rconst	0004395904859616	Q	0000000000000000
.rdata	0004396974554720	R	0000000000000000
.sbss	0004396974564384	S	0000000000000000
.sdata	0004396974558560	G	0000000000000000
.text	0004395904896224	T	0000000000000000
.xdata	0004396974555104	X	0000000000000000
_BASE_ADDRESS	0004395904794624	A	0000000000000008
_DYNAMIC_LINK	0000000000000002	A	0000000000000008
_F64_stat	0000000000000000	U	0000000000000008
_OtsDivide32	0000000000000000	U	0000000000000008
...			
_Psigwait	0004395904993456	T	0000000000000008
__Argc	0000000000000004	U	0000000000000000
__Argv	0000000000000008	U	0000000000000000
__CFE_fprint_puts_nl	0000000000000000	U	0000000000000008
...			
__krnAllowPreempt	0004395904930144	T	0000000000000008
__krnAllowResched	0004395904930176	T	0000000000000008

__kernBlockPreempt	0004395904930192 T 0000000000000008
__kernBlockResched	0004395904930224 T 0000000000000008
...	
__malloc_mutex_count	0000000000000000 U 0000000000000000
__malloc_mutex_list	0000000000000000 U 0000000000000000
__muDefAttr	0004396974542952 D 0000000000000000
...	
__pthread_cond_wait	0004395904903376 T 0000000000000008
__pthread_create	0004395905007136 T 0000000000000008
__pthread_create_hook	0004396974558736 G 0000000000000000
...	
__pthread_mutex_init	0004395904943952 T 0000000000000008
__pthread_mutex_lock	0004395904944768 T 0000000000000008
__pthread_mutex_trylock	0004395904945792 T 0000000000000008
__pthread_mutex_unlock	0004395904946000 T 0000000000000008
__pthread_once	0004395904946416 T 0000000000000008
...	
abort	0000000000000000 U 0000000000000008
...	
malloc	0000000000000000 U 0000000000000008
memcpy	0000000000000000 U 0000000000000008
...	
pthread_key_create	0004395905028160 T 0000000000000008
...	
pthread_mutexattr_init	0004395904942560 T 0000000000000008
...	
sigaction	0000000000000000 U 0000000000000008
...	
sprintf	0000000000000000 U 0000000000000008
strcat	0000000000000000 U 0000000000000008
...	
write	0000000000000000 U 0000000000000008

L'affichage résultant fournit non seulement les symboles, mais utilise aussi une initiale pour chacun d'entre eux indiquant la manière dont le symbole est défini (ou non défini). Les plus courantes sont :

- ◆ **T** : le symbole est bien défini (il apparaît en *Text*);
- ◆ **U** : le symbole n'est pas défini [*Undefined*] dans la librairie, mais celle-ci l'utilise;
- ◆ **W** : le symbole est défini, mais pourrait être redéfini dans une autre librairie [*Weak*];
- ◆ **D, G** : le symbole correspond à une donnée qui est initialisée ailleurs.

5. Les librairies utilisées par un programme

La commande **ldd** permet d'obtenir la liste des librairies dynamiques utilisées par des exécutables ou des objets partagés. Ainsi, si *c* est un exécutable qui utilise des threads :

```
copernic> ldd c
Main => c
libpthread.so => /usr/shlib/libpthread.so
libc.so => /usr/shlib/libc.so
libexc.so => /usr/shlib/libexc.so
copernic>
```

6. La construction d'une librairie statique

Pour fixer les idées, supposons donc avoir écrit une splendide fonction calculant le $n^{\text{ème}}$ terme d'une progression arithmétique dont on fournit le premier terme et la raison. Cela pourrait donner :

```
progArith.h
#ifndef PROGARITH_H
#define PROGARITH_H

float terme (float t1, float r, int n);
/* fournit le n eme terme d'une PA de raison r et de 1er terme t1 */

#endif
```

et

```
progArith.c
#include "progArith.h"

float terme (float t1, float r, int n)
/* fournit le n eme terme d'une PA de raison r et de 1er terme t1 */
{
    puts("Hello");
    return t1 + (n-1)*r;
}
```

En faire une librairie statique n'est pas bien compliqué. Commençons par construire le fichier objet :

```
copernic> cc -c progArith.c
```

ce qui nous produit un fichier **progArith.o**. Ensuite, nous allons effectivement créer le fichier librairie en utilisant la commande d'archivage :

```
ar [options] <fichier archive> <fichier(s)>
```

Les options les plus couramment utilisées sont :

options	effet
-c	crée le fichier archive précisé si il n'existe pas encore
-r	place les fichiers précisés dans l'archive; les symboles existant sont remplacés par leur nouvelle version, les nouveaux symboles sont placés à la fin
-s	met à à la table des associations symbole-membre

Pour ce qui nous intéresse, si nous choisissons de baptiser notre librairie libpa (pour rappel, débiter par "lib" est une convention), cela donnera :

```
copernic> ar rcs libpa.a progArith.o
```

On peut constater que le fichier libpa.a a bien été créé. Son contenu peut être vérifié avec la commande nm :

```
copernic> nm -mangled_name_only libpa.a

libpa.a[progArith.o]:

Name                               Value      Type      Size
_fpdata                            | 0000000000000000 | U | 0000000000000000
puts                               | 0000000000000000 | U | 0000000000000008
terme                             | 0000000000000000 | T | 0000000000000008
copernic>
```

Effectivement, seul le symbole "terme" est défini au sein de cette librairie, tandis que le symbole "puts" est bien utilisé mais non défini.

7. L'utilisation d'une librairie statique

Donnons-nous à présent un petit programme utilisant notre fabuleuse librairie :

```
testProgArith.c
/* test de progArith */
#include <stdio.h>

#include "progArith.h"

int main ()
{
    float a,r;
    int num;

    scanf("%f",&a); /* c'était vraiment trop dur de placer quelques printf() ??? ;-) */
    scanf("%f", &r);
    scanf("%d",& num);
    printf("Resultat = %f\n", terme(a,r,num));

    return 0;
}
```

Formidable, n'est-il pas ? Il nous faut donc le compiler et réaliser l'édition de liens avec notre bibliothèque. Celle-ci sera désignée selon le commutateur :

-l<nom de la librairie dont on a retiré "lib">

On remarquera l'utilisation de la convention selon laquelle une librairie porte toujours un nom débutant par "lib". Dans notre cas, cela donnera donc **-lpa** – ceci rappelle bien **-lpthread** ou **-lm**. Mais, précisément, ces librairies système se trouvent dans des répertoires prédéfinis (usuellement /usr/lib)). Ce n'est pas le cas de notre librairie : il nous faut donc préciser dans quel répertoire elle se trouve au moyen du commutateur

-L<répertoire où se trouve la librairie>

Pour nous, ce sera le répertoire courant. Il nous faut aussi préciser que notre librairie est statique : en effet, de nombreux UNIX (dont Linux) supposent par défaut que la librairie évoquée est dynamique, ce qui n'est pas le cas ici. Le commutateur nécessaire pour cc est **-non_shared** (**-static** sous Linux).

copernic> **cc -o progA testProgArith.c -non_shared -L. -lpa**

Un petite makefile (le strict minimum) en résumé :

```
progA: testProgArith.c libpa.a
    cc -o progA testProgArith.c -non_shared -L. -lpa

libpa.a: progArith.o
    ar rcs libpa.a progArith.o

progArith.o: progArith.c progArith.h
    cc -c progArith.c
```

et on peut vérifier que notre exécutable progA fonctionne bien :

```
copernic> progA
23
42
2
Hello
Resultat = 65.000000
copernic>
```

C'est bien juste ...

8. La construction d'une librairie dynamique

Reprenons notre progression arithmétique et entreprenons d'en faire une librairie dynamique. Bien sûr, nous supposons disposer déjà de progArith.o. Il nous faut à présent produire un objet partageable que les autres exécutables pourront utiliser au moment de leur exécution. Il suffit de le demander à cc en usant du commutateur **-shared**. Celui-ci se complète à son tour de diverses options dont nous retiendrons seulement :

-soname <soname de la librairie partagée>

Nous en arrivons ainsi à :

copernic> **cc -shared -soname libparith.so.1 -o libparith.so.1.0.0 progArith.o**

Le résultat est la production du fichier **libparith.so.1.0.0** :

```
-rw----- 1 vilvens prof 18592 Nov 26 12:35 libparith.so.1.0.0
```

et l'apparition du fichier **so_locations**, dont le contenu est :

```
libparith.so.1 \
```

```
:st = .text 0x000003ff bfff0000, 0x00000000000010000:\n:st = .data 0x000003ffff ffff0000, 0x00000000000010000:\n
```

En fait, on peut vérifier que la variable DT_SONAME est positionnée. Par défaut, elle désigne le nom du fichier de sortie.

Comme notre librairie ne se trouve pas dans /usr ou /usr/lib, il nous appartient de créer les liens symboliques entre ce fichier et ses correspondants soname et link. Pour ce faire, on utilise la commande des liens :

```
ln [-s] <fichier source> <alias cible>
```

Par défaut, ln produit des liens 'hardware'; nous utiliserons ici des liens 'symboliques' (commutateur -s). Pour ce qui nous concerne, nous allons lier :

- ◆ le fichier objet partagé et le soname, ce qui sera fort utile au *loader dynamique* : donc

```
copernic> ln -s libparith.so.1.0.0 libparith.so.1
```

ce qui produit l'apparition de libparith.so.1;

```
| lrwxrwxrwx  1 vilvens  prof   18 Nov 26 12:51 libparith.so.1 -> libparith.so.1.0.0
```

tandis que

```
copernic> ls -l libparith.so.1.0.0
```

```
| -rw-----  1 vilvens  prof   18592 Nov 26 12:35 libparith.so.1.0.0
```

Attention ! Les liens symboliques ne se voient pas dans la ligne du répertoire (seulement les liens hardware) ...

- ◆ le soname et le fichier .so qui sera pris en compte par le *linker ld* (qui verra sur sa ligne de commande -lparith, d'après les conventions de librairies) : donc

```
copernic> ln -s libparith.so.1 libparith.so
```

avec production de libparith.so

```
| lrwxrwxrwx  1 vilvens  prof   14 Nov 26 12:56 libparith.so -> libparith.so.1
```

et au total :

```
lrwxrwxrwx  1 vilvens  prof   14 Nov 26 12:56 libparith.so -> libparith.so.1\nlrwxrwxrwx  1 vilvens  prof   18 Nov 26 12:51 libparith.so.1 -> libparith.so.1.0.0\n-rw-----  1 vilvens  prof   18592 Nov 26 12:35 libparith.so.1.0.0
```

9. L'utilisation d'une librairie dynamique

Nous pouvons à présent envisager de tester notre librairie avec le même programme de test que celui que nous avons utilisé pour la librairie statique :

```
copernic> cc testProgArith.c -o testShared -L. -lparith
```

Tout semble logique, la compilation s'effectue mais pourtant une tentative d'exécution de testShared donnera :

```
| 274072:testshared: /sbin/loader: Fatal Error: cannot map libparith.so.1
```

En fait, le chargeur/linker dynamique **ld.so**, qui est donc responsable du link dynamique, utilise une variable d'environnement, appelée **LD_LIBRARY_PATH**, dans laquelle il trouve les chemins dans lesquels il doit rechercher les librairies dynamiques. Il convient donc de modifier cette variable (ici, elle n'existe pas encore et nous pouvons la définir directement) :

```
copernic> setenv LD_LIBRARY_PATH `pwd`
```

On peut vérifier que la variable d'environnement est bien présente :

```
| copernic> env  
TERM=vt220  
SHELL=/bin/csh  
USER=vilvens  
...  
LD_LIBRARY_PATH=/prof/vilvens/c  
copernic>
```

On peut à présent compiler le programme de test :

```
copernic> cc testProgArith.c -o testShared -L. -lparith
```

L'exécutable obtenu fonctionne parfaitement :

```
| copernic> testShared  
67  
23  
45  
Hello  
Resultat = 1079.000000  
copernic>
```

On peut vérifier l'utilisation de la librairie par :

```
| copernic> ldd testShared  
Main => testShared  
libparith.so.1 => /prof/vilvens/c/libparith.so.1  
libc.so => /usr/shlib/libc.so  
copernic>
```

10. L'utilisation d'une librairie dynamique (bis)

Si on place le programme de test dans un autre répertoire (disons cplus), il faut évidemment changer la directive d'inclusion :

```
#include "../c/progArith.h"
```

Faisons quelques essais :

```
copernic> cc testProgArith.c -o testShared -lparith
```

Bof ...

```
ld:
Can't locate file for: -lparith
copernic>
```

Evidemment, il faut rectifier l'indication du répertoire de la librairie :

```
copernic> cc testProgArith.c -o testShared -L../c -lparith
```

Tout est alors en ordre :

```
copernic> testShared
345
43523
323
Hello
Resultat = 14014751.000000
copernic>
```

Attention : `so_locations` doit rester dans son répertoire initial. Si il est dans déplacé dans un autre répertoire, disons 'tcp' :

```
copernic> cc testProgArith.c -o testShared -L../tcp -lparith
```

donne

```
ld:
Can't locate file for: -lparith
```

Même avec :

```
setenv LD_LIBRARY_PATH ../tcp
```

on n'obtient rien de mieux ...

<h2>Ouvrages consultés</h2>

Fowler, M. Le tout en poche : UML. Paris, France. Ed. CampusPress. 2002.

Janssens, A. UNIX sous tous les angles. Paris, France. Ed. Eyrolles. 1992.

Richter, J. Développer sous Windows 95 et Windows NT 4.0. Les Ulis, France. Microsoft Press. 1997.

Rifflet, J.M. La communication sous UNIX – Applications réparties. Paris, France. Ediscience international. 1995.

Stevens, W.R. UNIX networking programming – Networking APIs : Sockets and XTI (Volume 1). U.S.A. Prentice Hall Pub. 1998.

Tanenbaum, R. Réseaux. Paris/London, France/United Kingdom. InterEditions & Prentice Hall International. 1997.

Wall, K. et al. Linux programming. Indianapolis, Indiana, U.S.A. SAMS. 2001.