Chapitre 16

Les threads POSIX

La programmation par thread (actvité) est naturelle pour gérer des phénomènes asyncrones. Les entrées utilisateur dans les interfaces graphiques (souris, clavier) sont plus facile a gérer si l'on peut séparer l'activité principale du logiciel de la gestion des commandes utilisateur. Les entrées sorties multiples voir le chapitre 15 correspondant, sont gérées plus simplement en utilisant des threads.

Les actvitiés sont une nouvelle façon de voire les processus dans un système. L'idée est de séparer en deux le concept de processus. La première partie est l'environement d'exécution, on y retrouve une très grande partie des éléments constitutifs d'un processus en particulier les informations sur le propriétaire, la position dans l'arborescence le masque de création de fichier etc. La deuxième partie est l'`activité`, c'est la partie dynamique, elle contient une pile, un context processeurs (pointeur d'instruction etc), et des données d'ordonancement.

L'idée de ce découpage est de pouvoir associer plusieurs activité au même environement d'exécution. Pour CHORUS l'ensemble des ressources d'un environnement d'exécution est appelé des acteurs, MACH parle de tâches et AMOEBA de process. Mais tous désigne l'unité d'exécution par le terme de thread of control.

Organisation en mémoire pour un processus UNIX avec plusieurs threads : voir figure 16.1.

On peut grace au thread gérer plusieurs phénomènes asyncrone dans le même contexte, c'est à dire, un espace d'adressage commun, ce qui est plus confortable que de la mémoire partagée et moins couteux en ressource que plusieurs processus avec un segment de mémoire partagé.

Un processus correspond à une instance d'un programme en cours d'éxécution. Un thread correspond à l'activité d'un processeur dans le cadre d'un processus. Un thread ne peut pas exister sans processus (la tâche englobante), mais il peut y a voir plusieurs thread par processus, dans le cas de linux il ne peut y a voir de tâche sans au moins une activité.

16.0.1 Description

Un processus est composé des parties suivantes : du code, des données, une pile, des descripteurs de fichiers, des tables de signaux. Du point de vue du noyau, transférer l'exécution à un autre processus revient à rediriger les bons pointeurs et recharger les registres du processeur de la pile. Les divers threads d'un même processus peuvent partager certaines parties : le code, les données, les descripteurs de fichiers, les tables de signaux. En fait, ils ont au minimum leur propre pile, et partagent le reste.

16.0.2 `fork` et `exec`

Après un `fork`, le fils ne contient qu'une seule activité (celle qui a exécuté le `fork`). Attention aux variables d'exclusion mutuelle (qui font partie de l'espace d'adressage partagé) qui sont

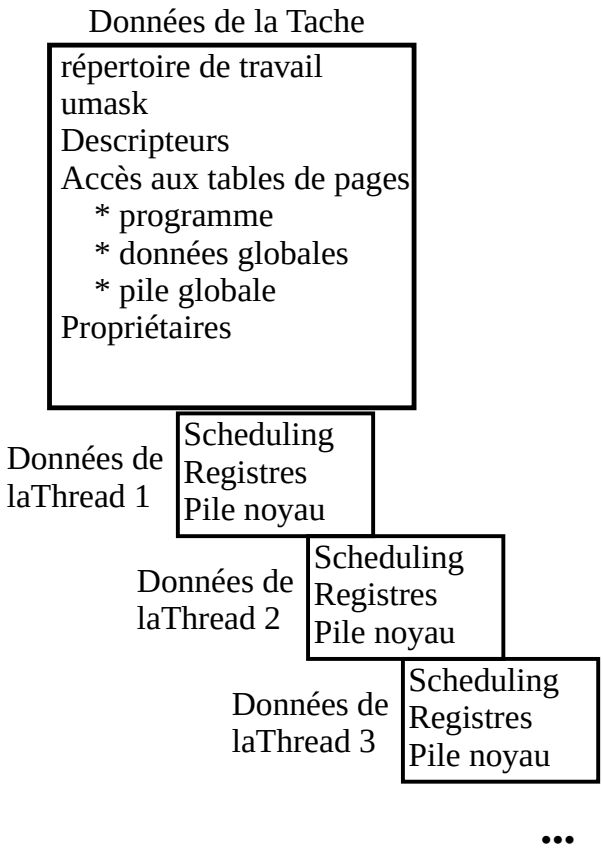

FIG. 16.1 – Organisation mémoire, partage des fonctions entre le processus et les activités

conservées après le `fork()` et dont le contenu ne varie pas. Ainsi si une activité a pris le sémaphore avant le `fork()`, si l'activité principale cherche à prendre ce sémaphore après le `fork()` elle sera indéfiniment bloquée.

Après un `exec`, le processus ne contient plus que la thread qui a exécuté l'une des six commandes `exec`. Pas de problème avec les sémaphores comme l'espace d'adressage a changé.

16.0.3 clone

Sous linux (et rarement sous les systèmes Unix) il existe un appel système un peut spécial. Cet appel système réalise un dédoublement du processus comme `fork` d'ou son nom de `clone`. Cet appel système permet de préciser exactement ce que l'on entend partager entre le processus père et le processus fils.
Eléments partageables :

ppid Création d'un frère au lieux d'un fils.

FS Partage de la structure d'information liée au système de fichier (".", "/", umask),

FILES Partage de la table des descripteurs,

SIGHAND Partage de la table des gestionnaires de Signaux, mais pas des masques de signaux,

PTRACE Partage du "crochet" (hook) de debug voire l'appel ptrace.

VFORK Partage du processeur ! le processus père est bloqué tantque le fils n'a pas exécuté soit `_exit` soit `execve`, c'est à dire qu'il s'est détaché de tout les élément partageable du processus père (sauf les FILEs),

VM Partage de la mémoire virtuelle, en particulier les allocations et désallocations par `mmap` et `munmap` sont visibles par les deux proccessus.

pid Les deux processus ont le même numéro.

THREAD Partage du groupe de thread, les deux processus sont ou ne sont pas dans le même groupe de threads.

16.0.4 Les noms de fonctions

`pthread[_objet]_operation[_np]`

où

objet désigne si il est présent le type de l'objet auquel la fonction s'applique. Les valeurs possibles de objet peuvent être

`cond` pour une variable de condition

`mutex` pour un sémaphore d'exclusion mutuelle

opération désigne l'opération a réaliser, par exemple `create`, `exit` ou `init`

le suffixe `np` indique, si il est présent, qu'il s'agit d'une fontion non portable, c'est-à-dire Hors Norme.

16.0.5 les noms de types

`pthread[_objet]_t`

avec `objet` prenant comme valeur `cond`, `mutex` ou rien pour une thread.

16.0.6 Attributs d'une activité

Identification d'une pthread : le TID de type pthread_t obtenu par un appel à la primitive :

```
pthread_t pthread_self(void);
```

pour le processus propriétaire

```
pid_t getpid(void);
```

En POSIX, le fait de tuer la thread de numéro 1 a pour effet de tuer le processus ainsi que toutes les autres threads éventuelles du processus.

Pour tester l'égalité de deux pthreads on utilise

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

16.0.7 Création et terminaison des activités

Création

```
int pthread_create (pthread_t      *p_tid,
                    pthread_attr_t attr,
                    void           *(*fonction) (void *arg),
                    void           *arg
                   );
```

La création et l'activation d'une activité retourne -1 en cas d'echec, 0 sinon.

- le tid de la nouvelle thread est placé à l'adresse `p_tid`
- `attr` attribut de l'activité (ordonnancement), utiliser `pthread_attr_default`
- la paramètre `fonction` correspond à la fonction exécutée par l'activité après sa création : il s'agit donc de son point d'entrée (comme la fonction `main` pour les processus). Un retour de cette fonction correspondra à la terminaison de cette activité.
- le paramètre `arg` est transmis à la fonction au lancement de l'activité.

Terminaison

a) les appels UNIX `_exit` et donc `exit` terminent toutes les threads du processus.
b) Terminaison d'une thread

```
int pthread_exit (int *p_status);
```

`p_status` code retour de la thread, comme dans les processus UNIX la thread est zombifiée pour attendre la lecture du code de retour par une autre thread. A l'inverse des processus, comme il peut y avoir plusieurs threads qui attendent, la thread zombie n'est pas libérée par la lecture du p_status, il faut pour cela utiliser une commande spéciale qui permettra de libérer effectivement l'espace mémoire utilisé par la thread.

Cette destruction est explicitement demandée par la commande

```
int pthread_detach (pthread_t *p_tid);
```

Si un tel appel a lieu alors que l'activité est en cours d'exécution, cela indique seulement qu'à l'exécution de `pthread_exit` les ressources seront restituées.

16.1 Synchronisation

Trois mécanismes de synchronisation inter-activités :

- la primitive `join`
- les sémaphores d'exclusion mutuelle
- les conditions (évènements)

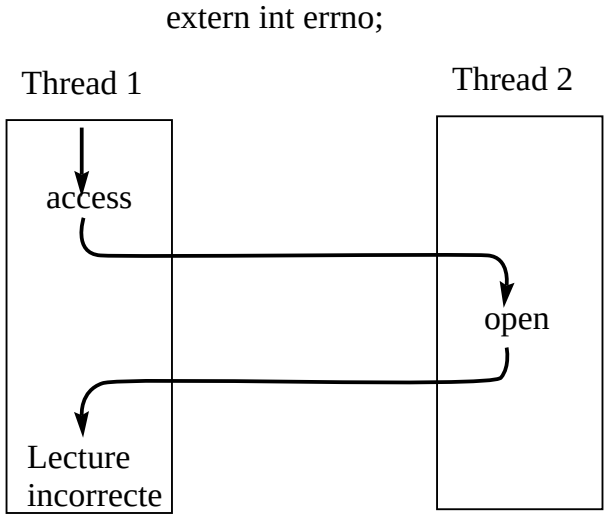

FIG. 16.2 – Changement de la valeur `errno` par une autre thread

16.1.1 Le modèle fork/join (Paterson)

Les rendez-vous : join
La primitive

```
int pthread_join (pthread_t tid, int **status);
```

permet de suspendre l'exécution de l'activité courante jusqu'à ce que l'activité `tid` exécute un appel (implicite ou explicite) à `pthread_exit`. Si l'activité `tid` est déjà terminée, le retour est immédiat, et le code de retour de l'activité visée est égal à **status (double indirection).
La primitive retourne :
0 en cas de succès
-1 en cas d'erreur

EINVAL si le `tid` est incorrect

ESRCH activité inexistante

EDEADLOCK l'attente de l'activité spécifiée conduit à un interblocage.

16.1.2 Le problème de l'exclusion mutuelle sur les variables gérées par le noyau

Il est nécessaire d'avoir plusieurs variables `errno`, une par activité. En effet cette variable globale pourrait être changée par une autre activité. Voir plus loin comment définir des variables globales locales à chaque activité.

16.1.3 Les sémaphores d'exclusion mutuelle

Ces sémaphores binaires permettent d'assurer l'exclusion mutuelle.
- Il faut définir un objet de type `pthread_mutex_t` qui correspond à un ensemble d'attributs de type `pthread_mutexattr_t`
  (on utilisera en général la constante `pthread_mutexattr_default`).
- Initialiser la variable par un appel à la fonction
  ```
  int pthread_mutex_init(pthread_mutex_t     *p_mutex,
                         pthread_mutexattr_t attr);
  ```
- On pourra détruire le sémaphore par un appel à la fonction
  ```
  int pthread_mutex_destroy(pthread_mutex_t *p_mutex);
  ```

16.1.4 Utilisation des sémaphores

Opération P :
Un appel à la fonction

```
pthread_mutex_lock (pthread_mutex_t *pmutex);
```

permet à une activité de réaliser une opération P sur le sémaphore. Si le sémaphore est déjà utilisé, l'activité est bloquée jusqu'à la réalisation de l'opération V (par une autre activité) qui libèrera le sémaphore.

Opération P non bloquante :

```
pthread_mutex_trylock (pthread_mutex_t *pmutex);
```

renvoie 1 si le sémaphore est libre
0 si le sémaphore est occupé par une autre activité
-1 en cas d'erreur.

Opération V :
Un appel à la fonction

```
pthread_mutex_unlock(pthread_mutex_t *pmutex);
```

réalise la libération du sémaphore désigné.

16.1.5 Les conditions (évènements)

Les conditions permettent de bloquer une activité sur une attente d'évènement. Pour cela l'activité doit posséder un sémaphore, l'activité peut alors libérer le sémaphore sur l'évènement, c'est-à-dire : elle libère le sémaphore, se bloque en attente de l'évènement, à la réception de l'évènement elle reprend le sémaphore.

Initialisation d'une variable de type `pthread_cond_t`

```
int pthread_cond_init (pthread_cond_t *p_cond, pthread_condattr_t attr);
```

L'attente sur une condition

```
int pthread_cond_wait (pthread_cond_t *p_cond, pthread_mutex_t *p_mutex);
```

Trois étapes

1. libération sur sémaphore *p_mutex
2. activité mise en sommeil sur l'évènement
3. réception de l'évènement, récupération du sémaphore

La condition est indépendante de l'événement et n'est pas nécessairement valide à la réception (cf. exemple).

Exemple, le programme suivant :

```
pthread_mutex_t m;
pthread_cond_t  cond;
int             condition = 0;

void *ecoute(void *beurk)
{
    pthread_mutex_lock(m);
    sleep(5);
    while (!condition)
        pthread_cond_wait(cond, m);
```

```
    pthread_mutex_unlock(m);

    pthread_mutex_lock(print);
    printf(" Condition realisee\n");
    pthread_mutex_unlock(print);
}

main()
{
    pthread_t lathread;

    pthread_create(lathread, pthread_attr_default, ecoute, NULL);
    sleep(1);
    pthread_mutex_lock(m);
    condition = 1;
    pthread_mutex_unlock(m);
    pthread_cond_signal(cond);
}
```

Un autre exemple d'utilisation de condition avec deux threads qui utilisent deux tampons pour réaliser la commande cp, avec une activité responsable de la lecture et l'autre de l'écriture. Les conditions permettent de synchroniser les deux threads. Ici nous utilisons la syntaxe NeXT/MACH.

```
#include <sdtio.h>
#include <fcntl.h>
#import  <mach/cthreads.h>

enum { BUFFER_A_LIRE = 1, BUFFER_A_ECRIRE = -1 };

mutex_t     lock1;  /* variables de protection et d'exclusion */
condition_t cond1;

char buff1[BUFSIZ];
int  nb_lu1;
int  etat1 = BUFFER_A_LIRE;
int  ds, dd;        /* descripteurs source et destination */

lire()         /* activite lecture */
{
    for(;;)  { /* lecture dans le buffer 1 */
        mutex_lock(lock1);
        while (etat1 == BUFFER_A_ECRIRE)
            condition_wait(cond1, lock1);
        nb_lu1 = read(ds, buff1, BUFSIZ);
        if (nb_lu1 == 0)
        {
            etat1 = BUFFER_A_ECRIRE;
            condition_signal(cond1);
            mutex_unlock(lock1);
            break;
        }
        etat1 = BUFFER_A_ECRIRE;
        condition_signal(cond1);
        mutex_unlock(lock1);
```

```
    }
}

ecrire()
{
    for(;;)
    { /* ecriture du buffer 1 */
        mutex_lock(lock1);
        while (etat1 == BUFFER_A_LIRE)
            condition_wait(cond1, lock1);
        if (nb_lu1 == 0)
        {
            mutex_unlock(lock1);
            exit(0);
        }
        write(dd, buff1, nb_lu1);
        mutex_unlock(lock1);
        etat1 = BUFFER_A_LIRE;
        condition_signal(cond1);
    }
}

main()
{
    ds    = open(argv[1], O_RDONLY);
    dd    = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0666);
    lock1 = mutex_alloc();
    cond1 = condition_alloc();

    cthread_fork((cthread_fn_t)lire, (any_t)0);
    ecrire(); /* la thread principale realise les ecritures */
}
```

16.2 Ordonnancement des activités

16.2.1 L'ordonnancement POSIX des activités

L'ordonnancement des activités DCE basé sur POSIX est très similaire à l'ordonnancement des activités sous MACH. Deux valeurs permettent de définir le mode d'ordonnancement d'une activité :
la politique et la priorité.
Pour manipuler ces deux valeurs, il vous faut créer un objet attribut d'activité (`pthread_attr`) en appelant `pthread_attr_create()`, puis changer les valeurs par défaut avec les fonctions décrites plus loin et créer la pthread avec cet objet `pthread_attr`. Ou bien la pthread peut elle-même changer ses deux valeurs, priorité et politique.

Les fonctions sont :

```
#include <pthread.h>
pthread_attr_setsched(pthread_attr_t *attr, int politique);
```

Les différentes politiques possibles sont :

SCHED_FIFO La thread la plus prioritaire s'exécute jusqu'à ce qu'elle bloque. Si il y a plus d'une pthread de priorité maximum, la première qui obtient le cpu s'exécute jusqu'à ce qu'elle bloque.

SCHED_RR Round Robin. La thread la plus prioritaire s'exécute jusqu'à ce qu'elle bloque. Les threads de même priorité maximum sont organisées avec le principe du tourniquet, c'est-à-dire qu'il existe un quantum de temps au bout duquel le cpu est préempté pour une autre thread (voire Chapitre 6 sur les Processus).

SCHED_OTHER Comportement par défaut. Tous les threads sont dans le même touniquet, il n'y a pas de niveau de priorité, ceci permet l'absence de famine. Mais les threads avec une politique `SCHED_FIFO` ou `SCHED_RR` peuvent placer les threads `SCHED_OTHER` en situation de famine.

SCHED_FG_NP (option DCE non portable) Même politique que `SCHED_OTHER` mais l'ordonnanceur peut faire évoluer les priorités des threads pour assurer l'équité.

SCHED_BG_NP (option DCE non portable) Même politique que `SCHED_FG_NP`, mais les threads avec une politique `SCHED_FIFO` ou `SCHED_RR` peuvent placer les threads `SCHED_BG_NP` en situation de famine.

```
pthread_attr_setprio(pthread_attr_t *attr, int prio);
```

La priorité varie dans un intervalle défini par la politique :
PRI_OTHER_MIN <= prio <= PRI_OTHER_MAX
PRI_FIFO_MIN <= prio <= PRI_FIFO_MAX
PRI_RR_MIN <= prio <= PRI_RR_MAX
PRI_FG_MIN_NP <= prio <= PRI_FG_MAX_NP
PRI_BG_MIN_NP <= prio <= PRI_BG_MAX_NP

Ces deux fonctions retournent 0 en cas de succès et -1 sinon. La valeur de **errno** indiquant si l'erreur est une question de paramètres ou de permission.

Les deux fonctions que l'on peut appeler sur une pthread pour changer sa priorité ou sa politique sont :

```
pthread_setprio(pthread_t *unepthread, int prio);
pthread_setsched(pthread_t *unepthread, int politique, int prio);
```

Il est possible de connaître la priorité ou la politique d'une pthread ou d'un objet pthread_attr avec :

```
pthread_attr_getprio(pthread_attr_t *attr,int prio);
pthread_attr_getsched(pthread_attr_t *attr,int politique);
pthread_getprio(pthread_t *unepthread, int prio);
pthread_getsched(pthread_t *unepthread, int politique);
```

16.3 Les variables spécifiques à une thread

Avec un processus multi-threads, nous sommes dans une situation de partage de données. Toutes les données du processus sont à priori manipulables par toutes les threads. Or certaines données sont critiques et difficilement partageables. Premièrement ce sont les données de la bibliothèque standard. Pour les fonctions de la bibliothèque standard, on peut résoudre le problème en utilisant un sémaphore d'exclusion mutuelle `pthread_mutex_t` pour POSIX.

Mais certaines variables ne peuvent être protégées. C'est le cas de la variables `errno`, comme nous l'avons vu précédemment. Pour cette variable, la solution est d'avoir une variable par thread. Ainsi le fichier `<errno.h>` est modifié et contient :

```
extern int *_errno();
#define errno (*_errno())
```

La valeur `errno` est obtenue par une fonction qui retourne la valeur de `errno` associée à la thread qui fait l'appel à `_errno` .

16.3.1 Principe général des données spécifiques, POSIX

L'idée des données spécifique est de créer un vecteur pour chaque donnée spécifique. Ainsi pour des données spécifique statiques, chaque thread possède son propre exemplaire. Les données spécifiques sont identifiées par des clés de type `pthread_key_t`.

16.3.2 Création de clés

La création d'une clé est liée à la création d'un tableau statique (variable globale), initialisé à NULL à la création. La fonction

```
#include <pthread.h>
int pthread_keycreate (pthread_key_t *p_cle,
                       void          (*destructeur)(void *valeur));
```

permet la création du tableau, 0 succès et -1 echec. La structure pointée par `p_cle` nous permettra d'accèder aux valeurs stockées, la clé est évidemment la même pour toutes les threads. Le paramètre `destructeur` de type pointeur sur fonction prenant un pointeur sur void en paramètre et renvoyant void, donne l'adresse d'une fonction qui est exécutée à la terminaison de la thread (ce qui permet de faire le ménage). Si ce pointeur est nul, l'information n'est pas détruite à la terminaison de l'activité.

16.3.3 Lecture/écriture d'une variable spécifique

La fonction

```
#include <pthread.h>
int pthread_getspecific (pthread_key_t *p_clé, void **pvaleur);
```

permet la lecture de la valeur qui est copié à l'adresse `pvaleur` retourne 0 ou -1 selon que l'appel à réussi ou non. La fonction

```
#include <pthread.h>
int pthread_setspecific (pthread_key_t *p_clé, void *valeur);
```

permet l'écriture à l'emplacement spécifié de `valeur` retourne 0 ou -1 selon que l'appel a réussit ou non.

16.4 Les fonctions standardes utilisant des zones statiques

Certaines fonctions standardes comme `ttyname()` ou `readdir()` retourne l'adresse d'une zone statique. Plusieurs threads en concurrence peuvent donc nous amener à des situations incohérentes. La solution des sémaphores d'exclusion étant coûteuse, ces fonctions sont réécrites pour la bibliothèque de thread de façon à être réentrantes.

Attention les problèmes de réentrance peuvent avoir lieu en utilisant des appels systèmes non réentrant dans les handlers de signaux ! Ceci sans utiliser de threads !