INP ENSEEIHT

Programmation multi-activités avec les POSIX threads

Philippe Quéinnec

14 novembre 2000

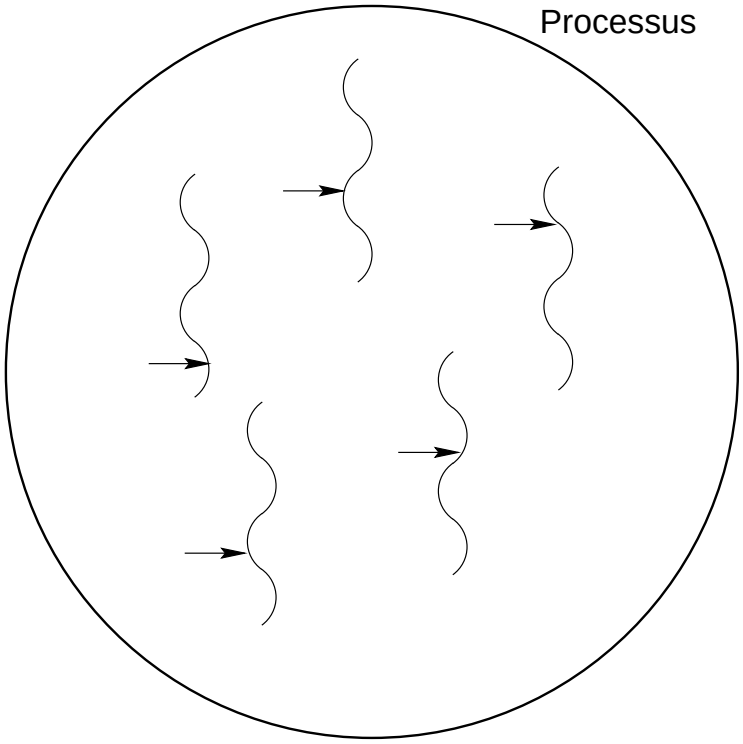
Programming with POSIX Threads, David Butenhof, Addison-Wesley 1997.

Programming with threads, Steve Kleiman, Devang Shah et Bart Smaalders, Prentice Hall 1996.

Systems programming with Modula-3, édité par Greg Nelson, chapitre "An Introduction to Programming with Threads" par Andrew Birell, Prentice Hall, 1991.

Table des matières

I. Généralités



1 espace d'adressage, plusieurs flots de contrôle.

⇒ plusieurs *activités* (ou processus légers) au sein d'un même processus UNIX.

Quelques définitions

asynchronisme : décrit le fait que des activités (ou événements) se produisent de manière indépendante.

synchronisation : apparaît quand il existe des dépendances entre les activités.

parallélisme (vrai parallélisme) : le fait que des activités s'exécutent simultanément. Possible uniquement sur un système multi-processeurs.

concurrence (ou parallélisme simulé, ou pseudo-parallélisme) : illusion que des activités s'exécutent simultanément.

1 Pour quoi faire ?

- le système d'exploitation (UNIX) est asynchrone : plusieurs processus en concurrence, synchronisés par des tubes ou des fichiers
- le monde est asynchrone, particulièrement l'utilisateur !

Les activités sont similaires à des processus mais :

- elles partagent le même espace d'adressage (codes et variables partagées)
- elles partagent les descripteurs de fichier
- elles sont moins « coûteuses » (souvent appelées « processus légers »)
- elles disposent de mécanismes évolués de synchronisation

2 Exemple : alarme sous Unix

Notre programme doit afficher un message après un délai lu au clavier.

```
int main ()
{
   int secondes;
   char msg[64];

   while (1) {
       scanf ("%d %s", &secondes, msg);
       sleep (secondes);
       printf ("Alarme %d: %s\n", secondes, msg);
   }
}
```

3 Exemple : plusieurs alarmes

On crée un nouveau processus pour chaque alarme.
Pb : comment se débarrasser des processus terminés ?

```
int main ()
{
   int secondes;
   char msg[64];

   while (1) {
       scanf ("%d %s", &secondes, msg);
       if (fork() == 0) {
           sleep (secondes);
           printf ("Alarme %d: %s\n", secondes, msg);
           exit (0);
       } else {
           /* ramasse tous les fils termines */
           while (waitpid (-1, NULL, WNOHANG) > 0)
                /*loop*/;
       }
   }
}
```

4 Exemple : plusieurs alarmes, un seul processus

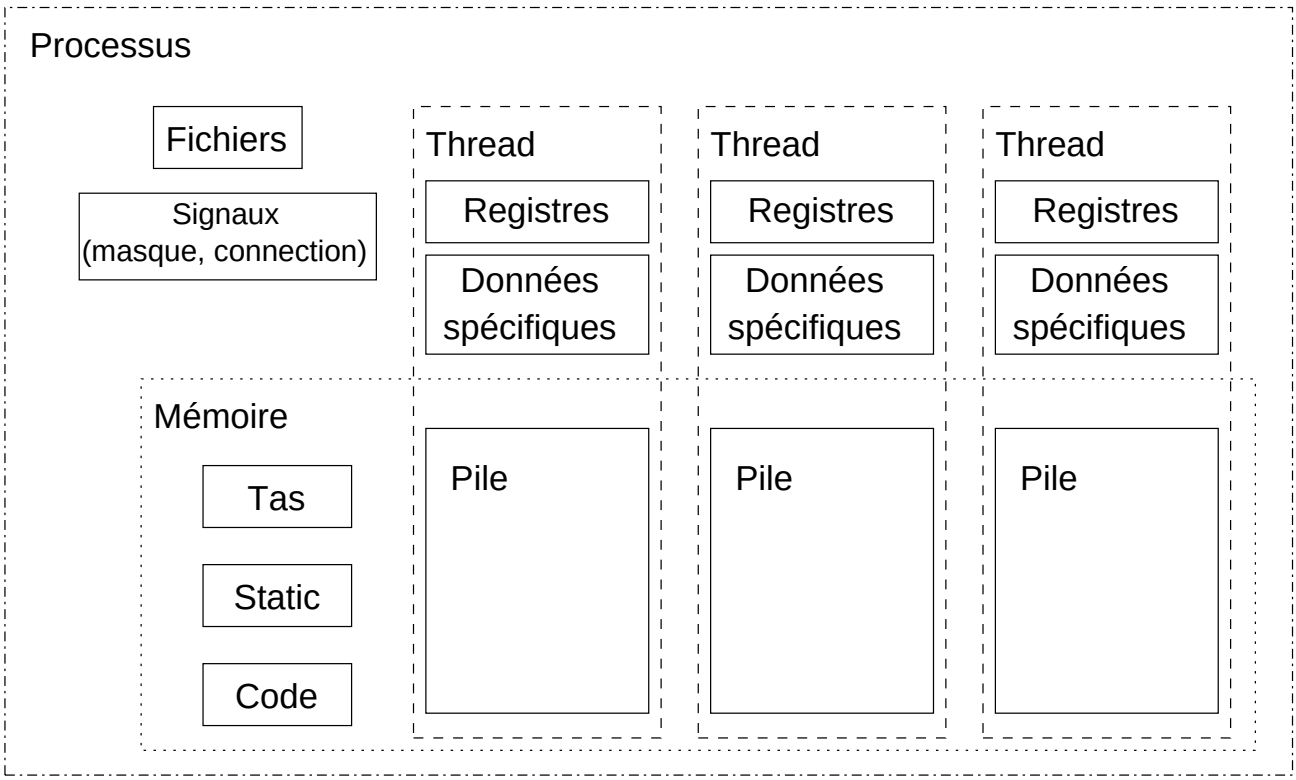
```
#include <pthread.h>

typedef struct { int sec; char msg[64]; } alarm_t;

void *alarm_thread (void *arg)
{
   alarm_t *al = (alarm_t *)arg;
   pthread_detach (pthread_self());
   sleep (al->sec);
   printf ("Alarme %d: %s\n", al->sec, al->msg);
   free (al);
   pthread_exit (NULL);
}

int main ()
{
   alarm_t *al;
   pthread_t thread;
   while (1) {
       al = malloc (sizeof (alarm_t));
       scanf ("%d %s", &(al->sec), al->msg);
       pthread_create (&thread, NULL,
                       alarm_thread, al);
   }
}
```

5 Thread et process



6 Réentrance

Si plusieurs activités s'exécutent simultanément, une procédure appelée depuis une première activité peut éventuellement être appelée par une deuxième activité en même temps.

Une procédure est *réentrante* si elle admet de telles exécutions sans dommage.

Une procédure modifiant des variables globales est généralement non réentrante. Une fonction pure est réentrante.

On place généralement les variables locales dans la *pile* de l'activité appelante ⇒ pas de conflit.

Solution : synchronisation (excl. mut., transactions. . .)

Une *bibliothèque* (ensemble de procédures et données) est *réentrante* si toutes ses procédures sont réentrantes les unes vis-à-vis des autres (et d'elles-mêmes).

Vocabulaire Posix :

- MT-safe = réentrance vis-à-vis du parallélisme
- async-safe = réentrance vis-à-vis des signaux

7 Avantages des activités

1. utilisation d'un système multi-processeurs.
2. utilisation de la concurrence naturelle d'un programme. Par exemple, une activité peut faire des calculs alors qu'une autre attend le résultat d'une E/S.
3. modèle de programmation naturel, en explicitant la synchronisation nécessaire.

8 Inconvénients

1. surcoût d'exécution (synchronisation, implantation du pseudo-parallélisme).
2. surcoût de développement : nécessité d'expliciter la synchronisation, vérifier la réentrance des bibliothèques, danger des variables partagées.
3. surcoût de mise-au-point : debuggage souvent délicat (pas de flot séquentiel à suivre) ; effet d'interférence entre des activités, interblocage...

9 Paradigmes de structuration

- travail délégué (faible priorité)
- travail anticipé (prévision des requêtes futures)
- actions événementielles ou périodiques (alarmes, surveillants. . .)
- maître-esclaves : décomposition du travail
- ouvriers (workpile) : ensemble (généralement fixe) d'activités réalisant le même travail (le même code), en extrayant des travaux parmi un ensemble de soumissions.
- tubes et filtres : chaîne d'activités

II. POSIX threads

Standard de librairie multi-activités, supporté par de nombreuses implantations plus ou moins conformantes (SUN/Solaris 2.5, Linux (avec linuxthreads), FreeBSD, Digital UNIX 4.0, AIX 4.3, HP-UX 11.0, IRIX 6.2, openVMS 7.0...)

Nom officiel : POSIX 1003.1c-1995.

Souvent associé a POSIX 1003.1b-1993 (temps-réel), POSIX 1003.1i-1995 (corrections de POSIX 1003.1b-1993), POSIX 1003.1j (extensions). Tous sont repris tels quels dans POSIX 1003.1-1996.

Repris avec quelques ajouts dans X/Open XSH5 (aka UNIX98).

Que contient la librairie :

- manipulation d'activités (création, terminaison...)
- synchronisation : verrous, variables condition.
- primitives annexes : données spécifiques à chaque activité, politique d'ordonnancement...
- ajustement des primitives standard : processus lourd, E/S, signaux, routines réentrantes.

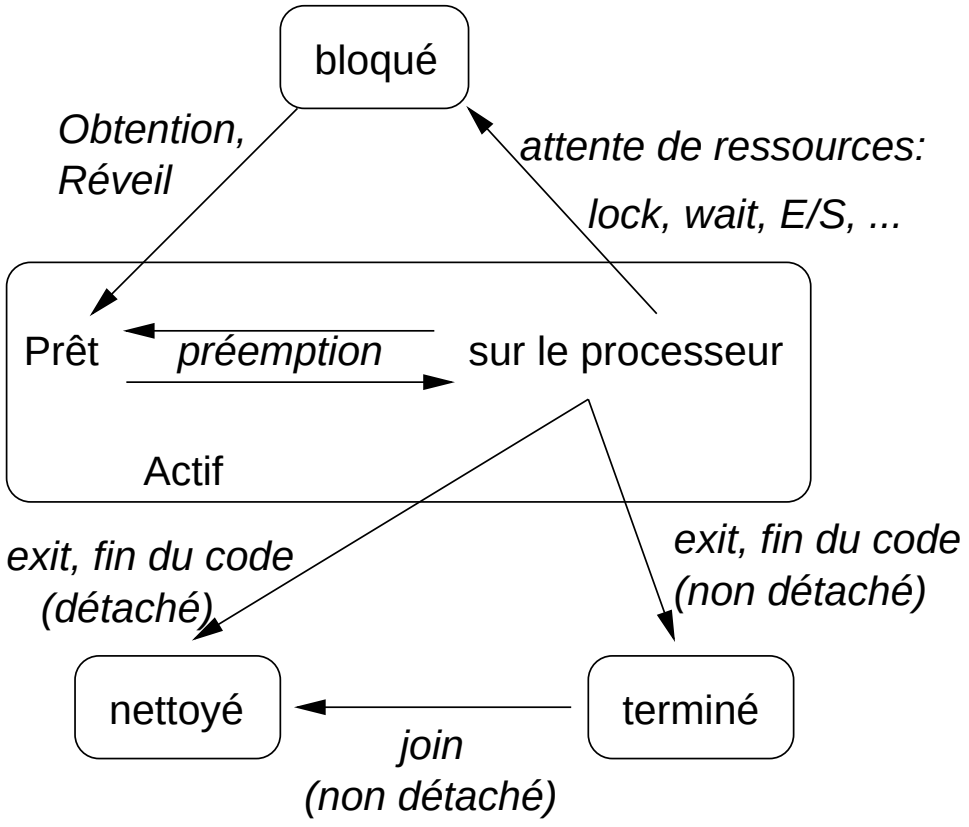
1 Compilation

- inclure `<pthread.h>` dans les sources
- compiler avec `-D_REENTRANT`
- faire l'édition de liens avec `-lpthread`

2 Erreurs

Les procédures de la librairie pthread renvoie 0 si ok, ou un code (> 0) issu de `errno.h`.

3 Le cycle de vie d'une activité



4 Manipulation des activités

a. Création

```
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void * (*start_routine)(void *),
                    void *arg);
```

Crée un nouvelle activité pour exécuter la routine indiquée, appelée avec l'argument `arg`. Les attributs sont utilisés pour définir la priorité et la politique d'ordonnancement (scheduling policy). `thread` contient l'identificateur de l'activité créée.

b. Terminaison

```
void pthread_exit (void *status);
```

Termine l'activité appelante en fournissant un code de retour. `pthread_exit(NULL)` est automatiquement exécuté en cas de terminaison du code de l'activité sans appel de `pthread_exit`.

c. Identification

```
pthread_t pthread_self (void);
int pthread_equal (pthread_t thr1, pthread_t thr2);
```

`self` renvoie l'identificateur de l'activité appelante.

`pthread_equal` renvoie vrai si les arguments désignent la même activité, faux sinon.

d. Attente de terminaison

```
int pthread_join (pthread_t thr, void **status);
```

Attend la terminaison de l'activité indiquée et récupère le code retour. L'activité ne doit pas être détachée ou avoir déjà été « jointe ».

e. Libération des ressources

```
int pthread_detach (pthread_t thread);
```

Détache l'activité `thread`.
Les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée et que :

- *ou* `join` a été effectué,
- *ou* l'activité a été détachée.

f. L'activité initiale

Au démarrage, une activité est automatiquement créée pour exécuter la procédure `main`. Elle exécute une procédure de démarrage qui contient le code :

```
{ int r = main(argc,argv); exit(r); }
```

Si la procédure `main` se termine, le process unix est ensuite terminé (par l'appel à `exit`), et non pas seulement l'activité initiale. Pour éviter que la procédure `main` ne se termine alors qu'il reste des activités :

- bloquer l'activité initiale sur l'attente de la terminaison d'une ou plusieurs autres activités (`pthread_join`) ;
- terminer explicitement l'activité initiale avec `pthread_exit`, ce qui court-circuite l'appel de `exit`.

5 Synchronisation

Principe : moniteurs de Hoare réalisés au moyen de verroux (pour assurer l'exclusion mutuelle) et de variables condition (pour attendre qu'un état, décrit par un prédicat, soit vrai).

a. Nécessité de protection

- modification concurrente
  ⟨ `x := 0x01` ⟩ ‖ ⟨ `x := 0x100` ⟩
  ⇒ `x = 0x01` ou `0x100` ou `0x101` ou `3.1415` !
- exécution concurrente
  ⟨ `a := x; x := a + 1` ⟩ ‖ ⟨ `b := x; x := b - 1` ⟩
  ⇒ `x = -1, 0` ou `1`
- cohérence mémoire
  ⟨ `x := 1; y := 2` ⟩ ‖ ⟨ `printf("%d %d",x,y);` ⟩
  ⇒ peut afficher `0 2`

b. Création/destruction d'un verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutex_attr *attr);
int pthread_mutex_destroy (pthread_mutex_t *m);
```

(on ignorera l'attribut).

c. Création/destruction d'une variable condition

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t *vc,
                       const pthread_cond_attr *attr);
int pthread_cond_destroy (pthread_cond_t *vc);
```

(on ignorera l'attribut).

d. Verrouillage/déverrouillage

```
int pthread_mutex_lock (pthread_mutex_t *m);
int pthread_mutex_trylock (pthread_mutex_t *m);
int pthread_mutex_unlock (pthread_mutex_t *m);
```

`lock` verrouille le verrou, avec blocage en attente si déjà verrouillé. Renvoie 0 si ok.

`trylock` Verrouille le verrou si possible et renvoie 0, sinon renvoie `EBUSY` si le verrou est déjà verrouillé (ou, comme toujours, `EINVAL` ou `EFAULT` en cas d'erreur).

`unlock` déverrouille. Seule l'activité qui a verrouillé `mutex` a le droit de le déverrouiller (en cas de tentative de déverrouiller un mutex verrouillé par un autre thread, le comportement est indéfini).

Si une ou plusieurs activités sont bloquées en attente du verrou, une d'entre elles obtient le verrou et est débloquée. Le choix de cette activité dépend des priorités et des politiques d'ordonnancement. Si les attributs (pour la création des activités et des mutex) ne sont pas utilisés, on considérera que l'ordre de déblocage est l'ordre de blocage (FIFO), *même si le système SOLARIS ne spécifie pas un tel ordre.*

Interblocage dû aux verrous

```
Activité A verrouille mutex M1
                    Activité B verrouille mutex M2
Activité A se bloque sur M2
                    Activité B se bloque sur M1
```

Solution : imposer un ordre (acyclique) sur les verrous, qui doivent nécessairement être demandés en respectant cet ordre.

Inversion de priorité Soit trois activités A, B et C, telles que A est de forte priorité, B de priorité moyenne, et C de faible priorité, ainsi qu'un verrou m.

A et B sont bloquées

C s'exécute et verrouille m

B se réveille et préempte C

A se réveille et préempte B

A tente de verrouiller m et se bloque

B reprend la main

Tant que B ne se bloque pas, C ne peut pas s'exécuter et libérer le verrou nécessaire à la poursuite de A. La solution (rarement implantée) consiste à modifier la priorité de C pour le rendre aussi prioritaire que A jusqu'à ce qu'il ait libéré le verrou.

e. Attente/signal

```
int pthread_cond_wait (pthread_cond_t *vc,
                       pthread_mutex_t *m);
int pthread_cond_timedwait (pthread_cond_t *vc,
                    pthread_mutex_t *m,
                    const struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *vc);
int pthread_cond_broadcast (pthread_cond_t *vc);
```

`cond_wait` : l'activité appelante doit posséder le verrou `m`. L'activité se bloque sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que `vc` soit signalée et que l'activité ait réacquis `m`.

`cond_timedwait` : comme `cond_wait` avec délai de garde. À l'expiration du délai de garde, le verrou est reobtenu et la procédure renvoie `ETIMEDOUT`.

`cond_signal` : signale la variable condition : une activité bloquée sur la variable condition est réveillée et tente de réacquérir le verrou de son appel de `cond_wait`. Elle sera effectivement débloquée quand elle le réacquerra. Pas d'ordre spécifié pour le choix de l'activité réveillée.

`cond_broadcast` : toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

f. Remarques

Contrairement à la définition des moniteurs de Hoare, l'activité signalée n'a pas priorité sur le signaleur : le signaleur ne perd pas l'accès au moniteur s'il le possédait, et le signalé reste bloqué tant qu'il n'obtient pas le verrou. C'est pourquoi il est nécessaire d'utiliser une boucle d'attente réévaluant la condition d'exécution. En effet, cette condition peut être invalidée entre le moment où l'activité est signalée et le moment où elle obtient effectivement le verrou, par exemple si une autre activité obtient le mutex et pénètre dans le moniteur avant l'activité signalée.

Pour des raisons d'efficacité, il est courant de faire l'appel à `cond_signal` hors de la zone `lock-unlock`, de sorte que l'activité signalée puisse acquérir plus facilement le verrou. Attention cependant à garantir l'atomicité des opérations du moniteur !

g. Exemple : producteur/consommateur

```
#include <pthread.h>
#include <string.h>
#include "prodcon.h"
static pthread_cond_t est_libre, est_plein;
static pthread_mutex_t protect;
static char *buffer;

/* Initialise le producteur/consommateur. */
void init_prodco (void)
{
    pthread_mutex_init (&protect, NULL);
    pthread_cond_init (&est_libre, NULL);
    pthread_cond_init (&est_plein, NULL);
    buffer = NULL;
}
```

```
/* Depose le message msg (qui est duplique).
 * Bloque tant que le tampon est plein. */
void deposer (char *msg)
{
    pthread_mutex_lock (&protect);
    while (buffer != NULL)
        pthread_cond_wait (&est_libre, &protect);
    /* buffer = NULL */
    buffer = strdup (msg); /* duplication de msg */
    pthread_cond_signal (&est_plein);
    pthread_mutex_unlock (&protect);
}

/* Renvoie le message en tete du tampon.
 * Bloque tant que le tampon est vide. */
char *retirer (void)
{
    char *result;
    pthread_mutex_lock (&protect);
    while (buffer == NULL)
        pthread_cond_wait (&est_plein, &protect);
    /* buffer != NULL */
    result = buffer;
    buffer = NULL;
    pthread_cond_signal (&est_libre);
    pthread_mutex_unlock (&protect);
    return result;
}
```

h. Règles de visibilité mémoire entre activités

- toute valeur mémoire visible par une activité avant un appel à `pthread_create` est aussi visible par la nouvelle activité à son démarrage ;
- toute valeur mémoire visible par une activité lorsqu'elle libère un mutex (explicitement ou par blocage sur une variable condition) sera aussi visible par toute activité qui verrouillera ce même mutex ;
- toute valeur mémoire visible par une activité quand elle signale (`cond_signal` ou `cond_broadcast`) est aussi visible par la ou les activités réveillées par ce signal ;
- toute valeur mémoire visible par une activité quand elle se termine (`pthread_exit` ou fin de la procédure de départ ou annulation) est aussi visible par l'activité qui la « joint » (`pthread_join`).

6 Primitives annexes

a. Initialisation

```
pthread_once_t once_controle = PTHREAD_ONCE_INIT;
int pthread_once (pthread_once_t *ctl,
                  void (*init_routine)(void));
```

Permet d'assurer que la procédure `init_routine` est appelée une et une seule fois.

Rq : peut être réalisée avec les primitives précédemment vues :

```
static pthread_mutex_t verrou
        = PTHREAD_MUTEX_INITIALIZER;
static int init_done = 0;
pthread_mutex_lock (&verrou);
if (! init_done) {
   init_done = 1;
   init_routine();
}
pthread_mutex_unlock (&verrou);
```

b. Annulation (cancellation)

```
int pthread_cancel (pthread_t qui);
int pthread_setcancelstate (int state,
                            int *oldstate);
int pthread_setcanceltype (int type,
                           int *oldtype);
void pthread_testcancel (void);
void pthread_cleanup_push (void (*routine)(void*),
                           void *arg);
void pthread_cleanup_pop (int execute);
```

`pthread_cancel` permet de « tuer » une activité.
Rarement utile : l'annulation est asynchrone, préférer une synchronisation avec test de terminaison.

L'état d'une activité peut être (positionné avec `pthread_setcancelstate`) :

- `PTHREAD_CANCEL_DISABLE` : l'annulation est mise en attente jusqu'à ce qu'elle soit autorisée.
- `PTHREAD_CANCEL_ENABLE` : l'annulation est prise en compte selon le type.

Le type d'annulation peut être (positionné avec `pthread_setcanceltype`) :

- `PTHREAD_CANCEL_DEFERRED` : l'annulation a lieu au prochain *point d'annulation.*
- `PTHREAD_CANCEL_ASYNCHRONOUS` : l'annulation peut avoir lieu à tout moment. *Extrêmement dangereux : impossible de connaître l'état des variables partagées.*

Lorsque l'annulation est déclenchée, le système positionne l'état à `CANCEL_DISABLE` et exécute les routines de nettoyage en ordre inverse de leur empilement.

Routines de nettoyage :

- `pthread_cleanup_push` empile une routine ;
- `pthread_cleanup_pop` dépile une routine et l'exécute si le paramètre `execute` est vrai.
- *Attention :* les appels push/pop doivent former des blocs syntaxiquement corrects (assimiler push à `{` et pop à `}`)

Point d'annulation : certaines fonctions sont des points d'annulation :

- `pthread_testcancel`
- les procédures pthread bloquantes « lourdes » : `pthread_cond_wait`, `pthread_cond_timedwait` (*pas* `pthread_mutex_lock`)
- certaines primitives bloquantes : `wait`, `sigwait`, `sleep`, `open`, `read`, `write`...
- certaines primitives *peuvent* être des points d'annulation : la librairie `stdio` (`gets`, `printf`...), manipulation de répertoire (`opendir`...)...

c. Données spécifiques (*thread-specific data*)

```
pthread_key_t clef;
int pthread_key_create (pthread_key_t *clef,
                        void (*destructeur)(void *));
int pthread_key_delete (pthread_key_t clef);

int pthread_setspecific (pthread_key_t clef,
                         const void *value);
void *pthread_getspecific (pthread_key_t clef);
```

Une *clef* est créée avec `pthread_key_create` et peut être détruite avec `pthread_key_delete` (rarement utile).

Données spécifiques : pour une clef donnée (partagée), chaque activité possède *sa propre valeur* associée à cette clef (ou `NULL` en absence de valeur positionnée).

Lors de la terminaison d'une activité, pour chaque clef ayant une valeur associée dans cette activité, le destructeur positionné lors de la création de la clef est exécuté, avec la valeur associée en paramètre.

Exemple : associer un *nom* à chaque activité.

```
#include <pthread.h>
#include <stdlib.h>
#include "t_ident.h"

static int numthread = 0;
static pthread_mutex_t excl_tid;
static pthread_key_t key_tid;

static pthread_once_t once_init_tid
                    = PTHREAD_ONCE_INIT;

static void free_tid (void *tid)
{
    if (tid != NULL)
      free (tid);
}

/* Initialise la clef pour obtenir le nom
 * et le mutex. */
static void init_tid (void)
{
    numthread = 0;
    pthread_key_create (&key_tid, free_tid);
    pthread_mutex_init (&excl_tid, NULL);
}
```

```
/* Get thread id.
 * If the thread has no id, set it now. */
int gettid (void)
{
    int *pi;
    pthread_once (&once_init_tid, init_tid);
    pi = (int *) pthread_getspecific (key_tid);
    if (pi == NULL) {
        pi = malloc (sizeof (int));
        pthread_mutex_lock (&excl_tid);
        *pi = numthread;
        numthread++;
        pthread_mutex_unlock (&excl_tid);
        pthread_setspecific (key_tid, pi);
    }
    return *pi;
}
```

7 Activités et signaux

Les activités sont des entités asynchrones synchronisées lorsque c'est nécessaire. Les signaux sont des événements asynchrones, qui cohabitent difficilement avec les activités.

- comment envoyer un signal à une activité précise?
- quelle action faut-il exécuter?
- dans quel contexte (quelle activité) faut-il délivrer un signal?
- une activité peut-elle masquer un signal?
- comment traiter de manière synchrone la délivrance d'un signal?

Deux types de signaux :

synchrones (ou déroutements): signaux directement causés par une action de l'activité : `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGILL`.

asynchrones : signaux issus de l'extérieur (`SIGCHLD`, `SIGALRM`, appel à `kill`...)

a. Envoi de signal

```
int pthread_kill (pthread_t thread, int sig);
```

Ne peut se faire qu'au sein du même processus UNIX.

b. Action associée à un signal

Pas de changement : on utilise `sigaction` (ou `signal`) comme précédemment. La liaison est faite au niveau du *processus UNIX* : il ne peut y avoir qu'un seul gestionnaire pour un ensemble d'activités exécutées au sein d'un même processus UNIX. L'action par défaut est identique au cas traditionnel (`SIGKILL` tue le processus donc toutes les activités, `SIGTSTP` suspend le processus donc toutes les activités...).

c. Qui reçoit le signal ?

Si le signal est synchrone ou émis vers une activité précise (`pthread_kill`), l'action est exécutée au sein de l'activité qui a engendré le signal. Si le signal est masqué dans cet activité, il reste en attente jusqu'au démasquage.

Si le signal est asynchrone, une activité quelconque où le signal n'est pas masqué reçoit le signal.

d. Masquage des signaux

```
int pthread_sigmask (int how,  /* BLOCK,UNBLOCK,SETMASK */
                     const sigset_t *set,
                     sigset_t *oldset);
```

Fonctionnement identique à `sigprocmask`, mais masque les signaux de l'ensemble `set` dans l'activité appelante seulement.

Le masque des signaux est hérité lors de la création d'une nouvelle activité.

e. Attente synchrone

```
int sigwait (const sigset_t *set, int *sig);
```

Bloque l'activité appelante jusqu'à réception d'un signal parmi l'ensemble `set`. Le signal reçu est placé dans `*sig`.

Les signaux de l'ensemble `set` doivent être masqués dans *toutes* les activités, *y compris l'activité qui appelle* `sigwait`.

Conseil : pour manipuler les signaux asynchrones, utiliser exclusivement `sigwait`, et non les gestionnaires traditionnels.

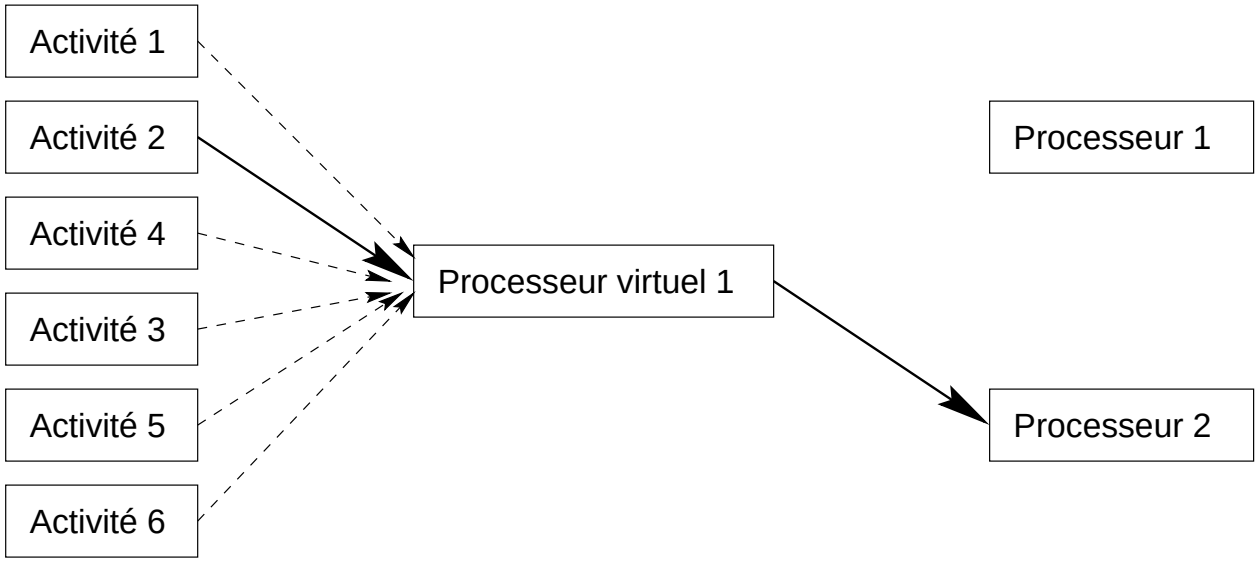
8 Activités et processeurs virtuels

Entre le processeur physique et les activités, il existe généralement une entité interne au noyau, appelé *kernel process* ou *processeur virtuel*.

Cette entité est *généralement* l'unité de blocage : un appel système bloquant (`read`...) bloque le processeur virtuel qui l'exécutait.

1. Many-to-one : 1 seul processeur virtuel par processus
2. Many-to-many : 1 processeur virtuel par activité
3. Many-to-few : plusieurs processeurs virtuels

a. Many-to-one



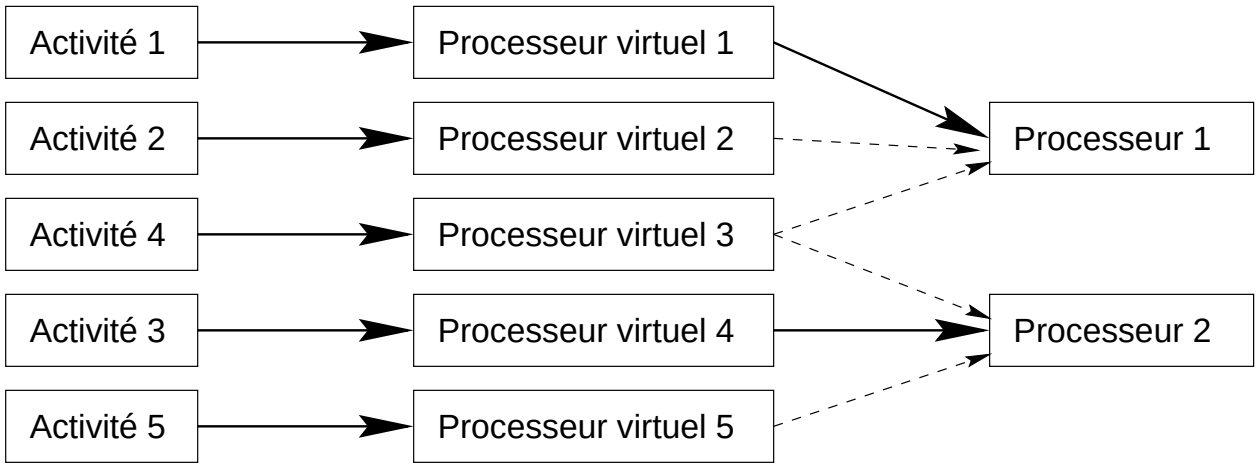
\+ commutation entre activités efficace

\+ implantation simple et portable

\− pas de bénéfice si plusieurs processeurs

\− blocage du processus (donc de toutes les activités) en cas d'appel système bloquant, ou implantation complexe

b. Many-to-many



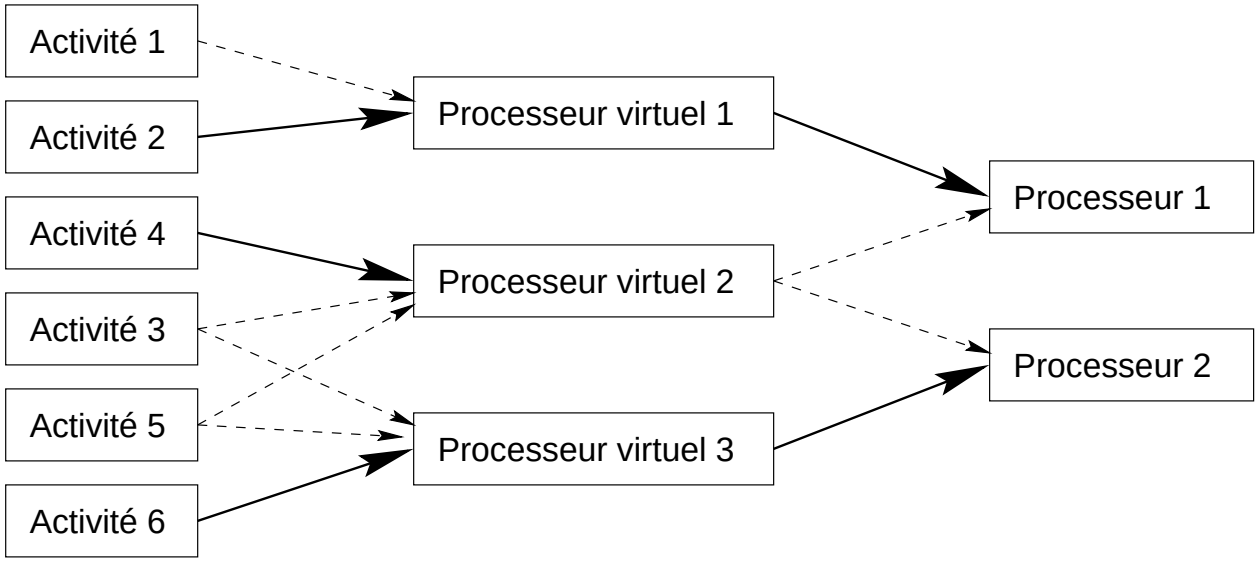
\+ vrai parallélisme si plusieurs processeurs physiques

\+ pas de blocage des autres activités en cas d'appel bloquant

– commutation moins efficace (dans le noyau)

– ressources consommées élevées

c. Many-to-few



+ vrai parallélisme si plusieurs processeurs physiques
+ meilleur temps de commutation
+ meilleur rapport ressources/nombre d'activités
+ pas de blocage des autres activités en cas d'appel bloquant
– complexe, particulièrement si création automatique de nouveaux processeurs virtuels
– faible contrôle des entités noyau
– implantations courantes souvent imparfaites

d. Solaris

Solaris 2.6 fournit une implantation de type « Many-to-few », avec quelques singularités :

- processeur virtuel = LWP (Light-Weight Process)
- Chaque processus contient des LWPs liés (bound) et un pool de LWPs non dédiés
- Pas de préemption au sein d'un LWP, mais les LWPs s'exécutent en concurrence avec préemption du processeur physique
- Un appel système bloquant bloque le LWP responsable. Certains appels (p.e. `read`, pas `sleep`) créent dynamiquement des LWPs non dédiés.
- Par défaut, les activités sont *unbound*, c'est-à-dire non liées à un LWP particulier et s'exécutent dans n'importe quel LWP non dédié.
  On peut forcer une activité à posséder son propre LWP (LWP lié) :

  ```
  pthread_attr_t attr;
  pthread_attr_init (&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  pthread_create(&thr, &attr, code, arg);
  ```

- On peut changer le nombre de LWPs non dédiés :

  ```
  thr_setconcurrency(4);
  ```

III. Autres synchronisations

1 Les barrières

Une barrière bloque un ensemble d'activités jusqu'à ce que toutes aient atteint la barrière.

```
int barrier_init (barrier_t *barrier,
                  barrier_attr_t *attr, int nombre);
int barrier_wait (barrier_t *b);
```

2 Verrous lecteur/rédacteur

Permet de réaliser une synchronisation de type lecteur/rédacteur

```
int rwlock_init (rwlock_t *lock, rwlock_attr_t *attr);
int rwlock_rlock (rwlock_t *lock);
int rwlock_wlock (rwlock_t *lock);
int rwlock_unlock (rwlock_t *lock);
```

3 Sémaphores

```
int sem_init (sem_t *sem, int pshared, int value);
int sem_post (sem_t *sem);
int sem_wait (sem_t *sem);
int sem_trywait (sem_t *sem);
```