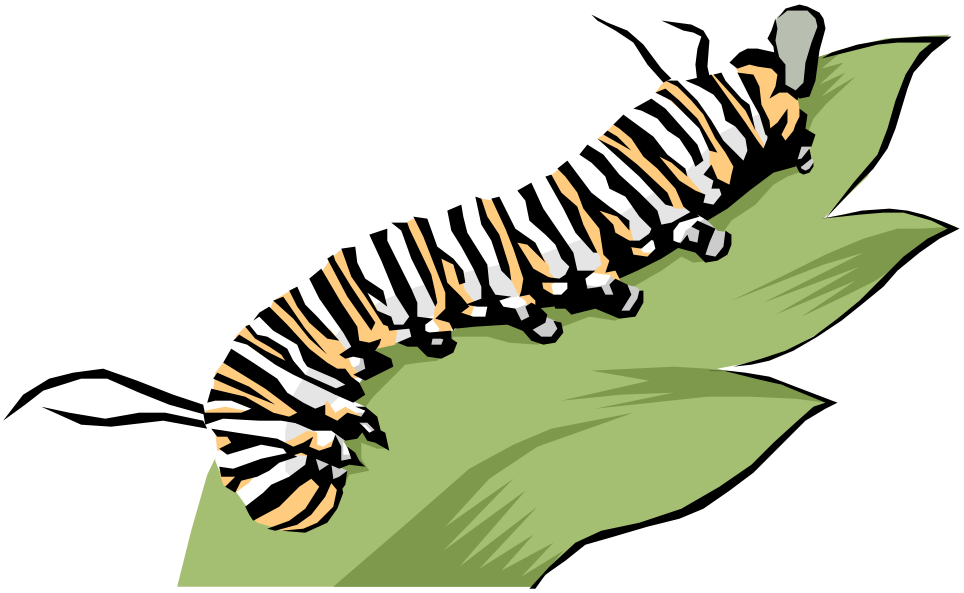
Les processus légers (Threads)



Patrick DEIBER

Probatoire CNAM Versailles

Patrick.Deiber@Wanadoo.fr

Février 1999

Table des matières

Liste des figures

1. Introduction

1.1. Présentation

Les processus légers[1] ont été des sujets de recherche pendant plusieurs années. Au travers de ces recherches, plusieurs bibliothèques (ou librairies) de processus légers furent expérimentées.

Les systèmes d'exploitation de type Unix, aussi bien que Windows NT et même OS/2 fournissent des librairies de processus légers.

Les processus légers servent également de support d'exécution pour certains compilateurs. Le langage Java fournit en natif les fonctions de processus légers.

Les processus légers sont souvent utilisés en multiprogrammation. Le *multithread* a été introduit dans les systèmes afin de permettre la mise en œuvre du parallélisme à l'intérieur des processus lourds. La multiprogrammation, à l'aide de processus légers, fournit un parallélisme plus fin et permet de détailler l'exécution d'un processus lourd.

La multiprogrammation avec les processus lourds, ne contenant qu'un seul processus léger, nécessite des ressources plus importantes et des mécanismes coûteux en temps d'exécution.

En 1995, un standard d'interface pour l'utilisation des processus légers a vu le jour, ce standard est connu sous le nom de *pthread* (ou *POSIX thread*).

Le présent document est une étude des processus légers. Au travers de cette étude détaillée, la définition et les concepts seront abordés pour ensuite aboutir aux différentes utilisations des processus légers. Des exemples simples illustreront leurs utilisations et clôtureront cette étude.

L'objectif de ce document est de présenter ce qu'apporte l'utilisation des processus légers.

1.2. Règles typographiques

Tout au long de ce document, les mots à mettre en valeur (définitions, points essentiels…) sont imprimés en **gras**, les termes étrangers en *italique* et les extraits de code façon `machine à écrire`.

1.3. Marques

Les différentes marques citées dans ce document sont déposées par leur propriétaire respectif.

[1] "processus léger", "fil d'exécution", "fil", "flot d'exécution", "flot" et "*thread*" sont synonymes.

2. Historique

Dans les années 1960, les premières réflexions sur la programmation concurrente dans un environnement monoprocesseur ont donné lieu à la notion de processus léger.

La notion de *thread*, en tant que flot de contrôle séquentiel date d'avant 1965 avec *Berkeley Timesharing System.*

En 1966, Dennis et van Horn décrivaient un système d'exploitation dans lequel un ensemble de processus pouvait se partager la mémoire et s'exécutait dans un contexte appelé "sphère d'influence".

Aux alentours des années 1970, Max Smith créa un prototype d'une implémentation des *threads* sur *Multics* en utilisant plusieurs piles (*stacks*) sur un processus lourd (*heavyweight process*) afin de supporter une compilation en tâche de fond (*background*).

Au début des années 70 avec l'arrivée d'UNIX, la notion de processus devient un fil de contrôle avec un espace d'adressage virtuel (la notion de processus était dérivée depuis les processus *Multics*).

C'est dans cette même période, qu'apparaissent les concepts de synchronisation des processus ainsi que les notions de moniteurs.

Les systèmes temps réel ont toujours mis en œuvre des systèmes de processus dont le changement de contexte étaient les plus légers possibles .

Les processus au sens UNIX sont des processus lourds, car ils ne peuvent pas partager la mémoire simplement (chaque processus possède son propre espace d'adressage, le partage ne peut s'effectuer par des mécanismes complexes). Ils communiquent entre eux par le biais de tubes communicants (*pipes*), de signaux… Le partage de la mémoire est arrivé bien plus tard par l'introduction de la notion de processus léger et des bibliothèques de communication (*Inter-Processes Communication*).

Vers la fin des années 1970, début des années 1980, les premiers systèmes micro-noyaux apparaissent avec les systèmes Toth (précurseur des systèmes V-kernel et QNX), Amoeba, Mach, Chorus…

La notion de processus léger a pris un essor avec le développement de systèmes parallèles.

Aujourd'hui, les processus légers sont implantés dans beaucoup de systèmes d'exploitation comme Unix, Linux, Solaris, Microsoft Windows NT (*WIN32 threads*)… et servent de support pour certains langages tels que Java et Ada.

Il existe une norme pour l'interface de programmation des processus légers : il s'agit du standard POSIX 1003.1c-1995 défini par l'IEEE PASC *(Institute of Electrical and Electronics Engineers Portable Application Standars Commitee*) et approuvé en juin 1995.

3. Caractéristiques d'un processus léger

Un processus léger (*lightweight processes*) est un **flot d'exécution** interne à une entité lui servant de contenant : le processus lourd.

Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** : l'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).

Un processus lourd contenant plusieurs fils d'exécution est dit **multiprogrammé** : l'exécution du code est réalisée par les différents fils d'exécution.

Le schéma ci-dessous présente les différents types de processus lourds.

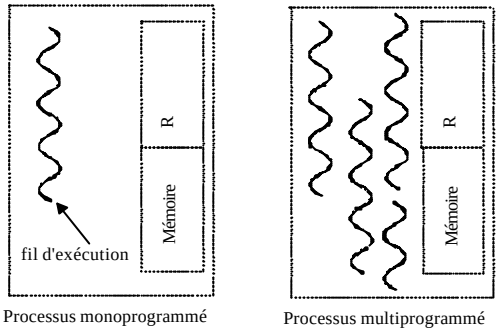


Figure 1 - *Processus lourds monoprogrammés et multiprogrammés.*

Chaque processus lourd implique la gestion d'un nouvel espace d'adressage virtuel et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).

Le principal avantage de ces processus est la protection mémoire entre les processus du même système.

Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

- leur création nécessite des appels système coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

Un processus lourd peut contenir plusieurs centaines (voire même plusieurs milliers) de processus légers constituant autant de flots d'exécution parallèles indépendants.

Le qualificatif "léger" est donné aux processus légers car, comparativement aux processus lourds, ils ne consomment que très peu de ressources (la taille d'un processus léger se compte en Kilo-octets, celle d'un processus lourd en Mega-octets). Chaque nouveau processus léger ne nécessite qu'une nouvelle pile et un nouveau jeu de registres. Les autres ressources sont partagées entre tous les processus légers s'exécutant dans le même processus lourd.

Les ressources partagées peuvent être :

- le code,
- les variables,
- le tas[2] (*heap*),
- la table des fichiers ouverts,
- la table des traitements des signaux.

Un processus léger est, par conséquent, une entité beaucoup plus efficace à gérer qu'un processus lourd comme le montrent les figures (2) et (3). La première figure présente deux processus légers au sein d'un processus lourd, la deuxième figure montre deux processus lourds issus d'une duplication (*fork*).

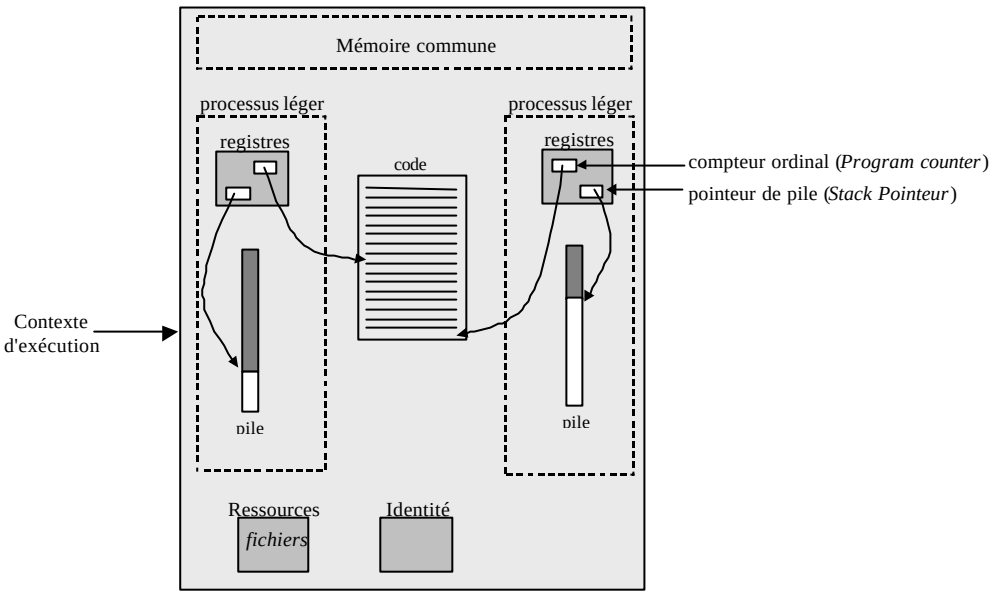


Figure 2 - *Processus lourd contenant deux fils de contrôle (processus légers)*

L'absence de protection des ressources partagées peut paraître comme un inconvénient, mais cela n'est pas vraiment gênant car les processus légers ont le même propriétaire au sein d'un processus lourd.

La création d'un processus léger nécessite peu d'opérations : allocation d'un descripteur et d'une pile, initialisation du contexte d'exécution (affectation des registres) et insertion dans la file locale des processus prêts. Pour réaliser une commutation de processus léger (changement de contexte), seuls les registres du processeur doivent être repositionnés.

La création et le changement de contexte d'un processus léger sont 10 à 100 fois meilleurs que dans le cas d'un processus lourd [4].

Ces performances s'expliquent essentiellement par la taille des ressources nécessaires au fonctionnement des processus légers, et au fait que l'ensemble des ressources de la gestion des processus légers se trouve dans l'espace d'adressage utilisateur. Ces ressources sont accessibles directement par le programme et ne nécessitent pas l'intervention du noyau.

[2] mémoire utilisateur

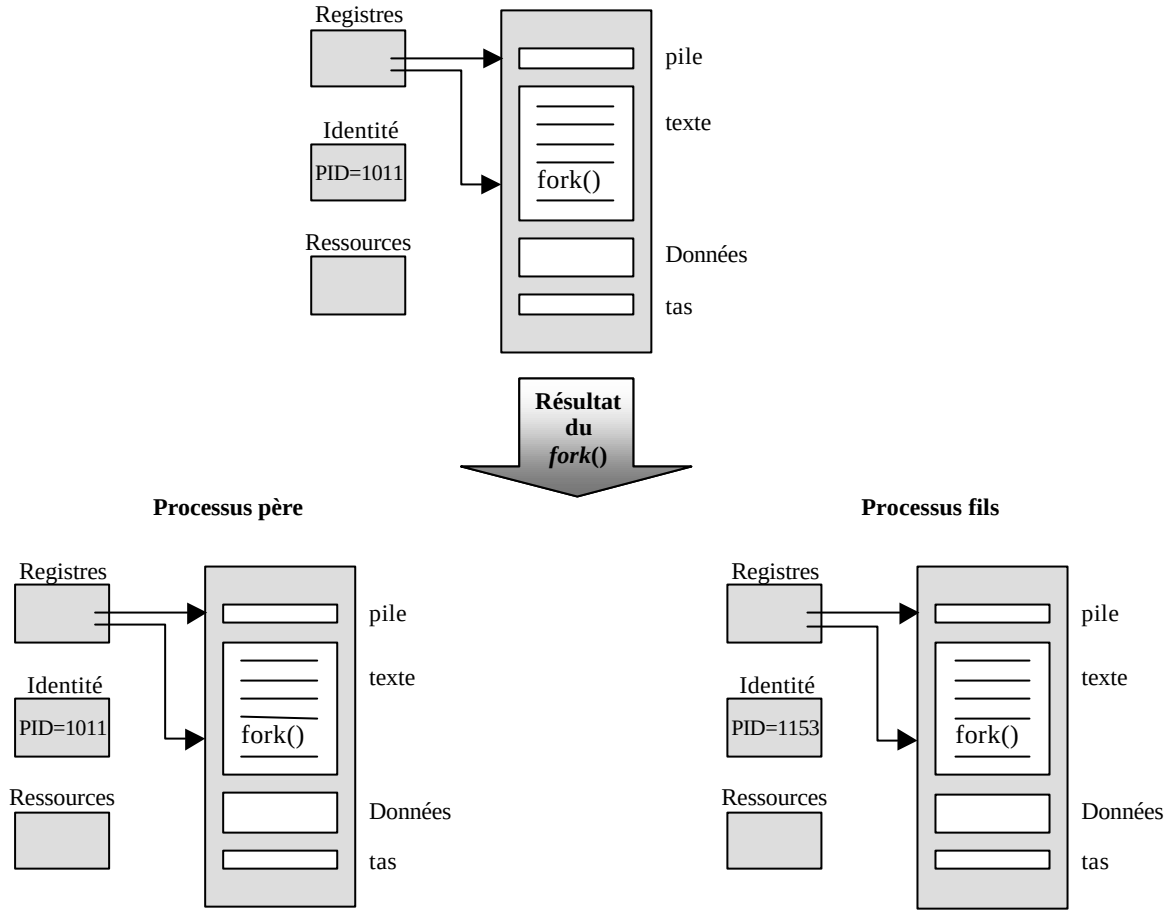


Figure 3 - *Duplication d'un processus lourd par un fork()*

En résumé, un processus léger est caractérisé par :

- Un numéro d'identification (*thread ID*), unique et affecté à la création du processus léger.
- Des registres : pointeur de pile, pointeur d'instruction (compteur ordinal)…
- Un masque de signaux permettant de spécifier quels sont les signaux à intercepter.
- Une priorité utilisée au moment de déterminer quel processus léger peut s'exécuter.
- Des données privées dont l'accès ne peut être réalisé qu'à l'aide d'une clé.

4. Raisons d'utiliser les processus légers

Les processus légers apportent de nombreux avantages à la mise en œuvre d'applications.

Les processus légers ont été surtout conçus pour faciliter la programmation parallèle. Dans ce type de programmation, on distingue deux types de parallélisme (figure 4) :

- parallélisme réel pour les systèmes multiprocesseurs où plusieurs processus sont exécutés en même temps,
- parallélisme virtuel (ou concurrence) sur les systèmes monoprocesseurs multitâches où plusieurs processus s'exécutent concurremment.

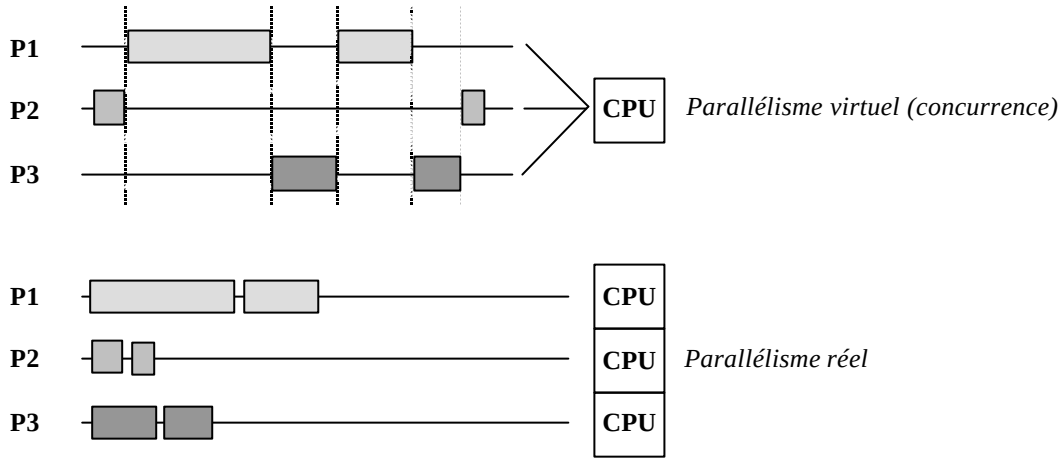


Figure 4 - *Concurrence et Parallélisme*

L'exécution des processus légers en parallèle ou en concurrence au sein d'un processus lourd est appelé le *multithreading*.

Les principaux aspects intéressants dans l'utilisation des processus légers sont :

Parallélisme et Multiprocesseur	Les machines multiprocesseurs fournissent plusieurs points d'exécution. Les processus légers exploitent pleinement le parallélisme induit par le matériel. Plusieurs processus légers s'exécutent simultanément sur plusieurs processeurs. Sur une machine monoprocesseur, les processus légers s'exécutent à tour de rôle sur un seul processeur. ☞ un bon exemple d'utilisation des processus légers sur un ordinateur multiprocesseur est la multiplication de matrices où chaque processus léger calcule chaque élément de la matrice résultante avec le vecteur multiplication correspondant (voir exemple §12.3).
Débit	Lorsqu'un processus monoprogrammé effectue un appel bloquant au système d'exploitation, le processus est bloqué tant que le service n'est pas terminé. Dans un processus multiprogrammé, seul le fil d'exécution réalisant l'appel est bloqué, les autres fils continuent leur exécution. ☞ gestion des périphériques (accès réseaux, disques, terminaux…)

Temps de réponse	Pour une application interactive, plusieurs fils d'exécution séparés traitant différentes opérations distinctes permettent d'obtenir une bonne réactivité. ☞ système de multi-fenêtrage où chaque commande est réalisée par un processus léger créé à cet effet. ☞ partage de ressources (serveurs des fichiers)
Communication	Une application utilisant plusieurs processus traditionnels pour réaliser ses fonctions peut être remplacée par une application composée de plusieurs fils d'exécution réalisant les mêmes fonctionnalités. Pour l'ancien programme, la communication entre processus traditionnel est réalisée par le biais des IPC (*Inter-Processes Communications*) ou d'autres mécanismes (*pipes* ou *sockets*). La communication entre processus légers est assurée par la mémoire partagée du processus traditionnel. Le principe des IPC peut être utilisé en plus de la mémoire partagée. ☞ application clients/serveur
Ressources systèmes	Un programme composé de plusieurs processus traditionnels consomme plus de ressources, car chaque processus est représenté par une structure incluant son espace d'adressage virtuel et son état au niveau du noyau. La création de processus traditionnels est coûteuse en temps. De plus, la séparation du programme en processus traditionnel nécessite la mise en place de mécanismes complexes de communication et de synchronisation. L'utilisation des processus légers facilite la mise en œuvre du programme et augmente sa rapidité d'exécution. Une application peut créer plusieurs centaines voire des milliers de processus légers, avec un impact mineur sur les ressources du système. Les processus légers n'utilisent qu'une partie des ressources systèmes utilisées par un processus traditionnel (voir exemple §12.2).
Facilité de programmation	Les programmes dont la structure implique une certaine concurrence sont adaptés pour les processus légers. L'utilisation des processus légers facilite l'écriture d'un programme complexe où chaque type de traitement est réalisé par un fil d'exécution spécifique. Ainsi, chaque fil d'exécution a une tâche simple à réaliser. L'évolution des programmes à base de processus légers est plus simple qu'un programme traditionnel.

L'utilisation des processus légers présente beaucoup d'avantages, mais elle nécessite la mise en place de mécanismes de protection des accès aux données et d'ordonnancement qui seront détaillés un peu plus loin (paragraphes §5.2 et §7).

5. Fonctionnalités des processus légers

5.1. Gestion des fils de contrôle

Un processus léger est créé par un appel à la fonction de création de la bibliothèque de processus légers[3]. Lors du premier lancement d'un processus, un fil d'exécution existe déjà et correspond au fil d'exécution d'un processus monoprogrammé. Ce premier fil d'exécution peut en créer d'autres, qui à leur tour pourront en créer d'autres et ainsi de suite. Le processus léger créé devient une entité indépendante avec son code et sa pile. Il peut effectuer des traitements et manipuler des données indépendamment des autres processus légers.

La taille de la pile du premier fil d'exécution peut être modifiée, contrairement aux autres, lors de son exécution (voir §5.3).

La primitive de création d'un processus léger exécute une fonction passée en paramètre et renvoie au processus léger créateur, l'identificateur du processus léger nouvellement créé.

Par rapport à un appel à une fonction, la primitive de création retourne immédiatement avant que le processus léger ne soit terminé (c'est l'intérêt des processus légers).

Un processus léger devant recueillir le résultat d'un autre processus peut attendre la terminaison de ce dernier en utilisant une primitive spécifique (primitive *join*). Dans cette primitive, il précise l'identificateur du processus dont il attend le résultat. La primitive de détachement (primitive *detached*) permet à un processus léger de ne pas attendre le résultat d'un autre processus.

Le schéma suivant présente les états d'un processus léger du standard POSIX :

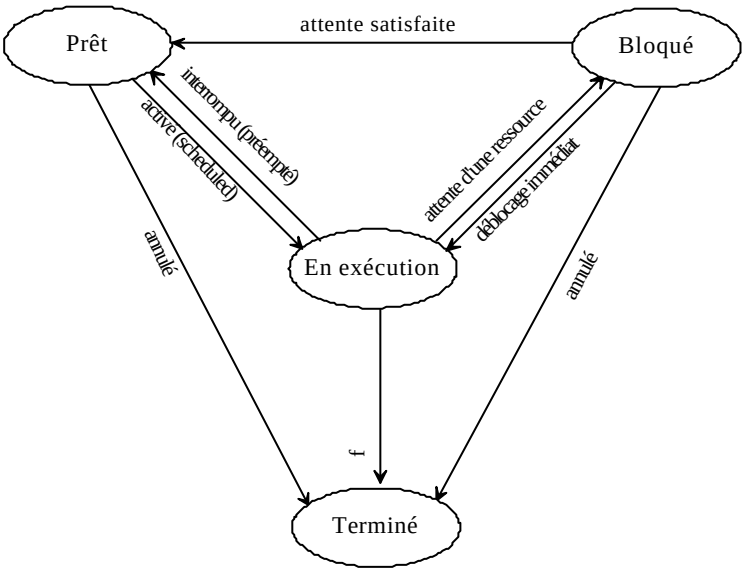


Figure 5 - *Transition d'état d'un fil d'exécution (POSIX)*

[3] appelé également "noyau de multiprogrammation".

Prêt	Le fil est prêt à être exécuté. Cas d'un fil nouvellement créé, d'un fil débloqué ou, d'un ou plusieurs fils occupant le ou les processeurs disponibles (utilisation de la primitive *yield* laissant le contrôle à un autre fil).
En exécution	Le fil est en cours d'exécution sur un processeur. Plusieurs fils peuvent être en exécution dans le cas d'une machine multiprocesseur.
Bloqué	Le fil est en attente sur une synchronisation ou sur la fin d'une opération entrées/sorties par exemple.
Terminé	Le fil a terminé son exécution ou a été annulé (*cancelled*). Les ressources du fil vont être libérées et le fil disparaîtra.

Remarques :

Dans le paragraphe §3, nous avons vu qu'il était possible de dupliquer un processus lourd en utilisant l'appel à la commande *fork()*. Que se passe-t-il si cette commande est invoquée par un des fils d'exécution du processus lourd ?

Plusieurs cas selon les bibliothèques de processus légers :

- Le processus lourd est dupliqué avec seulement le fil d'exécution qui a réalisé l'appel à la commande. Il s'agit du modèle *FORK-ONE*.
- Le processus lourd est dupliqué avec l'ensemble des fils d'exécution détenus. Il s'agit du modèle *FORK-ALL*.

5.2. Synchronisation

La cohérence des données ou des ressources partagées entre les processus légers est maintenue par des mécanismes de synchronisation. Il existe deux principaux mécanismes de synchronisation : *mutex* et variable condition.

5.2.1. Mutex

Un *mutex*[4] (verrou d'exclusion mutuelle) a deux états : verrouillé ou non verrouillé.

Quand le *mutex* n'est pas verrouillé, un fil d'exécution peut le verrouiller pour entrer dans une section critique de son code. En fin de section critique, le *mutex* est déverrouillé par le fil l'ayant verrouillé.

Si un fil essaye de verrouiller un mutex déjà verrouillé ; il est soit bloqué jusqu'à ce que le *mutex* soit déverrouillé, soit l'appel au mécanisme signale l'indisponibilité du mutex.

Trois opérations sont associées à un *mutex* : *lock* pour verrouiller le *mutex*, *unlock* pour le déverrouiller et *trylock* équivalent à *lock*, mais qui en cas d'échec ne bloque pas le processus.

Un *mutex* est souvent considéré comme un sémaphore binaire (sémaphore ne pouvant qu'avoir la valeur 0 ou 1).

Un **sémaphore** est un compteur d'accès à une ressource critique. L'utilisation d'un sémaphore s'effectue par des opérations de demande et de libération de la ressource critique. Par rapport à un *mutex*, les opérations utilisées pour un sémaphore peuvent être réalisées par un autre fil d'exécution (un fil n'est pas propriétaire d'un sémaphore quand il réalise des opérations sur celui-ci).

5.2.2. Variables conditions

Les variables conditions permettent de suspendre un fil d'exécution tant que des données partagées n'ont pas atteint un certain état. Une variable condition est souvent utilisée avec un *mutex*.

Cette combinaison permet de protéger l'accès aux données partagées (et donc à la condition variable).

Un fil bloqué en attente sur une variable condition est réveillé par un autre fil qui a terminé de modifier les données partagées.

Il existe également une attente limitée permettant de signaler automatiquement la variable condition si le réveil ne s'effectue pas pendant une durée déterminée.

Deux opérations sont disponibles : *wait*, qui bloque le processus léger tant que la condition est fausse et *signal* qui prévient les processus bloqués que la condition est vraie.

[4] ou verrou

5.2.3. Autres mécanismes

En combinant l'utilisation de ces deux primitives, les outils de synchronisation classique de type sémaphores à compteur et moniteurs peuvent être mis en oeuvre. Le document [6] contient un exemple de mise en œuvre des sémaphores en utilisant les mécanismes de base des processus légers.

Certaines bibliothèques fournissent des mécanismes de **sémaphores** [17]. L'accès à ces sémaphores ne nécessite pas l'utilisation de *mutex* comme c'est le cas avec les variables conditions (cf. §5.2.2).

Sur certains systèmes de processus léger [1][11], un mécanisme de **barrière** (*barrier*) permet de synchroniser un ensemble de processus légers à un point spécifique du programme. Les opérations disponibles pour ce mécanisme sont l'initialisation (*barrier_init*) dans laquelle le nombre de processus participants est spécifié, et la rencontre de la barrière (*barrier_hit*) correspondant au point de synchronisation. Ce mécanisme peut également être mis en œuvre en combinant les variables conditions et les *mutex.*

La figure ci-dessous illustre l'utilisation du mécanisme de la barrière dans le cas où trois processus participent à celle-ci :

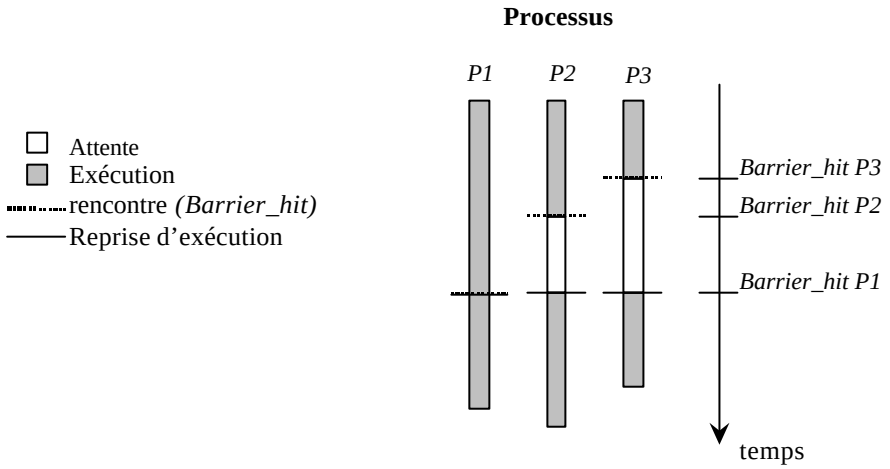


Figure 6 - *Principe de la barrière*

Le système Solaris (Sun) fournit des verrous entre lecteurs et rédacteurs (*Readers/Writer Locks*) [5][14][16]. Ce type de verrou permet des accès concurrents entre les lectures d'une ressource et des accès exclusifs entre les écritures, et entre les lectures et les écritures.

Un processus léger peut acquérir le verrou en lecture, si tous les processus légers, ayant déjà acquis le verrou, sont également en lecture. Si un processus léger veut acquérir un verrou en écriture, ce dernier est bloqué tant que les lecteurs n'ont pas libéré le verrou. Dès qu'un rédacteur est bloqué, les prochains processus légers souhaitant acquérir le verrou (en lecture et en écriture) seront bloqués.

5.2.4. Problèmes

Les mécanismes de synchronisation peuvent conduire aux problèmes suivants :

Interblocage (deadlocks) :

Le phénomène d'interblocage est le problème le plus courant.

Le cas le plus simple d'interblocage[5] (*deadlocks*) survient lorsqu'un processus léger verrouille un *mutex* et oublie de le déverrouiller. Une autre situation d'interblocage peut survenir lorsqu'un processus léger ayant acquis un verrou est prématurément terminé. Dans ce cas, le verrou n'est plus disponible.

L'exemple suivant illustre un autre cas, où deux processus légers verrouillent chacun un *mutex* dont l'autre a besoin [7] :

```
Soit M1 et M2 deux mutex.
Soit Pa et Pb deux processus légers.
Pa verrouille le mutex M1
Pb verrouille le mutex M2
Pa se bloque en tentant de verrouiller le mutex M2
Pb se bloque en tentant de verrouiller le mutex M1
```

Une solution efficace évitant le cas d'interblocage est de mettre en place un ordre d'acquisition d'un *mutex* (ordre hiérarchique). Ainsi, le verrouillage du mutex M2 de l'exemple, par un processus léger ne peut être réalisé que si le processus léger peut également verrouiller le mutex M1. Une autre solution consiste à vérifier si le *mutex* n'est pas déjà verrouillé (*trylock*).

Il existe des fonctions de nettoyage permettant de libérer les ressources et les verrous acquis par un fil d'exécution terminé (fonctions *pthread_cleanup_push* et *pthread_cleanup_pop* du standard POSIX) [10]. Ainsi, l'oubli de déverrouillage d'un verrou par un fil, ou une terminaison prématurée d'un fil possédant un verrou, sont réparés par ces mécanismes de nettoyage.

Des cas d'interblocage peuvent également survenir dans l'utilisation des variables conditions de la même manière.

```
Soit R1 et R2 deux ressources partagées.
Soit Pa et Pb deux processus légers.
Pa fait l'acquisition de la ressource R1
Pb fait l'acquisition de la ressource R2
Pa souhaite acquérir R2, et attend la variable condition de R2
Pb souhaite acquérir R1, et attend la variable condition de R1
```

Par conséquent, le développement d'applications utilisant les processus légers nécessite une attention particulière. De plus, une mauvaise utilisation des deux méthodes peut également conduire à des situations d'interblocage.

Famine (*starvation*) :

Un processus léger ne pouvant jamais accéder à un verrou se trouve dans une situation de famine.

Par exemple cette situation se produit, lorsqu'un processus léger, prêt à être exécuté, est toujours devancé par un autre processus léger plus prioritaire.

Endormissement (*dormancy*) : cas d'un processus léger suspendu qui n'est jamais réveillé.

[5] ou encore verrou mortel

5.3. Données privées et piles

Les différents fils d'exécution partagent l'espace d'adressage virtuel du processus lourd. Les fils ont accès à toutes les adresses mémoire de cet espace. Il n'est pas possible de protéger sélectivement les accès pour tel ou tel fil. Cependant, il est possible de créer des zones de données privées (*thread-specific data*) pour protéger des informations non partagées par tous les fils. L'accès à ces zones est réalisé à l'aide de clés communes. Ces clés servent à distinguer les différentes zones. Chaque zone mémoire distincte est définie par un couplet {clé; identité processus} [2][10].

Une autre zone mémoire qui n'est pas partagée entre les différents fils d'un processus lourd est la pile d'exécution de chaque fil. La pile ne peut être partagée car les exécutions des différents fils sont indépendantes. La pile d'exécution sert à stocker le contexte de retour lors d'un appel de fonction (adresse de retour, registres…) ainsi que les variables de ces fonctions.

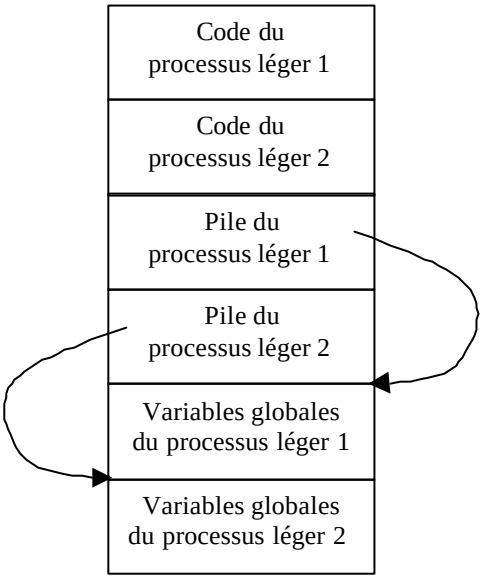


Figure 7 - *Les processus légers et leurs données privées*

Les piles des différents fils d'exécution sont allouées consécutivement en mémoire.

Comme nous l'avons vu au §5.1., les piles des fils d'exécution ne peuvent pas croître au-delà de la taille spécifiée lors de leur création. Seule la taille de la pile du fil initial peut être modifiée, car celle-ci croît vers la zone de mémoire utilisée pour le tas (comme la pile d'un processus monoprogrammé).

Une "certaine croissance" peut être permise aux piles des fils d'exécution dans la limite de la zone allouée à chaque pile et à l'espace entre les zones allouées. Il n'y a pas d'avantages à laisser de l'espace entre les zones allouées pour les piles pour permettre leur croissance [12]. En effet, dans le cas d'une allocation d'une large mémoire virtuelle, les pages de mémoire ne sont allouées (consommation de mémoire physique) qu'à partir du moment où le fil y fait un accès et génère une faute de page. Une allocation de larges zones consécutives en mémoire ou de petites zones espacées revient donc au même.

Des mécanismes de protection de pages (émission d'un signal au détenteur de la pile) permettent de s'assurer que la pile d'un fil ne déborde pas de la place qui lui est réservée.

5.4. Communications par messages

Les processus légers d'un même contexte communiquent au travers de la mémoire partagée.

Pour les processus légers d'un contexte différent, tel que les environnements répartis, une machine parallèle composée de machines indépendantes reliées par un réseau, des primitives de communication par messages sont fournies par certains systèmes. Ces primitives sont fournies souvent sous forme d'une bibliothèque qui s'interface avec la bibliothèque des processus légers [3][4].

Deux mécanismes d'échange de messages sont possibles :

- les "rendez-vous" par envoi et réception de messages (mécanisme bloquant),
- les files d'attente pour les messages reçus et envoyés de type boîte aux lettres (mécanisme non bloquant).

5.5. Signaux

L'utilisation des signaux conjointement à la multiprogrammation légère demande une attention particulière. Chaque processus léger dans un processus lourd peut avoir son propre masquage des signaux, pour indiquer les signaux qu'il souhaite traiter et ceux qu'il ignore.

Il existe deux catégories de signaux :

- les signaux asynchrones pour les interruptions,
- les signaux synchrones pour les exceptions et les trappes (*trap*)

Le traitement des signaux asynchrones (*signal handlers*) est défini pour l'ensemble du processus lourd. Il n'est pas possible d'envoyer un signal à un processus léger particulier d'un autre processus lourd : le signal est délivré à un processus léger arbitraire du processus lourd destinataire (exception faite des signaux synchrones générés par l'exécution elle-même qui sont délivrés au processus léger les ayant causés).

Une façon plus adaptée d'utiliser les signaux avec la multiprogrammation légère consiste à masquer les signaux dans tous les processus légers d'un processus lourd et d'attendre explicitement un ensemble de signaux (attente bloquante).

5.6. Annulation

Dans certains cas, il peut être utile d'arrêter un fil d'exécution avant la terminaison de son exécution (état "terminé" de la figure 5).

Des mécanismes d'annulation (*cancellation*) sont disponibles avec les processus légers du standard POSIX. L'annulation d'un fil, par un autre fil, provoque éventuellement un traitement de nettoyage (*cleanup handlers*) permettant de laisser les ressources partagées utilisées par le fil (verrou….) dans un état cohérent [10].

Trois modes de gestion des annulations sont possibles pour les fils d'exécution :

Annulation interdite	L'annulation n'a pas lieu tant que le fil reste dans ce mode. L'annulation reste en suspens.
Annulation différée	L'annulation pourra avoir lieu au prochain point d'annulation (*cancellation point*) atteint par le fil d'exécution. Certains appels systèmes sont définis comme étant des points d'annulation.
Annulation asynchrone	L'annulation est possible à tout moment.

6. Modèle de programmation

Il n'y a pas de règles de programmation pour l'utilisation des processus légers dans un programme.

Chaque programme comportant des processus légers est différent.

Cependant, certains modèles communs sont apparus. Ces modèles permettent de définir comment une application attribue une activité à chaque processus léger et comment ces processus légers communiquent entre eux.

6.1. Modèle répartiteur/travailleurs ou maître/esclaves (*Boss/Workers or Master/Slaves Model*)

Un processus léger, appelé le répartiteur (ou le maître), reçoit des requêtes pour tout le programme. En fonction de la requête reçue, le répartiteur attribue l'activité à un ou plusieurs processus légers travailleurs (ou esclaves).

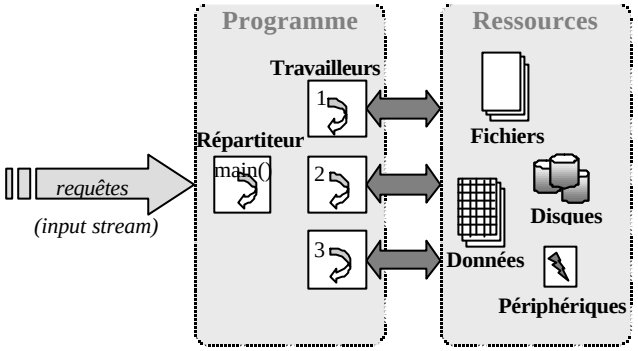


Figure 8 - *Le modèle répartiteur/travailleurs (Boss/Workers)*

Le répartiteur crée chaque processus travailleur en lui assignant une activité et attend, si cela est nécessaire, la fin du ou des processus travailleurs. Après avoir crée un processus léger, le répartiteur traite ou attend la prochaine requête.

L'exemple ci-dessous est le pseudo-code d'un programme appliquant le modèle répartiteur / travailleurs :

```
/* le répartiteur */
main()
{
      boucle infinie {
             réception d'une requête
             Suivant la requête :
                    si requête_1 : creation_processus_leger ( travailleur_1 )
                    si requête_2 : creation_processus_leger ( travailleur_2 )
                    …
      }
}
/*
 * Travailleur 1 : effectue les traitements spécifiques
 * pour les requêtes de type requete_1
 */
travailleur_1()
{
      Réalise les traitements correspondants,
      avec synchronisation si nécessaire avec les ressources partagées
}
```

Dans l'exemple précédent, les processus légers travailleurs sont créés dynamiquement lors de l'arrivée d'une requête. Une autre variante est la création, au départ, de tous les processus travailleurs (un processus léger par type de requête) et la création d'un processus intermédiaire (*thread pool*) s'occupant de la gestion des travailleurs. Ainsi, le répartiteur s'occupe principalement de la réception des requêtes et signale au processus intermédiaire le traitement à réaliser.

Le modèle répartiteur/travailleurs convient aux serveurs de bases de données, serveurs de fichiers, gestionnaires de fenêtres (*window managers*) et équivalents…

Le traitement de l'arrivée asynchrone des requêtes et de la communication avec les travailleurs est effectué par le répartiteur. La prise en charge des traitements correspondant aux requêtes est attribuée aux travailleurs.

Le plus important dans ce modèle est de minimiser la fréquence d'interaction entre le répartiteur et les travailleurs : le répartiteur ne doit pas perdre de temps pour la réception des requêtes. Il faut également veiller à ne pas utiliser trop de données partagées entre les travailleurs sous peine de ralentir le programme.

6.2. Modèle en groupe

Dans le modèle en groupe, chaque processus léger réalise les traitements concurremment sans être piloté par un répartiteur (les traitements à effectuer sont déjà connus). Le processus léger initial crée tous les autres processus légers lors du lancement du programme. Chaque processus léger traite ses propres requêtes (chaque processus peut être spécialisé pour un certain type de travail).

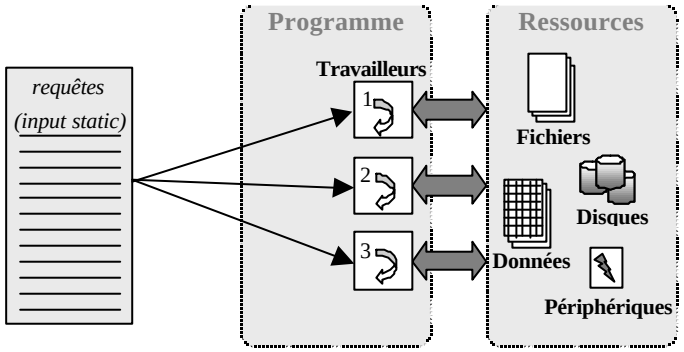


Figure 9 - *Le modèle en groupe*

L'exemple ci-dessous est le pseudo-code d'un programme appliquant le modèle en groupe :

```
main()
{
      creation_processus_leger ( travailleur_1 )
      creation_processus_leger ( travailleur_2 )
      …
      prévient les travailleurs qu'ils peuvent commencer les traitements
      attendre la terminaison de tous les travailleurs
      nettoyage éventuel
}

/* Travailleur 1 */
travailleur_1()
{
      Attente du départ
      traitements
      fin
}
…
```

Le modèle en groupe convient aux applications ayant des traitements définis à réaliser. Ces applications sont la multiplication de matrices, un moteur de recherche dans une base de données, un générateur de nombres premiers.

6.3. Modèle en *pipeline*

Dans ce modèle, l'exécution d'une requête est réalisée par plusieurs processus légers exécutant une partie de la requête en série. Les traitements sont effectués par étape du premier processus léger au dernier. Le premier processus engendre des données qu'il passe au suivant.

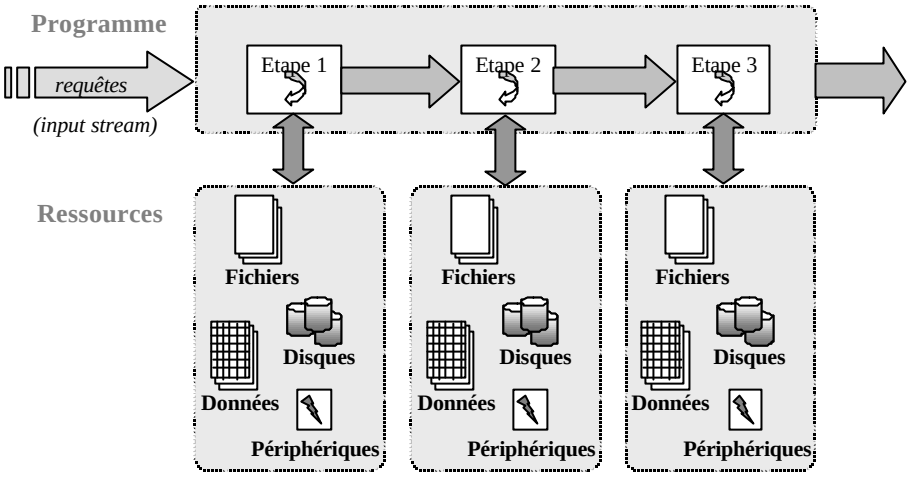


Figure 10 - *Le modèle en pipeline*

L'exemple ci-dessous est le pseudo-code d'un programme appliquant le modèle en ***pipeline*** :

```
main()
{
      creation_processus_leger ( Etape_1 )
      creation_processus_leger ( Etape_2 )
      …
      creation_processus_leger ( Etape_N )
      attendre la terminaison de tous les processus légers
      nettoyage éventuel
}

/* Etape 1 */
Etape_1()
{
      Boucle infinie
      {
            Récupération de la prochaine requête
            Réalise les traitements de l'étape 1
            Envoi des résultats au prochain processus du pipeline
      }
}
/* Etape 2 */
Etape_2()
{
      Boucle infinie
      {
            Récupération des données du précédent processus
            Réalise les traitements de l'étape 2
            Envoi des résultats au prochain processus du pipeline
      }
}
…
```

```
/* suite du programme… */

/* Etape N */
Etape_N()
{
       Boucle infinie
       {
              Récupération des données du précédent processus
              Réalise les traitements de l'étape N
              Envoi du résultat à la sortie du programme
       }
}
```

Le modèle en *pipeline* est utilisé pour les traitements d'images, traitements de textes, mais aussi pour les applications pouvant être décomposées en étapes.

Les traitements d'une étape peuvent être réalisés par plusieurs processus légers en parallèle. L'étape la plus lente définit les performances du modèle. Les étapes doivent être équilibrées au niveau des temps de traitement.

7. Ordonnancement

Ce paragraphe présente les politiques d'ordonnancement les plus utilisées dans les systèmes de processus légers.

La politique d'ordonnancement a pour but de gérer les processus légers lorsque le nombre de processeurs disponibles est inférieur au nombre de processus légers prêts à être exécutés.

L'ordonnancement permet de désigner le prochain processus léger à exécuter lorsqu'un autre processus rend le processeur disponible.

On distingue deux familles de politiques d'ordonnancement :

- les politiques **préemptives**, autorisant la perte du processeur par un fil au profit d'un autre,
- les politiques **non-préemptives** : un fil perd le processeur lorsqu'il est explicitement bloqué.

Des notions de priorités peuvent être utilisées dans les politiques d'ordonnancement permettant de modifier l'allocation de la ressource processeur aux fils d'exécutions.

Les ordonnanceurs gèrent également une file d'attente des fils prêts à s'exécuter (ordonnancement sans priorité) ou une file d'attente des fils prêts par niveau de priorité (ordonnancement avec priorité).

7.1. Ordonnancement FIFO

L'ordonnancement FIFO (*First In, First Out*) sans priorité est un ordonnancement non-préemptif : chaque fil s'exécute jusqu'à se terminer ou se bloquer, pour ensuite laisser le processeur au prochain fil qui attend depuis le plus longtemps et ainsi de suite.

L'ordonnancement FIFO avec priorités est différent du précédent. Un paramètre supplémentaire est à prendre en compte. Si un fil de plus haute priorité est prêt, il prend le processeur au fil de plus faible priorité et ce jusqu'à se terminer ou se bloquer. Tant qu'il y a des fils de priorités supérieures, aucun autre fil ne pourra s'exécuter. L'ordonnancement est dans ce cas préemptif.

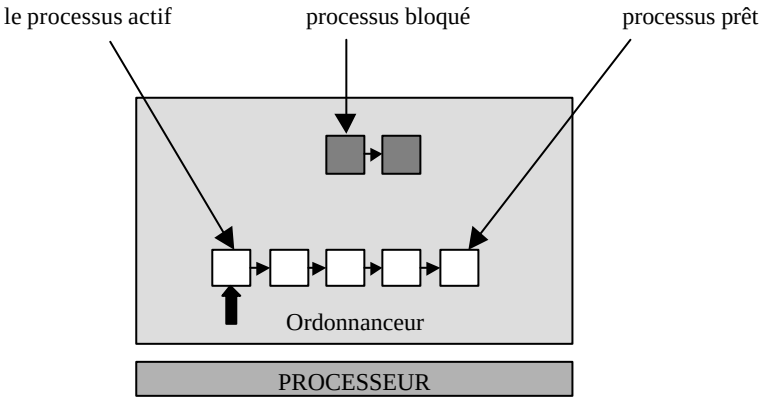


Figure 11 - *Ordonnancement FIFO (First In, First Out)*

Selon les niveaux d'implantations des processus légers (voir §8 pour les différentes implantations possibles des processus légers dans un système d'exploitation), il est possible de demander un ordonnancement de tous les fils d'exécution d'un même processus lourd ou de tous les fils d'exécutions s'exécutant sur le même processeur.

Un fil, qui ne se bloque pas ou ne se termine pas, bloque les autres fils du processus lourd dans le premier cas, mais dans le second cas, ce fil peut paralyser tout le système (pour une machine monoprocesseur).

7.2. Ordonnancement *round-robin*

L'ordonnancement round-robin ("à tour de rôle" ou en tourniquet[6]) fait intervenir la notion d'intervalle de temps[7].

Chaque fil s'exécute dans un intervalle de temps et laisse la place au prochain fil prêt.

Le prochain fil d'exécution peut être un fil d'une priorité plus importante, dans le cas d'un ordonnancement avec priorité.

L'ordonnancement round-robin, avec ou sans priorités, est un ordonnancement préemptif.

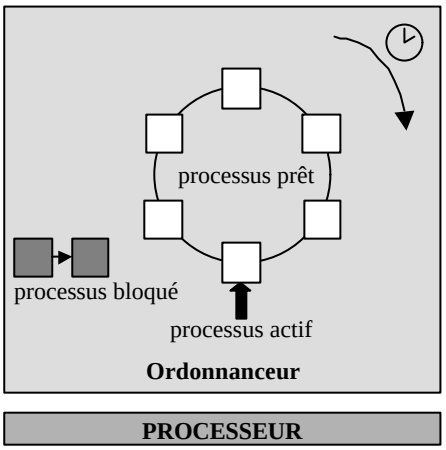


Figure 12 - *Ordonnancement en tourniquet*

7.3. Autres ordonnancements

Le standard POSIX fournit d'autres stratégies d'ordonnancement dont l'ordonnancement de type "temps partagé". Chaque fil de basse comme de haute priorité s'exécute à tour de rôle. Chaque fil est préempté après un certain quantum de temps variable en fonction de leur priorité (les fils de haute priorité s'exécutent plus longtemps que les fils de basse priorité).

[6] cette technique est également connue sous le nom d'ordonnancement circulaire
[7] quantum de temps

7.4. Priorités et *mutex*

Le problème cité dans ce paragraphe s'applique uniquement aux systèmes monoprocesseurs.

Dans un ordonnancement à priorités, des phénomènes "d'inversion de priorité" peuvent se produire lorsqu'un fil de basse priorité verrouille un *mutex* et qu'un autre fil de haute priorité désire également verrouiller ce *mutex*.

Il existe deux types "d'inversion de priorité".

- Bornée (*bounded priority inversion*) : le fil de haute priorité est bloqué pendant la durée de la section critique du fil de basse priorité (l'attente est dans ce cas bornée).
- Non bornée (*unbounded priority inversion*) : un fil de priorité moyenne préempte le fil de basse priorité qui est encore en section critique. Ainsi le fil de priorité moyenne empêche le fil de haute priorité (qui attend la libération du *mutex*) de s'exécuter . La durée de cette situation est indéterminée (figure 13).

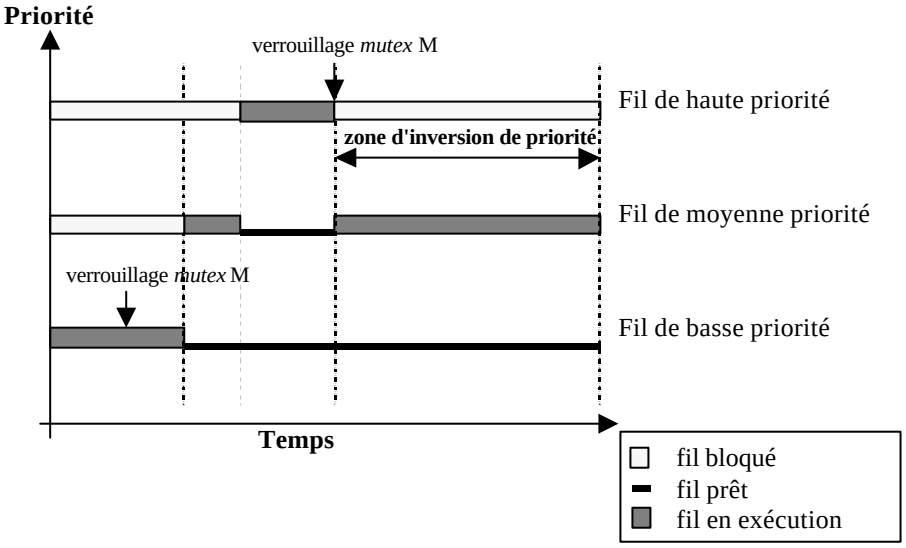


Figure 13 - *Inversion de priorité non bornée (unbounded priority inversion)*

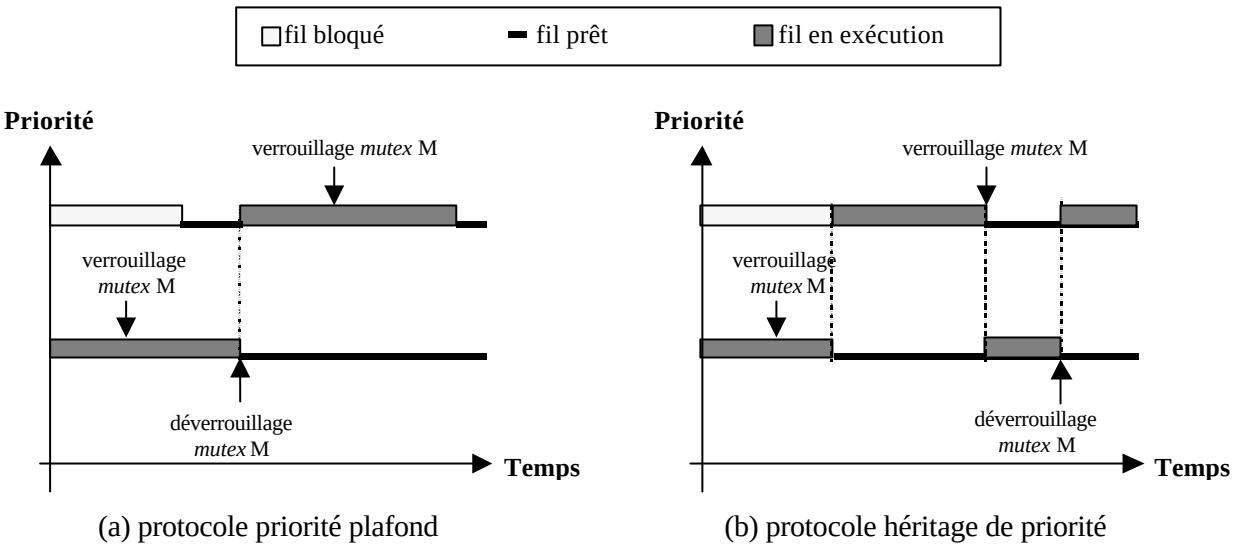
La solution, pour contourner ce problème, consiste à changer la priorité du fil qui verrouille le *mutex*.

Le changement de priorité peut s'effectuer selon deux protocole possibles [18] :

Le premier consiste à augmenter la priorité du fil qui verrouille un *mutex* jusqu'à une priorité plafond. Chaque *mutex* possède une valeur de priorité plafond. La difficulté réside sur le choix de la valeur de priorité plafond pour ne plus avoir d'inversion de priorité. Ce protocole est appelé "protocole de priorité plafond" (*priority ceiling protocol*).

Le deuxième est plus complexe, mais plus souple à utiliser. Le fil qui verrouille un *mutex* conserve sa priorité. Si avant qu'il ne déverrouille le *mutex*, d'autres fils tentent de le verrouiller et se bloquent, le fil possédant le *mutex* voit sa priorité augmentée pour prendre la plus grande valeur de priorité des fils en attente. Ce protocole est appelé "héritage de priorité" (*priority inheritance*).

La figure 14 présente les deux protocoles permettant de contourner les situations d'inversion de priorité.



(a) protocole priorité plafond (b) protocole héritage de priorité

Figure 14 - *Les protocoles "priorité plafond" et "héritage de priorité"*

8. Processus légers et les systèmes d'exploitation

L'implantation des processus légers dans un système d'exploitation soulève le problème de leur ordonnancement au sein d'un processus lourd.

En effet, les systèmes d'exploitation s'occupent de tous les aspects de l'exécution d'un processus lourd.

La gestion de l'ordonnancement des processus légers est répartie entre l'application et le système d'exploitation. Cette répartition conditionne l'indépendance de l'implantation des processus légers du système d'exploitation et de l'allocation des ressources aux processus légers.

L'ordonnancement de ces processus dépend principalement du choix d'implantation de la bibliothèque des processus légers (*threads package*) et de son intégration avec le système d'exploitation.

Il existe plusieurs façons d'implanter des processus légers dans un système d'exploitation. Les implantations sont classées selon l'interaction entre les processus légers et les entités noyau (*kernel entity*).

Une entité noyau est une entité qui se voit attribuer un processeur pour exécuter un processus léger. Cette entité peut être vue comme un processeur virtuel.

Les paragraphes suivants présentent les différentes implantations, ainsi que leurs avantages et leurs inconvénients. Quelques systèmes de processus légers sont présentés à la fin de ce chapitre.

8.1. Niveau utilisateur

Dans ce type d'implantation, tous les processus légers d'un processus lourd se partagent la même entité noyau pour leur exécution. On parle aussi de processus légers multiplexés sur une seule entité noyau. Cette implantation est compatible avec les systèmes d'exploitation non prévus pour les processus légers. Elle peut être réalisée sous forme de bibliothèque sans modification du noyau du système d'exploitation.

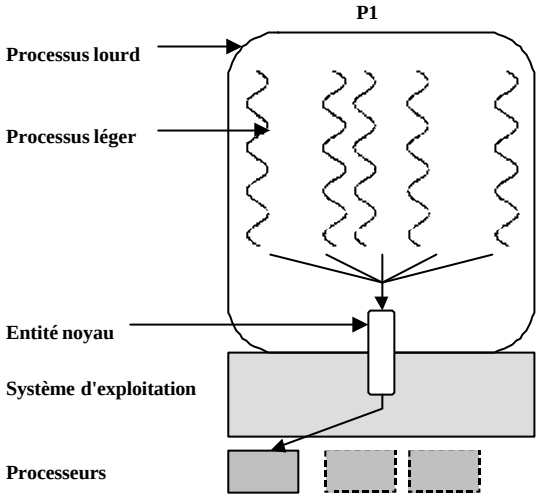


Figure 15 - *Plusieurs fils d'exécution sont liés à une entité noyau*

8.1.1. Avantages

Dans cette implantation, la gestion et la commutation des processus légers sont rapides car elles ne nécessitent pas d'appel au noyau (coûteux en temps).

Un autre avantage est la mise en oeuvre de cette implantation sur tout système d'exploitation.

De plus, chaque processus léger peut avoir un algorithme d'ordonnancement particulier.

8.1.2. Inconvénients

Quand un fil d'exécution fait un appel système bloquant, tous les fils du même processus lourd se bloquent. Une solution consiste à éviter les appels systèmes bloquants en utilisant d'autres moyens. Ces moyens peuvent devenir très lourds à mettre en œuvre (par exemple avant d'effectuer une lecture bloquante, le programme doit s'assurer au préalable de la présence de données sur le périphérique ☞ fonction *select* au lieu du *read* sous UNIX).

Les processus légers d'un même processus lourd ne peuvent pas exploiter plusieurs processeurs physiques, car l'entité noyau associée est placée sur un processeur physique donné. De plus, les processus légers sont invisibles du noyau.

La gestion de l'ordonnancement des processus légers est laissée à la charge de l'utilisateur[8].

[8] ici l'utilisateur est désigné par le "développeur d'applications"

8.2. Niveau noyau

Dans cette implantation, chaque processus léger est pris en charge par une entité noyau. Les processus légers sont totalement implantés dans le noyau du système d'exploitation. Ce type d'implantation est appelé "processus légers noyau" (*kernel threads*).

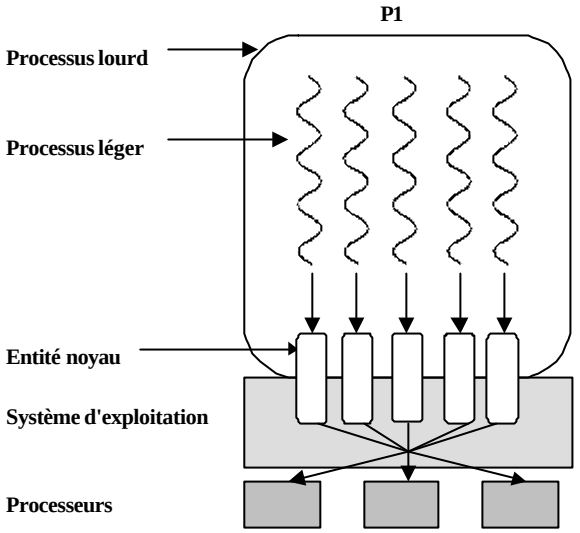


Figure 16 - *Chaque fil d'exécution est lié à une entité noyau*

8.2.1. Avantages

Les blocages des processus légers se font dans le noyau par le biais d'un blocage de l'entité noyau. Le système passe, alors, à l'exécution d'une autre entité noyau associée à un autre processus léger. Ainsi un processus léger en attente ne bloque plus les autres processus légers.

Les machines multiprocesseurs conviennent mieux à cette implantation, car le système d'exploitation peut placer des entités noyau différentes sur différents processeurs physiques.

L'ordonnancement des processus légers est laissé à la charge du noyau.

8.2.2. Inconvénients

La gestion des processus légers est réalisée par des appels systèmes coûteux en temps : la commutation de fil ou la synchronisation implique un changement de contexte et des vérifications par le noyau de la validité des paramètres.

Les entités noyau occupent de la place mémoire, or la mémoire disponible dans le noyau n'est pas illimitée. Cet aspect limite le nombre de processus légers disponibles pour l'ensemble du système.

8.3. Niveau "hybride"

Cette implantation permet de regrouper les différents avantages des deux implantations précédentes tout en évitant les inconvénients. Dans cette implantation, plusieurs processus légers en niveau utilisateur ont à leur disposition plusieurs entités noyau.

Lorsqu'une entité noyau est bloquée en attente d'une synchronisation ou d'une entrée/sortie (ou autre), le noyau informe la bibliothèque de niveau utilisateur. Un autre processus léger est engendré pour maintenir le nombre de processus légers en cours d'exécution.

Le nombre d'entités noyau disponible peut être modifiable par le programmeur.

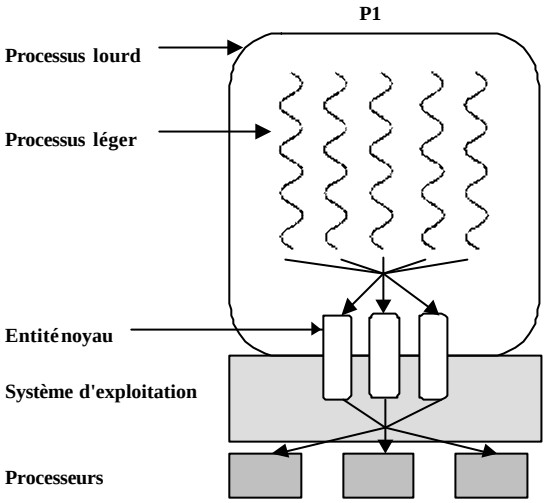


Figure 17 - *Un ensemble de fils d'exécution se partage un ensemble d'entités noyau*

8.3.1. Avantages

L'implantation en niveau utilisateur garantit des temps de commutation et de synchronisation très courts et favorise l'extension du système (*scalability*).

La présence de plusieurs entités noyau permet d'éviter les blocages des autres fils quand un fil se bloque.

La multiplicité des entités noyau rend efficace l'exploitation des multiprocesseurs.

La latence d'un processus bloqué au niveau du noyau est très courte, car une autre entité noyau est réactivée.

8.3.2. Inconvénients

Le seul inconvénient provient de la complexité de la mise en œuvre de cette implantation.

Cela nécessite une gestion rigoureuse dans la création et la destruction des entités noyau.

Les nombreuses opérations de création, destruction et re-création des entités noyaux peuvent conduire à des phénomènes d'instabilité.

8.4. Exemples d'implantation des processus légers dans les systèmes d'exploitation

Ce paragraphe présente quelques exemples d'implantation des processus légers dans les systèmes d'exploitation, d'autres exemples sont fournis dans les documents [1] et [8].

8.4.1. Les processus légers de SunOs (versions antérieures au système Solaris)

La bibliothèque des processus légers de SunOs est implantée au niveau utilisateur et s'appelle *Light Weight Processes* (LWP) [12].

Les objets LWP sont uniquement accessibles à l'intérieur d'un même processus lourd.

Les primitives fournies par cette bibliothèque comprennent :

- Création de processus léger, destruction, ordonnancement, désactivation, activation…
- Multiplexage de l'horloge (plusieurs processus légers peuvent "dormir" concurremment).
- Changement de contexte individuellement.
- Synchronisation des processus légers par moniteurs et variables conditions.
- Mécanisme de rendez-vous entre les processus légers (envoi/réception de messages et de réponses).
- Gestion des exceptions et des erreurs.

Dans cette bibliothèque, plusieurs processus légers partagent un même espace mémoire. Chaque LWP est représenté par une procédure convertie en processus léger. Chaque processus léger possède un contexte, une pile (*stack*) et il est prêt à être exécuté. Un ensemble de processus légers s'exécutant au sein d'un processus traditionnel est également appelé *pod*.

L'ordonnancement des processus légers est de type FIFO avec priorité (*first-come, first-served basis*). Deux processus légers de même priorité s'exécutent selon leur ordre de création. La bibliothèque fournit des primitives permettant à l'utilisateur de définir un ordonnancement spécifique.

Un processus léger peut explicitement laisser le contrôle à un autre (fonction *lwp_yield*) de même priorité.

Par exemple, pour mettre en œuvre un ordonnancement de type tourniquet et en temps partagé (*round-robin time-sliced scheduler*), le processus léger de plus haute priorité joue le rôle d'ordonnanceur des processus de moindre priorité. Le processus "ordonnanceur" se désactive durant un quantum de temps désiré et se réveille quand ce délai expire.

Les deux principaux mécanismes de synchronisation fournis sont les mécanismes de "rendez-vous" et le mécanisme des moniteurs.

Les "rendez-vous" s'effectuent par l'envoi et la réception d'un message. L'émetteur se bloque en attente d'une réception.

Les moniteurs sont similaires aux fonctions de désactivation (*sleep*) et de réveil (*wakeup*) du système UNIX. Les moniteurs sont souvent couplés avec les variables conditions.

8.4.2. Les processus légers de Solaris

Les processus légers du système *UNIX Solaris* utilisent l'implantation hybride. On trouve ainsi des fils de contrôle (*thread of control*) au niveau utilisateur (*user-level threads*) et les processus légers au niveau du noyau (*kernel-threads*) [5].

Les processus légers au niveau noyau sont appelés *LightWeight Processes* (*LWPs*). Les *LWP* entre *SunOs* et *Solaris* (à partir de *SunOs 5.0*) sont fondamentalement différents [8].

Les fils de contrôle sont uniquement visibles à l'intérieur d'un processus dans lequel ils partagent les ressources telles que la mémoire, les fichiers ouverts, etc… Les états suivants sont uniques pour chaque fil de contrôle :

- l'identification du fil de contrôle,
- le registre d'état (comportant le compteur ordinal et le pointeur vers la pile),
- la pile,
- le masquage des signaux,
- la priorité,
- les données privées.

Lorsqu'un fil de contrôle modifie une donnée partagée, cette modification est vue par les autres fils de contrôle. Ainsi les fils de contrôle communiquent sans invoquer le système d'exploitation.

Les fils de contrôle sont gérés dans l'espace utilisateur et évitent d'utiliser le noyau pour réaliser un changement de contexte. Une application peut posséder des milliers de fils de contrôle sans pour autant augmenter la consommation des ressources du noyau.

Les *LWPs* de *Solaris* peuvent être vus comme des processeurs virtuels disponibles pour l'exécution du code ou pour un appel système.

Les processus légers (*LWPs*) sont en fait un pont entre le niveau utilisateur et le niveau noyau. Chaque processus contient un ou plusieurs processus légers qui exécutent un ou plusieurs fils de contrôle.

Quand un fil de contrôle se bloque suite à une synchronisation, le *LWP* sur lequel il s'est bloqué est transféré vers un autre fil de contrôle prêt à s'exécuter. Ce transfert est réalisé sans intervention du système d'exploitation.

Le système d'exploitation décide de l'attribution des processeurs aux *LWPs*. Mais il n'a aucune visibilité des fils de contrôle des processus.

L'ordonnancement des *LWPs* est réalisé par le noyau selon leur classe d'ordonnancement et leur priorité.

Les fils de contrôle peuvent être, soit liés en permanence à un processus léger (*bound threads*), soit attachés ou détachés parmi un *pool* de processus légers (*unbound threads*).

Les fils de contrôle et les processus légers permettent au programmeur :

- de contrôler le degré maximum de parallélisme (*concurreny leve*l dans la terminologie *Solaris*),

- d'établir une correspondance (ou une liaison) entre les fils de contrôle et les *LWPs*.

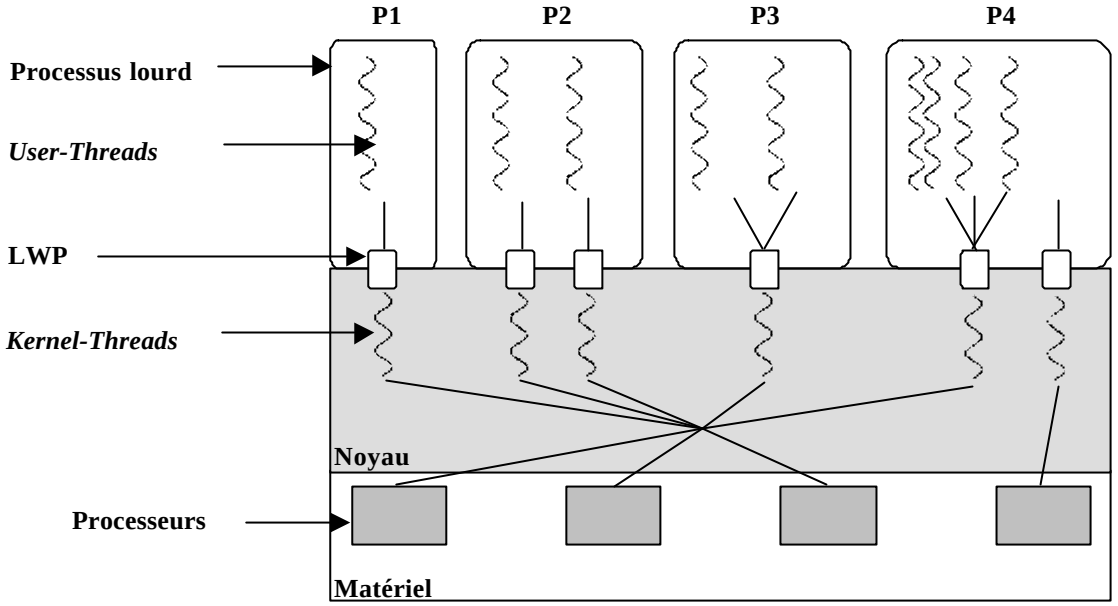


Figure 18 - *Le modèle de processus légers à deux niveaux du système Solaris*

8.4.3. Les processus légers de POSIX

La norme des processus légers POSIX est largement acceptée aujourd'hui et la plupart des constructeurs en proposent une implantation sur leurs machines. La norme POSIX définit l'interface pour la gestion des processus légers.

Les fonctionnalités suivantes sont présentes dans cette norme :

- gestion des processus légers (*thread management*) : initialisation, création, destruction…;
- synchronisation : exclusion mutuelle, variables conditions;
- données privées (*thread-specific data*);
- ordonnancement (*thread priority scheduling*) : gestion des priorités, ordonnancement préemptif;
- signaux : traitant des signaux (*signal handler*), attente asynchrone, masquage des signaux, saut dans un programme (*long jumps*);
- annulation (*cancellation*) : nettoyage des traitants (*cleanup handlers*), annulation asynchrone, synchrone, et désactivation.

Cette norme présente également une extension incluant d'autres règles d'ordonnancement et de contrôle de processus.

Les processus légers POSIX se déclinent en trois sous-groupes :

- Les "vrais" POSIX *threads*, basés sur le standard IEEE POSIX 1003.1c-1995 (également connus sous la norme ISO/IEC 9945-1:1996). Le standard POSIX est largement mis en œuvre sur les systèmes UNIX.
  - Le standard des processus légers POSIX est connu sous le nom de *pthread.*

  - Les processus légers POSIX sont souvent référencés en tant que POSIX.1c *threads* car la désignation 1003.1c correspond à la section du standard concernant les processus légers.
  - Le *draft* 10 de POSIX.1c qui a été standardisé.

- Un brouillon antérieur au standard précédent a également été implanté et a connu une large diffusion. Il s'agit du *draft* 4 de POSIX 1003.4a de 1990 (10003.4a/D4) qui a été intégré au *Distributed Computing Environment* (DCE) de l'*Open Software Foundation* (OSF) et est connu sous le nom de DCE *Threads.*
- Les processus légers de l'*Unix International* (*UI threads*), connus sous le nom de *Solaris thread*, sont basés sur le standard POSIX. Les variantes d'Unix supportant les *UI Threads* sont Solaris 2 de Sun et UnixWare 2 de SCO.

Le document [10] présente, de manière détaillée, la programmation des *pthreads* avec de nombreux exemples.

8.4.4. Les processus légers Microsoft

Il existe deux sous-groupes de processus légers Microsoft :

- Les processus légers WIN32 (*WIN32 threads*, *Microsoft Win32 API threads*) pour le standard des processus légers sur Windows 95/98 et Windows NT.
- Les processus légers OS/2 (*OS/2 threads*) pour le standard des processus légers sur OS/2 (d'IBM).

Ces processus légers ont été mis en œuvre à l'origine par Microsoft, mais leur implantation a divergé au cours des années.

Les processus légers *WIN32* sont implantés au niveau du noyau. L'unité d'exécution finale est le processus léger. L'ordonnancement est réalisé selon l'algorithme du tourniquet (*round-robin scheduling*). Tous les processus légers accèdent aux mêmes ressources du processus lourd, en particulier à son espace mémoire. Un processus lourd accède à une espace mémoire de 4 Gigaoctets (Go), découpé en 2 Go pour l'utilisateur et 2 Go pour le système d'exploitation. Chaque processus léger peut accéder à cette zone de 2 Go. Un processus léger d'un processus lourd ne peut pas accéder aux ressources d'un autre processus lourd. Ainsi, l'exécution n'est pas perturbée et cela rend le système d'exploitation stable.

La manipulation des processus légers nécessite l'utilisation d'objets noyaux : objets événements, des boîtes aux lettres, *mutex*, sémaphores…

Les processus légers sont considérés comme un objet noyau. Les objets noyaux sont différents aux objets systèmes, car ils possèdent des attributs de sécurité.

Les fonctionnalités suivantes sont fournies avec les processus légers *WIN 32 threads* :

- gestion des processus légers : création, terminaison, priorités, suspension.
- gestion de la synchronisation : sections critiques, *mutex*, sémaphores, événements.
- communication : une file de message associée à chaque processus léger permet d'envoyer des messages à des processus légers du système.

La synchronisation sur un objet noyau est réalisée en utilisant une fonction commune (*WaitForSingleObject*). Une autre fonction permet également l'attente multiple sur une synchronisation (*WaitForMultipleObject*).

8.4.5. Les processus légers sous Mach

Le système d'exploitation Mach est basé sur la technologie des micro-noyaux. Mach est construit de façon à pouvoir émuler UNIX et d'autres systèmes d'exploitation. Cette émulation est réalisée à l'aide d'une couche logiciel exécutée en dehors du noyau, dans l'espace utilisateur.

Dans la technologie des micro-noyaux, le noyau du système d'exploitation contient un nombre restreint de fonctions systèmes, les autres fonctions sont assurées par des serveurs extérieurs au noyau (figure 19).

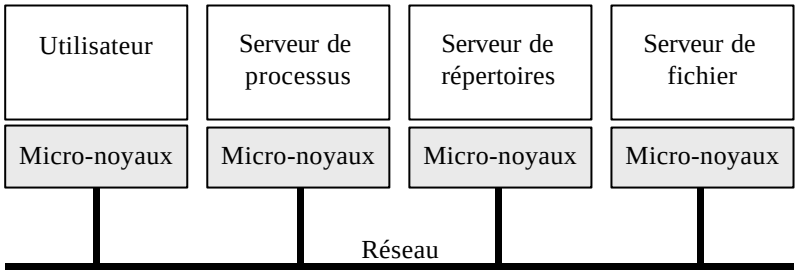


Figure 19 - *Architecture micro-noyau*

Les processus légers de Mach sont gérés par le noyau et sont les entités actives de Mach. Ils partagent l'espace d'adressage et toutes les ressources détenues par le processus lourd. Ils disposent également de ressources privées comme le port de processus léger (*thread port*) et le port d'exception des processus légers (*thread exception port*). Le processus léger utilise les ports pour appeler des services noyaux spécifiques comme le service de terminaison après la fin de l'exécution.

L'interface de base du noyau offre environ une vingtaine de primitives de gestion des processus légers. Un ensemble de ces primitives concerne le contrôle de l'ordonnancement.

Plusieurs bibliothèques peuvent être construites au-dessus de ces primitives. La bibliothèque des C *Threads* (*C Threads package*) en est un exemple. Elle permet la programmation parallèle en utilisant le langage C sous Mach. Elle permet la gestion des processus légers, le partage de la mémoire et la gestion de la synchronisation entre processus légers au moyen de *mutex* et de variables conditions.

Il existe trois variantes de cette bibliothèque :

- La première fonctionne entièrement dans l'espace utilisateur au sein d'un seul processus lourd. Cette approche effectue le partage de temps de tous les processus légers au-dessus d'un processus léger du noyau.
- La deuxième utilise un processus léger Mach par processus léger de la bibliothèque C.
- La troisième consiste à n'avoir qu'un seul processus lourd par processus léger. Dans cette variante, la mémoire est partagée entre tous les processus lourds. Cependant les ressources propres à un processus lourd ne peuvent être partagées. Cette dernière variante est peu utilisée.

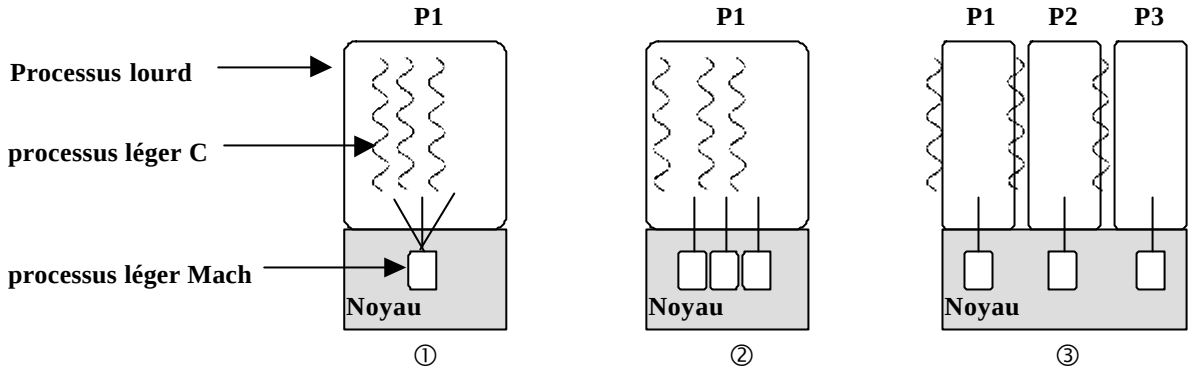


Figure 20 - *Les variantes de la bibliothèque C Threads*

8.4.6. Les processus légers sous Amoeba

Amoeba est également un système d'exploitation basé sur la technologie des micro-noyaux. Amoeba utilise un modèle simple de processus légers. Un processus lourd dispose d'au moins un processus léger au démarrage. Pendant l'exécution, le processus léger peut en créer ou terminer d'autres.

Tous les processus légers sont gérés par le noyau.

Tous les processus légers partagent le même texte de programme et les données globales. Chacun possède sa pile, son pointeur de pile et sa copie des registres de la machine. Un processus léger possède des données privées qui sont appelées des données *glocales*. Les données *glocales* sont un bloc de mémoire alloué dont seul le processus léger possède une visibilité de ce bloc.

Trois méthodes de synchronisation sont disponibles : les signaux, les verrous (ou *mutex*) et les sémaphores.

Les signaux sont des interruptions asynchrones envoyées d'un processus léger à un autre du même processus lourd. Le système des signaux est conceptuellement identique aux signaux UNIX, à ceci près qu'il s'agit de processus légers. Les signaux peuvent être traités, interceptés ou ignorés. Les interruptions asynchrones entre processus utilisent le mécanisme de paralysie.

Les sémaphores sont un peu plus lents que les *mutex*, mais leurs utilisations sont parfois nécessaires. Leurs utilisations sont standards et offrent la possibilité de demander un accès avec un temps de garde.

L'ordonnancement des processus légers est basé sur des priorités favorisant le noyau par rapport aux utilisateurs.

9. Les processus légers dans les supports d'exécution des langages

9.1. Cible de compilateurs

La difficulté de concevoir des applications parallèles au-dessus d'outils systèmes de bas niveau a entrainé l'apparition de langages parallèles. Ces langages fournissent un niveau d'abstraction au programmeur pour la manipulation d'entités actives au sein de leur application.

L'approche est simple pour le programmeur, mais cela augmente la difficulté dans la réalisation des compilateurs qui doivent ainsi prendre en charge toutes les opérations de gestion du parallélisme (création, commutation, ordonnancement, synchronisation…).

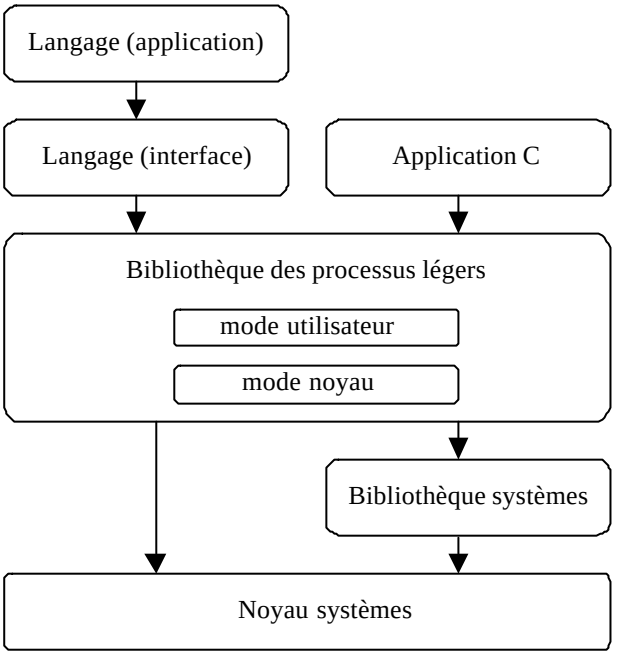


Figure 21 - *Niveau d'abstraction – couches logiciel*

Une autre approche consiste à choisir des bibliothèques de processus légers pour cible de la génération de code. Cela facilite le développement des compilateurs et augmente leur portabilité [18].

Les exécutables générés par ces compilateurs n'utilisent pas directement les fonctionnalités du système mais exploitent les primitives associées à la gestion des processus légers.

Cette approche présente les avantages suivants :

Simplicité du compilateur	Le compilateur s'appuie sur des primitives élaborées de gestion d'activités parallèles.
Portabilité du langage	Les bibliothèques de processus légers sont souvent disponibles sur plusieurs architectures. Si le compilateur n'utilise que les fonctionnalités standards des processus légers, le portage de celui-ci s'effectue sans modification importante.
Fiabilité du compilateur	Les risques d'erreur sont réduits (gestion séparée du parallélisme). Les bibliothèques de processus légers ont déjà subit de nombreux tests permettant de garantir leur robustesse.

Un exemple d'utilisation de cette approche est présenté dans l'article [13]. Cet exemple présente l'utilisation de la bibliothèque de processus légers *Pthreads* par le langage *Ada95.*

9.2. Support d'exécution pour les langages objets

La programmation objet simplifie la modélisation des programmes parallèles ou la simulation du monde réel. Les processus légers sont utilisés pour supporter les multiples activités concurrentes et/ou parallèles introduites dans le modèle de programmation.

9.3. Java

Les processus légers font partie du mode de fonctionnement de Java [15].

Le langage Java permet de développer très facilement des flots d'exécution concurrents (*threads* ou processus léger) dans une même application.

Les langages proposant des mécanismes de *multithreading* utilisent souvent des librairies externes. En intégrant la capacité de gestion des processus légers dans le langage, les programmes sont portables d'une plate-forme à l'autre.

Toute la librairie Java est réentrante (*thread safe*), c'est-à-dire que plusieurs processus légers peuvent appeler une fonction ou une méthode en même temps.

Java fournit des structures particulièrement simples pour synchroniser les activités des processus légers. Elles sont basées sur le concept des moniteurs pour la mise en place de verrous. La synchronisation des accès aux ressources est assez simple. Le langage gère l'initialisation et l'acquisition des verrous : il suffit au programmeur d'indiquer les ressources à verrouiller (*synchronized*).

Les cas d'interblocage, suite à un oubli de déverrouillage d'un verrou, ne peuvent pas se produire, car la gestion du verrou est effectuée par le langage.

Java fournit également des mécanismes de gestion de l'ordonnancement des processus légers.

9.4. Les processus légers dans les applications

De plus en plus d'applications utilisent les processus légers de manière directe pour exprimer efficacement et facilement le parallélisme naturel qu'elles contiennent.

Les applications, comportant une interface graphique, augmentent leur réactivité en utilisant le concept de processus légers.

Le principe consiste à attacher un ou plusieurs processus légers à chaque élément de l'interface (bouton, menu…). Ces processus, créés à chaque occurrence d'un événement, permettent de prendre en charge de façon asynchrone les traitements associés à cet événement.

Cela permet à l'application de se focaliser sur la détection des événements extérieurs (souris, clavier, port de communication…).

Un autre domaine d'application est celui des simulations. Dans ces applications, il s'agit d'observer le comportement collectif d'un certain nombre d'activités complexes souvent asynchrones.

L'utilisation de processus légers simplifie énormément la programmation de la simulation : un processus léger est associé à une activité de la simulation.

La description d'un comportement de chaque entité active est moins complexe que celle d'un ordonnanceur spécialisé, chargé de simuler l'évolution parallèle d'entités codées de manière synthétique.

L'utilisation des processus légers est bien implantée dans les jeux informatiques, en particulier les jeux de stratégie, où le joueur doit faire évoluer une population d'entité active (personnage, ennemi, animaux…).

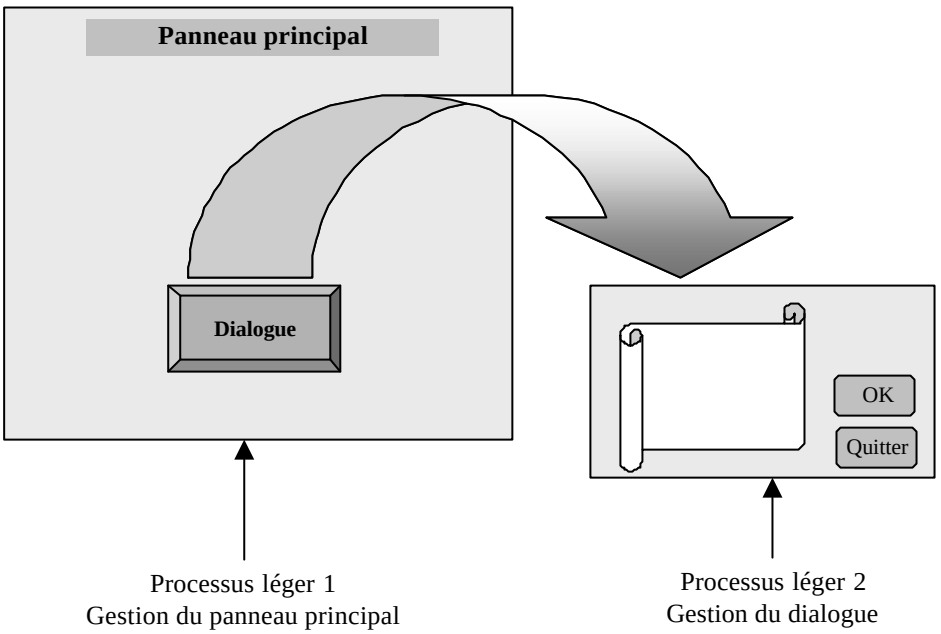


Figure 22 - *Gestion d'une interface graphique avec les processus légers*

10. Processus légers et codes existants

Les processus légers ont accès à l'intégralité d'un même espace d'adressage. Ainsi toutes les variables globales sont vues par les processus légers. Même les variables globales gérées par une autre bibliothèque, dont le source n'est pas souvent disponible, peuvent être accédées (directement ou indirectement) de la même manière.

Le risque se pose lors de l'accès à des données de manière concurrente entre la bibliothèque et les processus légers. Une synchronisation peut être mise en place au niveau des processus légers mais pas au niveau de la bibliothèque : pas de modifications possibles des codes sources.

Un accès ou une exécution pouvant être effectué de manière concurrente, est dit **réentrant.**

Il existe deux types de réentrance : celle relative à des multiples fils d'exécution (thread safe et thread aware) et celle relative au traitement des signaux (async-safe).

La non réentrance est appelée *thread unsafe*.

Thread unsafe	Un seul fil d'exécution peut appeler la fonction ou bibliothèque sous peine de comportement erroné. Parfois, seul le fil initial d'un processus multiprogrammé est autorisé à effectuer les appels pour simuler un processus monoprogrammé par rapport à la fonction appelée. ☞ Risque de comportement erroné.
Thread safe	Plusieurs fils peuvent accéder concurremment à la fonction ou à la bibliothèque. En cas d'appel bloquant par un fil, tout le processus lourd est bloqué. ☞ Comportement correct.
Thread aware	Plusieurs fils peuvent accéder concurremment à la fonction ou à la bibliothèque. Seul le fil ayant effectué un appel bloquant est bloqué. ☞ Fonctionnement correct.
Async-safe	Cette définition s'applique à une ou des fonctions particulières d'une bibliothèque en général. Il s'agit d'une fonction pouvant être appelée à partir d'un traitant de signal (*signal handler*). ☞ Comportement indéterminé.

Une fonction ou une bibliothèque peut ne pas fonctionner correctement en présence de plusieurs processus légers pour plusieurs raisons.

Raisons pour la fonction ou la bibliothèque :

- Des données privées à la fonction ou à la bibliothèque présentent des états incohérents lorsqu'elles sont accédées concurremment.
- Si une fonction maintient dans ses variables un état entre deux appels successifs, la valeur de l'état lue par un fil peut être incohérente si celui-ci est attribué à un autre fil.
- Utilisation des variables allouées statiquement pour le stockage des résultats intermédiaires d'un traitement non réentrant mais dont l'interface est réentrante.
- Une fonction retourne un pointeur vers des données statiques pouvant être à la fois modifiées par un fil, utilisées par un autre fil et également par la fonction.

Raisons pour les processus légers :

- l'appel à une fonction bloquante bloque tous les processus légers y compris le processus ayant réalisé l'appel.
- l'annulation (*cancellation*) des processus n'est pas gérée correctement par la bibliothèque.

Les solutions permettant de résoudre ces problèmes sont :

- interdiction par verrou à plusieurs fils de s'exécuter concurremment dans la bibliothèque. Le verrouillage peut être interne si la bibliothèque dispose de mécanismes adéquats et/ou externes en sérialisant les appels.
- utilisation de zones de données privées.
- utilisation de fonctions non-bloquantes de la bibliothèque
- utilisation de mécanismes de nettoyage lors de la disparition des fils, afin de libérer les ressources relatives à la bibliothèque.

Ces solutions ne résolvent pas tout. La réécriture de certaines fonctions de la bibliothèque est parfois la meilleure solution.

Il faut retenir que les processus légers sont récents et qu'il est difficile de les intégrer avec des anciennes bibliothèques non prévues à cet effet.

11. Performances des processus légers

A l'intérieur des différents documents sur les processus légers, des mesures de performances sont indiquées. L'ensemble de ces informations sont donc regroupées dans ce paragraphe. La manière dont a été réalisée la mesure est également précisée (lorsque l'explication est fournie).

11.1. Performance constructeur : le système Sun Solaris

Les mesures de performance indiquées sont obtenues à partir d'une station SPARCstation 2 (Sun 4/75) [5].

Création :

Le temps de création d'un processus léger est obtenu en utilisant une pile par défaut et est présent dans une zone de cache mémoire. Cette mesure inclut uniquement le temps de création et pas le changement de contexte initial.

Opérations	ms	Ratio
Création d'un processus léger non lié à un LWP (*unbound thread*)	52	-
Création d'un processus léger lié à un LWP	350	6,7
Création d'un processus lourd (*fork*)	1700	32,7

La colonne Ratio indique l'écart en pourcentage de la ligne courante par rapport à la première ligne.

Ce tableau montre que la création d'un processus lourd est plus de 30 fois plus gourmande en temps que la création d'un processus léger non lié à un LWP. Le temps de la création d'un processus léger lié (*bound*) est supérieur à un processus non lié, car sa création nécessite un appel au noyau pour créer le LWP associé.

Synchronisation :

Le temps est obtenu en utilisant la synchronisation de deux processus légers effectuant chacun une opération p et v des sémaphores.

Le programme suivant a été utilisé :

```
sema_t leSemaphore1, leSemaphore2;  /* declaration des semaphores */

/* processus léger effectuant la mesure de temps */
processus_leger1 ()
{
       …
       start_timer();
       sema_v(&leSemaphore1);
       sema_p(&leSemaphore2);
       t = end_timer();
       …
}

/* processus leger a synchroniser */
processus_leger2 ()
{
       …
       sema_p(&leSemaphore1);
       sema_v(&leSemaphore2);
       …
}
```

Opérations	ms	Ratio
Synchronisation de processus léger non lié à un LWP (*unbound thread*)	66	-
Synchronisation de processus léger lié à un LWP	390	5,9
Synchronisation entre processus lourds	200	3

11.2. Comparaison entre processus légers POSIX et Java

Le tableau ci-dessous présente les performances de synchronisation entre processus légers de POSIX et Java. Ces mesures ont été obtenues sur une *SPARCStation* 4 (110 MHz) sous *Solaris* 2.5.1. [11]

Opérations	POSIX (ms)	Java (ms)
Verrouillage/Déverrouillage d'un *mutex*	1,8	60
Tentative de verrouillage d'un *mutex*	1,3	-
Verrou de lecture/écriture	4,5	160
Sémaphore attente/réveil	4,3	56
Changement de contexte pour un processus léger non lié à un LWP (*unbounded threads*)	89	125
Changement de contexte pour un processus léger lié à un LWP (*bounded threads*)	42	-
Changement de contexte d'un processus lourd	54	-
Modification du masque d'un signal	18,1	-
Autorisation/interdiction d'une annulation	0,25	-
Création d'un processus léger non lié (*unbounded threads*)	330	1500
Création d'un processus léger lié (*bounded threads*)	720	-
Création d'un processus lourd par duplication (*fork*)	45000	-
Référence vers une variable globale	0,02	3
Référence vers les données privées	0,59	7

Les temps donnés dans ce tableau sont plus importants que les chiffres du tableau du §11.1.

Les chiffres du §11.1 sont des données constructeur et les chiffres ci-dessus sont des mesures réalisées par l'auteur du document [11]. Les mesures n'étant pas réalisées de la même manière, la différence est par conséquent normale. Ce qu'il faut surtout retenir des chiffres de ces tableaux, c'est les performances d'un processus léger par rapport à un processus lourd.

Les opérations sur les processus légers de Java nécessitent plus de temps par rapport aux processus légers du standard POSIX, car le code généré par le langage Java est un code interprété par la machine virtuelle de Java. L'interprétation d'un code consomme plus de temps qu'un code spécifique à un processeur.

Comme pour les processus légers du standard POSIX, les processus légers Java sont performants par rapport aux processus lourds.

12. Exemples de programmes

Cette partie présente quelques exemples sur les processus légers en utilisant l'interface définie par la norme POSIX.

12.1. Première approche simple

Cet exemple montre l'utilisation d'un processus léger effectuant un affichage vers la sortie standard, pendant que le fil d'exécution initial est bloqué en attente d'une saisie de l'entrée standard.

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

/*
 * Processus leger effectuant l'affichage vers la sortie standard
 */
void *writer_thread ( void *arg )
{
      sleep(5);      /* attente permettant au fil initial de se bloquer */
      printf("Processus léger -> sortie standard\n");
      return NULL;
}

/*
 * Programme principal (⇔ fil initial)
 */
int main ( int argc, char *argv[] )
{
      pthread_t      writer_id;                    /* identite du processus leger */
      char           input, buffer[255];           /* donnees pour la saisie */

      /* creation du processus leger realisant l'affichage */
      pthread_create ( &writer_id, NULL, writer_thread, NULL );

      /* realisation d'une lecture de l'entree standard : operation bloquante */
      input = fgets ( buffer, 255, stdin );

      if ( input != NULL )
            printf("Saisie de : %s\n", buffer);

      /* Terminaison du fil initial */
      pthread_exit( NULL );

      return 0;
}
```

Les paramètres passés à la fonction *pthread_create()* permettent de définir le processus léger *writer_thread()* et de récupérer l'identité de ce processus léger nouvellement créé. Les autres paramètres ayant la valeur NULL correspondent respectivement aux attributs et aux arguments du processus léger.

L'appel à la fonction *pthread_exit()* permet de terminer le fil initial *main()*.

12.2. Plusieurs processus légers

La légèreté des processus légers permettent d'en créer des milliers sans difficulté. La plupart des applications n'utilise pas un nombre aussi important de processus légers, mais cet exemple montre que cela est possible grâce aux propriétés des processus légers.

De plus, pour cet exemple, l'utilisation des processus légers est nécessaire. La version de cet exemple sans processus légers consommerait plus de ressources mémoires et processeurs.

Le programme prend en argument le nombre de processus légers à créer. Après leur création, chaque processus léger est bloqué par un *mutex*. Ce *mutex* est verrouillé avant la création de tous les processus légers pour bloquer leur exécution. Lorsque tous les processus légers sont créés, et que l'utilisateur appuie sur la touche *Entrée*, le *mutex* est déverrouillé autorisant l'exécution des processus légers. Le processus léger initial attend ensuite la terminaison de tous les processus légers.

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

/* declaration du verrou */
pthread_mutex_t verrou;

/*
 * Fonction executee par un processus leger
 */
void *processus ( void * )
{
	/*
	 * Essai de verrouillage de la variable mutex
	 * le programme principal possede le verrou
	 * le processus leger sera bloque jusqu'a la disponibilite
	 * du verrou
	 */
	pthread_mutex_lock (&verrou);

	printf("Fin de l'execution du processus leger %d\n", pthread_self());

	pthread_mutex_unlock (&verrou);

	return NULL;
}

/*
 * Programme principal
 */
int main ( int argc, char **argv )
{
	int i, nbProcessus = 100;
	char buffer;
	pthread_t	*idProcessus;

	/* l'utilsateur peut preciser le nombre de processus legers a creer */
	if ( argc == 2 )
		nbProcessus = atoi(argv[1]);

	/* allocation memoire permettant de recuperer tous les "id" des processus legers */
	idProcessus = (pthread_t *) malloc (nbProcessus * sizeof(pthread_t));

	/* Initialisation et verrouillage du mutex */
	pthread_mutex_init (&verrou, NULL);
	pthread_mutex_lock (&verrou);

	/* Creation des processus legers */
	printf("Creation de %d processus leger...\n", nbProcessus);

	for ( i = 0; i < nbProcessus; i++ )
		pthread_create ( &idProcessus[i], NULL, processus, NULL );
```

```
        printf("%d processus legers sont crees et en execution\n", i);
        printf("Appuyer sur <Entree> pour attendre la terminaison de tous les processus\n");

        /* attente de l'appui sur la touche entree */
        gets(&buffer);
        printf("Attente de la terminaison de %d processus leger\n", nb_processus);

        /* deverrouillage du mutex -> debloquage des processus legers */
        pthread_mutex_unlock (&verrou);

        /* Attente de la terminaison des processus legers */
        for ( i = 0; i < nbProcessus; i++)
                pthread_join(idProcessus[i], NULL);

        printf("Fin de l'execution de tous les processus legers\n");

        return 0;
}
```

12.3. Multiplication de matrices

La multiplication de matrices est l'exemple qui convient le mieux pour montrer le parallélisme dans les programmes.

Le programme, de cet exemple, doit calculer chaque élément de la matrice résultante. Si le programme n'utilise pas des processus légers, le temps de la multiplication de deux matrices est le temps de calcul de tous les éléments de la matrice résultante.

La performance de ce programme peut être améliorée en utilisant les processus légers. Plusieurs raisons permettent d'expliquer ce gain de performance. Le calcul de chaque élément nécessite une lecture des données (donc une opération d'entrée/sortie), ainsi l'utilisation de processus léger permet de continuer un calcul pendant qu'un autre processus léger effectue une lecture bloquante. Sur une machine multiprocesseur, chaque processus léger, effectuant le calcul d'un élément, est assigné à un processeur. Cela permet de diminuer le temps d'exécution du programme.

Les programmes présentés dans cet exemple sont la multiplication de deux matrices à deux dimensions.

La multiplication de deux matrices (a) et (b) s'effectue suivant la formule :

$$c_{ligne,colonne} = a_{ligne,1} \times b_{1,colonne} + a_{ligne,2} \times b_{2,colonne} ... + a_{ligne,n} \times b_{n,colonne}$$

Le programme ci-dessous est un exemple de multiplication de matrices sans utiliser les processus légers :

```
/*******************************************************
 * Programme de multiplication de matrices
 * (version sans processus léger)
 * (Matrice_A  X  Matrice_B) => Matrice_C
 *******************************************************/

#include <stdio.h>
#define TAILLE_TABLEAU 10

typedef int matrice_t[TAILLE_TABLEAU][TAILLE_TABLEAU];

matrice_t MA,MB,MC;

/*
 * Fonction effectuant la multiplication d'un ligne et d'une colonne pour placer
 * le resultat dans l'element de la matrice resultante
 */
void multiplication(int taille,
      int ligne,
      int colonne,
      matrice_t MA,
      matrice_t MB,
      matrice_t MC)
{
      int position;

      MC[ligne][colonne] = 0;
      for(position = 0; position < taille; position++) {
            MC[ligne][colonne] = MC[ligne][colonne] +
      ( MA[ligne][position]  *  MB[position][colonne] ) ;
      }
}
```

```
/*
 * Programme principal : Allocation des matrices, initialisation, et calcul
 */
int main(void)
{
      int     taille, ligne, colonne;

      size = TAILLE_TABLEAU;

      /* Initialisation des matrices MA et MB */
      …

      /* Calcul de la matrice resultante */
      for(ligne = 0; ligne < taille; ligne++) {
              for (colonne = 0; colonne < taille; colonne++) {
              multiplication(taille, ligne, colonne, MA, MB, MC);
              }
      }

      /* Affichage du resultat */
      printf("MATRICE: resultat de la matrice C;\n");
      for(ligne = 0; ligne < taille; ligne++) {
              for (colonne = 0; colonne < taille; colonne++) {
              printf("%5d ",MC[ligne][colonne]);
              }
              printf("\n");
      }
      return 0;
}
```

Les matrices dans ce programme sont appelées MA, MB et MC (MA x MB = MC).

La fonction "multiplication" calcule le résultat d'un élément de la matrice MC (en multipliant les lignes et colonnes des matrices MA et MB).

Pour la version avec processus légers, le programme utilise le modèle de programmation en groupe (voir §6.2), le schéma ci-dessous représente le modèle utilisé :

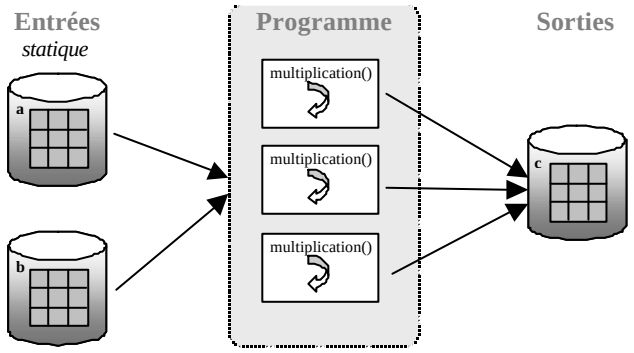


Figure 23 - *Utilisation des processus légers pour la multiplication de matrices*

Dans ce programme, le fil initial prépare la multiplication de deux matrices et crée autant de processus légers qu'il y a d'éléments à calculer dans la matrice résultante (matrice MC). Le fil initial attend ensuite que chaque processus léger ait terminé de réaliser le calcul d'un élément.

Avant la terminaison du fil initial, le résultat contenu dans la matrice MC est affiché.

Voici le code source de ce programme :

```
/*******************************************************
 * Programme de multiplication de matrices
 * (version avec processus légers)
 * (Matrice_A  X  Matrice_B) => Matrice_C
 *******************************************************/

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

#include <pthread.h>

#define TAILLE_TABLEAU 10

typedef int matrice_t[TAILLE_TABLEAU][TAILLE_TABLEAU];

typedef struct {
  int       id;
  int       taille;
  int       ligneA;
  int       colonneB;
  matrix_t  *MA, *MB, *MC;
} info_t;

matrice_t MA,MB,MC;

/*
 * Fonction effectuant la multiplication d'un ligne et d'une colonne pour placer
 * le resultat dans l'element de la matrice resultante
 * voir fonction multiplication() du source precedent
 */
 …


/*
 * Fonction de demarrage d'un processus leger travailleur.
 */
void *multiplication_worker(void *arg)
{
  info_t *p=(info_t *)arg;

  printf("Processus %d: traitement ligne A %d, colonne B %d\n",
       p->id, p->ligneA, p->colonneB );

  multiplication(p->taille, p->ligneA, p->colonneB, *(p->MA), *(p->MB), *(p->MC));

  free(p);
  printf("Processus %d: Fin\n", p->id);
  return(NULL);
}

/*
* Programme principal : Allocation des matrices, initialisation,
* creation des processus legers et calcule
*/
int main(int argc, char **argv)
{
  int       taille, ligne, colonne, num_processus, i;
  pthread_t *idProcessus;       /* Recuperation de toutes les identites des processus
                                   legers pour attendre leur terminaison. */
  info_t *p;                    /* argument pour les processus legers. */

  unsigned long taille_pile;
  pthread_attr_t *attribut, attribut_processus;


  /* Taille de la matrice */
  taille = TAILLE_TABLEAU;

  /* Allocation memoire pour recuperer les "id" des processus legers crees */
  idProcessus = (pthread_t *)malloc(taille*taille*sizeof(pthread_t));

  /* initialisation des matrices MA et MB */
  …
```

```
  /* Calcul de la matrice MC : creation des processus legers */
  num_processus = 0;
  for(ligne = 0; ligne < taille; ligne++) {
    for (colonne = 0; colonne < size; colonne++) {
      p           = (info_t *)malloc(sizeof(info_t));
      p->id       = num_processus;
      p->taille   = taille;
      p->ligneA   = ligne;
      p->colonneB = colonne;
      (p->MA)     = &MA;
      (p->MB)     = &MB;
      (p->MC)     = &MC;

      pthread_create(&idProcessus[num_processus],  /* id du processus leger */
                  NULL,                            /* attribut du processus */
                  multiplication_worker,           /* fonction */
                  (void *) p);                     /* parametres de la fonction */

      printf("Creation du processus leger %d\n", num_processus);
      num_processus++;
    }
  }

  /* Synchronisation : attente de la terminaison de tous les processus legers. */
  for (i = 0; i < (taille*taille); i++) {
       pthread_join(idProcessus[i], NULL);
       printf("Synchronisation avec processus %d.\n", i);
  }

/* Affichage du resultat */
   …

  return 0;
}
```

Dans cette version, chaque processus léger appelle la fonction *multiplication()*. Lors de la création des processus légers, les paramètres passés sont un pointeur vers une structure de données définissant le calcul à réaliser, l'identification du processus léger et les pointeurs vers les matrices.

Chaque processus léger dispose ainsi de ses propres données mais exploite en commun les matrices.

Ce programme n'a pas besoin d'utiliser des primitives de synchronisation car :

- le processus léger initial attend la terminaison de tous les autres processus pour l'affichage du résultat,
- chaque processus léger modifie uniquement l'élément de la matrice qu'il doit calculer,
- aucun risque d'incohérence des lectures des matrices sources, car ces dernières ne sont pas modifiées lors de la multiplication.

13. Conclusion

Au travers de cette étude, les caractéristiques des processus légers ont été présentées. Cela a permis de montrer la problématique à laquelle répondent les processus légers. En effet, les processus légers sont plus simples à gérer qu'un processus lourd, car ils nécessitent peu de ressources.

Le concept de "légèreté" de ces processus amène à démontrer les raisons de leurs utilisation. Après une présentation des fonctionnalités des processus légers, les différentes implantations au niveau des systèmes d'exploitation ont été invoquées avec leurs avantages et leurs inconvénients, pour ensuite aboutir aux différentes utilisations des processus légers.

Certains problèmes ont été soulevés tels que les situations d'interblocages, les inversions de priorité, afin de sensibiliser les utilisateurs des processus légers dans leur application.

L'utilisation sans cesse croissante des processus légers a amené à présenter un échantillon de modèles de programmation ainsi que des exemples simples d'utilisation des processus légers.

L'emploi des processus légers s'avère intéressant pour masquer les délais d'attente lors de l'exécution d'un programme (interaction avec l'utilisateur, entrées / sorties…).

Cependant, il ne faut pas vouloir utiliser à tout prix les processus légers dans une application. Si cette dernière est déjà simplifiée et optimisée pour son exécution, l'apport des processus légers ne fournira aucun gain en terme de performances.

Avec l'utilisation des bibliothèques de communications, les processus légers peuvent être utilisés dans des systèmes répartis (communication entre les processus légers de différents processus lourds étant situés sur des machines distantes).

L'introduction des processus légers au sein d'un processus lourd permet la mise en œuvre de parallélisme plus fin que celui des processus lourds. Ce parallélisme, à grain fin, charge moins le système et permet d'améliorer les performances des applications.

Le standard POSIX des processus légers est utilisé par la quasi-totalité des systèmes d'exploitation. Cette standardisation facilite l'écriture de programmes portables, et couvre les besoins des applications parallèles, distribuées et "classiques". Les bibliothèques de processus légers ont l'avantage d'être facilement disponibles, et leur grande diffusion a permis de les débarrasser d'un grand nombre d'erreurs (bogues ou *bug*).

Une évolution du standard 1003.1c-1995 POSIX intègre des mécanismes supplémentaires de synchronisation : verrou lecteur/rédacteur, barrières… Ce standard est actuellement connu sous la référence 1003.1j.

Les environnements intégrant les processus légers sont appelés à un bel avenir, en particulier sur des machines multiprocesseurs à mémoire commune (*Symmetric Multi Processor*), et pour les applications réalisant de puissants calculs.

Les processus légers permettent de simuler le comportement d'un système temps réel sur une station hôte au sein d'un même processus lourd (par exemple l'outil *vxSim* sur une station *Sun Solaris* pour simuler le noyau *vxWorks*).

Le couplage des bibliothèques de processus légers et de communication semble prometteur pour leur application avec un ensemble de machines interconnectées entre eux par l'intermédiaire de réseaux rapides.

14. Annexe 1 : Les systèmes d'exploitation supportant les processus légers

Dans le tableau ci-dessous figure une liste non exhaustive des processus légers supportés par les systèmes d'exploitation.

Vendeur	Version	Modèle	Interface POSIX utilisée
Digital	Digital UNIX 4.0	noyau / utilisateur	1003.1c-1995 1003.4a draft 4
	Digital UNIX 3.2	noyau	1003.4a draft 4
	Open VMS 7.0 (Alpha)	noyau / utilisateur	1003.1c-1995 1003.4a draft 4
	OpenVMS 7.0 (VAX)	utilisateur	1003.1c-1995 1003.4a draft 4
	OpenVMS 6.2	utilisateur	1003.4a draft 4
Hewlett Packard (HP)	HP/UX 11.00	noyau	1003.1c-1995
	HP/UX 10.20	utilisateur	1003.4a draft 4
	HP/UX 10.10	utilisateur	1003.4a draft 4
IBM	AIX 4.1 & 4.2	noyau	1003.4a draft 4 1003.4a draft 7
	AIX 3.5.x	utilisateur	1003.4a draft 4
	OS/2	noyau	1003.4a draft 4
Linux	Linux 1.2.13 et supérieur	noyau / utilisateur	1003.1c-1995 1003.4a draft 4
	Linux 2.x	noyau	-
Microsoft	Windows NT & 95/98	noyau (*WIN32 threads)*	1003.4a draft 4
SiliconGraphics (SGI)	Irix 6.2	noyau / utilisateur (patch)	1003.1c-1995
	Irix 6.1	noyau	-
Sun	Solaris 2.5 et supérieur	noyau / utilisateur	1003.1c-1995
	Solaris 2.4	noyau / utilisateur	1003.4a draft 8
	SunOS 4.x	utilisateur	-

15. Annexe 2 : Comparaison des processus légers POSIX, Solaris et Java

Le tableau ci-dessous représente la liste (non exhaustive) des fonctions permettant de gérer les processus légers.

Cette liste fournit les équivalences, lorsqu'elles existent, entre les processus légers POSIX, *Solaris* et Java.

Fonctions	***POSIX Pthreads***	***Solaris Threads***	***Java Threads***
Création d'un processus léger	pthread_create()	thr_create()	new Thread() t.start()
Terminaison d'un processus léger	pthread_exit()	thr_exit()	stop()
Détachement d'un processus léger	pthread_detach()	flag *THR_DETACHED* par thr_create()	
Attente de la terminaison d'un autre processus léger	pthread_join()	thr_join()	join()
Donne le contrôle à un autre processus	pthread_yield()	thr_yield()	yield()
Processus léger courant	pthread_self()	thr_self()	currentThread getName()
Envoi d'un signal	pthread_kill()	thr_kill()	interrupt()
Gestion du masque des signaux	pthread_sigmask()	thr_sigsetmask()	-
Paramètres et règles d'ordonnancement d'un processus léger	pthread_setschedparam()	thr_setprio()	setPriority()
	pthread_getschedparam()	thr_getprio()	
Gestion du degré de parallélisme	-	thr_setconcurrency()	-
	-	thr_getconcurrency()	-
Suspension d'un processus léger	-	thr_suspend()	suspend()
Reprise d'un processus léger	-	thr_continue()	resume()
Gestion des clés privées pour les données privées	pthread_key_create()	thr_keycreate()	via Subclass
	pthread_ key_delete()	-	-
Gestion des données privées	pthread_setspecific()	thr_setspecific()	var =
	pthread_getspecific()	thr_getspecific()	thread.var
Exécution unique d'une routine	pthread_once()	-	-
Comparaison de processus léger	pthread_equal()	-	-
Annulation d'un processus léger	pthread_cancel()	-	
Gestion de l'annulation d'un processus léger	pthread_testcancel()	-	stop()
	pthread_setcancelstate()	-	-
	pthread_setcanceltype()	-	-

Fonctions	***POSIX Pthreads***	***Solaris Threads***	***Java Threads***
Mise en place d'une routine de nettoyage dans la pile	pthread_cleanup_push()	-	
Retrait d'une routine de nettoyage de la pile	pthread_cleanup_pop()	-	via finally
Verrouillage d'un *mutex*	pthread_mutex_lock	mutex_lock()	synchronized
Déverrouillage d'un *mutex*	pthread_mutex_unlock()	mutex_unlock()	(implicit)
Essai de verrouillage d'un *mutex*	pthread_mutex_trylock()	mutex_trylock()	-
Initialisation d'un *mutex*	pthread_mutex_init()	mutex_init()	-
Destruction d'un *mutex*	pthread_mutex_destroy()	mutex_destroy()	-
Attente illimitée d'une variable condition (en association avec un mutex)	pthread_cond_wait()	cond_wait()	wait()
Attente limitée d'une variable condition (en association avec un mutex)	pthread_cond_timedwait()	cond_timedwait()	wait(long)
Déblocage d'un processus léger en attente sur une variable condition	pthread_cond_signal()	cond_signal()	notify()
Déblocage des processus légers en attente sur une variable condition	pthread_cond_broadcast()	cond_broadcast()	notifyAll
Initialisation d'une variable condition	pthread_cond_init()	cond_init()	-
Destruction d'une condition variable	pthread_cond_destroy()	cond_destroy()	-
Gestion d'un verrou lecture/écriture (*Readers/Writers Locks*)	-	rwlock_init()	-
	-	rwlock_destroy()	-
	-	rw_rdlock()	-
	-	rw_wrlock()	-
	-	rw_unlock()	-
	-	rw_tryrdlock()	-
	-	rw_trywrlock()	-
Gestion de sémaphores	sem_init() *POSIX 1003.4*	sema_init()	-
	sem_destroy() *POSIX 1003.4*	sema_destroy()	-
	sem_wait() *POSIX 1003.4*	sema_wait()	-
	sem_post() *POSIX 1003.4*	sema_post()	-
	sem_trywait() *POSIX 1003.4*	sema_trywait()	-
Priorité plafond d'un *mutex*	pthread_mutex_setprioceiling()	-	-
	pthread_mutex_getprioceiling()	-	-
Initialisation des attributs d'un mutex	pthread_mutexattr_init()	-	-

Fonctions	***POSIX Pthreads***	***Solaris Threads***	***Java Threads***
Destruction des attributs d'un *mutex*	pthread_mutexattr_destroy()	-	-
Gestion des attributs de partage d'un mutex	pthread_mutexattr_setpshared()	paramètre *type* dans cond_init()	-
	pthread_mutexattr_getpshared()	-	-
Gestion des attributs de la priorité plafond d'un mutex	pthread_mutexattr_setprioceiling()	-	-
	pthread_mutexattr_getprioceiling()	-	-
Gestion des attributs de protocole d'un *mutex*	pthread_mutexattr_setprotocol()	-	-
	pthread_mutexattr_getprotocol()	-	-
Initialisation des attributs d'une condition variable	pthread_condattr_init()	-	-
Destruction des attributs d'une condition variable	pthread_condattr_destroy()	-	-
Attribut d'une condition variable (lecture, modification)	pthread_condattr_getpshared()	-	-
	pthread_condattr_setpshared()	-	-
Initialisation des attributs d'un processus léger	pthread_attr_init()	-	-
Destruction des attributs d'un processus léger	pthread_attr_destroy()	-	-
Paramètres de détachement d'un processus léger	pthread_attr_getdetachstate()	-	-
	pthread_attr_setdetachstate()	-	-
Paramètres d'ordonnancement (héritage des propriétés, comportement, règles…)	pthread_attr_getinheritsched()	-	-
	pthread_attr_setinheritsched()	-	-
	pthread_attr_gestschedparam()	-	-
	pthread_attr_sestschedparam()		
	pthread_attr_gestschedpolicy()		
	pthread_attr_sestschedpolicy()	-	-
	pthread_attr_getscope()	-	-
	pthread_attr_setscope()	*THR_BOUND flag* dans thr_create()	
Paramètres de la pile (taille, adresse)	pthread_attr_getstacksize()	-	-
	pthread_attr_setstacksize()	paramètre *stack_size* dans thr_create()	
	pthread_attr_getstackaddr()	-	-
	pthread_attr_setstackaddr()	-	-

Le signe "-" indique l'indisponibilité de la fonction dans l'une des interfaces. Les lignes "sem_" suivies de "POSIX 1003.4" font partie du standard temps réel de POSIX et ne font pas partie de l'interface *pthreads*.

Certaines fonctions manquantes peuvent être fabriquées en combinant plusieurs fonctions.

Une description détaillée des fonctions citées dans le tableau ci-dessus est fournie dans les documents [5], [10], [11], [14] et [15].

16. Annexe 3 : Les processus légers de Windows 95/98/NT (*WIN 32 Threads*)

Le tableau suivant présente les différentes fonctions permettant de gérer les processus légers sous Windows 95/98 et Windows NT [17]. Ces fonctions n'ont pas été comparées aux processus légers de l'annexe précédente car leurs utilisations et les paramètres sont différents.

Fonctions	***WIN 32 Thread***
Création d'un processus léger	CreateThread()
Identification d'un processus léger	GetCurrentThread() GetCurrentThreadId()
Priorité d'un processus léger	SetThreadPriority() GetThreadPriority()
Suspension d'un processus léger	SuspendThread()
Reprise de l'exécution d'un processus léger	ResumeThread()
Terminaison normale d'un processus léger	ExitThread()
Terminaison anormale d'un processus léger	TerminateThread()
Code de sortie d'un processus léger	GetExitCodeThread()
Information d'exécution d'un processus léger	GetThreadTimes()
Gestion des sémaphores	CreateSemaphore() OpenSemaphore() ReleaseSemaphore()
Gestion des événements	CreateEvent() OpenEvent() SetEvent() ResetEvent() PulseEvent()
Exclusion mutuelle	InterlockedIncrement() InterlockedDecrement() InterlockedExchange()
Gestion de mutex	CreateMutex() OpenMutex() ReleaseMutex()
Attente sur un objet (mutex, sémaphore)	WaitForSingleObject() WaitForMultipleObject()
Gestion de sections critiques	InitializeCriticalSection() DeleteCriticalSection() EnterCriticalSection() LeaveCriticalSection()
Gestion des messages	PostMessage() PostThreadMessage() SendMessage() SendMessageTimeOut() ReplyMessage() GetMessage() WaitMessage() DispatchMessage()

17. Annexe 4 : Terminologie et sigles

DCE	*Distributed Computing Environment*
FAQ	*Frequently Asked Questions* (Foire Aux Questions ou Questions Fréquemment posées)
FIFO	*First In, First Out*
GNU	*Gnu is Not Unix*
IEEE	*Institute of Electrical and Electronics Engineers*
IPC	*Inter-Processes Communications*
LWP	*Light Weight Processes*
OSF	*Open Software Foundation*
PASC	*Portable Application Standards Committee*
POSIX	*Portable Operating System Interface for computer environments*
RPC	*Remote Procedure Call*
SMP	*Symmetric Multi Processor*
tas	mémoire utilisateur
UI	*Unix International*

18. Annexe 5 : Références documentaires

18.1. Documents de références

[1]	Isabelle Demeure et Jocelyne Farhat. *Systèmes de processus légers : concepts et exemples*. Thèse n°94D024 de l'Ecole Nationale Supérieure des Télécommunications, Paris, novembre 1994.
[2]	Andrew Tannenbaum. *Systèmes d'exploitation – Systèmes centralisés. Systèmes distribués.* InterEdition, Paris. Prentice Hall, London 1994.
[3]	Ilan Ginzburg. *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. PhD thesis, Laboratoire de Modélisation et de Calcul (LMC) de Grenoble, 1997.
[4]	Raymond Namyst. *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille, 1997
[5]	Sun Microsystems. Inc. *Multithread Programming Guide*. August 1994
[6]	Tom Wagner, Don Towsley. *Getting Started with POSIX Threads*. Department of Computer Science, University of Massachussets at Amherst, July 1995.
[7]	Andrew D. Birrel. *An introduction to Programming with Threads.* Digital Equipment Corporation, 1989.
[8]	Sun Microsystems Inc. *SunOs Multi-thread Architecture.* USENIX – Winter'91 – Dallas, Texas
[9]	J.-M. Rifflet, *La programmation sous UNIX*, Ediscience, 1993.
[10]	B. Nichols, D. Buttlar & J. Proulx Fareell. *Pthreads programming*. O'Reilly 1998.
[11]	Bil Lewis. *Multithread Programming, Concepts and Practice*. Transparents du séminaire du 5 octobre 1998.
[12]	Sun Microsystems Inc. *Programming Utilities & Libraries*, 1990. Chapitre 2*, Lightweight Processes*, pages 17 à 51.
[13]	F. Mueller, E. Giering, T. Baker. *Implementing ada 9x features using posix threads: Design issues*. Draft, 1993.
[14]	Sun Microsystems Inc. *pthreads and Solaris threads : A comparison of two user level threads APIs. Early Access Edition (May 1994); pthreads Based on POSIX 1003.4a / D8.*
[15]	Sun Microsystems Inc. Daniel J. Berg. *Java Threads. A Whitepaper*. Mai 1996
[16]	B. Zignin. *Techniques du multithread, du parallélisme dans les processus*. Collection synthèse informatiques du CNAM, Edition HERMES.
[17]	Nabil Cherifi. *Windows NT, Programmation 32 bits*. Edition Armand Colin, juin 1994.
[18]	F. Mueller. *A library Implementation of POSIX Threads under UNIX.* USENIX - Winter'93 - San Diego, CA.

18.2. Sites Internet

Les différents sites Internet énumérés ci-dessous ont permis d'obtenir un exemplaire électronique de certains articles cités dans le paragraphe précédent. Ces références permettent également d'obtenir de plus amples informations sur les processus légers.

http://www.enst.fr/~demeure *http://www.enst.fr/~farhat*	Pages personnelles d'Isabelle Demeure et de Jocelyne Farhat rédactrices de l'article [1] obtenu par e-mail sur simple demande.
http://www.sun.com/workshop/threads	Pages comportant des liens sur la documentation sur les processus légers.
http://www.humanfactor.com/pthreads/pthread-tutorials.html	Une autres pages comportant plusieurs liens vers la documentation sur les processus légers du standard POSIX
http://www.ens-lyon.fr/~rnamyst/pm2.html	Site Internet de l'auteur de l'article [4], on y trouve des bibliothèques et des documents concernant l'utilisation des processus légers dont : R. Namyst et J.-F. Méhaut. *Marcel : Une bibliothèque de processus légers* LIFL, Lille, 1995 R. Namyst et J.-F. Méhaut. *Madelaine : Une interface de communication efficace pour les environnements multithreads.* RenPar'10 – Strasbourg, juin 1998. R. Namyst et J.-F. Méhaut. *PM^2 : Parallel Multithreaded Machine – Guide d'utilisation, version 1*.
http://pauillac.inria.fr/~xleroy/linuxthreads/ *http://linas.org/linux/threads-faq.html*	Les threads et le système d'exploitation Linux : *The LinuxThreads library* *Linux Threads Frequently Asked Questions*
http://www.LambdaCS.com/	*Multithreaded Programming Education by Lambda Computer Science*. Ce site comporte de nombreux documents et des exemples sur les processus légers. Le document [11] provient de ce site.
newsgroup : comp.programming.threads *http://www.serpentine.com/~bos/threads-faq/*	Forum électronique sur les *threads*, ainsi que les réponses des questions fréquemment posées : *Answer to frequently asked question for comp.programming.threads*

19. Annexe 6 : A propos de ce document…

19.1. Sujet de probatoire

Ce présent document a été établi selon le sujet suivant :

Sujet n°49	
Titre	Processus léger – Problématique à laquelle répondent les processus légers, définition et utilisation
Bibliographie	*"Camelot and Avalon"*, De Eppinger, Mummert et Spector, Ed. Morgan Kaufman Documentation sur les THREAD Unix TSI Vol. 13, N°6 1994, Isabelle Demeure et Jocelyne Farhat *"Systèmes de processus légers : concepts et exemples"*

19.2. Version électronique

La version électronique de ce document, au format *postscript* ou *Word 97*, est disponible à l'adresse Internet suivante :

http://perso.wanadoo.fr/patrick.deiber/pub/

20. Glossaire

Cache : mémoire très rapide entre la mémoire centrale et le processeur. L'information est d'abord recherchée dans la cache. L'accès à la mémoire cache est moins coûteux qu'un accès à la mémoire centrale.

Changement de contexte (ou commutation de contexte) : le fait de passer d'un contexte à un autre. Le changement de contexte engendre la sauvegarde du contenu de la mémoire et le chargement avec d'autres valeurs.

Contexte d'exécution : environnement d'exécution d'un processus (registres, zones mémoires…)

Compteur ordinal : contient l'adresse de la prochaine instruction à exécuter.

Fil de contrôle (*thread of control*) : Un fil de contrôle est une exécution d'une portion de code séquentiel.

Fil d'exécution : voir Fil de contrôle.

Flot d'exécution : voir Fil de contrôle.

***Fork* :** opération permettant la duplication d'un processus.

Interblocage (*Deadlock*) : phénomène qui se produit lorsque deux processus (traditionnels ou légers) s'empêchent mutuellement d'accéder à une ressource d'exécution.

Intervalle de temps (*time-slice*) : intervalle de temps pendant lequel la ressource d'exécution est attribuée à une entité d'exécution.

***Inter-Processes Communication (IPC)* :** La couche IPC offre plusieurs mécanismes de communication entre processus UNIX d'une même machine. Les primitives d'obtention des ressources IPC utilisent une clé pour créer une ressource ou pour y accéder. Les IPC sont de trois types : sémaphores, messages et mémoire partagée.

***Join* :** opération permettant à un processus léger d'attendre la terminaison d'autres processus.

LWP (*Light Weight Process*) : processus léger. Terme servant à désigner les processus légers sur *Solaris* et *SunOs*. Attention, les LWP entre *Solaris* et *SunOs* sont fondamentalement différents.

Machine multiprocesseur *(Symmetric Multi Processor*) : C'est une machine dans laquelle tous les processeurs ont un même accès à la mémoire (donc partagée).

***Multithread* :** exécution de plusieurs processus légers (*thread*) en parallèle ou en concurrence.

***Mutex* :** verrou permettant de réaliser une exclusion mutuelle. Un *mutex* peut être assimilé à un sémaphore binaire (pouvant prendre la valeur 0 ou 1).

Ordonnancement (*scheduling*) : mécanisme de gestion des ressources d'exécution entre plusieurs entités exécutables.

Parallélisme : notion d'exécution de plusieurs entités exécutables en même temps.

***Pipeline* :** Le principe du pipeline est comparable à celui du travail à la chaîne : il consiste à décomposer l'exécution d'un programme en phases élémentaires de longueur fixe égale au temps de cycle (la phase la plus lente détermine le temps de cycle). Les phases du pipeline sont appelées étages et chaque étage est géré par une unité fonctionnelle donnée. Ainsi l'exécution de plusieurs instructions se recouvre et le débit d'exécution est plus important.

Préemption : "pouvoir" que possède une entité d'exécution d'interrompre l'exécution d'une entité de priorité plus faible. La préemption est différente de l'intervalle de temps (*time-slice*).

POSIX : groupe de travail sur la définition de standard en matière de système d'exploitation.

Processus (*process*) : souvent définit comme "un programme en exécution", parfois appelé tâche.

Processus "lourd" ou processus "traditionnel" : C'est une unité d'exécution qui comprend un fil de contrôle et un contexte mémoire dans lequel le fil de contrôle est seul à s'exécuter. Ce contexte comprend le code et les données du programme en exécution, la pile, ainsi qu'un contexte d'exécution contenant les informations telles que l'état du processus, du compteur ordinal et des autres registres du processeur, des informations sur la gestion de la mémoire, et des informations statistiques.

***Remote Procedure Calls (RPC)* :** Les RPC, appels de procédures à distance, permettent aux programmes d'appeler des procédures situées sur d'autres machines. Le but des RPC est de cacher au client le fait que l'appel va s'exécuter à distance.

Sémaphore : Permet de gérer l'entrée en section critique. Un sémaphore (*semaphore*) est une structure de données composée d'un compteur entier et d'une file d'attente gérée en FIFO. L'utilisation d'un sémaphore se fait par les opérations P (décrémentant le compteur) et V (incrémentant le compteur). L'opération P permet de demander l'accès à un élément de ressources (demande de ticket). L'opération V permet de libérer un élément de ressource (restitution du ticket).

***Solaris* :** système d'exploitation (type UNIX) développé par *Sun Microsystems* et utilisé sur leurs serveurs et stations de travail. Une version de ce système d'exploitation existe pour PC (*Sun Microsystems* propose une version de leur système d'exploitation gratuitement aux particuliers).

***Sockets* :** Les *sockets* permettent à deux processus de communiquer de manière bidirectionnelle sur deux machines différentes. Il existe deux modes d'utilisation des *sockets* : le mode non connecté où l'émetteur doit toujours spécifier l'adresse du destinataire (la connexion n'est pas continue), et le mode connecté où la connexion est tout le temps établie.

Temps de commutation : temps nécessaire à la réalisation d'un changement de contexte.

Tubes (*pipes*) : Les tubes sont un moyen de faire communiquer deux processus UNIX situés sur la même machine. Ce mécanisme crée un espace mémoire partagée. Les tubes fonctionnent de manière unidirectionnelle. Le mécanisme de tube peut être également utilisé avec les processus légers. L'accès au tube doit être protégé, si ce dernier est utilisé par plusieurs processus légers. Dans un contexte réparti où deux processus sont sur deux machines différentes, les tubes ne peuvent pas être utilisés. Les *sockets* résolvent ce problème.

Unité d'exécution : C'est une entité qui peut recevoir le contrôle de l'unité centrale pendant un certain temps pour exécuter un programme. L'unité d'exécution traditionnellement manipulée par les systèmes d'exploitation est le processus. On utilise également le terme **entité noyau** (*kernel entity*) pour désigner l'unité d'exécution.

Variable condition : élément de synchronisation constitué d'une file d'attente gérée en FIFO ou avec prise en compte de priorités.