

1 Gestion Mémoire

On considère un système de pagination à deux niveaux :

- les adresses (virtuelles et physiques) sont codées sur 32 bits ;
- les n_1 premiers bits d'une adresse virtuelle forment le premier index, les n_2 bits suivants forment le second index ($n_1 + n_2 = 20$), et les 12 bits restant forment le déplacement ;
- on suppose que chacune des entrées de ces tables occupe 32 bits.

Question 1 Discutez des avantages/inconvénients d'une configuration où n_1 serait très petit par rapport à n_2 . Même question dans le cas contraire.

Correction Si n_1 est très petit, alors la table de premier niveau contient peu d'entrées, au contraire des tables de second niveau qui sont grandes. Le problème posé par cette configuration est qu'il sera peu probable d'économiser de la place en évitant l'allocation d'une table de second niveau, puisqu'il faudrait pour cela qu'un processus possède un «trou» dans son espace d'adressage de 2^{n_2+12} octets.

D'un autre côté, si n_2 est vraiment trop petit, on va certes économiser de la place en allouant peu de tables de second niveau pour les processus qui possèdent des trous, mais la table de premier niveau (toujours allouée) sera plus imposante...

Question 2 Donnez une condition sur l'adresse de début et la taille des «trous» dans l'espace d'adressage d'un processus pour que l'économie mémoire faite en utilisant ce tableau à deux niveaux soit maximale. Expliquez à l'aide d'un dessin.

En fixant $n_1 = n_2 = 10$, chiffrez l'économie mémoire réalisée par rapport à une table des pages à un seul niveau dans le cas d'un processus ayant un seul trou d'un gigaoctet (2^{30} octets) dans son espace d'adressage.

Correction Pour que l'économie soit maximale, il faut qu'un «trou» coïncide avec l'ensemble des adresses couvertes par une table de second niveau. Cela concerne les trous qui commencent à une **adresse multiple de 2^{n_2+12}** et qui ont donc une **taille multiple de 2^{n_2+12}** .

En supposant $n_1 = n_2 = 10$, un trou d'un gigaoctet (2^{30} octets) occupe $2^8 = 256$ tables de second niveau (qui couvrent chacune 2^{22} octets = 4Mo) s'il est bien aligné, puisque $2^{30} = 2^8 \times 2^{22}$. Comme une table de second niveau occupe 2^{10} entrées de 4 octets chacune (ce qui fait donc 4Ko par table), on économise $256 \times 4\text{Ko} = 1\text{Mo}$. Si on veut mesurer l'économie par rapport à une table à un seul niveau, il faut décompter l'espace occupé par la table de premier niveau, qui «pèse» 4Ko. Au final, l'économie est donc de 1Mo – 4Ko.

2 Synchronisations entre threads

On souhaite implanter certaines synchronisations internes de la bibliothèque «Pthreads» à l'aide de sémaphores. Dans cet exercice, on s'appuiera sur la disponibilité des définitions suivantes :

```
typedef ... sem_t ;

void sem_init(sem_t *s, int value);
void P(sem_t *s);
void V(sem_t *s);
```

Concrètement, il s'agit dans un premier temps de réaliser la partie synchronisante des fonctions `pthread_exit` et `pthread_join`. Pour rappel, `pthread_join(pid)` est une fonction qui bloque le thread appelant tant que le thread `pid` ne s'est pas terminé, et qui permet de récupérer son code de retour (de type `void *`). Lorsqu'un thread «non détaché»¹ se termine, il doit attendre (à l'intérieur de `pthread_exit`) qu'un thread

¹C'est-à-dire dont la terminaison devra être récupérée par `pthread_join`.

appelle `pthread_join` avant de pouvoir disparaître. Dans cet exercice, on ne se préoccupera pas de la gestion des cas d'erreur (par exemple deux threads qui appelleraient `pthread_join` sur un même thread).

Dans les systèmes conformes à la norme Pthreads, un *thread* est manipulé par l'intermédiaire d'un objet de type `pthread_t`. Dans notre implémentation, cet objet est un pointeur sur un descripteur `pthread_desc_t` dans lequel sont stockées toutes les données propres au *thread*. Ce descripteur est donc l'équivalent de la classe `Thread` dans Nachos...

Les définitions de type ainsi que les squelettes des fonctions `pthread_create`, `pthread_exit` et `pthread_join` sont donnés en figure 1.

<pre>typedef struct { int detached; ... } pthread_desc_t; typedef pthread_desc_t *pthread_t; void pthread_create(pthread_t *pid, ...) { // Allocation et initialisation du descripteur *pid = (pthread_desc_t *)alloc_desc(...); (*pid)->detached = ...; ... // Initialisation des variables de synchro __pthread_init(*pid); // Lancement du thread ... }</pre>	<pre>void pthread_exit(void *status) { if(!pthread_self()->detached) { __pthread_exit(status); } // Destruction des ressources // (pile, descripteur, etc.) ... } void pthread_join(pthread_t pid, void **status) { if(pid->detached) { // Erreur ! ... } else { void *r = __pthread_join(pid); if(status != NULL) *status = r; } }</pre>
---	--

FIG. 1 – Squelette des fonctions Pthreads

Question 1 Pour réaliser la synchronisation nécessaire au sein de `pthread_join` et `pthread_exit`, il suffit de fournir le code des fonctions suivantes :

- `void __pthread_init(pthread_t pid);`
- `void __pthread_exit(void *status);`
- `void *__pthread_join(pthread_t pid);`

N'oubliez pas de préciser les champs que vous aurez besoin d'ajouter dans la structure `pthread_desc_t`.

Correction

```

typedef struct {
    int detached;
    sem_t child, father;
    void *status;
} pthread_desc_t;

void __pthread_init(pthread_t pid)
{
    sem_init(&pid->child, 0);
    sem_init(&pid->father, 0);
}

void __pthread_exit(void *status)
{
    pthread_self()->status = status;
    V(&pthread_self()->father);

    // Attention! Il faut attendre que le père ait récupéré
    // le résultat avant de se suicider
    P(&pthread_self()->child);
}

void *__pthread_join(pthread_t pid)
{
    void *status;

    P(&pid->father);
    status = pid->status;
    V(&pid->child);

    return status;
}

```

Question 2 On considère maintenant que le programme principal se termine toujours par un appel à `pthread_end(void)` (dans la fonction `main`) qui doit être bloquant tant que certains *threads* s'exécutent encore au sein du processus (y-compris les *threads* détachés).

Ecrivez le code de la fonction `pthread_end` et donnez également la nouvelle version des fonctions `__pthread_init` et `pthread_exit`. Vous aurez également besoin de déclarer quelques variables supplémentaires...

Correction

```

sem_t mutex; // initialisé à 1
sem_t wait_end; // initialisé à 0
int nb_threads; // initialisé à 0

void __pthread_init(pthread_t pid)
{
    sem_init(&pid->child, 0);
    sem_init(&pid->father, 0);

    P(&mutex);
    nb_threads++;
    V(&mutex);
}

void __pthread_exit(void *status)
{
    pthread_self()->status = status;
    V(&pthread_self()->father);
    P(&pthread_self()->child);

    // Il faut signaler qu'il y a un thread de moins
    P(&mutex);
    nb_threads--;
    V(&mutex);

    V(&wait_end);
}

void pthread_end(void)
{
    while(1) {
        P(&mutex);
        if (nb_threads == 0) {
            V(&mutex);
            return;
        }
        V(&mutex);
        P(&wait_end);
    }
}

```

Question 3 L'interface *Pthreads* propose une primitive `pthread_cancel(pid)` permettant de provoquer la terminaison d'un thread (similaire à `kill` pour les processus).

Que risque-t-il de se passer si on tue brutalement un thread non détaché? Expliquez comment² il faudrait implémenter la primitive `pthread_cancel` pour que `pthread_join` renvoie quand même un résultat dans cette situation?

Correction Si on tue brutalement un thread non détaché, il disparaîtra et non seulement il ne pourra jamais débloquent un éventuel processus qui tenterait un `pthread_join` sur lui, mais en plus son descripteur lui-même ne serait plus valide (donc impossible d'accéder aux champs `pid->...`).

Pour implanter proprement `pthread_cancel`, une solution serait d'envoyer un signal au thread victime pour le forcer à exécuter `pthread_exit`. Ainsi, le thread ne disparaîtra pas s'il n'est pas détaché : il attendra

²On ne demande pas d'écrire du code

(sur `P(&pthread->self()->child)` que quelqu'un le rejoigne au moyen de `pthread_join`. Ce sera un thread «*zombie*» en quelque sorte...