Programmation parallèle

Enjeu

- Certaines applications peuvent avoir besoin d'effectuer plusieurs traitements en parallèle. Elles peuvent être constituées de plusieurs processus, bénéficiant ainsi:
  - de l'ordonnancement du SE (multi-cœurs),
  - de l'arbitrage des ressources par le SE.
- Mais cette solution présente aussi des inconvénients :
  - obligation de passer par le SE pour communiquer (entrées-sorties, ...),
  - surcoût en temps lié à la commutation de contexte

Inconvénients des solutions multiprocessus

- Les inconvénients identifiés viennent du fait que les processus séparés sont la norme, et que la communication inter-processus est l"exception.
- Solution satisfaisante lorsque les tâches de l'application ont un couplage faible : les besoins d'arbitrage sont plus important que les besoins de communication.
- Le problème est qu'en parallélisme, on a souvent besoin de couplage fort : les besoins de communication sont très importants

Thread

- Définition : un thread (ou fil d'exécution) est l'exécution d'une procédure/fonction en parallèle de la fonction principale d'un processus
- Un processus comporte un ou plusieurs threads.

Plus souple que les processus

Le PCB du processus contient pour chaque thread:

- un état
- un contexte processeur
- En revanche, toutes les autres ressources (mémoire, fichiers, etc…) sont partagées par tous les threads du processus;
  c'est au programmeur de les arbitrer

Processus léger

- Les threads sont souvent appelés « processus légers ».
- Ils bénéficient des mêmes avantages en terme de parallélisme, que les processus (l'ordonnanceur considère chaque thread individuellement).
- Avantages par rapport aux processus :
  - la commutation de contexte est plus rapide (même PCB),
  - la communication entre eux est plus rapide (pas besoin d'appel système).

pthread_create()

- int pthread_create(pthread_t* p_tid, pthread_attr_t* attr, void *(*fonction)(void arg), void* arg);
- Le paramètre p_tid spécifie le numéro de l'activité. Elle est de type pthread_t (un entier).
- Le paramètre attr est un paramètre qui définit les attributs de l'activité. On passera en
- générale le pointeur NULL qui donne à l'activité les attributs standards.
- Le paramètre fonction est un pointeur sur la fonction que va exécuter le thread.
- Le paramètre arg est un pointeur sur les arguments de la fonction.
- Valeur renvoyée: retourne 0 si la création a réussi ou -1 en cas d 'erreur

Exemple

- pthread_t pthread_id[3];

```
void f_thread()
{printf( « Je suis le thread d'identite %d\n » ,getpid());}
int main()
{int i;
for(i=0;i<3;i++)
    pthread_create(pthread_id+i,NULL, (void*)
    f_thread,NULL)    ;
printf( « Je suis le processus initial de pid: %d\n »
,getpid());
sleep(3); //permet d'attendre la terminaison de tous les
threads
exit(1);
}
```

Terminaison d'un thread : pthread_exit()

int pthread_exit (void* p_status);
Cette fonction termine le thread.
Le paramètre p_status correspond à un pointeur sur le code de retour du thread.
Valeur renvoyée: retourne 0 en cas de réussite ou -1 en cas d ’erreur.

Synchronisation des threads

- int pthread_join (pthread_t tid, void** status);
- Cette fonction permet à une activité d’attendre la terminaison d’une autre activité et de récupérer le code de retour de cette activité (définit par l’appel pthread_exit()).
- Le paramètre tid est l’identité de l’activité pour laquelle on attend la terminaison.
- Le paramètre status est un pointeur sur un pointeur du code de retour. Attention, ce fonctionnement correspond à un retour effectué par pthread_exit().

Exemple synchronisé

```
pthread_t pthread_id[3];

void f_thread()

{printf( « Bonjour, je suis un thread);
int r=0;
pthread_exit(....);
}

int main()

{int i;
for(i=0;i<3;i++)

    pthread_create(&pthread_id[i],NULL, (void*) f_thread,NULL)   ;

pthread_join(pthread_id[0],....);
pthread_join(pthread_id[1], ...);
pthread_join(pthread_id[2], ....);

}
```


Attention

```
pthread_t pthread_id[100];
int a=0;

void go()

{a=a+1;
pthread_exit(....);
}

int main()

{int i;
for(i=0;i<100;i++)

    pthread_create(&pthread_id[i],NULL, (void*)go,NULL) ;

for(i=0;i<100;i++)pthread_join(pthread_id[i],....);

}
```

pthread_mutex_init()

- pthread_mutex_init()
- int pthread_mutex_init (pthread_mutex_t* p_mutex, pthread_mutexattr_t attr);
- Cette fonction initialise un mutex.
- Le paramètre p_mutex est un pointeur sur le mutex.
- Le paramètre attr indique les attributs du mutex.

pthread_mutex_lock

- int pthread_mutex_lock (pthread_mutex_t* p_mutex);
- Cette fonction verrouille un mutex. Si le mutex a déjà été verrouillé par une autre activité,

  la fonction se met en attente jusqu'au déverrouillage par l'autre activité.
- Le paramètre p_mutex est un pointeur sur le mutex.

pthread_mutex_unlock

- int pthread_mutex_unlock (pthread_mutex_t* p_mutex);
- Cette fonction déverrouille un mutex.
- Le paramètre p_mutex est un pointeur sur le mutex. Le mutex doit être initialisé au

Exemple final

```
pthread_t pthread_id[100];
int a=0;
pthread_mutex_t m;
void go()

{pthread_mutex_lock(&m);
a=a+1;
pthread_mutex_unlock(&m);
pthread_exit(....);
}

int main()

{int i;
pthread_mutex_init(&m,...)
for(i=0;i<100;i++)

    pthread_create(&pthread_id[i],NULL, (void*)go,NULL) ;

for(i=0;i<100;i++)pthread_join(pthread_id[i],....);

}
```

Green thread

- Dans certains contextes, les threads ne sont pas gérés par le noyau, mais émulés par un programme (par exemple la VM Java),
-
- qui s'exécute en mode utilisateur (d'où l'appellation thread utilisateur, qu'on oppose aux threads noyaux) ;
- et économise le temps de passer par le noyau (d'où l'appellation green thread).

Conclusion

- Il existe de multiples façon de programmer des tâches parallèles.
- processus lourds

  → couplage faible
- processus légers (thread noyau)

  → couplage fort, parallélisme réel