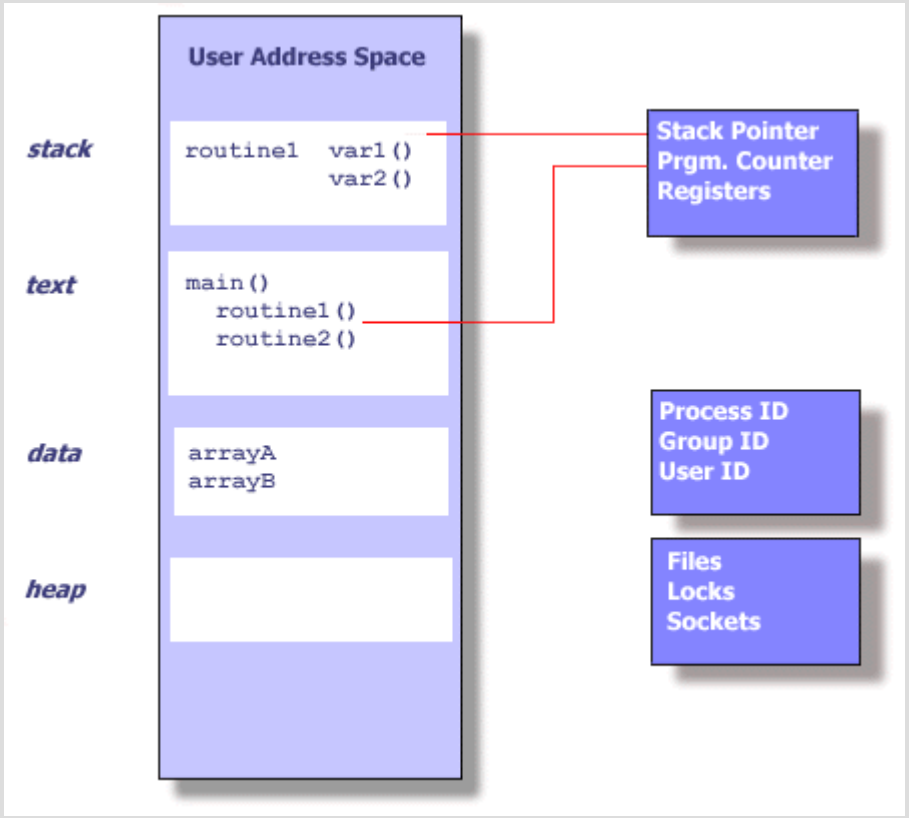
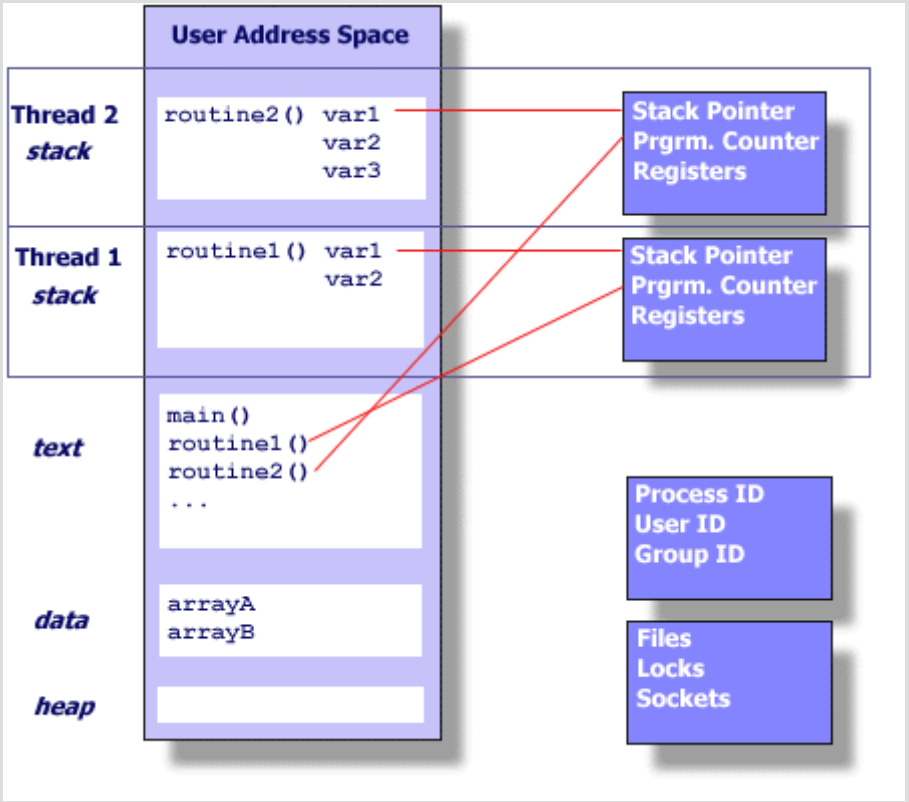
Threads POSIX

Threads

thread = « processus léger »



Processus



Processus ayant deux threads

Création

```
int pthread_create (
        pthread_t *tid, pthread_attr_t *attr,
        void* (*func)(void*), void* arg );
```

tid identifiant du thread créé

attr attributs (voir `man pthread_attr_init`)

Ex : joignable / détaché, politique d'ordonnancement, etc...
si NULL attributs par défaut

func fonction à appeler

arg argument de la fonction appelée (un pointeur)

Retour : code erreur (0 si succès)

Autres fonctions

`pthread_exit(void *retval);` → fin d'un thread

`pthread_join(pthread_t thread, void **retval);`

→ attend la fin d'un thread (correspond à `wait()`)

`pthread_cancel(pthread_t thread);`

→ demande à un thread de mourir (correspond à `kill()`)

```
pthread_ttid;
...
pthread_create (&tid, ...);
...
pthread_cancel (tid);
pthread_join (&tid, NULL);
```

Arrêt

- Chaque thread définit sa réaction aux demandes d'arrêt, selon deux paramètres :
  - accepte-t-il les demandes d'arrêt ? (cancel_state)
  - quand exactement s'arrêtera-t-il ? (cancel_type)
- Ces deux paramètres sont définis par les fonctions suivantes

Arrêt

```
int pthread_setcancelstate (int new, int *old);
```

new représente l’état du thread appelant

- **PTHREAD_CANCEL_ENABLE** : (par défaut) le thread accepte les demandes d’arrêt
- **PTHREAD_CANCEL_DISABLE** : le thread refuse les demandes d’arrêt (elles sont alors simplement ignorées)

old, s’il n’est pas NULL, reçoit l'état précédent.

Arrêt

```
int pthread_setcanceltype (int new, int *old);
```

new représente le type d’arrêt souhaité

- **PTHREAD_CANCEL_ASYNCHRONOUS** : l’arrêt a lieu immédiatement lorsqu'il est demandé
- **PTHREAD_CANCEL_DEFERRED** : (par défaut) l’arrêt a lieu au prochain *point d’arrêt* (fonction bloquante ou appel système)

old, s’il n’est pas NULL, reçoit le précédent type.

Mutex : création/destruction

Création d'un mutex :

```
int pthread_mutex_init (pthread_mutex_t* mutex,
        pthread_mutexattr_t *attr);
```

mutex le sémaphore à initialiser

attr les attributs

Destruction :

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

lock/unlock

L’usage classique des mutex est le suivant :

```
pthread_mutex mutex; // variable partagée
pthread_mutex_init (&mutex, NULL)
...
pthread_mutex_lock (&mutex);
// Section critique ici
pthread_mutex_unlock (&mutex);
...
```

trylock

Demander un mutex s’il est disponible sans rester bloqué en attendant qu'il le devienne :

```
if (pthread_mutex_trylock(&mutex) == 0)
{
   // Section critique ici
}
else
{
   // Autre chose, non critique ici
}
```

Variables conditions

- Mutex pour :
  - protéger section critique
  - Synchronisation
- Séparation des 2 rôles avec les variables conditions :
  - Mutex : section critique
  - Var. Condition : synchronisation

Variables conditions

Thread A

```
pthread_mutex_lock(&mutex);
// traitement
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Thread B

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond, &mutex);
// traitement
pthread_mutex_unlock(&mutex);
```

Variables conditions

- Type : pthread_cond_t cond ;
- Fonctions :
  - pthread_cond_init
  - pthread_cond_signal
  - pthread_cond_wait

Implémentation des threads

Contrairement aux processus, les threads partagent :

- la table des descripteurs (pas de copie)
- l'espace d'adressage (mais chacun possède sa propre pile dans cet espace)
- les signaux
- les variables d'environnement (répertoire courant, masque, etc.)

→ la création de thread est rapide.

La commutation d'un thread à l'autre est rapide car il n'y a pas de changement des registres de la MMU.