Cours 10 - PTHREADS

Partie 2

Luciana Arantes
Dec. 2005

Attributs des Threads (1)

- **Chaque thread possède un nombre d'attributs regroupé dans le type *pthread_attr_t*.**
  - Chaque attribut possède une valeur par défaut.
  - Attributs fixés lors de la création de la thread.
    - Paramètre du type *pthread_attr_t* de la fonction *pthread_create ()*.
      - NULL : attributs auront les valeurs par défaut.
    - Possibilité de changer dynamiquement les attributs.

Attributs des Threads (2)

- **Fonction pour créer une variable du type *pthread_attr_t :***

  int pthread_attr_init (pthread_attr_t * attributs);

  - Attributs initialisés avec les valeurs par défaut.
- **Fonction pour détruire une variable du type *pthread_attr_t :***

  int pthread_attr_destroy (pthread_attr_t * attributs);
- **Fonctions pour obtenir et modifier respectivement la valeur d'un attribut d'une variable du type *pthread_attr_t :***

  int pthread_attr_get**nom** (pthread_attr_t * attributs,...);

  int pthread_attr_set**nom** (pthread_attr_t * attributs,…);

  - **nom** : *nom* de l'attribut

Attributs des Threads (3)

- **Detachstate**
  - Thread joignable ou détachée :
    - PTHREAD_CREATE_JOINABLE (valeur par défaut)
    - PTHREAD_CREATE_DETACHED
  - Fonction pour obtenir l'état de détachement d'une thread :

pthread_attr_getdetachstate(const pthread_attr_t * attr, int * valeur);

  - Fonction pour modifier l'état de détachement d'une thread :

pthread_attr_setdetachstate(const pthread_attr_t * attr, int valeur);

Attributs des Threads (4)

- **Configuration de la pile :**
  - Obtenir et/ou modifier la taille et l'adresse de la pile
  - **Fonction pour obtenir la taille et l'adresse de la pile respectivement :**

    int pthread_attr_getstacksize (const pthread_attr_t * attr, size_t valeur);
    int pthread_attr_getstackaddr(const pthread_attr_t * attr, void ** valeur);
  - **Fonction pour modifier la taille et l'adresse de la pile respectivement :**

    int pthread_attr_setstacksize(const pthread_attr_t * attr, size_t valeur);
    int pthread_attr_setstackaddr(const pthread_attr_t * attr, void *valeur);
  - Peuvent entraîner des problèmes de portabilité

Attributs des Threads (5)

- **Configuration de la pile (cont.) :**
  - Valeurs de la taille et de l'adresse de la pile sont disponibles si les constantes suivantes on été définies dans le fichier <*unistd.h*> respectivement :
    - _POSIX_THREAD_ATTR_STACKSIZE
    - _POSIX_THREAD_ATTR_STACKADDR
  - Taille minimum d'une pile *(<unistd.h>*)
    - _PTHREAD_STACK_MIN

Attributs des Threads (6) - Exemple

```
….
void *thread_func (void *arg) {
….
  return NULL;
}
int main(int argc, char** argv) {
  int ret; size_t size; pthread_t tid;
  pthread_attr_t attr;

  if ((ret = pthread_attr_init (&attr)) !=0) {
    printf ("erreur  : %d\n", ret);  exit (1);
  }
  if ((ret = pthread_attr_getstacksize
(&attr,&size)) !=0) {
      printf ("erreur : %d \n", ret);  exit (1);
}
```

```
else
  printf ("Taille: %d; taille min :%d\n",
              size, PTHREAD_STACK_MIN);

 if ((ret = pthread_attr_setstacksize
  (&attr, PTHREAD_STACK_MIN*2)) !=0) {
   printf ("erreur : %d \n", ret);   exit (1);
 }

   if ((pthread_create(&tid, &attr, thread_func,
        NULL)) !=0) {
       printf ("erreur: %d \n", ret);
       exit (1);
   }
…
}
```


Attributs des Threads (7) Ordonnancement

- **Quatre attributs associés à l'ordonnancement :**
  - inheritsched
  - scope
  - schedpolicy
  - schedparam
  - Disponibles si _POSIX_THREAD_PRIORITY_SCHEDULING est définie dans <*unistd.h*>.
- **Attribut inheritedsched**

  int pthread_attr_getinheritsched (const pthread_attr_t * attr, int* valeur);
  int pthread_attr_setinheritsched (const pthread_attr_t * attr, int valeur);
  - Valeurs possibles :
    - PTHREAD_EXPLICIT_SCHED : l'ordonnancement spécifié à la création de la thread.
    - PTHREAD_IMPLICIT_SCHED : l'ordonnancement (valeurs schedpolicy et schedparam) hérité de la thread appelante

Attributs des Threads (8)
Ordonnancement

- **Attribut scope**
  - Pour les implémentations hybrides (ex. Solaris) :

    int pthread_attr_getscope (const pthread_attr_t * attr, int* valeur);

    int pthread_attr_setscope (const pthread_attr_t * attr, int valeur);
    - Valeurs possibles :
      - PTHREAD_SCOPE_SYSTEM : à chaque *Pthread* est associée une thread noyau.
      - PTHREAD_SCOPE_PROCESS : Les *Pthreads* d'un même processus sont prises en charge par un pool de threads noyau. Un nombre maximum de threads noyau existent pour un processus.
        - La taille du pool peut être consultée / modifiée par :

          int pthread_getconcurrency (void);

          int pthread_setconcurrency (int valeur);

Attributs des Threads (9)
Ordonnancement

- **Attribut schedpolicy**

  int pthread_attr_getschedpolicy (const pthread_attr_t * attr, int* valeur);

  int pthread_attr_setschedpolicy (const pthread_attr_t * attr, int valeur);
  - Valeurs possibles :
    - SCHED_OTHER : ordonnancement classique temps partagé
    - SCHED_FIFO : Temps-réel. Politique *FIFO*.
      - Thread avec priorité fixe et ne peut être préemptée que par une autre thread ayant une priorité strictement supérieure.
    - SCHED_RR: Temps-réel. Politique *Round-Robin*.
      - Après un quantum, la CPU peut être affectée à une autre Thread de priorité au moins égale.

Attributs des Threads (10) Ordonnancement

- **Attribut schedparam**

  int pthread_attr_getschedparam (const pthread_attr_t * attr, struct schedparam* sched);

  int pthread_attr_setschedparam(const pthread_attr_t * attr, struct schedparam sched);

  - *struct schedparam* possède le champ:
    - *int sched_priority :* priorité du processus.
    - Valeur comprise entre :
      - *sched_get_priority_min (classe)* et *sched_get_priority_max (classe).*
        - Classe : SCHED_OTHER, SCHED_RR, SCHED_FIFO

Attributs des Threads (11) Ordonnancement

- **Modification dynamique des attributs d'ordonnancement :**
  - Attributs *schedpolicy* et *schedparam* d'une thread peuvent être consultés et/ou modifiés avec les fonctions:

int pthread_getschedparam (pthread tid, int *classe, struct sched_param* sched);

int pthread_setschedparam (const pthread_attr_t * attr, int class, struct sched_param sched);

Attributs des Threads (12)
Ordonnancement – Exemple

```
void *thread_func (void *arg) {
    int ret; int policy; struct sched_param sched;
    ret = pthread_getschedparam (pthread_self(), &policy, &sched);
    if (ret)
       exit (1);
    else
         printf ("politique : %s, priorité :%d\n",
                  (policy == SCHED_OTHER ? "SCHED_OTHER" :
                   (policy == SCHED_FIFO ? "SCHED_FIFO" :
                   (policy == SCHED_RR ? "SCHED_RR" : "inconnu"))),
                    sched.sched_priority);
    return NULL;
}
```


Annulation d'une thread (1)

- **Une thread peut vouloir annuler l'autre.**
  - Une thread envoie une demande d'annulation à une autre qui sera prise en compte ou non en fonction de la configuration de celle-ci.
    - La thread qui reçoit une demande d'annulation peut la refuser ou la repousser jusqu'à atteindre un ***point d'annulation***.
  - Lorsque la thread annulée se termine, elle exécute toutes les fonctions de terminaison programmées.

Annulation d'une thread (2)

- **Demande d'annulation:**

  int pthread_cancel (pthread_t tid);

  - **Code de renvoi**: 0 ou *ESRCH* (si la thread n'existe pas).
- **Annulation d'une thread peut entraîner des incohérences :**
  - Exemples :
    - accès à une variable globale, abandon d'un mutex verrouillé, etc.
- **Solution :**
  - interdire temporairement les demandes d'annulation dans certaines portions du code.

Annulation d'une thread (3)

- **Interdiction temporaire des demandes d'annulation :**

  int pthread_setcancelstate (int etat_annulation, int *ancien_etat);

  - Fonction qui permet de configurer le comportement d'une thread vis-à-vis d'une requête d'annulation. Permet aussi de récupérer l'ancien état.
    - Possibles valeurs pour *etat_annulation* :
      - PTHREAD_CANCEL_ENABLE
        - La thread acceptera les requêtes d'annulation (**par défaut**)
      - PTHREAD_CANCEL_DISABLE
        - La thread ne tiendra pas compte des requêtes d'annulation.

Annulation d'une thread (4)

- **Interdiction temporaire des demandes d'annulation (PTHREAD_CANCEL_DISABLE) :**
  - Les requêtes d'annulation ne restent pas pendantes, contrairement aux signaux masqués.
    - Une thread désactivant temporairement les requêtes pendant une section critique ne se terminera pas lorsqu'elle autorise de nouveau les annulations.
  - Solution :
    - Utiliser un mécanisme de synchronisation afin de retarder des annulations jusqu'à atteindre des points bien définis dans le processus.

Annulation d'une thread (5)

- **Type d'annulation :**

int pthread_setcanceltype (int type_annulation, int *ancien_type);

- Possibles valeurs pour *type_annulation* :
  - PTHREAD_CANCEL_DEFERRED
    - La thread ne terminera qu'en atteignant un point d'annulation (**par défaut**)
  - PTHREAD_CANCEL_ASYNCHRONOUS
    - L'annulation prendra effet dès la réception de la requête d'annulation.

Annulation d'une thread (6)

- **Fonctions qui constituent des points d'annulation :**
  - *pthread_cond_wait ()* et *pthread_cond_timed_wait ();*
  - *pthread_join ();*
  - *pthread_testcancel ();*
    - Fonction *void pthread_testcancel (void)* :
      - Permet à une thread de tester si une requête d'annulation lui a été adressée.
        - Dès qu'elle est invoquée, la thread peut se terminer si une demande d'annulation est en attente.
      - Répartir des appels à *pthread_testcancel ()* aux endroits du code où on est sûr qu'une annulation ne posera pas de problème.

Annulation d'une thread (7)

- **Ensemble des fonctions et appels système bloquants qui sont des points d'annulation :**
  - Exemples :
    - *open (); close (); create ();*
    - *fcntl(); fsync();*
    - *pause ();*
    - *read();*
    - *sem_wait();*
    - *sigsuspend();*
    - *sleep();*
    - *wait, waitpid ();*
    - *etc.*

Annulation d'une thread (8)

- **Résumé des configurations**
  - PTHREAD_CANCEL_DISABLE
    - Région critique où on n'accepte pas des annulations.
  - PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_DEFERRED
    - Comportement par défaut.
  - PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_ASYNCHRONOUS
    - Boucle de calcul sans appels système qui utilise beaucoup de CPU.

Annulation d'une thread (9)

- **Une thread peut être annulée à tout moment**
  - nécessité de libérer les ressources que la thread possède avant qu'elle ne se termine.
    - Fichiers ouverts, mutex verrouillé, mémoire allouée, etc.
- **Solution :**
  - Lorsqu'une thread alloue une ressource qui nécessite une libération ultérieure, elle enregistre le nom d'une routine de libération dans une pile spéciale en utilisant la fonction ***pthread_cleanup_push ()***.
  - Quand la thread désire libérer explicitement la ressource, elle appelle ***pthread_cleanup_pop ()***.

Annulation d'une thread (10)

- Enregistrer des routines de libération dans une "pile de nettoyage":

  void pthread_cleanup_push (void (*fonction)(void *), void *arg);

  - Lorsque la thread se termine, les fonctions sont dépilées, dans l'ordre inverse d'enregistrement et exécutées.

  void pthread_cleanup_pop (int exec_routine);

  - Retire la routine au sommet de la pile.
  - *exec_routine* :
    - si non nul la routine est invoquée.
    - si 0, la routine est retirée de la pile de nettoyage sans être exécutée.

Annulation d'une thread (11)

```
void *func_thread (void *arg) {
 char * buf;  FILE * fich;
 ….
 buf = malloc (TAILLE_BUF);
 if ( buf != NULL) {
    pthread_cleanup_push(free,buf);
    fich = fopen ("FICH","r");
     if (fich !=NULL) {
        pthread_cleanup_push(fclose,fich);
        …
        pthread_cleanup_pop(1);
   }
   …
   pthread_cleanup_pop(1);
  }
}
```

• **Observation :**
Les appels aux fonctions *pthread_cleanup_push( )* et *pthread_cleanup_pop( )* doivent se trouver dans la même fonction et au même niveau d'imbrication.

Annulation d'une thread (12)

- **Conditions**
  - L'annulation d'une thread doit laisser le *mutex* associé à la *condition* dans un état cohérent
    - *mutex* doit être libéré.

```
pthread_mutex_lock (&mutex);
pthread_cleanup_push(pthread_mutex_unlock, (void*) &mutex);
while (! condition (var) ){
   pthread_cond_wait(&cond,&mutex);
}
….
/* pthread_mutex_unlock (&mutex); */
pthread_cleanup_pop(1);
```


Pthreads et les signaux (1)

- **La gestion d'un signal est assurée pour l'ensemble de l'application en employant la fonction *sigaction ().***
- **Chaque thread possède son masque de signaux et son ensemble de signaux pendants.**
  - Le masque d'une thread est hérité à sa création du masque de la thread la créant.
  - Les signaux pendants ne sont pas hérités.

```
int pthread_sigmask (int mode, sigset_t *pEns, sigset_t
 *pEnsAnc);
```

    - Permet de consulter ou modifier le masque de signaux de la thread appelante.

Pthreads et les signaux (2)

- **Envoi d'un signal à une thread**

  int pthread_kill (pthread_t tid, int signal);

  - Même comportement que la fonction *kill()*.
  - L'émission du signal au sein du même processus
  - Renvoie 0 en cas de succès ou ESRCH si la *thread tid* n'existe pas

- **Attente de signal**

  int sigwait (const sigset_t *ens, int *sig)

  - Extrait un signal de la liste de signaux pendants appartenant à *ens*. Le signal est récupéré dans *sig* et renvoyé comme valeur de retour de la fonction.

Pthreads et les signaux (3)

- **Signal traité par une thread spécifique**
  - **synchrone**
    - événement lié à l'exécution de la thread active. Signal est délivré à la thread fautive.
      - SIGBUS, SIGSEGV, SIGPIPE
    - Signal envoyé par une autre thread en utilisant *pthread_kill*
- **Signal traité par une thread quelconque**
  - **asynchrone** – reçu par le processus.
    - Le signal sera pris en compte par une des threads du processus parmi celles qui ne masquent pas le signal en question.

Pthreads et les signaux (4)

- **Observation :**
  - Les fonctions POSIX qui permettent de manipuler les *Pthreads* ne sont pas nécessairement réentrantes. Par conséquent elles ne doivent pas être appelées depuis un gestionnaire de signaux.
- **Solution :**
  - Créer une thread dédiée à la réception des signaux, qui boucle en utilisant *sigwait ()*.

Pthreads et les signaux (5)

int sigwait (const sigset_t * masque, int * num_sig);

- Attente de l'un des signaux contenus dans le champ *masque* .
  - Si un signal arrive, la fonction se termine en sauvegardant le signal reçu dans **num_sig*.
- Point d'annulation
- Possibilité d'utiliser les fonctions de la bibliothèque Pthreads.
- Toutes les autres threads doivent bloquer les signaux attendus.

Exemple 1 - signaux et Pthreads (6)

```
pthread_mutex_t mutex_sig = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_cont = PTHREAD_COND_INITIALIZER;
int sig_cont;

void * thread_sig (void *arg){
 sigset_t ens;  int sig;
 sigemptyset (&ens); sigaddset (&ens,SIGINT);

 while (1) {
   sigwait (&ens,&sig);
   pthread_mutex_lock (&mutex_sig);
   sig_cont ++;
   pthread_cond_signal (&cond_cont);
   if (sig_cont == 5) {
     pthread_mutex_unlock (&mutex_sig);
     pthread_exit ((void *)0);
   }
   pthread_mutex_unlock (&mutex_sig);
 }
}
```


Exemple 1- signaux et Pthreads (cont) (7)

```
void *thread_cont (void *arg) {
sigset_t ens;
  sigfillset (&ens);
 pthread_sigmask (SIG_SETMASK, &ens,
                  NULL);
 while (1) {
  pthread_mutex_lock (&mutex_sig);
  pthread_cond_wait(&cond_cont,&mutex_sig);
  printf ("cont: %d\n",sig_cont);
  if (sig_cont == 5) {
     pthread_mutex_unlock (&mutex_sig);
     pthread_exit ((void *)0);
  }
    pthread_mutex_unlock (&mutex_sig);
 }
}
int main (int argc, char ** argv) {
 pthread_t tid_sig, tid_cont;
 sigset_t ens;
sigfillset (&ens);
pthread_sigmask (SIG_SETMASK, &ens,NULL);

 if  ( (pthread_create (&tid_cont, NULL,
            thread_cont, NULL) != 0) ||
           (pthread_create (&tid_sig, NULL,
           thread_sig, NULL) != 0)) {
    printf ("pthread_create \n"); exit (1);
}
 if (pthread_detach (tid_sig) !=0 ) {
   printf ("pthread_detach \n"); exit (1);
 }
  if (pthread_join (tid_cont, NULL) !=0) {
    printf ("pthread_join"); exit (1);
  }
    printf ("fin \n");
 return 0;
}
```

Pthreads et sémaphores POSIX (1)

- **Sémaphore**
  - Variable du type *sem_t* permettant de limiter l'accès à une section critique.
    - #include <semaphore.h>
      - Si la constante _POSIX_SEMAPHORE est définie dans <unistd.h>
- **Création / Destruction**

  int sem_init (sem_t *sem, int partage, unsigned int valeur);
    - *Partage* : si valeur nulle, le sémaphore n'est partagé que par les threads du même processus
    - *Valeur* : valeur initiale du sémaphore
      - Valeur inscrite dans un compteur qui est décrémenté à chaque fois qu'une thread rentre en section critique et incrémenté à chaque sortie.

  int sem_destroy (sem_t *sem);

Pthreads et sémaphores POSIX (2)

- **Entrée/Sortie en section critique**

  int sem_wait (sem_t *sem);
  - Entrée en SC. Fonction bloquante
    - Attendre que le compteur soit supérieur à zéro et le décrémenter avant de revenir.

  int sem_post (sem_t *sem);
  - Sortie de SC. Compteur incrémenté; une thread en attente est libérée.

  int sem_trywait (sem_t *sem);
  - Fonctionnement égal à *sem_wait* mais non bloquante.
- **Consultation compteur sémaphore**

  int sem_getvalue (sem_t *sem, int *valeur);
  - Renvoie la valeur du compteur du sémaphore *sem*. dans *valeur.

Exemple 2 - Sémaphore POSIX (3)

```
#define _POSIX_SOURCE 1
#include <stdio.h>      #include <stdlib.h>
#include <pthread.h>  #include <unistd.h>
#include <semaphore.h>

#define NUM_THREADS 4
sem_t sem;

void *func_thread (void *arg) {
 sem_wait (&sem);
 printf ("Thread %d est rentrée en SC \n",
          (int) pthread_self ());
  sleep ((int) ((float)3*rand()/ (RAND_MAX +1.0)));
  printf ("Thread %d est sortie de la SC \n",
           (int) pthread_self ());
 sem_post(&sem);
 pthread_exit ( (void*)0);
}
```

```
int main (int argc, char ** argv) {
 int i;
 pthread_t tid [NUM_THREADS];

sem_init (&sem,0,2);

for (i=0; i < NUM_THREADS; i++)
   if  (pthread_create (&(tid[i]), NULL,
                    func_thread, NULL) != 0) {
     printf ("pthread_create"); exit (1);
   }
 for (i=0; i < NUM_THREADS; i++)
   if (pthread_join (tid[i], NULL) !=0) {
     printf ("pthread_join"); exit (1);
 }
 return 0;
}
```


Pthread et fork (1)

- **Lors du *fork***
  - Le processus est dupliqué, mais il n'y aura dans le processus fils que la thread qui a invoqué le *fork ()*.
- **Ce mécanisme doit être restreint à l'utilisation de *exec* après le *fork*.**
- **Problème :**
  - une autre thread du processus père a pris/verrouillé une ressource dont le fils aura besoin.

Pthread et fork (2)

- **Exemple Problème :**
  - Thread1 et Thread2.
    - *Thread1* verrouille une ressource partagée en utilisant *mutex*
    - *Thread2* appelle fork *()*;
      - La seule thread du fils sera *Thread2*
    - *Thread1* du processus père continue à exécuter et libère *mutex*
    - *Thread2* du processus fils veut accéder à la ressource critique.
      - Attend la libération du verrou par *Thread1*, mais celle-ci n'existe pas dans le processus fils.
        - Processus fils **bloqué pour toujours**.

Pthread et fork (3)

- **Solution pour les ressources partagées** :
  - fonction *pthread_atfork* qui permet d'enregistrer les routines qui seront automatiquement invoquées si une thread appelle le *fork*.

```
int pthread_atfork (void (*avant) (void),
                    void (*dans_pere) (void), void (*dans_fils) (void));
```

    - *avant*: fonction appelée avant le *fork*.
    - *dans_pere* et *dans_fils* : fonctions appelées par le père et par le fils respectivement après le *fork* () au sein de la thread ayant invoqué le *fork*.

Pthread et fork (4)

- **Solution Problème Thread1 et Thread2:**
  - Installer avec pthread_atfork() les fonctions :
    - *avant() : pthread_mutex_lock ()*
    - *dans_père () : pthread_mutex_unlock ()*
    - *dans_fils () : pthread_mutex_unlock ()*

- **Exécution :**
  - *Thread1* verrouille *mute*x ;
  - *Thread2* appelle fork *()*;
    - *avant ()* est exécutée en bloquant *Thread2.*
  - *Thread1* libère *mute*x
    - *avant ()* se termine et le *fork()* a lieu.
      - *dans_père ()* et *dans fils ()* sont exécutées, libérant le verrou dans les deux processus.

Exécution unique de fonction (1)

- **Lorsque plusieurs threads appellent une même fonction, parfois il est souhaitable que cette fonction soit exécutée une seule fois**
  - Utiliser une variable statique de type *pthread_once_t* initialisée à *PTHREAD_ONCE_INIT*;
  - La fonction à n'exécuter qu'une seule fois est appelée en utilisant :

```
int pthread_once (pthread_once_t *once, void (*func));
```

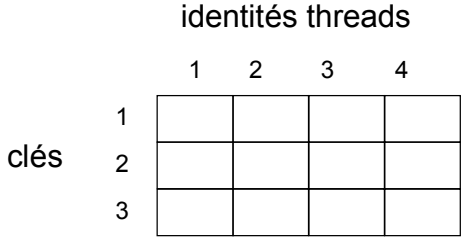
Exécution unique de fonction (2)
Exemple

```
static pthread_once_t once = PTHREAD_ONCE_INIT;
 void foncInit (void) {
      …..
}

void * func_thread (void *arg) {
     …
     pthread_once (&once, foncInit);
 ….
}

int main (int argc, char* argv[]) {
I   int i; pthread_t tid[3];
    for (i=0; i<3; i++)
        pthread_create(&tid [i], NULL, func_thread,NULL);
….
}
```

FoncInit ne sera qu'appelée par la première thread qui exécutera *pthread_once*

Données privées (1)

- **Une thread peut posséder des données privées.**
  - Un ensemble de données statiques est réservé et réparti entre les threads d'un même processus.
    - Ensemble peut être vu comme une matrice :



Données privées (2)

- Fonction qui permet la création d'un nouvelle clé :

  int pthread_key_create (pthread_key_t *cle,
  (void*) destruc (void*));

  - Si *destruc* égal à NULL, l'emplacement créé n'est pas supprimé à la terminaison de la *thread*. Sinon, la fonction spécifiée dans *destruc* est appelée à la terminaison de la thread.
- Fonction qui permet de consulter une donnée privée :

  void * pthread_getspecific(pthread_key_t cle);
- Fonction qui permet modifier une donnée privée :

  int pthread_setspecific(pthread_key_t cle, void *valeur);

  - valeur est typiquement l'adresse d'une zone allouée dynamiquement.

Données privées (3) - Exemple

```
…..
pthread_key_t cle;
void *thread_func (void *arg) {
 int i;  int *pt, *pt2;
  i = *((int*) arg);
  if ((pt = (int*) malloc (sizeof (int))) == NULL)
    exit (1);
  else     *pt = i;
  if (pthread_setspecific (cle, pt) !=0)
       exit (1);
  else   *pt +=2;
  if ((pt2= pthread_getspecific (cle ))== NULL)
      exit (1);
  else
      printf ( "Thread %d - valeur : %d \n",i, *pt2);
  return NULL;
}
```

```
int main(int argc, char** argv) {
 pthread_t tid [2]; int* pt_ind; int i;

 if (pthread_key_create
       (&cle,NULL) !=0)
       exit (1);
  for (i=0; i < 2; i++) {
     pt_ind = (int *) malloc (sizeof (i));
    *pt_ind =i;
      if  (pthread_create (&(tid[i]),
                 NULL, thread_func,
                 (void *)pt_ind ) != 0)
           exit (1);
  }
/* join */
……
}
```

```
Thread 0 – valeur : 2
Thread 1 – valeur : 3
```