Introduction aux Threads

Temps Réels & Embarqués

École Centrale de Lyon

2009-2010

Alexander Saidi

17 février 2010

1.1 Threads POSIX

- Pour implanter le parallélisme Mono et multi processeurs.
- Standardisé Posix IEEE 1003.1c (Pthreads), 1995 (original)

Définition :

Une suite d'instructions indépendantes chargée (scheduled) et exécutée par le SE.

➙ Comme une procédure / fonction qu'on appelle mais dont on revient immédiatement la laissant se dérouler de manière autonome.

➥ Dès que cet appel a lieu (via une création du thread), les 2 entités (l'appelant et l'appelé) s'exécutent en **parallèle**.

➥ On sera dans le cas du *multi-threading*.

1.2 Une introduction par des exemples

1.2.1 Création d'un thread

```
#include <pthread.h>
int pthread_create(pthread_t *tid, pthread_attr_t *attributs, void *(*fonction)(void *), void *arg);
```

• *pthread_create* démarre l'exécution d'un nouveau thread (une fonction) qui s'exécutera en parallèle de l'application (mais dans le même processus).

• Le deuxième paramètre spécifie les options de création du thread.

• Pour utiliser les options par défaut, il suffit de passer NULL comme valeur.

• Le troisième paramètre indique la fonction à démarrer.

Cette fonction doit avoir le prototype suivant : **void *fonction(void *).**

• Le dernier paramètre spécifie le paramètre d'appel de la fonction (cf. l'exemple ci dessous pour une utilisation possible de ce paramètre).

1.2.2 *Attente d’un thread*

```
#include <pthread.h
int pthread_join(pthread_t tid, void **thread_exit_code);
```

- *pthread_join* attend la fin du thread *tid* et permet de récupérer son code de sortie (renvoyé par *pthread_exit(code)*).

➙ Il est possible d’ignorer ce code de retour en plaçant NULL pour ce paramètre

➥ Voir l’exemple suivant..

1.2.3 Exemple-1 (trivial)

- Le *main* crée 1 thread (qui affiche un message) puis attend sa fin.

→ Remarquer l'utilisation générale de *(void *)*.

```
// 2010 : ex-super-simple.c
// Compiler avec : gcc -lpthread ex-super-simple.c

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

void *print_message_function( void * );
char *message1 = "Le message de Thread 1";

main()
{
  pthread_t thread1;

  int  iret1;

  /* Créer un thread  indépendant qui exécutera la fonction  print_message_function() */
  iret1 = pthread_create( &thread1, NULL, print_message_function,  NULL);


  /* Attendre la fin du thread avant de continuer. Sans join, l'exécution de  */
  /* exit(0)  mettra fin à tout le processus (threads compris) avant que */
  /* le thread se termine.*/

  pthread_join( thread1, NULL);
```

```
  printf("Thread 1 est terminé (son id = %d) \n",iret1);
  exit(0);
}

void *print_message_function( void *ptr )
{
  printf("%s \n", message1);
}
/* Trace :
Le message de Thread 1
Thread 1 est terminé (son id = 0)
*/
```

Listing 1.1 – ex-super-simple.c

• On peut aussi compiler avec : **gcc -lpthread -o ex-super-simple ex-super-simple.c**

➙ Cet exemple est "compilable" en C++ aussi (grâce à la forme du paramètre de la fonction *print_message_function*).

➙ Si on enlève*(void* ptr)*, *gcc* ne compilera plus.

1.2.4 Exemple-2 avec passage de paramètre

- Création de 2 threads qui affichent un message chacun + attente de leur fin.

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

void *function_affichage( void *ptr );     // Déclaration de la fonction attachée aux threads

int main()
{
  pthread_t thread1, thread2;
  char *message1 = "Thread 1";
  char *message2 = "Thread 2";
  int  iret1, iret2;

  /* Créer 2 threads indépendants qui exécuteront la même fonction avec 2 messages différents */

  iret1 = pthread_create( &thread1, NULL, function_affichage, (void*) message1);
  iret2 = pthread_create( &thread2, NULL, function_affichage, (void*) message2);

  /* Attendre que les threads terminent avant de continuer. Sinon, on risque d'exécuter  */
  /* exit() qui terminera  TOUT et donc les 2 threads avant qu'ils aient fini.  */

  pthread_join( thread1, NULL);    // NULL : on ignore toute valeur renvoyée par le thread.
  pthread_join( thread2, NULL);

  printf("Thread 1 retourne \n");
  printf("Thread 2 retourne  \n");
  exit(0);
}
```

```
void *function_affichage( void *ptr )
{ int i;
  char *message;
  message = (char *) ptr;
  for (i=0; i<10; i++) {
    printf("%d> %s \n", i, message);
  }
}
```

Listing 1.2 – Ex-1-creation-join.c

Compilation :

- C : *gcc -lpthread Ex-1-creation-join.c*
- C++ : *g++ -lpthread Ex-1-creation-join.c*

Test : Lancer *./a.out*

➙ Résultats (de gauche à droite et du haut en bas) :

```
0> Thread 1     1> Thread 1     2> Thread 1     3> Thread 1     4> Thread 1
5> Thread 1     0> Thread 2     1> Thread 2     2> Thread 2     3> Thread 2
4> Thread 2     5> Thread 2     6> Thread 2     7> Thread 2     8> Thread 2
9> Thread 2     6> Thread 1     7> Thread 1     8> Thread 1     9> Thread 1
Thread 1 retourne
Thread 2 retourne
```

Remarques :

- D'une exécution à l'autre, l'ordre des affichages (thread1 ou thread2) peut varier.
- Dans cet exemple, la fonction attachée aux threads est la même mais les arguments (messages) changent.
- *join()* permet au *main()* d'attendre la fin des threads.
- **Un threads peut terminer** (avec une différence pour *main()*) :

 1. Si un thread appelle *pthread_exit(n)*, il termine.
 2. Si *main()* appelle *pthread_exit(n)*, il termine mais les autres threads continuent.
 3. Si un thread finit son code normalement (sans valeur de retour), il termine.
 4. Si *main()* termine de cette façon, les autres threads terminent aussi.
 5. Un appel à *exit()* (main comprise) terminera TOUT le processus.
 6. *exit(0)* n'est pas équivalent à *pthread_exit(0)*.

Remarque sur la compilation :

L'exemple ci-dessus compilera avec g++ (C++).

Par contre, l'écriture suivante ne compilera seulement qu'avec gcc (C).

```
void print_message_function( void *ptr );
  ...
iret1 = pthread_create( &thread1, NULL, (void*)&print_message_function, (void*) message1);
  ...
```

- Une autre version compilée en C mais pas en C++ :

```
void function();       // écrire        void function(void* bidon)     pour compiler en C++
...
iret1 = pthread_create( &thread1, NULL,  function, NULL);
...
```

- Rappel : si *main()* termine en exécutant *pthread_exit()*, alors les autres threads continuent (*main* termine mais le processus associé subsiste jusqu'à la fin des threads).

Par contre, si *main()* termine normalement (sans rien), alors les autres terminent aussi.

➙ *exit(0)* n'est pas équivalent à *pthread_exit(0)* ➙ *exit(0)* arrête tout.

1.2.5 Exemple-3 : renvoi de valeur par un thread via *join()*

- On reprend l'exemple précédent mais cette fois en version C++.
- Chaque thread renvoie une valeur (son ID) transmise via *join*.
- Compilation : *g++ -w -lpthread Ex-2-join-recupere-params.cpp* (-w : warnings)

```
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
using namespace std;

void *function_affichage_avec_param_retour( void *ptr );

int main()
{
      pthread_t thread1, thread2;
      char *message1 = (char*)"Message Thread 1";
      char *message2 = (char*)"Message Thread 2";
      int  iret1, iret2;

      void * val_ret_th1, * val_ret_th2;   // récupérer des valeurs renvoyés par les threads

     /* Créer 2 threads indépendants qui exécuteront la même fonction avec 2 messages différents */
      iret1 = pthread_create( &thread1, NULL, function_affichage_avec_param_retour, (void*) message1);
      iret2 = pthread_create( &thread2, NULL, function_affichage_avec_param_retour, (void*) message2);

       cout << "Id du thread 1 : " <<(long)thread1 << endl;
```

```
     cout << "Id du thread 2 : " <<(long)thread2 << endl;

    /* Attendre que les threads terminent avant de continuer. Sinon, on risque d'exécuter  */
    /* exit qui terminera  TOUT et donc les 2 threads avant qu'ils aient fini.  */

    pthread_join( thread1,  &val_ret_th1);
    pthread_join( thread2,  &val_ret_th2);

    cout << "Thread 1 retourne : " << *(long*)val_ret_th1 << endl;
    cout << "Thread 1 retourne : " << *(long*)val_ret_th2 << endl;
    exit(0);
}

void *function_affichage_avec_param_retour( void *ptr )
{
    int i;
    char *message;
    message = (char *) ptr;
    for (i=0;i<10;i++) cout << (long) i << " : " << message << endl;

    // Affiche puis renvoie son propre id
    cout << "my self id : "<<(long)pthread_self()<<endl;

    pthread_t * ptr_sur_mon_id=new pthread_t(pthread_self());
    return  ptr_sur_mon_id;    // équivalent à   pthread_exit(ptr_sur_mon_id)
}
```

Listing 1.3 – Ex-2-join-recupere-params.cpp

- Bien remarquer les conversions et la nécessité de passer par une allocation (*new*).

➙ Au lieu de cette allocation dans chaque thread, on pourrait utiliser une variable

globale pour chacun, l'alimenter dans le thread et en renvoyer l'adresse.

- Une trace d'exécution :

```
/* une trace :  entrelacement possible
 Id du thread 1 : 139861895371024
Id du thread 2 : 139861886978320
0 : Message Thread 2
1 : Message Thread 2
2 : Message Thread 2
3 : Message Thread 2
4 : Message Thread 2
5 : Message Thread 2
6 : Message Thread 2
7 : Message Thread 2
8 : Message Thread 2
9 : Message Thread 2
my self id : 139861886978320
0 : Message Thread 1
1 : Message Thread 1
2 : Message Thread 1
3 : Message Thread 1
4 : Message Thread 1
5 : Message Thread 1
6 : Message Thread 1
7 : Message Thread 1
8 : Message Thread 1
9 : Message Thread 1
my self id : 139861895371024
Thread 1 retourne : 139861895371024
Thread 1 retourne : 139861886978320
*/
```

Remarques :

- Lorsqu'un thread se termine, s'il n'est pas détaché (à la création) et tant qu'il n'y a pas eu *join()* avec lui, il conserve les ressources qui restent disponibles.
- En particulier, la valeur renvoyée par le thread est récupérable par *pthread_join()*.
- Détacher un thread (par *join()* ou en changeant son statut détaché/joignable en cours d'exécution) permet au système de récupérer ses ressources mais rend le thread non joignable.

1.2.6 Exemple-4 : main laisse finir les threads

- *main()* termine par un appel à *void pthread_exit(void *info) ;*

➙ Elle laisse les threads finir.

```
// 2010 : terminaison des threads (via main)
// Compil : gcc terminaison_de_main.c -lpthread
// Chaque thread reçoit un entier 0..NB_threads et l'utilise par créer une lettre (de 'A'..'Z')
// Affiche sa lettre et l'indice de son itération qui va de 0 à 3*valeur reçu
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NB_threads 5
void * ma_fonction(void * ptr);

int main()
{  int i, rc;
   pthread_t tab_id1[NB_threads];
   int tab_valeurs[NB_threads];

   printf("Créations des threads \n");
   for (i=1; i< NB_threads; i++){
      tab_valeurs[i]=i;
      rc = pthread_create(&tab_id1[i] , NULL, ma_fonction, (void *) &tab_valeurs[i]);
   }

    printf("Sans Join, Main termine avec  pthread_exit(0) \n");
    printf("Les threads continuent\n");
    pthread_exit(0);     // Laisser finir
```

```
}

void * ma_fonction(void * ptr) {
     int valeur = *((int *) ptr), i;
     valeur ++;  // pour commencer à 1;
     char mon_signe= valeur+'A';      // donnera de 'A'.. 'Z'
     printf("%c> Je compte jusqu'au %d\n", mon_signe, 3*valeur);
     for (i=0; i<valeur*3; i++){
          printf("%c-%d\n", mon_signe,i);
           usleep(500000);
     }
}
/* Trace :
Créations des threads
C> Je compte jusqu'au 6
C-0
D> Je compte jusqu'au 9
D-0
E> Je compte jusqu'au 12
Sans Join, Main termine avec  pthread_exit(0)
Les threads continuent
E-0
F> Je compte jusqu'au 15
F-0
C-1
D-1
F-1
E-1
...
Suite des affichages..
*/
```

Listing 1.4 – terminaison_de_main.c

1.2.7 Exemple-5 : renvoi d'une valeur par un thread

• *pthread_exit()* termine l'exécution du thread ; elle permet de renvoyer des informations au processus/thread père.

```
//2010 : le thread renvoie une valeur au main

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

#define NB_THREAD 5

void *fn_thread(void *arg);

int compteur = 0;

int main()
{
  unsigned int i, j;
  pthread_t threads[NB_THREAD];
  int threadnb[NB_THREAD];
  void *res;

  // Création des threads
  for ( i = 0; i < NB_THREAD; ++i ){
    threadnb[i] = i;
    if ( pthread_create(&threads[i], NULL, fn_thread, (void*)&threadnb[i]) ){
      fprintf(stderr, "Erreur lors de pthread_create\n");
```

```
      break;
    }
  }

// Attendre la fin des threads et récupérer leur compte rendu
  for ( j = 0 ; j < i; ++j ){
    pthread_join(threads(j), &res);
    printf("Valeur retournée par le thread num %d: %d\n", j, *(int *)res);
  }

  // Afficher la valeur récupérée
  printf("Valeur finale du compteur: %d\n", compteur);
  return EXIT_SUCCESS;
}

// Les threads :
void *fn_thread(void *arg)
{
  int i;
  int no_thread = *(int*)arg;

  // On affiche des choses
   for ( i = 0; i < 100; ++i ){
    compteur = compteur + (compteur + 3) % (compteur + 2);
        pthread_yield();        // ou sched_yield();
  }

  printf("Valeur du compteur à la fin du thread num %d: %d\n",no_thread, compteur);
  usleep(1);

  // On transmet la valeur du compteur
  pthread_exit((void *)&compteur);
}
```

```
/* Une TRACE : (trace différente d'une exécution à l'autre)

Valeur du compteur à la fin du thread num 0: 153
Valeur du compteur à la fin du thread num 1: 184
Valeur du compteur à la fin du thread num 2: 284
Valeur du compteur à la fin du thread num 3: 384
Valeur du compteur à la fin du thread num 4: 484
Valeur retournée par le thread num 0: 484
Valeur retournée par le thread num 1: 484
Valeur retournée par le thread num 2: 484
Valeur retournée par le thread num 3: 484
Valeur retournée par le thread num 4: 484
Valeur finale du compteur: 484
*/
```

Listing 1.5 – Ex-recup-param-par-join.c

1.2.8 Abandon du temps d'exécution

```
#include <pthread.h>
int pthread_yield(void);                 ou           sched_yield()   ;
```

• Cette fonction permet à un thread d'abandonner son temps d'exécution au profit d'un autre thread (choisi par l'ordonnanceur / arbitre). Elle est très utile lorsqu'un thread boucle sur une activité et bloque constamment un verrou pour réaliser sa tâche (cf. Synchronisation et Section Critique ci dessous).

• NB : On peut "passer la main" avec

→ **sleep(n)** attendre n secondes (1...)

→ **usleep(1..1000000)** micro secondes (usleep(1000000)=sleep(1))

→ **sched_yield()** (ou *pthread_yield()* qui n'est pas std).

Exemple d'utilisation de pthread_yield

```
int tab[10];
pthread_mutex_t verrou;
void *thread1(void *voidparam)
{
    for(;;)
    {
        for(i= 0; i< 10; i++) tab[i]= 0;

        /* sans le yield, le thread va reprendre tout de suite le verrou sans laisser
        le temps aux autres threads de travailler sur le tableau
        */
        pthread_yield();
    }

    pthread_exit(NULL);   // termine le thread
    return NULL;   // pour que le compilateur soit content, mais ça ne sert à rien
}
```

1.2.9 *Erreurs à éviter lors du passage des paramètres*

Cas 1 : l'exemple (partiel) suivant montre un cas incorrect de passage d'argument :
➙ L'argument **t** se modifie alors qu'il est partagé.

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

➥ Solution : s'arranger pour que la valeur passée en argument ne se modifie pas dans la fonction qui passe l'argument (ici, *main()*).

Exemple : bien remarquer les valeurs affichées par les threads.

➙ On s'attend que chaque thread affiche une valeur différente en sortant. Ce n'est pas le cas !

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

void *affiche(void *voidparam)
{  printf("Thread lancé : %d\n",*((int *) voidparam));
   sleep(1);
   printf("termine %d\n", *((int *) voidparam));
   return NULL;     //  (= pthread_exit(NULL))
}

int main(void)
{  pthread_t tid(10);
   int code, i;

   // creation des threads
   for(i= 0; i < 10; i++) {
      code= pthread_create(&tid(i), NULL, affiche, (void *) &i);
      if(code < 0) perror("creation du thread");
   }

   // attente des threads
   for(i= 0; i < 10; i++) code= pthread_join(tid(i), NULL);
   return 0;
}
```

```
/* Une trace :
Thread lancé : 1
Thread lancé : 1     // le 2e thread a accédé au paramètre avant son incrémentation dans main
Thread lancé : 3
Thread lancé : 4
Thread lancé : 4
Thread lancé : 6
Thread lancé : 6
Thread lancé : 8
Thread lancé : 9
Thread lancé : 0

// A sa sortie , chaque thread affiche n'importe quoi !

termine 0
termine 0
termine 1
termine 1
termine 0
termine 1
termine 6
termine 1
termine 7
termine 7
*/
```

Listing 1.6 – Ex-pb-param.c

➥ Voir aussi ci-dessous.

Cas 2 : l'exemple suivant montre un autre cas incorrect de passage d'arguments :
➙ On aimerait passer les entiers 1 et 2 en argument aux deux threads.

```
void * ma_fonction(void * ptr);
int main()
{   int rc;
    pthread_t id1, id2;
    printf("Créations des threads \n",);
    rc = pthread_create(&id1 , NULL, ma_fonction, (void *) 1);
    rc = pthread_create(&id2, NULL, ma_fonction, (void *) 2);
    ...
}
```

- Ce cas provoque une erreur car l'entier 1 (idem 2) est considéré comme une adresse (dans l'écriture **(void*)1**). Or, cette adresse a beaucoup de chance d'être réservée au système !

➥ Solution : ne pas donner une valeur constante à cet endroit mais donner l'adresse d'une variable (voir ci-dessous).

OK : l'exemple suivant montre une forme correcte de passage d'arguments :

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

void * ma_fonction(void * ptr);

int main()
{   int rc;
    pthread_t id1, id2;

    int param1=1 , param2=2;

    printf("Créations des threads \n");
    rc = pthread_create(&id1 , NULL, ma_fonction, (void *) &param1);
    rc = pthread_create(&id2, NULL, ma_fonction, (void *) &param2);
    //...
}

void * ma_fonction(void * ptr) {
    int valeur = *((int *) ptr);
    printf("la valeur reçu = %d\n", valeur);
    //...
}
```

- Ces divers paramètres peuvent être des valeurs d'un tableau. Par exemple :

```
int params[]={12, 25, 36, ...};
rc = pthread_create(&id1 , NULL, ma_fonction, (void *) &params[0]);
rc = pthread_create(&id2, NULL, ma_fonction, (void *) &params[1]);
//...
```

1.2.10 Mutex : fonctions de manipulation des verrous

- **Création d'un verrou**

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr_t *mutex_attr);
```

➙ Initialise un verrou avant utilisation. Le premier paramètre désigne le verrou et le second les options de création. Les options par défaut sont utilisées lorsque le paramètre est NULL.

- **Bloquer un verrou**

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- **Tentative de bloquer un verrou**

```
#include <pthread.h>
extern int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

➙ = tentative non bloquante : l'appelant n'est pas bloqué si Mutex n'est pas libre.

• Relâcher un verrou

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

• Détruire un verrou

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Schéma d'utilisation des mutex :

```
    pthread_mutex_t verrou;   // déclaration d'un verrou

    // initialisation, nécessaire pour pouvoir utiliser le verrou
    pthread_mutex_init(&verrou, NULL);

    // demande de blocage du verrou passé en paramètre
    pthread_mutex_lock(&verrou);

    // manipulation sur les données
    { ... }

    // relâche le verrou passé en paramètre
    pthread_mutex_unlock(&verrou);

    // destruction du verrou / libération des ressources utilisées
    pthread_mutex_destroy(&verrou);
```

1.2.11 Exemple-6 : Mutex simple

- Les threads incrémentent un compteur à tour de rôle.

```
// 2010 : ex simple de  mutex : on fait +1 sur un compteur
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10

void *thread_function(void *);

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id(NTHREADS);
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id(i), NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id(j), NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
```

```
    /* have been completed.                                          */

    printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
    printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}

/* sortie (sur MAc)
Thread number 140459706796304
Thread number 140459698403600
Thread number 140459690010896
Thread number 140459681618192
Thread number 140459673225488
Thread number 140459664832784
Thread number 140459656440080
Thread number 140459648047376
Thread number 140459639654672
Thread number 140459631261968
Final counter value: 10
*/
```

Listing 1.7 – Ex-mutex-join.c

Remarques :

• Cette introduction est détaillée dans la suite.

• Pour la compléter, il faudra lire les sections consacrées aux **Conditions** (avec *timeout*) ainsi que la partie **Sémaphores** (avec attente bornée).

• Néanmoins, cette introduction contient les outils pour réaliser les mêmes effets.

• A un titre indicatif :

➙ Une **condition** = 2 Mutex + une variable

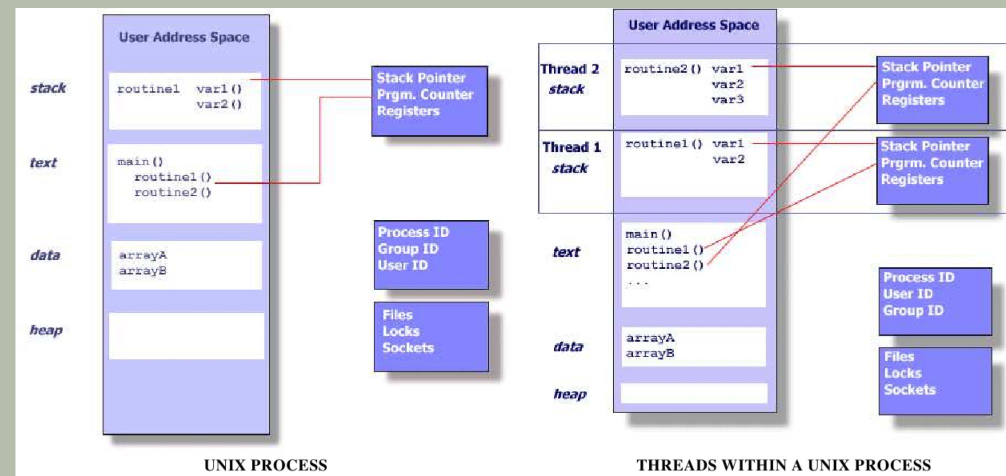
➥ Attente sur une Condition : boucle d'attente sur un Mutex

➥ Attente bornée sur une condition : attente sur la condition + décompte du temps.

• Néanmoins, le code équivalent comporte souvent une attente active alors que les outils de POSIX équivalents sont basés sur une attente passive.

1.3 Thread comme processus léger

• Dans le cas simple, l'ensemble des threads évolue à l'intérieur d'un processus (cf. ADA/Java, ...)



• Les threads partagent les ressources du programme (processus principale = main()) mais peuvent être pris en charge de manière indépendante par le SE.

➞ Ils copient le strict minimum des ressources du processus principal pour pouvoir être exécutés.

- Chaque thread possède :
  - ➙ ptr de pile, registres, propriétés de scheduling (e.g. . priorité) ;
  - ➙ ensemble des signaux (suspendus ou bloqués) ;
  - ➙ Données spécifiques au thread.
- Un thread doit son existence à son parent (son créateur).
- Du fait de partager les mêmes ressources que le processus parent, une modif faite sur ces ressources (e.g. . fermeture d'un fichier) est visible par les autres threads/parent.
- 2 pointeurs ayant le même valeur pointent la même données (vs. fork).
- Le programmeur doit veiller aux accès concurrents à une même zone de mémoire.
- Les threads Posix sont définis en C (structures de données et appels de fonctions).
  - ➙ inclusion par "pthread.h" et compilation avec (-pthread)
  - ➙ Les threads possèdent leur librairie (qui peut faire partie du *libc*).

1.4 Intérêts des threads

- Gain de performances
- Leur création et gestion coûte moins cher que pour les processus

➙ en terme du temps CPU/espace

- Du fait du partage d'un même espace, les échanges inter-threads sont plus rapides.

➙ Les échanges IPC par copie (Shmem) coûtent chers tandis que les threads n'ont pas besoin de ces copies.

1.5 Exemple fork vs. thread

- On crée un processus et on attend sa fin avant de créer le suivant.

```
#include <stdio.h>
#include <stdlib.h>
#define NFORKS 50000

void do_nothing() { int i; i= 0; }

int main(int argc, char *argv()) {
  int pid, j, status;
  for (j=0; j<NFORKS; j++) {
      /*** error handling ***/
    if ((pid = fork()) < 0 )
        { printf ("fork failed with error code= %d\n", pid);
         exit(0);
        }

      /*** this is the child of the fork ***/
    else if (pid ==0)
        { do_nothing(); exit(0); }
      /*** this is the parent of the fork ***/
    else { waitpid(pid, status, 0);   }
  }
}
```

Et la version avec thread

- On crée et on attend la fin avant de créer le suivant.

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

#define NTHREADS 50000

void *do_nothing(void *null) {
  int i; i=0;
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  int rc, i, j, detachstate;
  pthread_t tid;
  pthread_attr_t attr;

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);



/* Création des threads */
  for (j=0; j<NTHREADS; j++) {
      rc = pthread_create(&tid, &attr, do_nothing, NULL);
      if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
        }
```

```
      /* Wait for the thread */
      rc = pthread_join(tid, NULL);
      if (rc) {
        printf("ERROR; return code from pthread_join() is %d\n", rc);
        exit(-1);
        }
  }

  pthread_attr_destroy(&attr);
  pthread_exit(NULL);
}
```

1.6 Thread or not thread !

Comparaison vs. une application sans thread :

• Performances : si l'application doit attendre des E/S longues, un thread peut s'en occuper pendant que le reste de l'application continue les calculs (attention aux dépendances)

➙ C'est le principe de **recouvrement**.

• Une application peut avoir une section qui a besoin d'une priorité plus élevée que le reste (ou les autres applications du même prio).

• Dans le cas de la prise en charge d'événement asynchrone (cf. embarqué)

➙ E.g., un serveur Web peut à la fois s'occuper de répondre à une requête et en même temps en traiter une autre.

Quand se servir des threads ?

- Lorsqu'une application possède des fonctions indépendantes dont l'ordre d'exécution n'a pas d'importance (interchangeables)
- Si les données supportent des manipulations concurrentes (e.g. . Qsort, mult-Mat)
- Si l'application doit procéder à des E/S qui peuvent être longues (attente)
- Si l'application doit prendre en charge des événements asynchrones
- Si les différentes parties de l'application nécessitent des prios différentes.

Les modèles courant d'utilisation des Pthreads :

• Patron/ouvrier : le patron distribue des tâches (comme sur un chantier) après avoir reçu les données (le matériel)

➞ Se décline en 2 sous modèles : statique ou dynamique (nbr d'ouvriers changeant)

➞ Le patron peut (ou non) ensuite faire comme les ouvriers.

• Pipeline : le travail à réaliser est découpé en parties parallèles indépendantes et chaque partie (ses sous opérations) est achevée en série .

➞ Un exemple est une chaîne de montage de véhicules.

Mémoire partagée

• Les threads se partagent une même zone globale mais chacun peut posséder des données privées (comme les ados dans la maison de leurs parents).

➙ L'accès concurrent aux données partagées est organisé par le programmeur.

NB important : faire attention lors d'utilisation des librairies : elles peuvent ne pas être *thread-safe* (et / ou **ré entrant**).

➙ Dans le doute, faire l'hypothèse pessimiste.

➙ S'organiser pour faire des appels en série ou contrôlés.

1.7 Les APIs Pthread

• Dernier standard : IEEE Std 1003.1, 2004 (évolution depuis 1995).

Organisation des APIs :

✔ **Gestion** : création, détachement, jointure, etc.

➙ Demande d'attributs des threads : joignable, scheduling, etc.

✔ **Mutex** : création, destruction, verrouillage/déverrouillage des mutex.

➙ Ainsi que les routines de modification des attributs des mutex.

../..

✔ **Variables conditionnelles** : les routines de gestion des communications entre les threads qui partagent des mutex.

➙ fonctions de création, destruction, attente et signal basées sur les valeurs spécifiées.

➙ Ainsi que les fonctions de modification d'attributs des variables conditionnelles.

✔ **Synchronisation** : routines qui gèrent les verrous de lecture/écriture et les barrières.

Convention de nommage POSIX :

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

- Voir la liste des routines API à la fin de ce document.

Compilation sous Linux / macos (GNU) :

➙ **gcc -pthread** en C pour le link, ajouter *-lpthread*

➙ **g++ -pthread** en C++ ... idem ...

➥ Ne pas oublier *#include <pthread.h>* dans le code et *-lpthread* si link séparé.

- Un programme C/C++ utilisant pthread contient un thread associé au *main()*.

➙ Tous les autres seront crées par le programmeur.

- Une fois crées, les threads peuvent en créer d'autres à leur tour :

➙ Il n'y a pas de hiérarchie de threads ni dépendance entre les threads.

➙ On ne peut pas savoir quand chaque thread créé est chargé/exécuté par l'OS

➙ On peut néanmoins utiliser les mécanismes de scheduling des Pthreads.

1.8 Création et terminaison des threads

- Les fonctions :

  pthread_create (thread, attr, start_routine,arg)

  pthread_exit (status) → exit explicite.

  pthread_attr_init (attr) → pour modifier les attributs

  pthread_attr_destroy (attr)

- NB : prendre l'habitude, dans les threads, de "passer la main" avec

  → **sleep(n)** attendre n secondes (1...)

  → **usleep(1..1000000)** micro secondes (usleep(1000000)=sleep(1))

  → **sched_yield()** (ou *pthread_yield()* qui n'est pas std).

1.8.1 *Création des threads : détails*

```
int pthread_create(pthread_t * thread_id,  const pthread_attr_t * attr,
                   void * (routine_associee)(void *),  void *arg) ;
```

➙ Renvoie 0 si la création réussit. Voir le fichier *error.h* pour les différents codes.

- *thread_id* : l'identifiant du thread créé (du type *unsigned long int* défini dans le fichier *bits/pthreadtypes.h*)
  - ➙ Un thread peut demander son propre identifiant par *pthread_self()*.
  - ➙ On compare deux identifiants par *pthread_equal(ID1, ID2)*.
- *void * (routine_associee)(void*)* : pointeur sur la fonction associée au thread.
  Cette fonction devra avoir un seul paramètre du type *void**
- **arg* : pointeur vers l'argument de la routine associée au thread.
  - ➙ Pour passer plusieurs arguments, définir une struct.

- *attr* : NULL par défaut. Sinon, fournir une struct *pthread_attr_t* (type défini dans le fichier *bits/pthreadtypes.h*).

  ➙ Les attributs que l'on peut spécifier par le *struct pthread_attr_t* sont :

  ✔ *detachstate* : joignable ou pas (PTHREAD_CREATE_JOINABLE par défaut)

  ➥ Autre option possible : PTHREAD_CREATE_DETACHED

  ✔ Politique de scheduling pour les aspects temps réel.

  ➥ Une valeur parmi : *PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER*

  ✔ *schedpolicy* : Paramètres de scheduling (par défaut SCHED_OTHER).

  ➙ Exécution préemptée par une plus forte priorité, bloqué ou *yield*.

  ✔ *inheritsched* : (PTHREAD_INHERIT_SCHED par défaut)

  ➙ Paramètres de scheduling hérités du parent.

  ➥ Autre valeur : PTHREAD_EXPLICIT_SCHED

- ✔ *scope* (portée) : (par défaut PTHREAD_SCOPE_PROCESS)
  - ➥ Autre : PTHREAD_SCOPE_SYSTEM (chargé par le kernel)
  - ➙ Si PTHREAD_SCOPE_PROCESS (simple thread utilisateur), le kernel ne voit que le processus principal, pas les threads. Dans ce cas, les threads de l'application sont gérés à l'intérieur du processus (vs. kernel).
- ✔ *stackaddr* : Adresse Pile (NULL) ; voir le fichier *unistd.h* et *bits/posix_opt.h* et le paramètre *PTHREAD_ATTR_STACKADDR*
- ✔ *stacksize* : minimum par défaut (1 MB) : PTHREAD_STACK_SIZE définie dans le fichier *pthread.h*

• les fonctions pthread_attr_getxxx et pthread_attr_setxxx où xxx est une des valeurs ci-dessus sont disponibles pour cet attribut.

• Voir les exemples et détails en section 1.11 page 74

1.8.2 *Terminaison des threads*

Fonction pthread_exit : void pthread_exit(void *retval) ;

➥ *retval* : pointeur sur la valeur renvoyée par le thread.

Cet appel termine (et détruit) le thread ; *pthread_exit* ne retourne donc jamais rien.

➙ Si le thread concerné n'est pas détaché (*detached*), la valeur thread_id et la valeur de retour (retval) peuvent être examinées par un autre thread en utilisant *pthread_join*.

N.B. : le pointeur renvoyé **retval* ne devrait pas être une donnée locale car elle disparaîtra lorsque le thread disparaît.

1.8.3 *Sur la terminaison des threads*

• Un thread peut terminer :

➙ s'il revient de la fonction qui lui est attachée (fin de *main()* pour le thread initial)

➙ s'il appelle *pthread_exit()* (à utiliser à la fin normale du thread)

➙ est annulé (cancel) par un autre via l'appel à *pthread_cancel()* (voir la bibliothèque)

➙ est terminé (comme Tout le programme) par un appel à *exit* ou *exec*.

NB : si *main()* se termine avant la fin de ses threads en exécutant *pthread_exit*, alors les autres threads crées **continuent**.

➙ Mais si *main()* se termine normalement (sans *pthread_exit*) alors les autres threads **terminent** aussi.

• A l'appel de *pthread_exit (status)*, la variable *status* peut être utile à un autre thread qui voudrait joindre (par join) ce thread (voir plus loin).

• *pthread_exit* ne ferme pas les fichiers ouverts.

☞ NB : appeler *pthread_exit* dans main si l'on veut que les autres continuent (le thread de main() reste vivant même si la fonction *main()* est terminée).

☞ Ne pas appeler *pthread_exit* à la fin du *main()* (laisser finir normalement ou exécuter *return(0)*) si l'on veut mettre fin à TOUS.

1.8.4 Exemple (multilingue)

- Création de 5 threads qui chacun affiche un message et termine.

➙ Il n'y a pas de *join* : *main()* n'attend pas la fin des threads ;

➙ Chaque thread (*main()* compris) exécute *pthread_exit()*.

➙ Si *main()* termine en premier, elle laisse les threads finir.

```
/*************************************************************************
* 2010 :  affichage de HELLO dans différentes langues.
* les données sont passées via une structure
* Compiler avec g++ -w -lpthread xx.cpp
*************************************************************************/
#include <pthread.h>
#include <iostream>
#include <string>
#include <stdlib.h>

#define NB_THREADS  8

std::string messages(NB_THREADS);

struct donnees_thread
{
  int id_tache, somme;
  std::string message;
};
```

```
donnees_thread thread_data_array(NB_THREADS);

void *Dis_Bonjour(void *arg)
{
  int id_tache, somme;
  std::string hello_msg;
  struct donnees_thread *mes_donnees;


  // sched_yield(); // = pthread_yield() mais standard. Passer la main
  sleep(1);// mettre ceci pour attendre 1 sec.

  mes_donnees = (struct donnees_thread *) arg;
  id_tache = mes_donnees->id_tache;
  somme = mes_donnees->somme;
  hello_msg = mes_donnees->message; // on récupère l'adresse

  std::cout << "Thread " << (long)id_tache << " : " << hello_msg;
  std::cout << " Somme = " << somme << std::endl;

  pthread_exit(NULL);
}

int main(int argc, char *argv()) {
  pthread_t threads(NB_THREADS);
  int *taskids(NB_THREADS);
  int rc, t, somme;

  somme=0;
  messages(0) = "English: Hello World!";
  messages(1) = "French: Bonjour, le monde!";
  messages(2) = "Spanish: Hola al mundo";
  messages(3) = "Klingon: Nuq neH!";
```

```
  messages(4) = "German: Guten Tag, Welt!";
  messages(5) = "Russian: Zdravstvytye, mir!";
  messages(6) = "Japan: Sekai e konnichiwa!";
  messages(7) = "Latin: Orbis, te saluto!";

  for(t=0;t<NB_THREADS;t++) {
    somme = somme + t;
    thread_data_array(t).id_tache = t;
    thread_data_array(t).somme = somme;
    thread_data_array(t).message =  messages(t);
    std::cout << "Creation du  thread" << t << std::endl;
    rc = pthread_create(&threads(t), NULL, Dis_Bonjour, (void *) &thread_data_array(t));
    if (rc) {
      std::cout << "ERROR; code retour de  pthread_create() est " << rc << std::endl;
      exit(-1);
    }
  }

    pthread_exit(NULL);
}
/* Une trace :
Creation du  thread0
Creation du  thread1
Creation du  thread2
Creation du  thread3
Creation du  thread4
Creation du  thread5
Creation du  thread6
Creation du  thread7
Thread Thread 0 : 1 : English: Hello World!French: Bonjour, le monde! Somme =  Somme = 10
Thread 3 : Klingon: Nuq neH! Somme = 6
Thread 5 : Russian: Zdravstvytye, mir! Somme = 15
Thread 6 : Japan: Sekai e konnichiwa! Somme = 21
Thread 4 : German: Guten Tag, Welt! Somme = 10
```

```
Thread 2 : Spanish: Hola al mundo Somme = 3
Thread 7 : Latin: Orbis, te saluto! Somme = 28
*/
```

Listing 1.8 – hello_arg2.cpp

1.8.5 *Notes sur la terminaison*

1. par un appel explicite à *pthread_exit(n)*

   ➙ En l'absence de join(), si cet appel a lieu par *main()*, les threads continuent .

2. en laissant la fonction terminer normalement (sans valeur de retour)

   ➙ En l'absence de join(), dans le cas de *main()*, les threads terminent aussi.

3. par un appel à *exit()* qui terminera TOUT le processus (tous terminent).

   ➙ *exit(0)* n'est pas équivalent à *pthread_exit(0)*.

- Un exemple :

➙ Déplacer les parties mises en commentaire pour voir les différentes manières de terminer un ensemble de threads. ../..

```c
// 2010 : terminaison des threads (via main)
// Compil : gcc terminaison_de_main.c -lpthread
// Chaque thread reçoit un entier 0..NB_threads et l'utilise par créer une lettre (de 'A'..'Z')
// Affiche sa lettre et l'indice de son itération qui va de 0 à 3*(valeur reçu)

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

#define NB_threads 5

void * ma_fonction(void * ptr);
int main()
{   int i, rc;
    pthread_t tab_id1(NB_threads);
    int tab_valeurs(NB_threads);

    printf("Créations des threads \n");
    for (i=1; i< NB_threads; i++){
        tab_valeurs(i)=i;
        rc = pthread_create(&tab_id1(i) , NULL, ma_fonction, (void *) &tab_valeurs(i));
}

// CAS 1 :
//     printf("Sans Join, Main termine normalement, sans rien retourner. \n");
//     printf("Les threads terminent de suite\n");


// CAS 2:
//     printf("Sans Join, Main termine avec exit(0) \n");
//     printf("Les threads terminent de suite\n");
//     exit(0);
```

```
// CAS 3 :
//      printf("Sans Join, Main termine avec return(0) \n");
//      printf("Les threads terminent de suite\n");
//      return(1);

// CAS 4 :
     printf("Sans Join, Main termine avec  pthread_exit(0) \n");
     printf("Les threads continuent\n");
     pthread_exit(0);     // Laisser finir
}

void * ma_fonction(void * ptr) {
     int valeur = *((int *) ptr);
     valeur ++;  // pour commencer à 1;
     int i;
     char mon_signe= valeur+'A';      // donnera de 'A'.. 'Z'
     printf("%c> Je compte jusqu'au %d\n", mon_signe, 3*valeur);
     for (i=0; i<valeur*3; i++){
          printf("%c-%d\n", mon_signe,i);
          usleep(500000);
     }
}

/* Trace :
Créations des threads
C> Je compte jusqu'au 6
C-0
D> Je compte jusqu'au 9
D-0
E> Je compte jusqu'au 12
Sans Join, Main termine avec  pthread_exit(0)
Les threads continuent
E-0
F> Je compte jusqu'au 15
```

```
F-0
C-1
D-1
F-1
E-1
...
Suite des affichages..
*/
```

Listing 1.9 – terminaison_de_main.c

1.9 Joindre ou détachement de threads

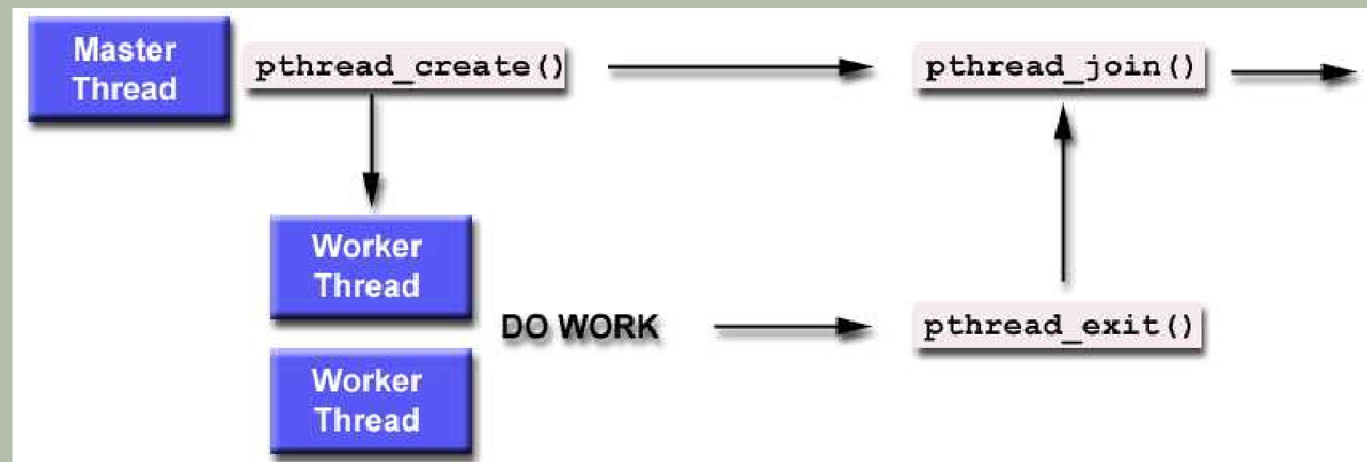
- La jointure (*join*) est une manière de synchronisation des threads.

➙ *pthread_join (threadid,status)*

➙ *pthread_detach (threadid,status)*

➙ *pthread_attr_setdetachstate (attr,detachstate)*

➙ *pthread_attr_getdetachstate (attr,detachstate)*



• *pthread_join (threadid,status)* bloque l'appelant A jusqu'à la terminaison du thread B identifié par threadid (une sorte de synchronisation).

➙ Ex : A envoie B faire des courses et guète son retour.

• On peut accéder au code de terminaison de B s'il a exécuté *pthread_exit()*.

➙ B est unique pour cet appel à *join* : A on ne peut pas attendre plusieurs threads (Bs) sur un même *join* (il faudrait une *barrière*). Voir aussi les autres manières de synchro (mutex et condition var) plus loin.

Peut-on être **joignable** ?

• Attribut joignable ou détaché à la création du thread.

➙ Seuls les threads créés "joignables" sont joignables !

➙ Si un thread est créé détaché, jamais on pourra le joindre.

Join : le standard POSIX préconise de procéder par :

➙ Créer les threads tous joignables.

➙ Utiliser l'attribut *attr* lors de l'appel de *pthread_create()* en

- déclarer une donnée du type *pthread_attr_t*
- l'initialiser à l'aide de *pthread_attr_init()*
- assigner une valeur (joignable) avec *pthread_attr_setdetachstate()*
- penser à libérer l'espace alloué à l'attribut*pthread_attr_destroy()*

- On peut rendre un thread détaché par *pthread_detach()* même s'il a été crée joignable.
- Le seul argument en faveur d'une création détachée est un petit gain de ressource.

➙ Mais il vaut mieux créer les thread joignables par défaut : on peut toujours les détacher alors que l'inverse n'est pas possible.

1.9.1 Exemple-1 (simple)

- Lire les commentaires !

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

void *affiche(void * voidparam)
{  int v= (int*) voidparam;      // vite vite, récupérer le paramètre avant son changement (voir risque) !
   printf("  affiche %d\n", v);
   sleep(2);
   printf("Le thread %d termine\n", v);
   return NULL;     //  (= pthread_exit(NULL))
}

int main(void)
{  pthread_t tid(10);
   int code, i;

   // creation des threads
   for(i= 0; i < 10; i++) {
      code= pthread_create(&tid(i), NULL, affiche, (void *) i); // on prend un RISQUE ! Lequel ?
      if(code < 0) perror("creation du thread");
   }

   // attente des threads
   for(i= 0; i < 10; i++) code= pthread_join(tid(i), NULL);

   return 0;
}
```

1.9.2 Exemple-2

- Après création, on modifie l'attribut (joignable) des threads.
- Attente de la fin des threads par *join*.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     4

void *BusyWork(void *t)
{   int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)
    {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
```

```
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
       printf("Main: creating thread %ld\n", t);
       rc = pthread_create(&thread(t), &attr, BusyWork, (void *)t);
       if (rc) {
          printf("ERROR; return code from pthread_create()
                 is %d\n", rc);
          exit(-1);
          }
       }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for(t=0; t<NUM_THREADS; t++) {
       rc = pthread_join(thread(t), &status);
       if (rc) {
          printf("ERROR; return code from pthread_join()
                 is %d\n", rc);
          exit(-1);
          }
       printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
       }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

```
//=======> les sorties :
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = -3.153838e+06
Thread 0 done. Result = -3.153838e+06
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Thread 3 done. Result = -3.153838e+06
Thread 2 done. Result = -3.153838e+06
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

Remarques :

- Un thread peut être détaché à l'aide de son identifiant de la création via l'appel *pthread_detach(ID)*.

- Un thread peut se détacher par *pthread_detach(pthread_self())* .

- Un appel à *pthread_join()* rejoint le thread et le détache (libération des ressources).

- Rendre un thread détaché à la création :

```
pthread_attr_t    attr ;
pthread_attr_init(&attr) ;
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

pthread_create(&ID, &attr, .... ) ;
```

1.9.3 Exemple plus complet : alarme

- On boucle à l'infini :

  On lit *nb* secondes et un *message* puis on crée un thread en lui passant ces 2 paramètres. Chaque thread attend *nb* secondes puis affiche son message et termine.

- Du fait de se détacher, le thread n'est plus joignable (le *main* ne l'attend pas).
- Compiler (avec *gcc -pthread alarm_thread.c*) puis lancer.

  ➙ A chaque invite, donner *nb* secondes et un *message*.

  ➙ Selon *nb* secondes donné, on verra les messages s'afficher dans un ordre quelconque. Par exemple, si on donne :

  ➙ *1 message1*

  ➙ *2 message2*

  ➙ *1 message3*

On verra *message1* et *message3* s'afficher avant *message2*. Couper avec *CTRL-C*.

```
/* 2010
* alarm_thread.c
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _alarme {
  int         sec;
  char        message[64];
} Type_alarme;

void *alarm_thread (void *arg)
{
  Type_alarme *alarme = (Type_alarme*)arg;
  int status;

  status = pthread_detach (pthread_self ());
  if (status != 0) {perror("Erreur detach_thread");;exit(1);}
  sleep (alarme->sec);
  printf ("(%d) %s\n", alarme->sec, alarme->message);
  free (alarme);
  return NULL;
}

int main (int argc, char *argv[])
{
  int status;
  char line[128];
  Type_alarme *alarme;
  pthread_t thread;
```

```
  printf("\nDevant le prompte 'Alarme ?>', donner un entier (nb. sec d'attente) et un message\n");
  printf(" ... Après nb.sec, le message sera affiché \n");
  while (1) {
    printf ("Alarme ?> ");
    if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
    if (strlen (line) <= 1) continue;
        alarme = (Type_alarme*)malloc (sizeof (Type_alarme));
    if (alarme == NULL) {perror("Err Allocation alarme");exit(1);}

        /* Lecture du clavier :
    * Le message consiste en 1..64 caractères ou un retour chariot
    */

    if (sscanf (line, "%d %64[^\n]", &alarme->sec, alarme->message) < 2) {
      fprintf (stderr, "Bad cmd ! \n");
      free (alarme);
    } else {
      status = pthread_create ( &thread, NULL, alarm_thread, alarme);
      if (status != 0) {perror("Err Creation thread alarme");exit(1);}
    }
  }
}
/* TRACE :
Devant le prompte 'Alarme ?>', donner un entier (nb. sec d'attente) et un message
... Après nb.sec, le message sera affiché
Alarme ?> 4 quatre secondes
Alarme ?> 1 sec
Alarme ?> (1) sec
(4) quatre secondes
Alarme ?> ...
*/
```

Listing 1.10 – alarm_thread.c

1.10 Gestion de la pile

• POSIX n'impose pas une taille de pile pour les threads (chaque plate forme décide).

• Le débordement de la pile est un problème important. Il vaut donc mieux ne pas s'en remettre aux valeurs par défaut qui varient selon les architectures :

➥ imposer une taille adéquate par *pthread_attr_setstacksize*.

➙ *pthread_attr_getstacksize (attr, stacksize)*

➙ *pthread_attr_setstacksize (attr, stacksize)*

➙ *pthread_attr_getstackaddr (attr, stackaddr)*

➙ *pthread_attr_setstackaddr (attr, stackaddr)*

• On peut implanter une pile de thread à un endroit précise de la mémoire avec *pthread_attr_setstackaddr (attr, stackaddr)*.

1.10.1 Un exemple

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
      A[i][j] = ((i*j)/3.452) + (N-i);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc;
   long t;
```

```
    pthread_attr_init(&attr);
    pthread_attr_getstacksize (&attr, &stacksize);
    printf("Default stack size = %li\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li\n",stacksize);
    pthread_attr_setstacksize (&attr, stacksize);
    printf("Creating threads with stack size = %li bytes\n",stacksize);
    for(t=0; t<NTHREADS; t++){
        rc = pthread_create(&threads(t), &attr, dowork, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    printf("Created %ld threads.\n", t);
    pthread_exit(NULL);
}
```

- Voir aussi la section suivante.
- NB : sous Linux, la commande **ulimit -s** donne la taille par défaut allouée à la pile.

1.11 Utilisation des attributs de thread

Le format général :

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void * arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* call an appropriate functions to alter a default value */
ret = pthread_attr_*(&tattr, SOME_ATRIBUTE_VALUE_PARAMETER);

/* create the thread */
ret = pthread_create(&tid, &tattr, start_routine, arg);
  ....
  ....
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

- La fonction *ret = pthread_attr_init(&tattr) ;* initialise le thread par les valeurs par défaut.

→ Détruire l'attribut après l'avoir utilisé pour récupérer la place. ../..

Les valeurs par défaut des champs de l'attribut :

scope : PTHREAD_SCOPE_PROCESS

detachstate : PTHREAD_CREATE_JOINABLE

stackaddr : NULL

stacksize : 1 MB

inheritsched : PTHREAD_INHERIT_SCHED

schedpolicy : SCHED_OTHER

schedparam : priorité (entier), voir l'exemple plus loin.

Exemple d'utilisation pour la valeur **detachstate**

- Valeurs possibles : PTHREAD_CREATE_JOINABLE et PTHREAD_CREATE_DETACHED.

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void * arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);

...........

int detachstate;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);

/* renvoie 0 si tout va bien. */
```

Exemple d'utilisation pour la valeur **scope**

- Valeurs possibles : PTHREAD_SCOPE_SYSTEM et PTHREAD_SCOPE_PROCESS.

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void *start_routine;
void * arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
/* BOUND behavior */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);

............


int scope;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

Exemple d'utilisation pour la valeur **schedpolicy**

- Les valeurs possibles (POSIX) sont : SCHED_FIFO (first-in-first-out), SCHED_RR (round-robin) ou SCHED_OTHER (selon l'implantation).

➙ SCHED_FIFO et SCHED_RR sont optionnelles sous POSIX et sont uniquement offertes pour les threads temps réels (avec deadline).

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);

....

pthread_attr_getschedpolicy(... ) ;
```

- La fonction *pthread_attr_getschedpolicy()* renvoie la valeur de *schedpolicy*. Mais, en général, la valeur renvoyée est la valeur par défaut *SCHED_OTHER*.

Exemple d'utilisation pour la valeur **inheritsched**

- Les valeurs possibles sont : *PTHREAD_INHERIT_SCHED* (default). Cette valeur (celle du parent) inhibe les valeurs données pour scheduling dans *pthread_create()*.

➙ Par contre, si la valeur *PTHREAD_EXPLICIT_SCHED* est utilisée, les valeurs spécifiées dans *pthread_create()* sont utilisées.

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
...

/* format getinheritsched */
pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit) ;
```

Exemple d'utilisation pour la valeur **schedparam**

- Les paramètres de scheduling sont définis par la struct sched_param.
- Seule la valeur de la **priorité** via **sched_param.sched_priority** est implantée.

➙ C'est un entier où la plus grande valeur = la plus grande priorité.

- Les nouveaux threads crées auront cette priorité.
- La fonction pthread_attr_setschedparam() permet de fournir cette valeur

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;


/* set the priority; others are unchanged */
newprio = 30;
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);

.........
/* format de getschedparam pour obtenir la priorité */

pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param)
```

Exemple d'utilisation pour la valeur **stacksize**

• On utilise cet attribut pour obtenir une taille plus grande ou bien pour réduire la taille par défaut (1 MB) allouée.

• Cette valeur tient compte de l'espace nécessaire à l'appel, variables locales et autre structure d'information.

• Quand cet attribut est précisé, le thread associé **doit être** PTHREAD_CREATE_JOINABLE.

• La place sera libérée après *pthread_join()* (la fin du thread).

• Habituellement la taille du stack de 1 méga octets par défaut suffit.

• On peut obtenir la taille allouée pour une implantation via la macro PTHREAD_STACK_MIN (définie dans pthread.h) qui renvoie la taille min pour une fonction (du thread) dont le corps = NULL.

➙ Cette valeur représente le strict minimum pour un thread (dont la fonction est vide !)

• Exemple de modification de la taille (en octets).

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;


/* setting a new size */
stacksize = (PTHREAD_STACK_MIN + 0x4000);
ret = pthread_attr_setstacksize(&tattr, stacksize);
...........

/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &stacksize);
```

☞ Attention : pthread_attr_getstacksize renvoie la taille minimum, pas la taille actuelle.

• On peut également spécifier l'adresse de base du stack par la fonction pthread_attr_setstacka

➙ La valeur par défaut est NULL (pas de déplacement perso de la base).

• L'exemple suivant montre un cas de manipulation de la taille et de l'adresse de base de la pile.

• Exemple de modification de la taille et de la base du stack :

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);
/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);
/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);
/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

• La fonction *pthread_attr_getstackaddr(pthread_attr_t *tattr, void * *stackaddr)* permet d'obtenir l'adresse de base de la pile du thread courant.

1.12 Gestion dynamique des threads

➙ *pthread_self ()* : renvoie l'Id (unique) du thread appelant.

➙ *pthread_equal (thread1,thread2)* : compare 2 Ids et renvoie 0 si différents.

➥ Ne pas utiliser l'opérateur '==' dans les comparaisons d'Id d'un thread (aucune garantie).

➙ *pthread_once (once_control, init_routine)* : exécute la fonction *init_routine* (qui est une initialisation) une seule fois et sans paramètre. Les autres tentatives d'exécution seront vaines.

➥ Le paramètre *once_control* est pour la synchronisation ; on doit l'initialiser avant l'appel de *pthread_once* .

➙ Par exemple : *pthread_once_t once_control* = *PTHREAD_ONCE_INIT ;*

1.13 Mutex

1.13.1 Un exemple simple

On se donne 3 threads (créés dans cet ordre) et deux variables globales X et Y :

- ➙ thread 1 fera X = 2
- ➙ thread 2 : X = 3
- ➙ thread 3 : Y = X

• Selon l'ordre aléatoire d'exécution (scheduling) des threads, on peut obtenir différentes valeurs dans Y. Par exemple :

- ➙ l'ordre thread 3, thread 1, thread 2 donne *X = 3 et Y= 0*
- ➙ l'ordre thread 2, thread 1, thread 3 donne *X = 2 et Y = 2*
- ➙ l'ordre 1, 2, 3 donne *X = 3 et Y = 2*

../..

```
// compiler avec    g++ -w -lpthread xx.cpp

#include <iostream>
#include <stdlib.h>
#include <pthread.h>
using namespace std;

void *function1( void *ptr );      // fera X_globale=2
void *function2( void *ptr );    // fera X_globale=2
void *function3( void *ptr );    // fera X_globale=2

int X_globale, Y_globale;

int main()
{
     pthread_t thread1, thread2, thread3;
     int  iret;

     iret = pthread_create( &thread1, NULL, function1, NULL);
     iret = pthread_create( &thread2, NULL, function2, NULL);
     iret = pthread_create( &thread3, NULL, function3, NULL);

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     pthread_join( thread3, NULL);

    cout << "On a X_globale = " << X_globale << " et Y_globale = " << Y_globale<< endl;
     exit(0);
}

void *function1( void *ptr )
{   cout << "thread 1" << endl;
   X_globale=2;
```

```
    return NULL;
}

void *function2( void *ptr )
{    cout << "thread 2" << endl;
    X_globale=3;
    return NULL;
}

void *function3( void *ptr )
{    cout << "thread 3" << endl;
    Y_globale=X_globale;
    return NULL;
}

/* plusieurs traces :

thread 3
thread 1
thread 2
On a X_globale = 3 et Y_globale = 0

thread 2
thread 1
thread 3
On a X_globale = 2 et Y_globale = 2

ZZ : ici, un \n a été entrelacé
thread 1
thread 2thread 3

On a X_globale = 3 et Y_globale = 2
 */
```

1.14 Non déterminisme et concurrence

- Le non déterminisme peut venir de plusieurs causes :

➙ ordre imprévisible d'avancement des threads sur un seul processeur

➙ ordre imprévisible d'avancement sur différents processeurs

➙ Utilisation de mécanisme non déterministes dans un programme qui fera un choix aléatoire en cas de multiples possibilités (e.g. un tirage aléatoire dans un parcours de graphe).

- Le non déterminisme n'est pas forcément une erreur.

➙ Les threads modélisent les actions du monde réel dans une nature par définition non déterministe.

- Le non déterminisme peut être efficace.

➙ Exemple : 2 ouvriers (robots) travaillant sur une table à une vitesse variable dont le

1er prépare des pièces et dont le 2e les assemble.

➥ Le 2e peut assembler dès que les pièces nécessaires sont prêtes.

➥ Un ordre séquentiel est plus simple mais moins efficace (temps) et moins flexible.

- Le non déterminisme et la concurrence sont des concepts liés.

➙ La concurrence peut être modélisée par le non déterminisme.

➙ Le non déterminisme est inhérent à la concurrence.

➙ Dans l'exemple des 2 robots, l'utilisation d'un tampon (buffer) permet de rendre le travail flexible ; à condition d'organiser la gestion de ce tampon (empty, full, ...)

- Le modèle client serveur est un exemple classique.
- Il est difficile de déverminer et de tester des programmes (threads) concurrents.
- Il est de la responsabilité du concepteur de déterminer les actions *atomiques* ou *insécables* (les sections critiques) et celles qui doivent avoir lieu en séquence.

1.14.1 *Fonctionnement d'un Mutex*

- Les variables mutex pour la synchronisation et protection des données.
- Une variable mutex joue le rôle d'un verrou.
- En accès mutex : si plusieurs threads tentent d'y accéder, seul un thread pourra le faire à un instant donné ; les autres attendront la libération de la variable.
- Un exemple de *race condition* justifiant les verrous (ici, pas de synchronisation) :

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

➙ Il faut synchroniser l'accès à *Balance* (i.e. section critique=SC).

- On doit penser à créer la variable mutex, organiser l'accès aux Sc en verrouillant / déverrouillant la variable et enfin, libérer l'espace alloué.
- Une demande de verrou est **bloquant** mais on peut faire un appel non bloquant (*trylock* au lieu de *lock*).

Les fonction de gestion des Mutex :

➙ *pthread_mutex_init (mutex,attr)*

➙ *pthread_mutex_destroy (mutex)*

➙ *pthread_mutexattr_init (attr)*

➙ *pthread_mutexattr_destroy (attr)*

- On peut aussi initialiser un mutex à la création :

  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER ;

- *attr* doit être du type *pthread_mutexattr_t*.

  ➙ Si l'on spécifie *NULL* à sa place, les valeurs par défaut seront utilisées.

- Il y a 3 valeurs pour l'attribut *attr* (à vérifier sur votre plate forme) :

  ➙ *Protocol* : spécifie le protocole utilisé pour le problème d'inversion des priorités

  ➙ *Prioceiling* : spécifie la priorité minimum du mutex

  ➙ *Process-Shared* : spécifie le partage du mutex.

- N'utiliser *pthread_mutexattr_destroy (attr)* que si plus aucun thread ne l'utilise.

Les fonction de verrouillage/déverrouillage de Mutex :

➙ *pthread_mutex_lock (mutex)* : demande bloquante.

➙ *pthread_mutex_trylock (mutex)* : demande non bloquante.

➙ *pthread_mutex_unlock (mutex)* : déverrouillage si mutex possédé par l'appelant.

➥ Erreur si le mutex était déjà déverrouille ou si l'appelant ne possède pas le verrou.

- Un exemple qui peut poser problème :

Thread 1	Thread 2	Thread 3
Lock	*Lock*	
A = 2	*A = A+1*	*A = A*B*
Unlock	*Unlock*	

➙ Résultat aléatoire (sauf si l'on utilise les stratégie de scheduling / prio du système).

1.14.2 Exemple-1 simple

- Deux threads sont créés. Chacun fait +1 sur un compteur commun (*counter*).
- Les threads affichent la valeur de ce compteur.

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {printf("Thread creation failed: %d\n", rc1);}

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   { printf("Thread creation failed: %d\n", rc2);  }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */
```

```
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    exit(0);
}

void *functionC()
{
    pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
    pthread_mutex_unlock( &mutex1 );
}
```

• Un mutex initialisé par *PTHREAD_MUTEX_INITIALIZER* est statique (ne passe par par la fonction d'initialisation *pthread_mutex_init()*) et n'a pas besoin d'être détruit par *pthread_mutex_destroy()*.

1.14.3 Initialisation de Mutex

Les mutex peuvent être initialisés. Dans l'exemple suivant, l'initialisation a lieu lors de la création d'un thread qui utilisera un Mutex.

Exemple d'initialisation non statique :

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;


void foo_init()
{
     pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
     pthread_once(&foo_once, foo_init);
     pthread_mutex_lock(&foo_mutex);
    /* Do work. */
     pthread_mutex_unlock(&foo_mutex);
}
....
```

- La même initialisation de manière statique.

Exemple d'initialisation statique :

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

 void foo()
 {
      pthread_mutex_lock(&foo_mutex);
     /* Do work. */
      pthread_mutex_unlock(&foo_mutex);
 }
```

1.14.4 Exemple-2

- Avec plusieurs threads. Voir la boucle d'attente (*join*) dans *main()*.
- Chaque thread affiche son propre ID et fait +1 sur le compteur commun.
- Cette fois, c'est *main()* qui affiche la valeur finale du compteur.

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

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{   pthread_t thread_id(NTHREADS);
    int i, j;

    for(i=0; i < NTHREADS; i++) { pthread_create( &thread_id(i), NULL, thread_function, NULL ); }

    for(j=0; j < NTHREADS; j++)  {pthread_join( thread_id(j), NULL); }

    /* Now that all threads are complete I can print the final result.     */
    /* Without the join I could be printing a value before all the threads */
    /* have been completed.                                                */

    printf("Final counter value: %d\n", counter);
}
```

```
void *thread_function(void *dummyPtr)
{    printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}

//   Résultats:
     Thread number 1026
     Thread number 2051
     Thread number 3076
     Thread number 4101
          .....
     Thread number 9226
     Thread number 10251
     Final counter value: 10
```

Listing 1.11 – Ex-mutex-join.c

- Compilation : *gcc -lpthread Ex-mutex-join.c*

  Puis exécuter par : *./a.out*

Rappel : Un mutex peut être simplement initialisé par *PTHREAD_MUTEX_INITIALIZER*.

1.14.5 Exemple-3 : estimation parallèle de PI

- Estimation de la valeur de PI par une estimation de $\pi = \int_0^1 \frac{4}{1+x^2}\, dx$
- On découpe l'intervalle 0-1 en 1000000 rectangles et on charge 4 threads de calculer chacun 1/4 de la région sous la fonction.
- Chaque thread calcule sa part et ajoute celle-ci à la somme en prenant la précaution de vérrouiller la somme par un mutex.
- La fonction main récupère les résultats à la fin.

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

#define nb_rectangles 1000000    // découpage en 1000000 battons
#define NB_THREADS 4

double somme = 0.0;       // estimation de la somme

pthread_mutex_t verrou;

void *threadFunction(void *pArg)
{
    int mon_numero = *((int *)pArg);
```

```
    double batton = 0.0, largeur = 1.0 / nb_rectangles, x;
    for (int i = mon_numero; i < nb_rectangles; i += NB_THREADS)
    {
        x = (i + 0.5f) / nb_rectangles;
        batton += 4.0f / (1.0f + x*x);
    }
    pthread_mutex_lock(&verrou);
    somme += batton * largeur;
    pthread_mutex_unlock(&verrou);
}

void main()
{
    pthread_t id_threads(NB_THREADS);
    int tNum(NB_THREADS);
    pthread_mutex_init(&verrou, NULL);
    for (int i = 0; i < NB_THREADS; i++ ) {
        tNum(i) = i;
        pthread_create(&id_threads(i), NULL, threadFunction,(void *)&tNum(i));
    }
    for ( int j=0; j<NB_THREADS; ++j ) {
        pthread_join(id_threads(j), NULL);
    }
    pthread_mutex_destroy(&verrou);
    printf("Valeur calculée de Pi: %f\n", somme);
}
```

1.14.6 Exemple-4 : produit vectoriel parallèle

• Réalisation du produit vectoriel (*dot product*).

• La donnée est globale et chaque thread s'occupe d'un partie.

• La *main()* attend la fin de tous puis affiche le résultat (à comparer avec la version sans thread = sériale).

```
/******************************************************************************
* FILE: dotprod_mutex.c
* DESCRIPTION:
*   This example program illustrates the use of mutex variables
*   in a threads program. This version was obtained by modifying the
*   serial version of the program (dotprod_serial.c) which performs a
*   dot product. The main data is made available to all threads through
*   a globally accessible  structure. Each thread works on a different
*   part of the data. The main thread waits for all the threads to complete
*   their computations, and then it prints the resulting sum.
* SOURCE: Vijay Sonnad, IBM
* LAST REVISED: 01/29/09 Blaise Barney
******************************************************************************/

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

```
/* The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure.  This structure is
unchanged from the sequential version.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd(NUMTHRDS);
   pthread_mutex_t mutexsum;


/*
The function dotprod is activated when the thread is created.
As before, all input to this routine is obtained from a structure
of type DOTDATA and all output from this function is written into
this structure. The benefit of this approach is apparent for the
multi-threaded program: when a thread is created we pass a single
argument to the activated function - typically this argument
is a thread number. All  the other information required by the
function is accessed from the globally accessible structure.
*/
```

```
void *dotprod(void *arg)
{

/* Define and use local variables for convenience */

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

/*
Perform the dot product and assign result
to the appropriate variable in the structure.
*/

   mysum = 0;
   for (i=start; i<end ; i++)
     {
       mysum += (x(i) * y(i));
     }

/*
Lock a mutex prior to updating the value in the shared
structure, and unlock it upon updating.
*/
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);
```

```
   pthread_exit((void*) 0);
}

/*
The main program creates threads which do all the work and then
print out result upon completion. Before creating the threads,
The input data is created. Since all threads update a shared structure, we
need a mutex for mutual exclusion. The main thread needs to wait for
all threads to complete, it waits for each one of the threads. We specify
a thread attribute value that allow the main thread to join with the
threads it creates. Note also that we free up handles when they are
no longer needed.
*/

int main (int argc, char *argv())
{
long i;
double *a, *b;
void *status;
pthread_attr_t attr;

/* Assign storage and initialize values */

a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

for (i=0; i<VECLEN*NUMTHRDS; i++) {
  a(i)=1;
  b(i)=a(i);
  }

dotstr.veclen = VECLEN;
dotstr.a = a;
```

```
dotstr.b = b;
dotstr.sum=0;

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  {
  /* Each thread works on a different set of data.
   * The offset is specified by 'i'. The size of
   * the data for each thread is indicated by VECLEN.
   */
   pthread_create(&callThd(i), &attr, dotprod, (void *)i);
   }

pthread_attr_destroy(&attr);
/* Wait on the other threads */

for(i=0;i<NUMTHRDS;i++) {
  pthread_join(callThd(i), &status);
  }
/* After joining, print out the results and cleanup */

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

1.15 Problème d'interblocage

- Une difficulté lors de l'utilisation des mutex est d'éviter les interblocages (deadlocks).

- Ce cas de figure se produit, par exemple, si un thread tente d'acquérir un mutex détenu par un autre thread lui-même en attente.

- Exemple d'interblocage :

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

#define NB_THREAD 5


pthread_mutex_t mutex1=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2=PTHREAD_MUTEX_INITIALIZER;

void *fn_thread(void *arg);

int main()
{
  pthread_t thread;

  pthread_mutex_lock(&mutex1);
  if ( pthread_create(&thread, NULL, fn_thread, NULL) )
```

```
  {
    fprintf(stderr, "Erreur lors de pthread_create\n");
    return EXIT_FAILURE;
  }
  sleep(1);  //Permet d'attendre le blocage de mutex2 par le thread
  pthread_mutex_lock(&mutex2);
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return EXIT_SUCCESS;
}

void *fn_thread(void *arg)
{
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
  pthread_exit((void *)0);
}

.....
```

1.16 Schéma Lecteurs / Rédacteurs

• Il s'agit d'une classe de problèmes de synchronisation *à la Client-Serveur* appelée également *Producteur-Consommateur*.

➙ Les rédacteurs produisent (écrivent) et les lecteurs consomment (lisent).

• Différents variations existent :

➙ On peut avoir un seul livre à écrire, relu par des lecteurs : les lecteurs doivent-ils attendre la fin d'une rédaction.

➙ Les lectures peuvent-ils avoir lieu en parallèle ou une par une.

➙ On peut avoir plusieurs rédacteurs.

➙ Le nombre d'exemplaires du livre peut être limité : les lecteurs doivent gérer cela.

➙ Que faire si une lecture est en cours et un rédacteur arrive ? ...

1.16.1 L/R Sans synchronisation

- Aucune garantie sur la validité des données modifiées.
- Le rédacteur modifie les données à son rythme et le lecteur tente de lire ces données.
- D'une exécution à l'autre, les résultats diffèrent.

```
/* 2010 : Lecteur / Rédacteur de base :
Cas d'une variable partagée NON PROTÉGÉE qui est modifiée par une tâche, et lue par une autre
*/

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

struct Donnee_partagee {int donnees;};

struct donnees
{ /* paramètres */
  int nb;
  char const *sid;

  /* contexte partagé : L et R partagent cette même donnée via son adresse*/
  struct Donnee_partagee *psh;
};

static void *redacteur (void *p)
{if (p != NULL)
  {
    /* recuperer le contexte applicatif */
```

```
    struct donnees *p_data = p;
    int i;

    for (i = 0; i < p_data->nb; i++)
    { int x = p_data->psh->donnees;
      x++;
      p_data->psh->donnees = x;

      printf ("'%s' (%d) donnees <- %d\n", p_data->sid, i, p_data->psh->donnees);
    }
  }
    return NULL;
}

static void *lecteur (void *p)
{ if (p != NULL)
  {
    /* recuperer le contexte applicatif */
    struct donnees *p_data = p;
    int i;

    for (i = 0; i < p_data->nb; i++)
    {
      printf ("'%s' (%d) donnees == %d\n", p_data->sid, i, p_data->psh->donnees);
    }
  }
    return NULL;
}

int main (void)
{
  pthread_t id_redacteur;
  pthread_t id_lecteur;
```

```
    struct Donnee_partagee sh ={0};
    struct donnees data_redact ={5, "Rédacteur",&sh};
    struct donnees data_lect ={7,"Lecteur",&sh};

    puts ("Début main");
    pthread_create (&id_redacteur, NULL, redacteur, &data_redact);
    pthread_create (&id_lecteur, NULL, lecteur, &data_lect);

    pthread_join (id_redacteur, NULL);
    pthread_join (id_lecteur, NULL);
    puts ("Fin main");
    return 0;
}

/* Une Trace :
Début main
'Rédacteur' (0) donnees <- 1
'Rédacteur' (1) donnees <- 2
'Lecteur' (0) donnees == 1
'Lecteur' (1) donnees == 3
'Lecteur' (2) donnees == 3
'Lecteur' (3) donnees == 3
'Lecteur' (4) donnees == 3
'Lecteur' (5) donnees == 3
'Rédacteur' (2) donnees <- 3
'Rédacteur' (3) donnees <- 4
'Rédacteur' (4) donnees <- 5
'Lecteur' (6) donnees == 3
Fin main
```

Listing 1.12 – lect-redac1.c

1.16.2 Avec Mutex : principe + exercice

Principe : synchronisation du schéma lecteur / rédacteur

```
*** thread lecteur
   pthread_mutex_lock(&mutex_lecteurs);

   nombre de lecteurs ++
   if(nombre de lecteurs == 1) pthread_mutex_lock(&mutex_redacteur);
   pthread_mutex_unlock(&mutex_lecteurs);

   // lire les données
   { ... }

   pthread_mutex_lock(&mutex_lecteurs);

   nombre de lecteurs --
   if(nombre de lecteurs==0) pthread_mutex_unlock(&mutex_redacteur);

   pthread_mutex_unlock(&mutex_lecteurs);

*** thread rédacteur
   pthread_mutex_lock(&mutex_redacteur);

   // modifier les données
   { ... }

   pthread_mutex_unlock(&mutex_redacteur);
```

- NB : cette solution bloque les rédacteurs en faveur des lecteurs ...

Exercice : coder cet exemple en C++.

1.17 Variables de condition

- Un autre moyen de synchronisation.

- Le principe de synchronisation est ici basé sur le signalement d'un événement (= la réalisation d'une condition

➙ La réalisation d'une condition peut également refléter un changement d'état/valeur de certaines données.

- Par exemple, un thread peut signaler l'affaiblissement du niveau d'une batterie (c'est la condition) et préciser également le niveau actuel (c'est la donnée).

➙ A l'autre bout, un autre thread attend ce signalement pour entreprendre la recharge.

- Pour réaliser cet effet soi même, il faut gérer une/des Sc avec des files d'attente ; vérifier sans cesse si une valeur est atteinte et/ou une condition est remplie... :

➥ c'est lourd et en plus, c'est de l'attente **active** (consomme des ressources) !

• Le mécanisme de condition permet de procéder à une attente **passive** : on ne consomme rien pendant que l'on attend un événement ; on sera réveillé.

➙ Comparer : surveiller l'arrivée des invités ou installer une sonnette !

• Une variable de condition est toujours utilisée avec un Mutex.

➙ Car un thread peut se préparer à attendre sur une condition alors qu'un autre thread signale cette même condition avant que le 1er ait débuté son attente.

- Un schéma classique d'utilisation (**main()**) (non unique) :

➙ Créer les données globales qui nécessitent une synchronisation (e.g. . **niveau** gradué de la batterie).

➙ Créer/initialiser les variables de **condition** (e.g. . *changement* état de la batterie
→ faible / rechargé / ...).

➙ Créer/initialiser un **Mutex** de protection de la condition ; ...
puis créer les threads concernés.

- Chacun des threads (par exemple Thread **Je_depanne**) procédera par :

→ Vaquer ... (mettre son bleu !)

→ Verrouiller **Mutex** et appeler *pthread_cond_wait()* pour se mettre en attente (wait libère automatiquement Mutex).

→ Quand on sera réveillé (par signal), Mutex sera automatiquement (re) verrouille.

→ Traiter l'événement : consulter le niveau gradué de la batterie et décider.

→ Déverrouiller Mutex explicitement (avait été verrouille au réveil)

→ Continuer .... recommencer, ...

- Un thread **Je_previens** concurrent aura :

➙ Vaquer ... puis Verrouiller le Mutex

➙ Changer/constater la valeur graduée de la batterie (variable globale puisque accessible par plusieurs threads).

➙ Vérifier si elle mérite d'être signalée (signal attendu par *Je_depanne*)

➙ Si elle est à la valeur attendue, signaler (= réveiller) *Je_depanne*

➙ Déverrouiller Mutex

➙ Continuer.

- A la suite de ces séquences, main() peut continuer/joindre/...

1.17.1 Création et destruction des VarConds

➙ *pthread_cond_init (condition,attr)*

➙ *pthread_cond_destroy (condition)*

➙ *pthread_condattr_init (attr)*

➙ *pthread_condattr_destroy (attr)*

• Les Variables de Condition sont déclarées du type *pthread_cond_t* puis initialisées.

➙ Initialisation à la volée : *pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER ;*

Ou par *pthread_cond_init (condition,attr)* qui renvoie l'Id de la VarCond dans le paramètre *condition*.

- La paramètre optionnel *attr* permet de préciser la valeur *process-shared* qui rend la VarCond accessible aux autres threads d'autres processus.

  attr doit être du type *pthread_condattr_t* (peut être précisé NULL pour les valeurs par défaut).

- On fera attention aux variations (selon les plate formes) de la valeur *process-shared.*

Attendre et signaler :

- *pthread_cond_wait (condition,mutex)* : **bloquante** jusqu'à signaler.

  ➥ Doit être appelé après avoir verrouille Mutex.

  ➥ Libérera automatiquement le Mutex pendant l'attente.

  ➥ Après le signal, le thread bloqué sera réveillé et le Mutex automatiquement verrouille pour lui (il n'a pas à le refaire)

  ➥ Le programmeur DOIT déverrouiller le Mutex quand il aura fini avec.

- *pthread_cond_signal (condition)*

  ➥ Doit être appelé après avoir eu Mutex afin que *pthread_cond_wait()* puisse s'exécuter.

  ➥ Doit déverrouille le Mutex : signaler ne fait rien automatiquement.

• *pthread_cond_broadcast (condition)* : signale une condition à tous les threads qui l'attendent. A utiliser si plusieurs threads attendent la même condition (e.g. . le dernier arrivée réveille tout le monde).

• *pthread_cond_timedwait(cond, mutex, deadline)* : Attente bornée dans le temps.

➙ Si aucun signal n'arrive avant la fin de la durée spécifiée (durée = deadline-maintenant), on sort de l'attente avec time out (code renvoyé par wait = *ETIMEDOUT*).

➙ Se comporte comme un wait ordinaire quant à la gestion du Mutex. Le time out est comme une sortie ordinaire du wait (re verrouillage du Mutex à la sortie du wait, même à cause du time out).

➙ Deadline = le temps actuel + durée : on ne précise pas une durée mais l'heure du deadline (utiliser **time(NULL) + k** pour attendre k secondes.)

Attention :

- Ne pas appeler *signal* avant d'avoir appelé *wait*.

- Si on ne verrouille pas Mutex avant d'appeler wait, l'appel de wait peut ne pas être bloquant (car risque de croisement, voir ci-dessus).

- Si on ne libère pas Mutex avant d'appeler signal ; cela peut empêcher le wait correspondant de sortir de l'attente (on restera bloqué).

1.17.2 Schéma général de signale/attendre

→ sommeil et activation

```
**** thread 1

     bloquer l'accès
     if(manipulation impossible)
     {
         relâcher l'accès
         attendre_evenement(manipulation possible)
         bloquer l'accès
     }

     // manipuler les données
     { ... }

     relâcher l'accès


**** thread 2

     bloquer l'accès

     // manipuler les données
     { ... }

     signaler_evenement(manipulation possible)
     relâcher l'accès
```

1.17.3 Un premier exemple (Robot R1)

- 2 threads : *main()* et *reveiller_moi_si_le_robot_bouge()*
- La *main()* change la position du robot (X,Y) et le signale puis attend un certain temps
- *reveiller_moi_si_le_robot_bouge()* attend qu'un changement soit signalé et affiche les nouvelles coordonnées du robot.
- Dans cet exemple, il n'y a pas de valeur particulière a atteindre, seulement un changement.

```c
// Exemple varcond avec thread (Ex-mutex-cond-simple.c)
// compiler avec : gcc -pthread

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

// mutex d'accès à la position du Robot
pthread_mutex_t mutex_position_robot = PTHREAD_MUTEX_INITIALIZER;

// La condition signalée et son mutex
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

// les données de la section Critique
int X_robot, Y_robot;

void *reveiller_moi_si_le_robot_bouge();

int main() {
  int i;
  int code;
  srand(time(NULL));

  pthread_t th_affichage_cond_mutex_id;

  // création du thread d'affichage
  code=pthread_create(&th_affichage_cond_mutex_id, NULL, reveiller_moi_si_le_robot_bouge, NULL );
  if (code) {printf("Pb creation thread  : %d\n",code);exit(1);}

  X_robot= Y_robot=0;
  for(i=1;i<50; i++) {// répéter 50 fois
```

```c
    // changer la position du robot
      pthread_mutex_lock( &mutex_position_robot );
      X_robot++; Y_robot++;
      pthread_mutex_unlock( &mutex_position_robot );

    // signaler ce changement
    pthread_mutex_lock( &condition_mutex );
    pthread_cond_signal( &condition_cond );
    pthread_mutex_unlock( &condition_mutex );

    usleep(50000* (rand()%3+1));      //attendre aléatoirement
  }
// Une fin normale met fin a tous les threads
}
//-----------------------------------------------
// Attendre que X,Y du robot soient modifiées et signalées pour se réveiller
void *reveiller_moi_si_le_robot_bouge() {
    for(;;) {
      pthread_mutex_lock( &condition_mutex );
      pthread_cond_wait( &condition_cond, &condition_mutex );
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &mutex_position_robot );
      printf("Nouvelle Position Robot<%d,%d> \n", X_robot, Y_robot);
      pthread_mutex_unlock( &mutex_position_robot );
      usleep(500);  // passer la main
    }
}
```

Listing 1.13 – Ex-mutex-cond-simple.c

Rappel sur Wait/Signal et Mutex :

- Verrouiller le Mutex avant d'appeler wait.
- Le Mutex est libéré (unlock) automatiquement lors d'un wait.
  - → Il est automatiquement déverrouille quand on est réveillé
  - → Donc penser à le libérer une fois réveillé (par signal) via un appel à *unlock*.
- De même, penser verrouiller Mutex avant de signaler.
  - → Et déverrouiller quand le signal a été donné.
- *Signal* est par défaut non bloquant et peut être sans effet si aucun thread ne l'attend.

- La séquence du coté celui qui attend (par *wait*) :
  - → *Verrouiller ; . Wait, ...(on est réveillé).. Déverrouiller*
- La séquence du coté celui qui signale (par *signal*) :
  - → *Verrouiller ; . Signaler, ...(on réveille).. Déverrouiller*

- Si tous les threads (qui vont signaler) verrouillent Mutex avant de signaler, on aura la garantie qu'une condition ne peut pas être signalée (et donc être ignorée) entre le moment ou un thread verrouille Mutex et le moment où il attend la condition (par wait) puisque le verrouillage du Mutex par le signaleur évite que wait et signal se croisent ainsi.

1.17.4 Exemple 2 : Robot R1 (bis)

• Dans cette version, on regroupe les données dans une struct pour en faciliter l'accès.

➙ On guète la sortie du robot du cadre de l'échiquier.

• 3 threads :

1- *main* change la position du robot et le signale puis attend un certain temps

2- Le thread *affichage_systematique* affiche sans cesse les nouvelles coordonnées.

3- Le thread *avertir_si_sortie_du_cadre* attend un changement ; puis vérifie que les nouvelles coordonnées sont hors cadre échiquier auquel cas affiche un avertissement.

• Le robot est hors cadre si son $X < 1$ ou $X > 20$ (idem son Y)

```
 // 2009-2010
// Dans cette version, on regroupe les données dans une struct
// on guète la sortie du robot du cadre de l'échiquier
// 3 threads :
//     1- main change la position du robot et le signale puis attend un certain temps
//     2- Le thread affichage_systematique  affiche sans cesse les nouvelles coordonnées.
//     3- Le thread avertir_si_sortie_du_cadre attend un changement et que les nouvelles coordonnées
//        soient hors cadre échiquier et affiche un avertissement
// Le robot est hors cadre si son X<1 ou x>20 (idem son Y)

// compiler : gcc -pthread Ex-mutex-cond-et-VarCond.c

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

#define LIMITE_ECHIQUIER 20

typedef struct _donnees_robot {// n'accéder au reste que si mutex est locké !
  pthread_mutex_t mutex;
  pthread_cond_t  condition; // ce que l'on signale
  int X_robot, Y_robot;
} Donnees_robot;

Donnees_robot data_pour_robot  = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 1, 1};

void *affichage_systematique();
void *avertir_si_sortie_du_cadre();

int main() {
   int i;
   int code;
```

```
    pthread_t th_affichage_systematique_mutex_id;
    pthread_t th_avertir_si_sortie_cadre_cond_mutex_id;

    // création du thread d'affichage
    code=pthread_create(&th_affichage_systematique_mutex_id, NULL, affichage_systematique, NULL );
    if (code) {printf("Pb creation thread 1 : %d\n",code);exit(1);}
    code=pthread_create( &th_avertir_si_sortie_cadre_cond_mutex_id, NULL, avertir_si_sortie_du_cadre, NULL );
    if (code) {printf("Pb creation thread 2 : %d\n",code);exit(1);}

    for(i=1;i<60; i++) {// répéter 60 fois
        pthread_mutex_lock(&data_pour_robot.mutex);

        data_pour_robot.X_robot = (data_pour_robot.X_robot+2)%(LIMITE_ECHIQUIER+2);// pour tester
        data_pour_robot.Y_robot = (data_pour_robot.Y_robot+3)%(LIMITE_ECHIQUIER+2);

        pthread_cond_signal(&data_pour_robot.condition);
        pthread_mutex_unlock(&data_pour_robot.mutex);
        usleep(500000); // passer la main.
    }
// Une fin normale met fin a tous les threads
}
//-----------------------------------------------
void *affichage_systematique() {
    while(1) {
        pthread_mutex_lock(&data_pour_robot.mutex);
        printf("> Position Robot<%d,%d> \n", data_pour_robot.X_robot, data_pour_robot.Y_robot);
        pthread_mutex_unlock(&data_pour_robot.mutex);
            usleep(500000);   // sans cela, ça va trop vite
   }
}
//-----------------------------------------------
// Attend que X,Y du robot soient hors cadre
void *avertir_si_sortie_du_cadre() {
for(;;) {
```

```
    pthread_mutex_lock(&data_pour_robot.mutex);

    // le wait délock le mutex si l'on attend et lock quand on se réveille
    while(data_pour_robot.X_robot>0 && data_pour_robot.Y_robot>0 &&
      data_pour_robot.X_robot<LIMITE_ECHIQUIER && data_pour_robot.Y_robot<LIMITE_ECHIQUIER){
      pthread_cond_wait( &data_pour_robot.condition, &data_pour_robot.mutex );
    }
    printf("Hors cadre : Position Robot<%d,%d> \n", data_pour_robot.X_robot, data_pour_robot.Y_robot);
    pthread_mutex_unlock(&data_pour_robot.mutex);
    usleep(500000);
  }
}
```

1.17.5 Exemple 3 : condition timeout

• Le thread *wait_thread* attend k secondes (au moins 1) puis modifie la variable *value* par *data.value = 1 ;* et **signale** cet acte.

➙ Le thread prend la précaution de demander d'abord le mutex (puisque SC).

• De son côté, la fonction *main()* accède à la même variable *value*. Si *value != 1*, *main()* attend au plus 2 secondes.

➙ A la sortie de cette attente, soit *value=1* s'est réalisé (*main()* l'affiche) ; soit le deadline de 2 secondes a été atteint auquel cas *main* n'affiche rien.

• La variable hibernation initialisée par défaut à 1 = le délai avant de faire *data.value = 1 ;* par le thread *wait_thread* . Si on lance ce programme avec une valeur, ce sera la nouvelle durée d'attente avant de faire *data.value = 1 ;*

```c
/* Ex-condition-timeout.c : Demonstrate a simple condition variable wait. */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

typedef struct my_struct_tag {
  pthread_mutex_t      mutex;   /* Protects access to value */
    pthread_cond_t       cond;    /* Signals change to value */
    int                  value;   /* Access protected by mutex */
} my_struct_t;

my_struct_t data = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};

int hibernation = 1;            /* Default to 1 second */

/*
* Thread start routine. It will set the main thread's predicate
* and signal the condition variable.
*/
void * wait_thread (void *arg) {
  int status;

  sleep (hibernation);
  status = pthread_mutex_lock (&data.mutex);
  if (status != 0) {perror( "Lock mutex"); exit(1);}
    data.value = 1;             /* Set predicate */
    status = pthread_cond_signal (&data.cond);
  if (status != 0) {perror("Signal condition");exit(1);}
    status = pthread_mutex_unlock (&data.mutex);
  if (status != 0) {perror( "Unlock mutex");exit(1);}
```

```
    return NULL;
}

int main (int argc, char *argv[]) {
  int status;
  pthread_t wait_thread_id;
  struct timespec timeout;

  /*
  * If an argument is specified, interpret it as the number
  * of seconds for wait_thread to sleep before signaling the
  * condition variable. You can play with this to see the
  * condition wait below time out or wake normally.
  */
  if (argc > 1) hibernation = atoi (argv[1]);

  /*
  * Create wait_thread.
  */
  status = pthread_create (&wait_thread_id, NULL, wait_thread, NULL);
  if (status != 0) {perror("Create wait thread");exit(1);}

    /*
  * Wait on the condition variable for 2 seconds, or until
  * signaled by the wait_thread. Normally, wait_thread
  * should signal. If you raise "hibernation" above 2
  * seconds, it will time out.
  */
  timeout.tv_sec = time (NULL) + 2;
  timeout.tv_nsec = 0;
  status = pthread_mutex_lock (&data.mutex);
  if (status != 0) {perror("Lock mutex");exit(1);}

    while (data.value == 0) {
```

```
    status = pthread_cond_timedwait (&data.cond, &data.mutex, &timeout);
    if (status == ETIMEDOUT) {
      printf ("Condition wait timed out.\n");
      break;
    }
        else if (status != 0) {perror("Wait on condition");exit(1);}
  }

    if (data.value != 0) printf ("Condition signalée.\n");
  status = pthread_mutex_unlock (&data.mutex);
  if (status != 0) {perror("Unlock mutex");exit(1);}
    return 0;
}
/* trace
Condition signalée.
*/
```

Listing 1.14 – Ex-condition-timeout.c

1.17.6 Exemple 4 : timeout (bis)

- *main()* crée 3 threads dont 2 mettent à jour un compteur ; le 3e attend qu'une valeur particulière du compteur soit atteinte.

```
/****************************************************************************
* DESCRIPTION:
*   Example code for using Pthreads condition variables.  The main thread
*   creates three threads.  Two of those threads increment a "count" variable,
*   while the third thread watches the value of "count".  When "count"
*   reaches a predefined limit, the waiting thread is signaled by one of the
*   incrementing threads. The waiting thread "awakens" and then modifies
*   count. The program continues until the incrementing threads reach
*   TCOUNT. The main program prints the final value of count.
* SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
*   et al. O'Reilly and Associates.
* LAST REVISED: 07/16/09  Blaise Barney
****************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int       count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

```
void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
      reached.  Note that this occurs while mutex is locked.
      */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ", my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
    }
      printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
  }
    pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
```

```
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
    will automatically and atomically unlock mutex while it waits.
    Also, note that if COUNT_LIMIT is reached before this routine is run by
    the waiting thread, the loop will be skipped to prevent pthread_cond_wait
    from never returning.
    */
  pthread_mutex_lock(&count_mutex);
  if (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld going into wait...\n", my_id);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
  }
    pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joignable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
```

```
  pthread_create(&threads(2), &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads(i), NULL);
  }
    printf ("Main(): Waited on %d threads. Final value of count = %d. Done.\n", NUM_THREADS, count);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
/* TRACE :
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 2, count = 3, unlocking mutex
inc_count(): thread 3, count = 4, unlocking mutex
......
inc_count(): thread 2, count = 11, unlocking mutex
inc_count(): thread 3, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 3, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 2, count = 138, unlocking mutex
inc_count(): thread 3, count = 139, unlocking mutex
*/
```

Listing 1.15 – condvar.c

1.18 Sémaphores Posix

- Un sémaphore est un mécanisme plus évolué qu'un Mutex.

➙ Un Mutex ne permet pas de savoir le nombre de ceux qui sont en attente

➙ De plus, l'ordre des réveils n'est pas défini

➙ Un sémaphore permet de connaître le nombre de ceux qui attendent et on aura la garantie qu'ils seront réveillés dans leur ordre d'arrivée.

- Un sémaphore est comme un distributeur de tickets (par exemple d'un parking à N places) :

➙ les N premiers arrivés entrent

➙ Le N+1 ème et les suivants attendront ;

➙ Dès qu'une place est libérée, le première de la fil entre ...

- Le nombre de tickets peut varier de 0 à ... (limite du système)

Quelques opérations définies sur les sémaphore Posix :

#include <semaphore.h> ➙ directive d'inclusion du fichier d'interface

sem_t

➙ type associé à un sémaphore (valeur quelconque, opérations P et V standard).

int sem_init(sem_t *sem, int partage, unsigned int valeur)

➙ initialise le sémaphore non nommé (pointé par sem) à la valeur spécifiée en paramètre.

Le paramètre partage a le rôle suivant :

➙ s'il est non nul, le sémaphore peut être utilisé pour synchroniser des threads de processus différents

➙ s'il est nul, l'utilisation du sémaphore est limitée aux threads du processus appelant.

int sem_destroy(sem_t *sem)

➙ détruit le sémaphore pointé par sem (supposé avoir été initialisé par sem_init)

int sem_wait(sem_t *sem)

➙ l'opération P : on prend un ticket

int sem_trywait(sem_t *sem)

➙ l'opération P en mode non bloquant

int sem_post(sem_t *sem)

➙ l'opération V : on rend un ticket

int sem_getvalue(sem_t *sem, int *valeur)

➙ permet de récupérer la valeur (positive ou nulle) courante du sémaphore spécifié

- Voir la documentation pour les autres opérateurs.

1.18.1 Utilisation des Sémaphores POSIX

Rappel : inclure < *semaphore.h* >

- *sem_t S ;*
- *int sem_init(&S, 0,1) ;* propre au processus courant, val init=1.
- *int sem_wait(&S) ;* // P(S) renvoie 0 si OK
- *int sem_post(&S) ;* //V(S)
- *int sem_trywait(&S) ;* // P(S) non bloquant
- *int sem_timedwait(&S, &abs_timeout) ;*

→ Avec *abs_timeout* pointeur sur :

struct timespec { time_t tv_sec ; / Secondes */*

long tv_nsec ; } ; / Nanosecondes [0 .. 999999999] */*

A propos des fonctions principales des sémaphores :

- *sem_wait()* décrémente (verrouille) le sémaphore pointé par sem.

  Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement.

  Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible pour effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle), soit un gestionnaire de signaux interrompe l'appel.

- *sem_trywait()* est pareil que *sem_wait()*, excepté que si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur (errno vaut EAGAIN) plutôt que bloquer.

- *sem_timedwait()* est pareil que *sem_wait()*, excepté que *abs_timeout* spécifie une limite sur le temps pendant lequel l'appel bloquera si la décrémentation ne peut pas être

effectuée immédiatement.

L'argument *abs_timeout* pointe sur une structure qui spécifie un temps absolu en secondes et nanosecondes depuis l'Époque (1er janvier 1970, 00 :00 :00)

➙ Cette structure est définie de la manière suivante :

```
struct timespec {
    time_t tv_sec;        /* Secondes */
    long   tv_nsec;       /* Nanosecondes (0 .. 999999999) */
};
```

➙ Si le délai est déjà expiré à l'heure de l'appel et si le sémaphore ne peut pas être verrouillé immédiatement, *sem_timedwait()* échoue avec l'erreur d'expiration de délai (errno vaut ETIMEDOUT).

➙ Si l'opération peut être effectuée immédiatement, *sem_timedwait()* n'échoue jamais avec une valeur d'expiration de délai, quelque soit la valeur de *abs_timeout*.

➙ De plus, la validité de *abs_timeout* n'est pas vérifiée dans ce cas.

1.18.2 Exemple

```
 $ Utilisations :
$ ./a.out 2 3
 About to call sem_timedwait()
 sem_post() from handler
 sem_getvalue() from handler; value = 1
 sem_timedwait() succeeded

 $ ./a.out 2 1
 About to call sem_timedwait()
 sem_timedwait() timed out

// Le code :

 #include <unistd.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <semaphore.h>
 #include <time.h>
 #include <assert.h>
 #include <errno.h>
 #include <signal.h>

 #define die(msg) { perror(msg); exit(EXIT_FAILURE); }

 sem_t sem;

 static void handler(int sig)
 {
     int sval;

     printf("sem_post() from handler\n");
```

```
    if (sem_post(&sem) == -1)  die("sem_post");

    if (sem_getvalue(&sem, &sval) == -1) die("sem_getvalue");
    printf("sem_getvalue() from handler; value = %d\n", sval);
} /* handler */

int main(int argc, char *argv[])
{
    struct sigaction sa;
    struct timespec ts;
    int s;

    assert(argc == 3);  /* Usage: ./a.out alarm-secs wait-secs */

    if (sem_init(&sem, 0, 0) == -1) die("sem_init");

    /* Establish SIGALRM handler; set alarm timer using argv[1] */

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, NULL) == -1) die("sigaction");

    alarm(atoi(argv[1]));

    /* Calculate relative interval as current time plus
       number of seconds given argv[2] */

    if (clock_gettime(CLOCK_REALTIME, &ts) == -1) die("clock_gettime");
    ts.tv_sec += atoi(argv[2]);

    printf("main() about to call sem_timedwait()\n");
    while ((s = sem_timedwait(&sem, &ts)) == -1 && errno == EINTR)
        continue;       /* Restart when interrupted by handler */
```

```
    /* Check what happened */

    if (s == -1) {
        if (errno == ETIMEDOUT) printf("sem_timedwait() timed out\n");
        else die("sem_timedwait");
    }
    else printf("sem_timedwait() succeeded\n");

    exit(EXIT_SUCCESS);
}
```

Remarques sur les sémaphores :

• Contrairement aux mutex, les sémaphores permettent à plusieurs threads d'accéder en même temps à une portion de code critique.

Si la valeur courante d'un sémaphore est n, un nombre d'appels jusqu'à n à *sem_wait()* pourront être effectués sur ce sémaphore sans être bloqués (si entre-temps aucun appel à *sem_post()* n'a eu lieu).

• Ceci peut s'avérer utile pour gérer l'accès à des ressources en nombre limité.

• Par ex., un serveur vocal permettant de recevoir et envoyer des appels de manière automatisée et disposant d'un nombre fixé de lignes téléphoniques peut utiliser un sémaphore pour gérer l'accès de threads réalisant des appels automatiques à ces lignes.

• Un thread peut réaliser un appel à *sem_post()* même s'il n'a pas effectué d'appel à *sem_wait()* sur le même sémaphore précédemment.

- Suite à un appel à *sem_post()*, la valeur d'un sémaphore peut dépasser sa valeur initiale.

1.18.3 Exercice

- Créer un programme qui crée 10 threads qui exécutent une fonction dont le corps est protégée par un sémaphore initialisé à 3 dans le thread principal et qui en empêche donc l'exécution par plus de 3 threads en même temps.
- La portion protégée par le sémaphore affichera sa valeur, endort le thread pendant une seconde puis ré affiche la valeur du sémaphore.
- Les valeurs affichées pourront parfois sembler incohérentes car un thread peut être interrompu entre le moment où il décrémente un sémaphore et le moment où il affiche sa valeur par un autre thread qui effectue une incrémentation ou une décrémentation.
- Pour forcer la manifestation de ce phénomène, placer un appel à *usleep()* entre l'acquisition du sémaphore et l'affichage de sa valeur.
- Un appel à *sleep()* au début de la fonction de thread laissera le temps au thread

principal de créer tous les threads avant que ceux-ci ne tentent d'accéder à la section critique.

- Utiliser des threads détachés et la terminaison du thread principal avant celles des autres threads.

1.18.4 Solution

```
// 2010 :  exercice sur les sémaphores
// Dans la section critique, chaque thread demande la valeur du sémaphore, l'affiche
// s'endort puis redemande la valeur du sémaphore
// puis sort de la section critique.
// Les 2 valeurs affichées sont souvent différentes car il y a eu des waits sur le sémaphore
// pendant le sommeil du thread.

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>

#define NB_threads 10

sem_t S;     // visible par tous

void * fonction_avec_SC(void * mon_numero);

int main()
{  int i, rc;
   pthread_t tab_id1(NB_threads);
   int tab_num_ligne_a_ecran(NB_threads);    // les valeurs envoyées aux threads

   srand(time(NULL));

   rc=sem_init(&S, 0, 3);    // partagé dans ce processus seulement. Val init=0 (privé)
   if (rc !=0) {printf("Pb sem \n"); exit(0);}

   printf("Créations des threads \n");
   for (i=0; i< NB_threads; i++){
```

```
      tab_num_ligne_a_ecran(i)=i+1;
      rc = pthread_create(&tab_id1(i) , NULL, fonction_avec_SC, (void *) &tab_num_ligne_a_ecran(i));
   }

   printf("Main termine et laisse les threads se terminer ... \n");
   pthread_exit(0) ;

}

void * fonction_avec_SC(void * p_mon_num) {
   int mon_numero = *((int *) p_mon_num);
   int i=0, j;

   sleep(1);    // laisser les threads se créer

   pthread_detach(pthread_self()) ; // On se détache
   do {
    int val_sem=-1;
    sem_wait(&S);

         // demander la valeur du sémaphore
    if (sem_getvalue(&S, &val_sem)) {printf("pb sem_getvalue"); exit(1);}

    for(j=0; j<mon_numero;j++) putchar(' ');    // décaler l'affichage
    printf("Thread numéro %d affiche val_sem (avant sleep)= %d\n",mon_numero,val_sem);

    usleep(500000*(rand()%3+1));

         // Redemander la valeur du sémaphore
    if (sem_getvalue(&S, &val_sem)) {printf("pb sem_getvalue"); exit(1);}

    for(j=0; j<mon_numero;j++) putchar(' ');    // décaler l'affichage
    printf("Thread numéro %d affiche val_sem (après sleep)= %d\n",mon_numero,val_sem);
```

```
    sem_post(&S);
   } while(i++ < 5);

    return NULL;
}

// Une trace :
Créations des threads
Main termine et laisse les threads se terminer ...
     Thread numéro 3 affiche val_sem (avant sleep)= 0
 Thread numéro 1 affiche val_sem (avant sleep)= 1
Thread numéro 2 affiche val_sem (avant sleep)= 1
   Thread numéro 1 affiche val_sem (après sleep)= 0
 Thread numéro 1 affiche val_sem (avant sleep)= 0
Thread numéro 2 affiche val_sem (après sleep)= 0
  Thread numéro 2 affiche val_sem (avant sleep)= 0
  Thread numéro 2 affiche val_sem (après sleep)= 0
  Thread numéro 2 affiche val_sem (avant sleep)= 0
   Thread numéro 3 affiche val_sem (après sleep)= 0
   Thread numéro 3 affiche val_sem (avant sleep)= 0
 Thread numéro 1 affiche val_sem (après sleep)= 0
 Thread numéro 1 affiche val_sem (avant sleep)= 0
   Thread numéro 3 affiche val_sem (après sleep)= 0
   Thread numéro 3 affiche val_sem (avant sleep)= 0
  Thread numéro 2 affiche val_sem (après sleep)= 0
  Thread numéro 2 affiche val_sem (avant sleep)= 0
 Thread numéro 1 affiche val_sem (après sleep)= 0
 Thread numéro 1 affiche val_sem (avant sleep)= 0
   Thread numéro 3 affiche val_sem (après sleep)= 0
   Thread numéro 3 affiche val_sem (avant sleep)= 0
 Thread numéro 1 affiche val_sem (après sleep)= 0
 Thread numéro 1 affiche val_sem (avant sleep)= 0
....
```

1.19 Application : Robot R1

1.19.1 Approche décentralisée à base de réflexe (cybernétique)

- Plus récent ; Vient du MIT (R. Brooks, Labo IA)

- Idée : combiner le contrôle Temps Réel distribué avec le réflexe à base de capteur.

- Les réflexes (**Behavior**) sont les couches du **contrôle** du système.

➥ Un module d'**arbitrage** permet de résoudre les conflits et mettre en oeuvre un scénario donné.

- Attention : une réaction (réflexe) ne contrôle pas (n'appelle pas) une autre.

➥ Les Réflexes sont exécutées en parallèle, mais un réflexe d'un niveau <u>supérieur</u> peut suspendre une d'un niveau inférieur.

Approche décentralisée réactive (suite) ..

➥ Si les Réflexes du niveau supérieur sont déactivés (par exemple, coupure liaison capteur), les réflexes du niveau inférieur reprennent la main.

- Les capteurs interagissent à travers les Réflexes.
- Il n'y a pas de modèle global du monde réel contrôlant l'ensemble, ni de structure de données unifiée.

1.19.1.1 R1 : un Robot simple

• Un robot muni d'un anneau de capteurs sonars (balayage à 360°), un détecteur IR, un processeur basique (simple) + un peu de RAM.

• On lui assigne le but d'éviter de se cogner dans les obstacles.

• Un diagramme :

• Sonar est un module qui opère sur les capteurs sonars qui s'occupe en permanence de la distance mesurée.

• Motor est un module qui envoie du courant au moteurs en réponse de la commande qu'il reçoit.

... Suite Robot R1 ...

• Entre les deux, le module Avoid s'appuie sur les données sonars et calcule en permanence des commandes envoyées au Motor.

• Le but de ce robot est de rester à distance (loin) de tous les obstacles (repérables par les sonars)... par réflexe / réaction !

• Avoid contient un code qui implante un comportement (réflexe) simple et réflexif.

• Si la donnée Sonar de tête est trop "courte", Avoid arrête l'avancée du Robot.

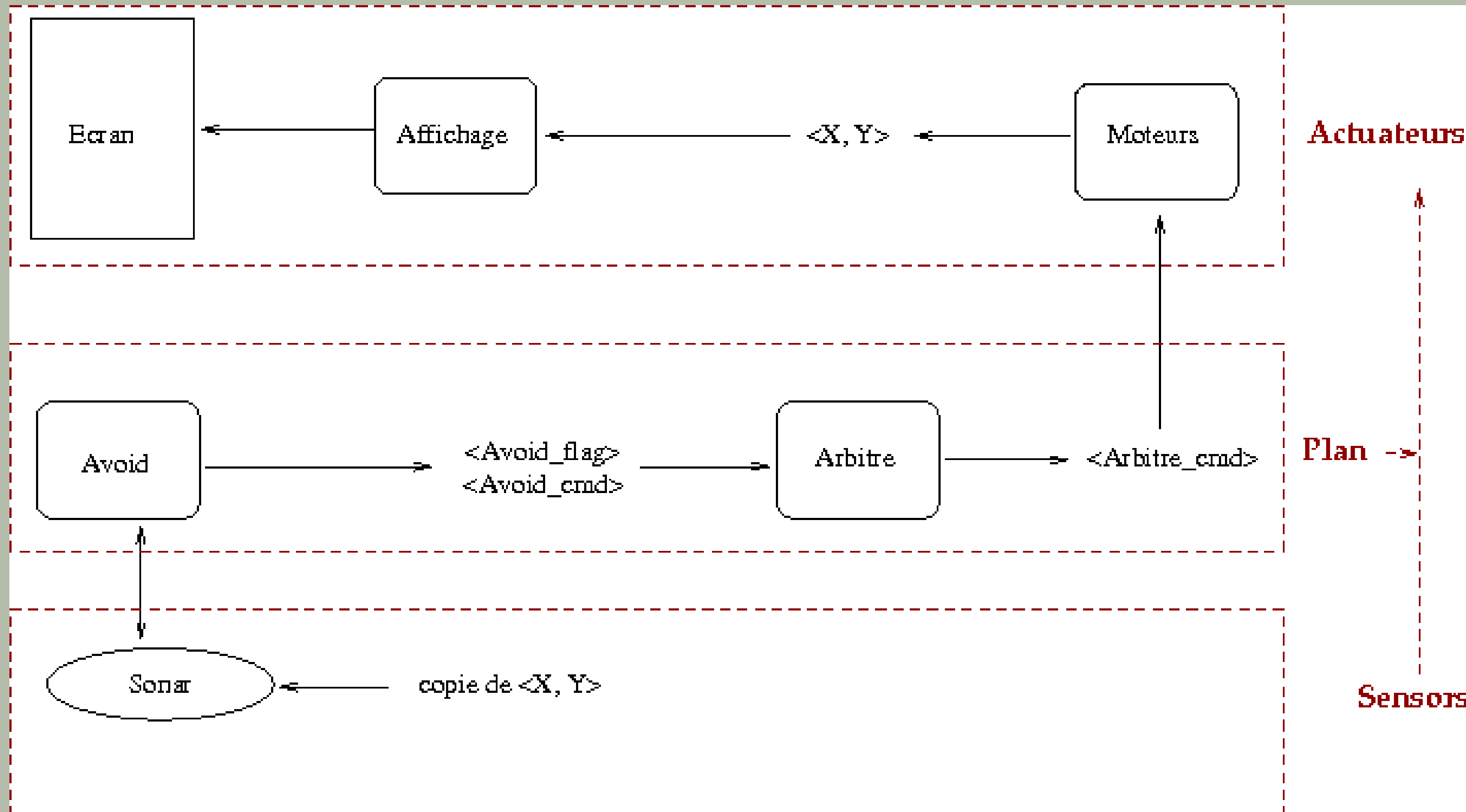
• Si, hormis les mesures arrières (à contre sens), un capteur sonar mesure une distance la plus proche, Avoid fait tourner le Robot pour pointer dans cette direction.

... Suite Robot R1 ...

- Quand le sonar arrière mesure la distance la plus proche, les moteurs reçoivent la commande d'avancer (de s'éloigner).
- Si les données sonars dépassent un certain seuil, *Avoid* ne fait rien (n'envoie pas de commande).

- L'ensemble de ces opérations (simples) est traitable par un petit MC68HC11A0 capable d'exécuter tout le code très rapidement (plusieurs fois par seconde).
- La réflexe de cette stratégie permet au robot de rester à distance (loin) de tous les obstacles (repérables par les sonars).

1.19.1.2 Code R1



Main() :

Initialiser Affichage

Initialiser Coordonnées (X,Y) du robot au hasard (dans la limite de l'échiquier)

Initialiser le rayon de sensibilité du sonar R

Créer un thread **Avoid** (lit le Sonar, avec distinction G/D)

Créer un thread **Moteur** (pour tous les moteurs)

Créer un thread **Arbitre**

Créer un thread **Affichage**

Se mettre en attente.

Thread Avoid : est seul à consulter les données Sonar

Répéter

Lire données Sonar **Val** (2 bits)

Si Val == 00 *// ne rien changer (pas de détection)*

Alors Avoid_output_flag=false ;

Sinon Avoid_output_flag=true ; *// mur à proximité*

Si Val=01 *// proximité mur à droite*

Alors Avoid_cmd = TURN_LEFT

Sinon Avoid_cmd = TURN_RIGHT *// en face ou à gauche : tourner*

Finsi *// on peut éventuellement décider de reculer si val==3*

Fin si

Fin Répéter

Lecture données Sonar : une simple fonction faisant partie du thread Avoid

Accéder (mutex) à <X,Y> position du robot et Dir = sa direction

Si proche d'un mur (selon rayon R, <X,Y> et Dir)

Alors

indiquer un valeur appropriée pour chaque capteur G/D

Fin si

- Pas besoin d'affecter un thread à cette fonction qui n'est appelée que par Avoid.

Thread Affichage :

Répéter

Accéder (mutex) à <X,Y> positon du robot

Vérifier que <X,Y> est dans la limite de l'échiquier

Effacer la position actuelle

Afficher le Robot

Fin Répéter

- Attention : voir si besoin de ré afficher tout l'échiquier

Thread Arbitre :

Accéder (mutex) à Avoid_output_flag et Avoid_cmd

// Avoid_cmd parmi {TURN_LEFT, TURN_RIGHT}

Arbitre_cmd=Forward *// par défaut*

Si Avoid_output_flag== true *// drapeau levé*

Alors Arbitre_cmd=Avoid_cmd

Fin si

- Arbitre_cmd sera prise en charge par Moteur

Thread Moteur : simulation déplacement du Robot

Répéter

Accéder (mutex) à <X,Y> position du robot

Accéder (mutex) à Arbitre_cmd

// Arbitre_cmd parmi {Forward, TURN_LEFT, TURN_RIGHT}

Calculer nouvelle position <newX, newY> en fonction de la commande

Publier (mutex) <newX, newY> as <X,Y>

Fin Répéter

- La nouvelle position affichée par Affichage.
- Voir solutions avec VarCond

1.20 Annexe-1 : Bibliothèque des threads

Une liste des routines de la bibliothèque des threads :

```
pthread_atfork

// Attribut de création des threads
pthread_attr_destroy               pthread_attr_getdetachstate     pthread_attr_getguardsize
pthread_attr_getinheritsched    pthread_attr_getschedparam      pthread_attr_getschedpolicy
pthread_attr_getscope               pthread_attr_getstack                pthread_attr_getstackaddr
pthread_attr_getstacksize          pthread_attr_init                            pthread_attr_setdetachstate
pthread_attr_setguardsize          pthread_attr_setinheritsched     pthread_attr_setschedparam
pthread_attr_setschedpolicy       pthread_attr_setscope                  pthread_attr_setstack
pthread_attr_setstackaddr         pthread_attr_setstacksize

// Barrières
pthread_barrier_destroy            pthread_barrier_init                            pthread_barrier_wait
pthread_barrierattr_destroy     pthread_barrierattr_getpshared      pthread_barrierattr_init
pthread_barrierattr_setpshared

// Intall / remove cleanup handlers
pthread_cleanup_pop     pthread_cleanup_push

// Conditions
pthread_cond_broadcast        pthread_cond_destroy            pthread_cond_init
pthread_cond_signal               pthread_cond_timedwait          pthread_cond_wait
pthread_condattr_destroy     pthread_condattr_getclock     pthread_condattr_getpshared
pthread_condattr_init            pthread_condattr_setclock      pthread_condattr_setpshared

// Création des threads / join state/ exit / kill
pthread_create      pthread_detach    pthread_equal     pthread_exit
pthread_join           pthread_kill          pthread_once
```

```
// Thread cancel
pthread_cancel     pthread_setcancelstate      pthread_setcanceltype    pthread_testcancel


// Divers propriétés des threads et leur comportement
pthread_getconcurrency      pthread_getcpuclockid        pthread_getschedparam
pthread_getspecific             pthread_key_create           pthread_key_delete     pthread_self
pthread_setconcurrency      pthread_setschedparam      pthread_setschedprio
pthread_setspecific             pthread_sigmask
pthread_spin_destroy          pthread_spin_init                 pthread_spin_lock
pthread_spin_trylock           pthread_spin_unlock

// Mutex
pthread_mutex_destroy                     pthread_mutex_getprioceiling       pthread_mutex_init
pthread_mutex_lock                           pthread_mutex_setprioceiling       pthread_mutex_timedlock
pthread_mutex_trylock                     pthread_mutex_unlock                 pthread_mutexattr_destroy
pthread_mutexattr_getprioceiling      pthread_mutexattr_getprotocol
pthread_mutexattr_getpshared         pthread_mutexattr_gettype
pthread_mutexattr_init                       pthread_mutexattr_setprioceiling
pthread_mutexattr_setprotocol        pthread_mutexattr_setpshared      pthread_mutexattr_settype

// Readers Writers
pthread_rwlock_destroypthread_once        pthread_rwlock_initpthread_once     pthread_rwlock_rdlock
pthread_rwlock_timedrdlock         pthread_rwlock_timedwrlock              pthread_rwlock_tryrdlock
pthread_rwlock_trywrlock             pthread_rwlock_unlock                    pthread_rwlock_wrlock
pthread_rwlockattr_destroy     pthread_rwlockattr_getpshared        pthread_rwlockattr_init
pthread_rwlockattr_setpshared
```

Fonctions de manipulation des mutex lecteur / rédacteurs

NB : compiler avec gcc -D_REENTRANT -D_GNU_SOURCE

```
#include <pthread.h>

/* Initialize read-write lock RWLOCK using attributes ATTR, or use the default values if later is NULL.  */
extern int pthread_rwlock_init(pthread_rwlock_t *rwlock, pthread_rwlockattr_t *attr);

/* Destroy read-write lock RWLOCK. */
extern int pthread_rwlock_destroy(pthread_rwlock_t *);

/* Acquire read lock for RWLOCK. */
extern int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

/* Try to acquire read lock for RWLOCK. */
extern int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

/* Acquire write lock for RWLOCK. */
extern int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

/* Try to acquire write lock for RWLOCK. */
extern int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

/* Unlock RWLOCK. */
extern int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

• Fonctions de manipulation des barrières

```
#include <pthread.h>

extern int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr, unsigned int count);

extern int pthread_barrier_destroy(pthread_barrier_t *barrier);

extern int pthread_barrier_wait(pthread_barrier_t *barrier);
```

A voir (compliation) :

gcc -Wall -o programme -g -D_REENTRANT -D_GNU_SOURCE source.c -lpthread

1.21 Exemple-1 : manipulation de la Pile

```
/*
 The program below demonstrates the use of pthread_create(), as well as a number of other functions in the
  pthreads API.

In the following run, on a system providing the NPTL threading implementation, the stack size defaults to the
 value given by the "stack size" resource limit:

 $ ulimit -s
 8192            # The stack size limit is 8 MB (0x80000 bytes)

 $ ./a.out hola salut servus
 Thread 1: top of stack near 0xb7dd03b8; argv_string=hola
 Thread 2: top of stack near 0xb75cf3b8; argv_string=salut
 Thread 3: top of stack near 0xb6dce3b8; argv_string=servus
 Joined with thread 1; returned value was HOLA
 Joined with thread 2; returned value was SALUT
 Joined with thread 3; returned value was SERVUS


In the next run, the program explicitly sets a stack size of 1MB (using pthread_attr_setstacksize(3)) for the
 created threads:

 $ ./a.out -s 0x100000 hola salut servus
 Thread 1: top of stack near 0xb7d723b8; argv_string=hola
 Thread 2: top of stack near 0xb7c713b8; argv_string=salut
 Thread 3: top of stack near 0xb7b703b8; argv_string=servus
 Joined with thread 1; returned value was HOLA
 Joined with thread 2; returned value was SALUT
 Joined with thread 3; returned value was SERVUS

*/
```

```
#include <pthread.h>
 #include <string.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <errno.h>
 #include <ctype.h>

 #define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

 #define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while (0)

 struct thread_info {    /* Used as argument to thread_start() */
     pthread_t thread_id;        /* ID returned by pthread_create() */
     int       thread_num;       /* Application-defined thread # */
     char     *argv_string;      /* From command-line argument */
 };

 /* Thread start function: display address near top of our stack,
    and return upper-cased copy of argv_string */

static void * thread_start(void *arg)
 {
     struct thread_info *tinfo = (struct thread_info *) arg;
     char *uargv, *p;

     printf("Thread %d: top of stack near %p; argv_string=%s\n",
             tinfo->thread_num, &p, tinfo->argv_string);

     uargv = strdup(tinfo->argv_string);
     if (uargv == NULL) handle_error("strdup");
```

```
    for (p = uargv; *p != '\0'; p++)
        *p = toupper(*p);

    return uargv;
}


int main(int argc, char *argv[])
{
    int s, tnum, opt, num_threads;
    struct thread_info *tinfo;
    pthread_attr_t attr;
    int stack_size;
    void *res;

    /* The "-s" option specifies a stack size for our threads */

    stack_size = -1;
    while ((opt = getopt(argc, argv, "s:")) != -1) {
        switch (opt) {
        case 's':
            stack_size = strtoul(optarg, NULL, 0);
            break;

        default:
            fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }

    num_threads = argc - optind;
```

```
/* Initialize thread creation attributes */

s = pthread_attr_init(&attr);
if (s != 0)  handle_error_en(s, "pthread_attr_init");

if (stack_size > 0) {
    s = pthread_attr_setstacksize(&attr, stack_size);
    if (s != 0) handle_error_en(s, "pthread_attr_setstacksize");
}

/* Allocate memory for pthread_create() arguments */

tinfo = calloc(num_threads, sizeof(struct thread_info));
if (tinfo == NULL)
    handle_error("calloc");

/* Create one thread for each command-line argument */

for (tnum = 0; tnum < num_threads; tnum++) {
    tinfo(tnum).thread_num = tnum + 1;
    tinfo(tnum).argv_string = argv(optind + tnum);

    /* The pthread_create() call stores the thread ID into
       corresponding element of tinfo() */

    s = pthread_create(&tinfo(tnum).thread_id, &attr,
                       &thread_start, &tinfo(tnum));
    if (s != 0) handle_error_en(s, "pthread_create");
}
```

```
    /* Destroy the thread attributes object, since it is no
       longer needed */

    s = pthread_attr_destroy(&attr);
    if (s != 0) handle_error_en(s, "pthread_attr_destroy");

    /* Now join with each thread, and display its returned value */

    for (tnum = 0; tnum < num_threads; tnum++) {
        s = pthread_join(tinfo(tnum).thread_id, &res);
        if (s != 0) handle_error_en(s, "pthread_join");

        printf("Joined with thread %d; returned value was %s\n",
                tinfo(tnum).thread_num, (char *) res);
        free(res);      /* Free memory allocated by thread */
    }

    free(tinfo);
    exit(EXIT_SUCCESS);
}
```

Listing 1.16 – Ex-stack.c

1.22 Exemple-2 : manipulation des attributs

```
/*
The program below optionally makes use of pthread_attr_init() and various related functions to initialize a
 thread attributes object that is used to create a single thread. Once
created, the thread uses the pthread_getattr_np() function (a non-standard GNU extension) to retrieve the
 thread's attributes, and then displays those attributes.

If the program is run with no command-line argument, then it passes NULL as the attr argument of pthread_create
 (3), so that the thread is created with default attributes. Running
the program on Linux/x86-32 with the NPTL threading implementation, we see the following:

 $ ulimit -s       # No stack imit ==> default stack size is 2MB
 unlimited
 $ ./a.out
 Thread attributes:
         Detach state        = PTHREAD_CREATE_JOINABLE
         Scope               = PTHREAD_SCOPE_SYSTEM
         Inherit scheduler   = PTHREAD_INHERIT_SCHED
         Scheduling policy   = SCHED_OTHER
         Scheduling priority = 0
         Guard size          = 4096 bytes
         Stack address       = 0x40196000
         Stack size          = 0x201000 bytes


When we supply a stack size as a command-line argument, the program initializes a thread attributes object,
 sets various attributes in that object, and passes a pointer to the
object in the call to pthread_create(3). Running the program on Linux/x86-32 with the NPTL threading
 implementation, we see the following:

 $ ./a.out 0x3000000
 posix_memalign() allocated at 0x40197000
 Thread attributes:
```

```
        Detach state          = PTHREAD_CREATE_DETACHED
        Scope                 = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler     = PTHREAD_EXPLICIT_SCHED
        Scheduling policy     = SCHED_OTHER
        Scheduling priority   = 0
        Guard size            = 0 bytes
        Stack address         = 0x40197000
        Stack size            = 0x3000000 bytes


Program source
*/

#define _GNU_SOURCE     /* To get pthread_getattr_np() declaration */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static void
display_pthread_attr(pthread_attr_t *attr, char *prefix)
{
    int s, i;
    size_t v;
    void *stkaddr;
    struct sched_param sp;

    s = pthread_attr_getdetachstate(attr, &i);
    if (s != 0) handle_error_en(s, "pthread_attr_getdetachstate");
    printf("%sDetach state        = %s\n", prefix,
```

```
        (i == PTHREAD_CREATE_DETACHED) ? "PTHREAD_CREATE_DETACHED" :
        (i == PTHREAD_CREATE_JOINABLE) ? "PTHREAD_CREATE_JOINABLE" :
        "???");

s = pthread_attr_getscope(attr, &i);
if (s != 0) handle_error_en(s, "pthread_attr_getscope");
printf("%sScope              = %s\n", prefix,
        (i == PTHREAD_SCOPE_SYSTEM)  ? "PTHREAD_SCOPE_SYSTEM" :
        (i == PTHREAD_SCOPE_PROCESS) ? "PTHREAD_SCOPE_PROCESS" :
        "???");

s = pthread_attr_getinheritsched(attr, &i);
if (s != 0) handle_error_en(s, "pthread_attr_getinheritsched");
printf("%sInherit scheduler  = %s\n", prefix,
        (i == PTHREAD_INHERIT_SCHED)  ? "PTHREAD_INHERIT_SCHED" :
        (i == PTHREAD_EXPLICIT_SCHED) ? "PTHREAD_EXPLICIT_SCHED" :
        "???");

s = pthread_attr_getschedpolicy(attr, &i);
if (s != 0)  handle_error_en(s, "pthread_attr_getschedpolicy");
printf("%sScheduling policy  = %s\n", prefix,
        (i == SCHED_OTHER) ? "SCHED_OTHER" :
        (i == SCHED_FIFO)  ? "SCHED_FIFO" :
        (i == SCHED_RR)    ? "SCHED_RR" :
        "???");

s = pthread_attr_getschedparam(attr, &sp);
handle_error_en(s, "pthread_attr_getschedparam");
printf("%sScheduling priority = %d\n", prefix, sp.sched_priority);

s = pthread_attr_getguardsize(attr, &v);
if (s != 0)  handle_error_en(s, "pthread_attr_getguardsize");
printf("%sGuard size          = %d bytes\n", prefix, v);
```

```
    s = pthread_attr_getstack(attr, &stkaddr, &v);
    if (s != 0) handle_error_en(s, "pthread_attr_getstack");
    printf("%sStack address       = %p\n", prefix, stkaddr);
    printf("%sStack size          = 0x%x bytes\n", prefix, v);
}

static void *
thread_start(void *arg)
{
    int s;
    pthread_attr_t gattr;

    /* pthread_getattr_np() is a non-standard GNU extension that
       retrieves the attributes of the thread specified in its
       first argument */

    s = pthread_getattr_np(pthread_self(), &gattr);
    if (s != 0)  handle_error_en(s, "pthread_getattr_np");

    printf("Thread attributes:\n");
    display_pthread_attr(&gattr, "\t");

    exit(EXIT_SUCCESS);          /* Terminate all threads */
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    pthread_attr_t attr;
    pthread_attr_t *attrp;       /* NULL or &attr */
    int s;

    attrp = NULL;
```

```
/* If a command-line argument was supplied, use it to set the
   stack-size attribute and set a few other thread attributes,
   and set attrp pointing to thread attributes object */

if (argc > 1) {
     int stack_size;
     void *sp;

     attrp = &attr;

     s = pthread_attr_init(&attr);
     if (s != 0) handle_error_en(s, "pthread_attr_init");

     s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
     if (s != 0) handle_error_en(s, "pthread_attr_setdetachstate");

     s = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
     if (s != 0)   handle_error_en(s, "pthread_attr_setinheritsched");

     stack_size = strtoul(argv[1], NULL, 0);

     s = posix_memalign(&sp, sysconf(_SC_PAGESIZE), stack_size);
     if (s != 0)  handle_error_en(s, "posix_memalign");

     printf("posix_memalign() allocated at %p\n", sp);

     s = pthread_attr_setstack(&attr, sp, stack_size);
     if (s != 0)  handle_error_en(s, "pthread_attr_setstack");
}

s = pthread_create(&thr, attrp, &thread_start, NULL);
if (s != 0) handle_error_en(s, "pthread_create");
```

```
    if (attrp != NULL) {
        s = pthread_attr_destroy(attrp);
        if (s != 0)  handle_error_en(s, "pthread_attr_destroy");
    }

    pause();     /* Terminates when other thread calls exit() */
}
```

Listing 1.17 – Ex-attributs.c

1.23 Annexe2- Using Barrier Synchronization

- In cases where you must wait for a number of tasks to be completed before an overall task can proceed, barrier synchronization can be used.
- POSIX threads specifies a synchronization object called a **barrier**, along with barrier functions.
- The functions create the barrier, specifying the number of threads that are synchronizing on the barrier, and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier.
- When the last thread arrives at the barrier, all the threads resume execution.
- See Parallelizing a Loop on a Shared-Memory Parallel Computer for more about barrier synchronization.

Initializing a Synchronization Barrier

Use *pthread_barrier_init* to allocate resources for a barrier and initialize its attributes.

```
int pthread_barrier_init(pthread_barrier_t  *barrier,
          const pthread_barrierattr_t *restrict attr,
          unsigned count);

#include <pthread.h>
pthread_barrier_t barrier;
pthread_barrierattr_t attr;
unsigned count;
int ret;
ret = pthread_barrier_init(&barrier, &attr, count);

ret values :
  EINVAL : The value specified by count is equal to 0, or the value specified by attr is invalid
  EAGAIN : The system lacks the necessary resources to initialize another barrier.
  ENOMEM : Insufficient memory exists to initialize the barrier.
  EBUSY : There was an attempt to destroy a barrier while it is in use (for example, while being used in a
pthread_barrier_wait() call) by another thread.
```

- The*pthread_barrier_init()* function allocates any resources required to use the barrier referenced by barrier and initializes the barrier with attributes referenced by attr.

- If attr is NULL, the default barrier attributes are used ; the effect is the same as passing the address of a default barrier attributes object.

• The count argument specifies the number of threads that must call pthread_barrier_wait() before any of them successfully return from the call.

• The value specified by count must be greater than 0.

• *pthread_barrier_init()* returns zero after completing successfully. Any other return value indicates that an error occurred.

Waiting for Threads to Synchronize at a Barrier

Use pthread_barrier_wait(3C) to synchronize threads at a specified barrier. The calling thread blocks until the required number of threads have called pthread_barrier_wait() specifying the barrier. The number of threads is specified in the pthread_barrier_init() function.

When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD is returned to one unspecified thread and 0 is returned to each of the remaining threads. The barrier is then reset to the state it had as a result of the most recent pthread_barrier_init() function that referenced it.

```
int pthread_barrier_wait(pthread_barrier_t  *barrier);

#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_wait(&barrier);

ret value :
EINVAL : The value specified by barrier does not refer to an initialized barrier object.
```

- When *pthread_barrier_wait()* completes successfully,

  → the function returns PTHREAD_BARRIER_SERIAL_THREAD,

  which is defined in pthread.h, for one arbitrary thread synchronized at the barrier.

- The function returns zero for each of the other threads. Otherwise an error code is returned.

Destroying a Synchronization Barrier

- When a barrier is no longer needed, it should be destroyed.

Use the pthread_barrier_destroy(3C) function to destroy the barrier referenced by barrier and release any resources used by the barrier.

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);

#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_destroy(&barrier);

ret values :
EINVAL : Indicates that the value of barrier was not valid.
EBUSY : An attempt was made to destroy a barrier while it is in use (for example, while being used in
a pthread_barrier_wait() by another thread.
```

- pthread_barrier_destroy() returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

Initializing a Barrier Attributes Object

• The *pthread_barrierattr_init* function initializes a barrier attributes object attr with the default values for the attributes defined for the object by the implementation.

• Currently, only the process-shared attribute is provided, and the *pthread_barrierattr_getpshared()* and *pthread_barrierattr_setpshared()* functions are used to get and set the attribute.

• After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) does not affect any previously initialized barrier.

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);

#include <pthread.h>
pthread_barrierattr_t attr;
int ret;
ret = pthread_barrierattr_init(&attr);

ret value
ENOMEM : Insufficient memory exists to initialize the barrier attributes object.
```

- *pthread_barrierattr_init()* returns zero after completing successfully.
- Any other return value indicates that an error occurred.

Setting a Barrier Process-Shared Attribute

• The *pthread_barrierattr_setpshared()* function sets the process-shared attribute in an initialized attributes object referenced by attr.

• The process-shared attribute can have the following values :

➞ *PTHREAD_PROCESS_PRIVATE*

The barrier can only be operated upon by threads created within the same process as the thread that initialized the barrier. This is the default value of the process-shared attribute.

➞ *PTHREAD_PROCESS_SHARED*

The barrier can be operated upon by any thread that has access to the memory where the barrier is allocated.

```
int pthread\_barrierattr\_setpshared(pthread\_barrierattr\_t *attr, int pshared);

pthread\_barrierattr\_setpshared() Return Values

ret value :
EINVAL :Indicates that the value of attr was not valid, or the new value specified for the pshared is not valid.
```

- *pthread_barrierattr_setpshared()* returns zero after completing successfully. Any other return value indicates that an error occurred.

Getting a Barrier Process-Shared Attribute

• The *pthread_barrierattr_getpshared* function obtains the value of the process-shared attribute from the attributes object referenced by attr. The value is set by the *pthread_barrieratt* function.

```
int pthread\_barrierattr\_getpshared(const pthread\_barrierattr\_t *restrict attr,
          int *restrict pshared);

pthread\_barrierattr\_getpshared() Return Values


ret value :
EINVAL : Indicates that the value of attr was not valid.
```

• *pthread_barrierattr_getpshared()* returns zero after completing successfully, and stores the value of the process-shared attribute of attr into the object referenced by the psha-red parameter. Any other return value indicates that an error occurred.

Destroying a Barrier Attributes Object

- The *pthread_barrierattr_destroy()* function destroys a barrier attributes object.

- A destroyed attr attributes object can be reinitialized using *pthread_barrierattr_init()*.

- After a barrier attributes object has been used to initialize one or more barriers, destroying the object does not affect any previously initialized barrier.

```
#include <pthread.h>
int  ret= pthread\_barrierattr\_destroy(pthread\_barrierattr\_t *attr);

ret value :
EINVAL : Indicates that the value of attr was not valid.
```

- *pthread_barrierattr_destroy()* returns zero after completing successfully. Any other return value indicates that an error occurred.

1.23.1 Schéma d'utilisation des Barrières

- compteur_barrière= nombre de threads de l'étape A,
- initialiser le mutex et la variable condition,
- démarrer les threads de l'étape A,
- démarrer les threads de l'étape B,
- attendre que les threads des 2 étapes soient terminés (comment ? avec une autre barrière sur les threads de l'étape B !).

```
thread A
    // réaliser la tache de la partie A de l'algorithme
    { ... }
    // une tache est terminée, décrémenter le compteur de taches terminées
    pthread_mutex_lock(&mutex_barriere);
    compteur_barrière= compteur_barrière -1;
    if(compteur_barrière == 0)
       // la dernière tache de la partie A vient de se terminer, réveiller les threads B
       pthread_cond_broadcast(attente_barrière);
    pthread_mutex_unlock(&mutex_barrière);

thread B
    // attendre que toutes les taches de la partie A se terminent
    pthread_mutex_lock(&mutex_barriere);
    while(compteur_barrière != 0)
       pthread_cond_wait(&attente_barrière, &mutex_barrière);
    pthread_mutex_unlock(&mutex_barrière);
    // réaliser la tache de la partie B de l'algorithme
    { ... }
```

Ex (douche.c)

Table des matières

1.1 Threads POSIX 1

1.2 Une introduction par des exemples 2

1.2.1 Création d’un thread 2

1.2.2 Attente d’un thread 3

1.2.3 Exemple-1 (trivial) 4

1.2.4 Exemple-2 avec passage de paramètre 6

1.2.5 Exemple-3 : renvoi de valeur par un thread via join() 10

1.2.6 Exemple-4 : main laisse finir les threads 14

1.2.7 Exemple-5 : renvoi d’une valeur par un thread 16

1.2.8 Abandon du temps d’exécution 19

1.2.9 Erreurs à éviter lors du passage des paramètres 21

1.2.10 Mutex : fonctions de manipulation des verrous 26

1.2.11 Exemple-6 : Mutex simple . . . 28
1.3 Thread comme processus léger . . . 31
1.4 Intérêts des threads . . . 33
1.5 Exemple fork vs. thread . . . 34
1.6 Thread or not thread ! . . . 37
1.7 Les APIs Pthread . . . 41
1.8 Création et terminaison des threads . . . 45
1.8.1 Création des threads : détails . . . 46
1.8.2 Terminaison des threads . . . 49
1.8.3 Sur la terminaison des threads . . . 50
1.8.4 Exemple (multilingue) . . . 52
1.8.5 Notes sur la terminaison . . . 56
1.9 Joindre ou détachement de threads . . . 60
1.9.1 Exemple-1 (simple) . . . 63
1.9.2 Exemple-2 . . . 64
1.9.3 Exemple plus complet : alarme . . . 68

1.10 Gestion de la pile 71

1.10.1 Un exemple 72

1.11 Utilisation des attributs de thread 74

1.12 Gestion dynamique des threads 84

1.13 Mutex 85

1.13.1 Un exemple simple 85

1.14 Non déterminisme et concurrence 88

1.14.1 Fonctionnement d'un Mutex 90

1.14.2 Exemple-1 simple 95

1.14.3 Initialisation de Mutex 97

1.14.4 Exemple-2 99

1.14.5 Exemple-3 : estimation parallèle de PI 101

1.14.6 Exemple-4 : produit vectoriel parallèle 103

1.15 Problème d'interblocage 108

1.16 Schéma Lecteurs / Rédacteurs 110

1.16.1 L/R Sans synchronisation 111

1.16.2 Avec Mutex : principe + exercice . . . 114
1.17 Variables de condition . . . 116
1.17.1 Création et destruction des VarConds . . . 121
1.17.2 Schéma général de signale/attendre . . . 126
1.17.3 Un premier exemple (Robot R1) . . . 127
1.17.4 Exemple 2 : Robot R1 (bis) . . . 132
1.17.5 Exemple 3 : condition timeout . . . 136
1.17.6 Exemple 4 : timeout (bis) . . . 140
1.18 Sémaphores Posix . . . 144
1.18.1 Utilisation des Sémaphores POSIX . . . 147
1.18.2 Exemple . . . 150
1.18.3 Exercice . . . 155
1.18.4 Solution . . . 157
1.19 Application : Robot R1 . . . 160
1.19.1 Approche décentralisée à base de réflexe (cybernétique) . . . 160
1.19.1.1 R1 : un Robot simple . . . 162

1.19.1.2 Code R1 165
1.20 Annexe-1 : Bibliothèque des threads 172
1.21 Exemple-1 : manipulation de la Pile 176
1.22 Exemple-2 : manipulation des attributs 181
1.23 Annexe2- Using Barrier Synchronization 187
1.23.1 Schéma d'utilisation des Barrières 199

Listings

1.1 ex-super-simple.c 4
1.2 Ex-1-creation-join.c 6
1.3 Ex-2-join-recupere-params.cpp 10
1.4 terminaison_de_main.c 14
1.5 Ex-recup-param-par-join.c 16
1.6 Ex-pb-param.c 22
1.7 Ex-mutex-join.c 28
1.8 hello_arg2.cpp 52
1.9 terminaison_de_main.c 57
1.10 alarm_thread.c 69
1.11 Ex-mutex-join.c 99
1.12 lect-redac1.c 111
1.13 Ex-mutex-cond-simple.c 128
1.14 Ex-condition-timeout.c 137

1.15 condvar.c 140
1.16 Ex-stack.c 176
1.17 Ex-attributs.c 181