OS: Synthèse C

Bruno Sylin

9 janvier 2018

Table des matières

1 Intro

Programme minimal :

```
int main (void)
{
   return 0;
}
```

Pour afficher quelque chose sur la sortie standard (souvent l'écran), on peut utiliser la fonction *puts* qui prend comme paramètre un pointeur vers une chaîne de caractère :

```
#include <stdio.h>

int puts (char const *);
```

Les différents types et objets possibles + glossaire :

Bit Le bit (BInary digiT) ou chiffre binaire est l'unité de mesure d'information. Il peut prendre 2 valeurs : 0 ou 1.

Byte Le byte (ou multiplet) est le plus petit objet adressable pour une implémentation donnée. En langage C, il fait au moins huit bits.

Caractère Un caractère est une valeur numérique qui représente un glyphe visible ('A', '5', '*' etc.) ou non (CR, LF etc.). Un caractère est de type int. Cependant, sa valeur tient obligatoirement dans un type char.

Chaîne de caractères Une chaîne de caractères est une séquence de caractères encadrée de double quotes.

```
"Hello world!"
```

La représentation interne d'une chaîne de caractères est spécifiée. C'est un tableau de char terminé par un «\0», le caractère de fin de chaine.

```
char s[] = "Hello";
```

est équivalent à :

```
char s[] = {'H','e','l','l','o',\0};
```

«Suite d'objets»

Tableau Un tableau est une suite d'objets identiques consécutifs.
Exemple :

```
#include <stddef.h>
   size_t size_of_tab = sizeof (long); // type size_t
   long tab[5] = {12.34, 56.78};
   size_t size_of_tab = sizeof tab / sizeof tab[0];
   // donne le nombre d'éléments dans le tableau tab
```

Pour un tableau a 2 dimensions :

```
p = (int *)t;  // pas sur a 100 %
<type> t[N][M];
t[i][j] = *(p + N*i + j) /* ou encore p[N*i + j] */
// permet de parcourir le tableau
```

Types : valeur minimal / valeur maximal les termes entre « () » sont facultatifs.

char 0 127

unsigned char 0 255

signed char -127 127

(signed) short (int) -32767 32767

unsigned short (int) 0 65535

(signed) int -32767 32767

unsigned (int) 0 65535

(signed) long (int) -2147483647 2147483647

unsigned long (int) 0 4294967295

1.0.1 Le mot-clé const

Il permet de déclarer une variable constante par exemple :

```
const int n = 10;
```

1.1 printf

Formateur de base Rôle Type attendu Types compatibles

% affiche le glyphe du caractère % Type attendu : int Types compatibles : short, char

c affiche le glyphe d'un caractère imprimable Type attendu : int Types compatibles : short, char

d affiche la valeur d'un entier en décimal Type attendu : int Types compatibles : short, char

f affiche la valeur d'un réel en décimal avec virgule fixe Type attendu : double Type compatibles : float

s affiche les glyphes d'une chaîne de caractères Type attendu : char *

```
#include <stdio.h>

int main (void)
{
    printf ("%c\n", 'A');
    printf ("%d\n", 123);
    printf ("%d, '%c'\n", 'A', 'A');

    printf ("Hello world\n");
    printf ("Hello %s\n", "world");
    printf ("%s\n", Hello world);

    return 0;
}
```

Renvoie :

```
A
123
65, 'A'
Hello world
Hello world
Hello world
```

1.2 Fonction

Une fonction est une séquence d'instructions dans un bloc nommé. Une fonction est constituée de la séquence suivante :

type (ou le mot clé void)

identificateur (le nom de la fonction, ici main)

parenthèse ouvrante « (»

liste de paramètres ou une liste vide (dans ce cas, on écrit void)

parenthèse fermante «) »

accolade ouvrante «{»

liste d'instructions terminée par un point virgule «;», ou rien

accolade fermante «}».

L'utilisateur peut créer ses propres fonctions. Une fonction peut appeler une autre fonction. Généralement, une fonction réalisera une opération bien précise. L'organisation hiérarchique des fonctions permet un raffinement en partant du niveau le plus élevé (main()) en allant au niveau le plus élémentaire (atomique).
On gardera cependant en tête que la multiplication des niveaux introduit une augmentation de la taille du code et du temps de traitement.
Une fonction ne peut être appelée qu'à partir d'une autre fonction.
Il faut savoir qu'en C, un paramètre ne fait que transmettre une valeur. Modifier la valeur d'un paramètre n'aura jamais d'effet sur la valeur originale :

```
#include <stdio.h>

void fonction (int x)
{
  printf ("x = %d (dans la fonction)\n", x); /* x = 123 */

  x = 456;
  printf ("x = %d (dans la fonction)\n", x); /* x = 456 */
}

int main (void)
{
  int a = 123;

  printf ("a = %d\n", a); /* a = 123 */

  fonction (a);
  printf ("a = %d\n", a); /* a = 123 */
  return 0;
}
```

1.2.1 Déclaration / implémentation

Comme en C++
Exemple de déclaration

```
int function (void);
```

1.2.2 Les macros

```
#define <macro> <le texte de remplacement>
```

Peut également être utilisé comme petite fonction, par exemple :

```
#define carre(x) ((x) * (x))
carre(3)  // sera remplacé par 3 * 3
```

1.2.3 Typedef

Le C dispose d'un mécanisme très puissant permettant au programmeur de créer de nouveaux types de données en utilisant le mot clé typedef. Par exemple :

```
typedef int ENTIER;
ENTIER a, b;
```

Définit le type ENTIER comme n'étant autre que le type int. Bien que dans ce cas, un simple #define aurait pu suffire, il est toujours recommandé d'utiliser typedef qui est beaucoup plus sûr.

1.3 Compilation

TODO :

1.4 Les pointeurs

Le langage C dispose d'un opérateur «&» permettant de récupérer l'adresse en mémoire d'une variable ou d'une fonction quelconque. Par exemple, si n est une variable, &n désigne l'adresse de n.
Le C dispose également d'un opérateur «*» permettant d'accéder au contenu de la mémoire dont l'adresse est donnée. Par exemple, supposons qu'on ait :

```
n = 10;
// est tout a fait identique à
* ( &n ) = 10;
```

Les pointeurs en langage C sont typés et obéissent à l'arithmétique des pointeurs que nous verrons un peu plus loin. Supposons que l'on veuille créer une variable « p » destinée à recevoir l'adresse d'une variable de type int. « p » s'utilisera alors de la façon suivante :

```
int n;
int * p;
p = &n;
*p = 5;
```

Exemple dans une fonction qui permute deux variables :

```
#include <stdio.h>

void permuter(int * addr_a, int * addr_b);

int main(){
    int a = 10, b = 20;

    permuter(&a, &b);
    printf("a = %d\nb = %d\n", a, b);

    return 0;
}

void permuter(int * addr_a , int * addr_b)
/***************\
* addr_a <-- &a *
* addr_b <-- &b *
\***************/
{
    int c;

    c = *addr_a;
    *addr_a = *addr_b;
    *addr_b = c;
}
```

1.4.1 Les pointeurs générique

Le type des pointeurs génériques est void *. Comme ces pointeurs sont génériques, la taille des données pointées est inconnue et l'arithmétique des pointeurs ne s'applique donc pas à eux. De même, puisque la taille des données pointées est inconnue, l'opérateur d'indirection * ne peut être utilisé avec ces pointeurs, un cast est alors obligatoire. Par exemple :

```
int n;
void * p;

p = &n;
*((int *)p) = 10;
/* p étant désormais vu comme un int *, on peut alors lui appliquer l'opérateur *.
*/
```

Étant donné que la taille de toute donnée est multiple de celle d'un char, le type char * peut également être utilisé en tant que pointeur universel. En effet, une variable de type char * est un pointeur sur octet autrement dit il peut pointer n'importe quoi. Cela s'avère pratique des fois (lorsqu'on veut lire le contenu d'une mémoire octet par octet par exemple) mais dans la plupart des cas, il vaut mieux toujours utiliser les pointeurs génériques. Par exemple, la conversion d'une adresse de type différent en char * et vice versa nécessite toujours un cast, ce qui n'est pas le cas avec les pointeurs génériques.
Dans printf, le spécificateur de format %p permet d'imprimer une adresse (void *) dans le format utilisé par le système.
Et pour terminer, il existe une macro à savoir NULL, définie dans stddef.h, permettant d'indiquer qu'un pointeur ne pointe nulle part. Son intérêt est donc de permettre de tester la validité d'un pointeur.Il est conseillé de toujours initialiser un pointeur à NULL.

1.5 lvalue et rvalue

Dois-je vraiment rappeller comment ça fonctionne ? En gros, une lvalue est quelque chose qui peut ce situer à gauche du «=» et rvalue a droite. Une rvalue possède une valeur mais pas d'adresse.

1.6 Opérateur usuel et logique

usuel :

< Inférieur à

> Supérieur à

== Égal à

<= Inférieur ou égal à

>= Supérieur ou égal à

!= Différent de

logique :

&& ET

|| OU

! NON

Dans une opération ET, l'évaluation se fait de gauche à droite. Si l'expression à gauche de l'opérateur est fausse, l'expression à droite ne sera plus évaluée car on sait déjà que le résultat de l'opération sera toujours FAUX.
Dans une opération OU, l'évaluation se fait de gauche à droite. Si l'expression à gauche de l'opérateur est vrai, l'expression à droite ne sera plus évaluée car on sait déjà que le résultat de l'opération sera toujours VRAI.

On peut séparer plusieurs expressions à l'aide de l'opérateur virgule. Le résultat est une expression dont la valeur est celle de l'expression la plus à droite. L'expression est évaluée de gauche à droite.

```
(a = -5, b = 12, c = a + b) * 2 ;    // renvoie 14.
```

1.6.1 l'opérateur ternaire « ? : »

Une expression conditionnelle est une expression dont la valeur dépend d'une condition. L'expression :

```
p ? a : b ;
```

vaut a si p est vrai et b si p est faux.

1.7 Cast

```
float f;
f = (float)3.1416;
```

2 Les instructions

2.1 if

```
if ( <expression> )
  {
    les instructions
  }
else if (<expression>)
{
  les instructions
}
else
  {
    les instructions
  }
```

2.2 do

```
do
  {
    les instructions
  }
while ( <expression> );
```

2.3 while

```
while ( <expression> )
  {
    les instructions
  }
```

2.4 for

```
for ( <init> ; <condition> ; <step>)
  les instructions
```

2.5 break

L'instruction break termine l'exécution de l'instruction *do*, *for*, *switch* ou *while* englobante la plus proche dans laquelle elle figure. Le contrôle est transmis à l'instruction qui suit l'instruction terminée.

2.6 switch

```
#include <stdio.h>

int main()
{
  int n;
```

```
  printf("Entrez un nombre entier : ");
  scanf("%d", &n);

  switch(n)
  {
  case 0:
    printf("Cas de 0.\n");
    break;
  case 1:
    printf("Cas de 1.\n");
    break;
  case 2: case 3:
    printf("Cas de 2 ou 3.\n");
    break;
  case 4:
    printf("Cas de 4.\n");
    break;
  default:
    printf("Cas inconnu.\n");
  }
  return 0;
}
```

2.7 continue

Dans une boucle, permet de passer immédiatement à l'itération suivante. Par exemple, modifions le programme table de multiplication de telle sorte qu'on affiche rien pour n = 4 ou n = 6.

```
#include <stdio.h>

int main()
{
  int n;
  for(n = 0; n <= 10; n++)
  {
    if ((n == 4) || (n == 6))
    continue; // termine la boucle actuelle du for
  printf("5 x %2d %2d\n", n, 5 * n);
  }
  return 0;
}
```

3 L'allocation dynamique de mémoire

3.1 Les fonctions malloc et free

L'intérêt d'allouer dynamiquement de la mémoire se ressent lorsqu'on veut créer un tableau dont la taille dont nous avons besoin n'est connue qu'à l'exécution par exemple. On utilise généralement les fonctions malloc et free.

```
int t[10];
...
/* FIN */
```

Peut être remplacé par :

```
int * p;

p = malloc(10 * sizeof(int));
...
free(p); /* libérer la mémoire lorsqu'on n'en a plus besoin */
```

```
  /* FIN */
```

Les fonctions malloc et free sont déclarées dans le fichier stdlib.h. malloc retourne NULL en cas d'échec. Voici un exemple qui illustre une bonne manière de les utiliser :

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

int main()
{
  int * p;

  /* Creation d'un tableau assez grand pour contenir 10 entiers */
  p = malloc(10 * sizeof(int));

  if (p != NULL)
  {
    printf("Succes de l'operation.\n");
    p[0] = 1;
    printf("p[0] = %d\n", p[0]);
    free(p); /* Destruction du tableau. */
  }
  else
    printf("Le tableau n'a pas pu etre cree.\n");
  return 0;
}
```

3.2 La fonction realloc

```
void * realloc(void * memblock, size_t newsize);
```

Permet de « redimensionner » une mémoire allouée dynamiquement (par malloc par exemple). Si memblock vaut NULL, realloc se comporte comme malloc. En cas de réussite, cette fonction retourne alors l'adresse de la nouvelle mémoire, sinon la valeur NULL est retournée et la mémoire pointée par memblock reste inchangée.

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

int main()
{
    int * p = malloc(10 * sizeof(int));

    if (p != NULL)
    {
        /* Sauver l'ancienne valeur de p au cas ou realloc echoue. */
        int * q = p;
        /* Redimensionner le tableau. */
        p = realloc(p, 20 * sizeof(int));

        if (p != NULL)
        {
            printf("Succes de l'operation.\n");
            p[0] = 1;
            printf("p[0] = %d\n", p[0]);
            free(p);
        }
        else
        {
            printf("Le tableau n'a pas pu etre redimensionne.\n");
            free(q);
        }
```

```
    }
    else
        printf("Le tableau n'a pas pu etre cree.\n");

    return 0;
}
```

4 Les caractères

4.1 Le jeu de caractères du C

Voir Bonne pratique de codage en C

4.2 Commentaire

Préférer les commentaires en dessous ou au dessus d'une ligne plutôt que en bout de ligne, car souvent ça rend la ligne trop longue.
Pour les commentaires de plusieurs lignes, utiliser plutôt /* */.
S'il faut isoler provisoirement une portion de code, le mieux est de ne pas utiliser les commentaires (il pourrait y avoir des commentaires imbriqués), mais plutôt les directives du préprocesseur : #ifdef .. #endif ou #if .. #endif

```
#if 0
  /* Compteur */
  int cpt ;
#endif
```

Commenter le moins possible. Le principe est de ne commenter que ce qui apporte un supplément d'information. Il est d'usage d'utiliser en priorité l'auto-documentation, c'est à dire un choix judicieux des identificateurs qui fait que le code se lit 'comme un livre'...

5 Les threads, ce qui est important quoi

On sait tous ce qu'est un tread, si tu sais pas, tu es dans la merde pour ton oral et va d'abord voir ton cours !

5.1 Compilation

Toutes les fonctions relatives aux threads sont incluses dans le fichier d'en-tête <pthread.h> et dans la bibliothèque libpthread.a (soit -lpthread à la compilation).
Exemple :
Voici la ligne de commande qui vous permet de compiler votre programme sur les threads constitué d'un seul fichier.
gcc -lpthread <nom du fichier>.c -o <Output>
Et n'oubliez pas d'ajouter #include <pthread.h> au début de vos fichiers. Bon, tout ça on aura pas à l'examen, c'est juste pour le projet de C.

5.2 Les threads en eux-mêmes

5.2.1 Créer un thread

Pour créer un thread, il faut déjà déclarer une variable le représentant.Celle-ci sera de type pthread_t (qui est, sur la plupart des systèmes, un typedef d'unsigned long int). Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

```
#include <pthread.h>

  int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                     void *(*start_routine) (void *), void *arg);
```

Ce prototype est un peu compliqué, c'est pourquoi nous allons récapituler ensemble.

La fonction renvoie une valeur de type int : 0 si la création a été réussie ou une autre valeur si il y a eu une erreur.

Le premier argument est un pointeur vers l'identifiant du thread (valeur de type pthread_t).

Le second argument désigne les attributs du thread. Vous pouvez choisir de mettre le thread en état joignable (par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...). Dans nos exemple, on mettra généralement NULL.

Le troisième argument est un pointeur vers la fonction à exécuter dans le thread. Cette dernière devra être de la forme «void *fonction(void* arg)» et contiendra le code à exécuter par le thread.

Enfin, le quatrième et dernier argument est l'argument à passer au thread.

5.2.2 Suprimer un thread

```
#include <pthread.h>

void pthread_exit(void *ret);
```

Elle prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

5.2.3 Exemple simple

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

void *thread_1(void *arg) {
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;  // le warning enlevé est celui-ci:
    /* format '%s' expects argument of type 'char*',
    *  but argument 2 has type 'void*' [-Wformat=]
    *  en gros, il faut reconvertir l'argument
    *  s'il y en a un pour pas avoir d'erreur de format */
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thread1;  // thread1 contiendra le thread

    printf("Avant la création du thread.\n");

    /* appelle la fonction thread_1 dans la condition du if
    * (pour direct renvoyer une erreur si besoin) */
    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
            perror("pthread_create");
      return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");

    return EXIT_SUCCESS;
}
```

On a bien sur un soucis, c'est que ici, le programme s'arrête avant que le thread ait pu finir son calcul (bon, ici, ce n'est qu'afficher quelque chose, mais quand-même). Pour éviter ce soucis, on peut utiliser « pthread_join »

5.2.4 Attendre un thread avec pthread_join

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Elle prend donc en paramètre l'identifiant du thread et son second paramètre, un pointeur, permet de récupérer la valeur retournée par la fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de pthread_exit).

Du coup dans l'exemple précédent, on peut rajouter ceci(après la création du thread, bien sur) :

```
  if (pthread_join(thread1, NULL)) {
    perror("pthread_join");
    return EXIT_FAILURE;
    }
```

Donc, pour ceux qui n'ont pas suivi :

1. créer une fonction qui va permettre de faire tous les calculs qu'on veut dans les autres threads que le thread principal ;
2. initialiser une variable de type pthread_t pour chaque thread qu'on veut créer ;
3. appeler cette fonction (dans un if, souvent, ça permet de gérer les éventuelles erreurs plus facilement) ;
4. appeler la fonction pthread_join pour "fusionner" les deux threads (en réalité, attendre le thread en retard sur l'autre)

5.3 Mutex et toute ces belles petites choses

Bon, c'est sympa tout ça, mais il y a encore un soucis. Toutes les variables sont partagées, et on se rend vite compte que ça peut poser problème, si on veut faire plusieurs calculs sur une même variable, mais l'un à la suite de l'autre, ça risque de coincer, on ne sait pas dans quel ordre vont se faire les calculs. Ce qui veut dire que nous risquons de modifier une valeur dans un thread alors qu'un autre thread en avait besoin sans modifications.(je ne sais pas si j'en ai pas perdu par hasard, si c'est le cas, sachez juste que les mutex sont vachement utiles pour garder visible une variable qu'à un seul thread et la cacher à tous les autres).
Il faut aussi de nouveau créer une variable, mais cette fois avec un type «pthread_mutex_t».
Le problème, c'est qu'il faut que le mutex soit accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale. Mais pour le faire plus proprement, on peut utiliser une structure avec la donnée à protéger. Allez, encore un exemple :

```
typedef struct data {
    int var;
    pthread_mutex_t mutex;
} data;
```

Ainsi, nous pourrons passer la structure en paramètre à nos threads quand nous les créons.
Du coup, voilà comment on initialise un mutex en prenant en compte la convention qui veut qu'on l'initialise mutex avec la valeur de la constante PTHREAD_MUTEX_INITIALIZER, déclarée dans pthread.h.

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

typedef struct data {
    int var;
    pthread_mutex_t mutex;
} data;

int main(void){
    data new_data;

    new_data.mutex = PTHREAD_MUTEX_INITIALIZER;

    return EXIT_SUCCESS;
}
```

5.3.1 Pour verrouiller un mutex de nom mut

```
int pthread_mutex_lock(pthread_mutex_t *mut);
```

5.3.2 Pour déverrouiller un mutex de nom mut

```
int pthread_mutex_unlock(pthread_mutex_t *mut);
```

5.3.3 Pour détruire un mutex de nom mut

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

5.4 Les conditions (permettent d'attendre un autre thread)

Quand un thread est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre thread.

5.4.1 Initiation

```
static pthread_cond_t cond_stock = PTHREAD_COND_INITIALIZER;
```

Les conditions reposent essentiellement sur deux fonctions. l'une permet de mettre en attente un thread et la seconde permet de signaler que la condition est remplie ce qui réveille alors le thread qui est en attente de cette condition. Plusieurs threads peuvent surveiller la même condition.

5.4.2 Exemple complet de mutex

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;  // init le mutex

void* doSomeThing(void *arg){
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;

    printf("\n Job %d started\n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void){
    int i = 0;
    int err;

    if (pthread_mutex_init(&lock, NULL) != 0){
        printf("\n mutex init failed\n");
        return 1;
    }
    while(i < 2){
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

ma source pour cet exemple

5.4.3 Le thread attend la condition

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Cette fonction permet de mettre le thread appelant en attente de la condition, il suspend donc son exécution temporairement. Ses deux arguments sont :

L'adresse de la variable condition de type pthread_cond_t.

L'adresse d'un mutex Une condition est en effet, toujours associée à un mutex

5.4.4 Réveiller un thread

pthread_cond_signal est la fonction qui permet de signaler la condition au thread qui l'attend. Elle prend en paramètre l'adresse de la variable-condition surveillée. Cette fonction ne permet de réveiller qu'un seul thread.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

5.4.5 Réveiller plusieurs thread

Cette fonction permet de réveiller tous les thread qui surveille la condition cond. Tout comme pthread_cond_signal, elle prend en paramètre l'adresse de la variable-condition surveillée.

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

cours complet sur les threads

5.5 Exemple avec les conditions et les mutex

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

// Création de la condition
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
//Création du mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void){
        pthread_t monThreadCompteur;
        pthread_t monThreadAlarme;

        pthread_create (&monThreadCompteur, NULL, threadCompteur, (void*)NULL);
    // Création des threads
        pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL);

        pthread_join (monThreadCompteur, NULL);
    // Attente de la fin des threads
        pthread_join (monThreadAlarme, NULL);

        return 0;
}

void* threadCompteur (void* arg) {
        int compteur = 0, nombre = 0;

        srand(time(NULL));

    // Boucle infinie
        while(1) {
        // On tire un nombre entre 0 et 10
                nombre = rand()%10;
        // On ajoute ce nombre à la variable compteur
                compteur += nombre;

                printf("\n%d", compteur);

        // Si compteur est plus grand ou égal à 20
                if(compteur >= 20) {
            // On verrouille le mutex
                        pthread_mutex_lock (&mutex);
            // On délivre le signal : condition remplie
                        pthread_cond_signal (&condition);
            // On déverrouille le mutex
                        pthread_mutex_unlock (&mutex);

            // On remet la variable compteur à 0
                        compteur = 0;
                }
        // On laisse 1 seconde de repos
                sleep (1);
        }
    // Fin du thread
        pthread_exit(NULL);
```

```
}

void* threadAlarme (void* arg) {
    // Boucle infinie
        while(1) {
        // On verrouille le mutex
                pthread_mutex_lock(&mutex);
        // On attend que la condition soit remplie
                pthread_cond_wait (&condition, &mutex);
                printf("\nLE COMPTEUR A DÉPASSÉ 20.");
        // On déverrouille le mutex
                pthread_mutex_unlock(&mutex);
        }
    // Fin du thread
        pthread_exit(NULL);
}
```

Lien de l'exemple

6 Gestion des processus

Pour cette partie je me suis plus inspiré des slides des tp.

6.1 Création des processus

6.1.1 fork

Fork permet de cloner intégralement le processus courant avec un pid différent pour son fils. Du coup, fork est la seule fonction qui renvoie deux valeur :

1. dans le processus fils (le clone), fork renvoie 0
2. dans le processus père (l'original), fork renvoie le pid du fils

Exemple :

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork();
// renvoie 0 si c'est l'enfant
// < 0 si il y a une erreur dans le fork
// > 0 si c'est le parent
```

Exemple complet :

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int var_glb; /* A global variable*/

int main(void) {
  pid_t childPID;
  int var_lcl = 0;

  childPID = fork();

  if(childPID >= 0) { // fork was successful
    if(childPID == 0) { // child process
      var_lcl++;
      var_glb++;
      printf("\n Child :: var_lcl = [%d], var_glb[%d]\n", var_lcl, var_glb);
    }
    else { //Parent process
      var_lcl = 10;
      var_glb = 20;
      printf("\n Parent :: var_lcl = [%d], var_glb[%d]\n", var_lcl, var_glb);
    }
  }
  else { // fork failed
    printf("\n Fork failed, quitting !!!!!!\n");
    return 1;
  }
  return 0;
}
```

Les variables var_lcl et var_glb sont utilisé pour montrer que le père et le fils travail bien sur des variables séparé.
Output :
Parent : : $\text{var}_lcl = [10], var_glb[20]$
$Child :: var_lcl = [1], var_glb[1]$
Liens de l'exemple
Fork pas a pas
Reference anglais pour fork

6.1.2 getpid()

Comment savoir le pid du processus courant ?

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

Et pour le pid du père :

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid(void);
```

6.2 Execution d'un programme depuis un programme C, "systeme calls"

Utilise la famille exec
Lorsqu'un processus fait appel à exec, le processus appelant est remplacé par le programme passé en paramètre. Si cela fonctionne, l'appel à exec ne retourne pas.
les différentes variantes :

```
// permet de passer les paramètres du programme
// en utilisant une liste d'arguments terminée par NULL
int execl(const char *program, const char *arg, ...);
execl("/bin/ls", "/bin/ls", "-l", "-a", NULL );

// cherche le programme dans la variable PATH
int execlp(const char *program, const char *arg, ...);
execlp("ls", "ls", "-l", "-a", NULL );

// avec un vecteur pour les paramètres et dois donner le chemin absolu
int execv(const char *program, const char *argv [ ]);
char *arguments [4];
arguments[0] = "/bin/ls";
arguments[1] = "-l";
arguments[2] = "-a";
arguments[3] = NULL;
execv("/bin/ls", arguments);


// avec un vecteur pour les paramètres et dois donner le chemin absolu
int execvp(const char *program, const char *argv [ ]);
char *arguments [4];
arguments[0] = "ls";
arguments[1] = "-l";
arguments[2] = "-a";
arguments[3] = NULL;
execvp("ls", arguments);
```

Référence en anglais pour approfondir

6.2.1 Attendre la fin d'un processus

Attendre la fin d'un processus peut être intéressant quand on attend de ce processus un résultat.
Utilise la famille exec

```
#include<sys/types.h>
#include<sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait : attent la fin d'un des processus fils, retourne son PID et stocke dans l'entier pointé par status (si status est différent de NULL), le code de retour du processus fils (ce qui est renvoyé par la fonction main du processus fils).

waitpid : comme wait mais attend uniquement le processus fils précisé par le paramètre PID. Le paramètre entier «option» permet de rentre l'appel à waitpid non bloquant avec la constante WNOHANG, si cela n'est pas nécessaire, on utilise la valeur 0.

Pour savoir si le processus c'est terminé correctement, on utile la macro WIFEXITED(status) qui renvoie vrai si le processus fils s'est terminé normalement.

Ensuite pour avoir le résultat que le processus a renvoyé (ce que son main a renvoyé), on utilise la macro WEXITSTATUS(status).

Et voilà, encore bien sur un exemple :

```
/* CELEBW02

   The following function suspends the calling process using &waitpid.
   until a child process ends.

*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = fork()) < 0)
    perror("fork() error");
  else if (pid == 0) {
    sleep(5);
    exit(1);
  }
  else do {
    if ((pid = waitpid(pid, &status, WNOHANG)) == -1)
      perror("wait() error");
    else if (pid == 0) {
      time(&t);
      printf("child is still running at %s", ctime(&t));
      sleep(1);
    }
    else {
      if (WIFEXITED(status))
        printf("child exited with status of %d\n", WEXITSTATUS(status));
      else puts("child did not exit successfully");
    }
  }
  while (pid == 0);
}
```

Output :
child is still running at Mon Jan 8 11 :05 :43 2018
child is still running at Mon Jan 8 11 :05 :44 2018
child is still running at Mon Jan 8 11 :05 :45 2018
child is still running at Mon Jan 8 11 :05 :46 2018
child is still running at Mon Jan 8 11 :05 :47 2018
child is still running at Mon Jan 8 11 :05 :48 2018
child is still running at Mon Jan 8 11 :05 :49 2018
child exited with status of 1
Liens de l'exemple avec des explications plus poussées (anglais)

6.3 Les pipes

Les pipes permettent a deux processus sur la même machine de communiquer de manière unidirictionnels entre eux. (pour pouvoir communiquer entre des machines différentes, on va utiliser des serveurs, mais on verra ça plus tard).
Il en existe deux type :

1. anonymes : communication parentale, en famille quoi (père -> fils ; fils -> fils)

2. nommés : utilisés pour la communication entre deux processus indépendants (pas de lien de parenté direct entre les deux), du coup il faudras le nommé d'où le nom.

6.3.1 Comment les créer ?

Les pipes anonymes :

```
#include <unistd.h>

int pipe(int fd[2]);
```

C'est quoi tout ça ?

L'appel à pipe crée une paire de «file descriptors» stockée dans fd.
fd[0] sert à la lecture du pipe et fd[1] sert à l'écriture dans le pipe.

La fonction pipe renvoie 0 si elle réussie, autre chose sinon

Pour établir une communication (unidirectionelle !) à l'aide d'un pipe, il faut que le processus père crée un pipe avant de faire un (ou plusieurs) appel(s) à fork (sinon l'autre processus ne pourras pas connaître l'entré ou la sortie du pipe). Dès lors, après le fork, le processus père et le(s) processus fils peuvent accéder aux pipe en utilisant fd.

Pour une communication bidirectionelle, il faut bien sur créer deux pipes.

Pour écrire dans un tube (ce n'est pas dans les tp, mais ça me semble plutôt utile) :

```
ssize_t write(int entreeTube, const void *elementAEcrire, size_t nombreOctetsAEcrire);
```

La fonction prend en paramètre l'entrée du tube (on lui enverra fd[1]), un pointeur générique vers la mémoire contenant l'élément à écrire, ainsi que le nombre d'octets de cet élément.
Elle renvoie une valeur de type ssize_t correspondant au nombre d'octets effectivement écrits.

Et pour lire dedans :

```
ssize_t read(int sortieTube, void *elementALire, size_t nombreOctetsALire);
```

La fonction prend en paramètre la sortie du tube (fd[0]), un pointeur vers la mémoire contenant l'élément à lire et le nombre d'octets de cet élément.
Elle renvoie une valeur de type ssize_t qui correspond au nombre d'octets effectivement lus. On pourra ainsi comparer le troisième paramètre (nombreOctetsALire) à la valeur renvoyée pour vérifier qu'il n'y a pas eu d'erreurs.

Petit example offert par openclassrooms :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à écrire */

int main(void) {
    pid_t pid_fils;
    int fd[2];

    char messageEcrire[TAILLE_MESSAGE];

    if (pipe(fd) != 0) {
      perror "erreur a la créeation du pipe"
    }

    pid_fils = fork();

    if(pid_fils != 0) { /* Processus père */
        sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !");
        /* La fonction sprintf permet de remplir
        *  une chaîne de caractère avec un texte donné */

        write(fd[1], messageEcrire, TAILLE_MESSAGE);
    }
    else if(pid_fils == 0) /* Processus fils */
    {
        read(fd[0], messageLire, TAILLE_MESSAGE);
        printf("Message reçu = \"%s\"", messageLire);
    }

    return EXIT_SUCCESS;
}
```

Les pipes nommés :

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Le pipe nommé passe par un fichier spécial qui est créer lors de l'appel de cette fonction

avec comme path le pathname donné en paramètre

et comme permissions, ce qui est indiqué dans le paramètre mode, il y a plusieurs moyen d'utiliser ce paramètre :

1. S_IRWXU : read, write, execute/search by owner
2. S_IRUSR : read permission, owner
3. S_IWUSR : write permission, owner
4. S_IXUSR : execute/search permission, owner
5. S_IRWXG : read, write, execute/search by group
6. S_IRGRP : read permission, group
7. S_IWGRP : write permission, group
8. S_IXGRP : execute/search permission, group
9. S_IRWXO : read, write, execute/search by others
10. S_IROTH : read permission, others

11. S_IWOTH : write permission, others
12. S_IXOTH : execute/search permission, others
13. S_ISUID : set-user-ID on execution
14. S_ISGID : set-group-ID on execution
15. S_ISVTX : on directories, restricted deletion flag
16. Ils peuvent être combiné avec : «|»
17. Ou encore utiliser les valeursdes droits avec des chiffres(sous mode octal)
    (a) Le premier chiffre correspond au propriétaire
    (b) Le second au groupe du propriétaire
    (c) Le troisième a tous les autres

    (d) Chaque chiffre peut avoir une valeur différentes :
        i. 1 exécution
        ii. 2 écriture
        iii. 4 lecture

        Et on fait une somme quand on veut combiner plusieurs permissions
        Exemple :
        Pour attribuer toutes les permissions à vous, seule la lecture pour le groupe, et aucune pour les autres, la valeur correspondante est 0720
        Premier chiffre = 0 (obligatoire)
        Deuxième chiffre = 1 (Exécution) + 2 (Ecriture) + 4 (Lecture)
        Troisième chiffre = 0 (aucune permission)

Et renvoie 0 si elle réussit, ou -1 en cas d'erreur.

Un dernier "petit" exemple : (Ecrivain.c)

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

#define TAILLE_MESSAGE  256

int main(void) {
        int entreeTube;
        char nomTube[] = "essai.fifo";
        char chaineAEcrire[TAILLE_MESSAGE] = "Bonjour";

        if(mkfifo(nomTube, 0644) != 0)  { // créer le tube
                fprintf(stderr, "Impossible de créer le tube nommé.\n");
                exit(EXIT_FAILURE);
        }

        if((entreeTube = open(nomTube, O_WRONLY)) == -1)        {
                fprintf(stderr, "Impossible d'ouvrir l'entrée du tube nommé.\n");
                exit(EXIT_FAILURE);
        }

        write(entreeTube, chaineAEcrire, TAILLE_MESSAGE); // écris dans le tube

        return EXIT_SUCCESS;
}
```

Lecteur.c

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

#define TAILLE_MESSAGE  256

int main(void) {
        int sortieTube;
        char nomTube[] = "essai.fifo";
        char chaineALire[TAILLE_MESSAGE];

        if((sortieTube = open ("essai.fifo", O_RDONLY)) == -1)  {
                fprintf(stderr, "Impossible d'ouvrir la sortie du tube nommé.\n");
                exit(EXIT_FAILURE);
        }

        read(sortieTube, chaineALire, TAILLE_MESSAGE); // lit dans le tube
        printf("%s", chaineALire);

        return EXIT_SUCCESS;
}
```

Les pipes (deux exemples), openclassrooms
Très très bon site pour les pipes

7 TODO :

1. Expliquer comment compiler
2. socket

3. bind
4. connect
5. listen
6. accept
7. send
8. write ?
9. recv
10. shutdown
11. getpeername
12. gethostname
13. gethostbyname
14. select

Comme vous pouvez le voir, il me reste beaucoup moins à faire en fait, il ne me reste plus que les serveurs et la compilation d'un programme. La compilation d'un programme ne risque pas de tomber a l'examen, mais les serveurs, c'est possible, donc c'est sur quoi je me pencherais mardi 9/01.

8 Sponsors

Ce document est complètement sponsorisé par OpenClassrooms, developpez.com et certainement plein d'autres que j'ai déjà oublié.
Liens des bonnes pratiques pour C (oui, il est déjà apparu en haut)
initiation au language C Si vous êtes complètement perdu car vous ne savez pas ce que veut dire C++ et que vous n'en avez jamais fait, c'est assez long, je préviens (en même temps, vous venez de loin)
Le C en 20h askip, bon, je ne pense pas que ce soit la référence la plus intéressante
Un autre cours du C mais uniquement pour les bases (mais déjà bien poussé tout en restant sur la base)
3 tuto en vidéo, je ne les ai pas encore visionné, donc je sais pas vous dire ce que ça vaut, mais je pose ça la quand même
Le cours de OpenClassrooms, 40 h de cours, si vous avez le temps, ou si vous voulez le parcourir rapidement, mais il ne parle pas de threads, juste les fonctionnalités classiques (pointeur, tableau, etc)