```
# Included files:
 * 0_exercice1.c
 * 1_exercice2.c
 * 2_distributeur.c


# Ignored C and H files




commit 76a07bc23662aebdb343d3a07d9eb62c7afd091d
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:47:12 2023 +0200

    🎨 prefix files to correct pdf annexes order

 TP/TP1-threads/{exercice1.c => 0_exercice1.c}       | 0
 TP/TP1-threads/{exercice2.c => 1_exercice2.c}       | 0
 TP/TP1-threads/{distributeur.c => 2_distributeur.c} | 0
 3 files changed, 0 insertions(+), 0 deletions(-)

commit d01d536f0f1f08986f83c1b776f215bb11f96464
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:46:02 2023 +0200

       add proj2pdf and Makefile

 TP/TP1-threads/Makefile |  48 ++++++++++
 TP/TP1-threads/proj2pdf | 239 ++++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 287 insertions(+)

commit 53e04f16eec08310dd028d7d46e510ea37a7beb0
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:45:02 2023 +0200

       exercice 3

 TP/TP1-threads/README.md      |  20 +++++-
 TP/TP1-threads/distributeur.c | 142 +++++++++++++++++++++++++++++++-----------
 2 files changed, 123 insertions(+), 39 deletions(-)

commit f77459ec79e6b30c836a020aedef6c42921e4286
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:44:18 2023 +0200

       exercice 2

 TP/TP1-threads/README.md   |  9 ++++++++
 TP/TP1-threads/exercice2.c | 56 ++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 65 insertions(+)

commit fe9c7c10167849ecb87ca56037cf765c484d8527
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:43:24 2023 +0200

       exercice 1

 TP/TP1-threads/README.md   | 13 +++++++++++++
 TP/TP1-threads/exercice1.c | 43 ++++++++++++++++++-------------------------------
 2 files changed, 31 insertions(+), 25 deletions(-)

commit 4f33b9e85d57aeedf62ded4467a2396e109881ad
Author: AlexisHamon <alexis.hamon24@gmail.com>
Date:   Thu Sep 28 03:41:08 2023 +0200

    🎉 TP1 initial commit

 TP/TP1-threads.tgz                | Bin 0 -> 190839 bytes
 TP/TP1-threads/TP1-threads.pdf    | Bin 0 -> 200830 bytes
 TP/TP1-threads/distributeur.c     |  94 +++++++++++++++++++++++++++++++++++++++++
 TP/TP1-threads/exercice1.c        |  41 ++++++++++++++++++
 4 files changed, 135 insertions(+)
```

TP1 : Threads POSIX

Exercice 1: Comprendre du code avec des threads

Question 1: Que s'est-il passé ?

Les deux threads ont obtenus le verrou l'un après l'autre permettant d'attribuer à `i` la même valeur de `count`. Même si le compteur s'incrémente deux fois par la suite, les deux threads vont pouvoir prétendre avoir incrémenter le même multiple de 10000.

Question 2: Comment corriger le problème ?

Pour corriger le problème, il suffit de mettre l'obtention du compteur et son incrémentation dans la même section critique. C'est ce qui est fait dans le code en annexe dans la fonction `getCount()`.

Exercice 2: Écrire de petits programmes avec des threads

En suivant les questions du sujet jusqu'à la question optionelle 6, on obtient le code en annexe.

En évitant `pthread_exit()`, élimine le psuedo-mem leak de valgrind (bug connu).

`atoi()` est préféré à `sscanf("%d", ...)` dans la lecture des arguments pour éviter de parser la chaîne de formattage.

L'utilisation du mutex n'est nécessaire que lors de l'incrémentation de la somme partielle de chaque thread à la somme globale.

Exercice 3: Communication inter-threads

Pour des raisons de simplification de passage d'arguments, c'est le `distributeur` qui va créer deux threads fils `chiffre` et `lettres`.

On aurai bien pu déplacer le code de `distributeur` dans `main` et éviter la création d'un thread mais cela aurai eu pour effet de changer le skelette de code donné pour l'exercice.

Pour la communication inter-threads, nous sommes dans un cas classique de producteur/consommateur dans lequel le producteur, ici `distributeur`, répartie les produits entre deux consommateurs, `chiffre` et `lettres`.

L'idée va alors d'alouer un espace mémoire afin que le producteur puisse déposer ses produits qui seront consomés par le consommateur sans être bloquer.

Si cela peut se faire avec une variable et deux mutex en attente passive, on en atteint vite les limitations.

On va donc opter pour une solution plus compliquée mais intéressante, l'allocation d'un espace mémoire circulaire `syncbuffer`. On utilise alors des sémaphores plutôt que des mutex pour leur compteurs.

Le fait de disposer d'un seul producteur/consomateur par produit permet d'éviter de devoir poser des sections critiques sur les têtes d'écriture et de lecture du tableau circulaire.

```c
/*
 ********************************************************
 ********* Fichier 0_exercice1.c *********
 ********************************************************
 */
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
long long count = 0;

long long incrementCount() {
  pthread_mutex_lock(&count_mutex);
  long long c = ++count;
  pthread_mutex_unlock(&count_mutex);
  return c;
}

void *threadFunA(void *ignored) {
  long long i;
  while ((i = incrementCount()) < 50000)
    if (i % 10000 == 0)
      printf("A: i = %lld\n", i);
  return NULL;
}
void *threadFunB(void *ignored) {
  long long i;
  while ((i = incrementCount()) < 100000)
    if (i % 10000 == 0)
      printf("B: i = %lld\n", i);
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t pth_a, pth_b;
  pthread_create(&pth_a, NULL, threadFunA, NULL);
  pthread_create(&pth_b, NULL, threadFunB, NULL);
  pthread_join(pth_a, NULL);
  pthread_join(pth_b, NULL);
}


////////////////////////////////// clang-tidy output
```

```c
/*
 ********************************************************
 ********* Fichier 1_exercice2.c *********
 ********************************************************
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

struct thread_args {
  int i;
  int *sum;
  pthread_mutex_t *mutex;
};

void *threadFun(void *ptr) {
  struct thread_args *args = (struct thread_args *)ptr;
  printf("%d: Hello World !\n", args->i);
  printf("%d: Pid: %d\n", args->i, getpid());
  printf("%d: %p\n", args->i, (void *)pthread_self());

  int sum = 0;
  for (unsigned i = 0; i < 1000000; ++i)
    sum += args->i;

  pthread_mutex_lock(args->mutex);
  *(args->sum) += sum;
  pthread_mutex_unlock(args->mutex);

  free(args);
  return NULL;
}

int main(int argc, char **argv) {
  if (argc < 2)
    return 1;
  int n = atoi(argv[1]);

  if (n < 1)
    return 0;

  pthread_mutex_t sum_mutex = PTHREAD_MUTEX_INITIALIZER;
  int sum = 0;

  pthread_t *pth_t = malloc(n * sizeof(pthread_t));

  for (int i = 0; i < n; ++i) {
    struct thread_args *arg = malloc(sizeof(struct thread_args));
    arg->i = i;
    arg->sum = &sum;
    arg->mutex = &sum_mutex;
    pthread_create(&pth_t[i], NULL, threadFun, (void *)arg);
  }
  for (unsigned i = 0; i < n; ++i)
    pthread_join(pth_t[i], NULL);

  free(pth_t);

  printf("Somme: %d\n", sum);
}


////////////////////////////// clang-tidy output
```

```c
/*
 ******************************************************
 ******** Fichier 2_distributeur.c *********
 ******************************************************
 */
/*****************************************************************************
**
** Version initiale.
** ----------------
** Ce programme se compose de trois fonctions. La fonction de distribution
** est chargée de lire les caractères en provenance de l'entrée standard,
** d'envoyer les chiffres à la fonction chiffre (par l'intermédiaire d'un
** appel de fonction classique) et les lettres à la fonction lettre (aussi par
** l'intermédiaire d'un appel de fonction).
** Une fois la fin de fichier atteinte, les fonctions renvoient leurs
** résultats à la fonction de distribution qui affiche ensuite les résultats.
**
** © 2002, Éric Renault pour l'Institut National des Télécommunications.
** © 2003, Denis Conan
** © 2005-2020, Martin Quinson
**
*****************************************************************************/

#include <assert.h>
#include <ctype.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct syncbuffer {
  unsigned length;
  sem_t s_filled;
  sem_t s_empty;
  unsigned r_head;
  unsigned w_head;
  char *buffer;
} syncbuffer;

syncbuffer *syncbufferAlloc(unsigned value) {
  syncbuffer *sbuf = malloc(sizeof(syncbuffer));
  sbuf->length = value;
  sem_init(&sbuf->s_filled, 0, 0);
  sem_init(&sbuf->s_empty, 0, value);
  sbuf->r_head = 0;
  sbuf->w_head = 0;
  sbuf->buffer = malloc(value * sizeof(char));
  return sbuf;
}

void syncbufferFree(syncbuffer *sbuf) {
  sem_destroy(&sbuf->s_filled);
  sem_destroy(&sbuf->s_empty);
  free(sbuf->buffer);
  free(sbuf);
}

void syncbufferAppend(syncbuffer *sbuf, char car) {
  sem_wait(&sbuf->s_empty);
  sbuf->buffer[sbuf->r_head] = car;
  sbuf->r_head = (sbuf->r_head + 1) % sbuf->length;
  sem_post(&sbuf->s_filled);
}

char syncbufferGet(syncbuffer *sbuf) {
  sem_wait(&sbuf->s_filled);
  char car = sbuf->buffer[sbuf->w_head];
  sbuf->w_head = (sbuf->w_head + 1) % sbuf->length;
  sem_post(&sbuf->s_empty);
  return car;
}

/*
** Somme des chiffres.
*/
void *chiffre(void *ptr) {
  syncbuffer *chiffre_buffer = (syncbuffer *)ptr;

  static int somme = 0;
  char car;

  while ((car = syncbufferGet(chiffre_buffer)) != EOF) {
    somme += (car - '0');
  };

  return (void *)&somme;
}

/*
** Fréquence des lettres.
*/
void *lettre(void *ptr) {
  syncbuffer *lettre_buffer = (syncbuffer *)ptr;

  static int frequence['z' - 'a' + 1];
  char car;

  while ((car = syncbufferGet(lettre_buffer)) != EOF) {
    frequence[tolower(car) - 'a']++;
  };

  return (void *)frequence;
}

/*
** Distribution des caractères et affichage des résultats.
*/
void *distributeur(void *ignored) {
  /* Création des Threads Concurrents */

  syncbuffer *chiffre_buffer = syncbufferAlloc(512);
  pthread_t pth_chiffre;
  pthread_create(&pth_chiffre, NULL, chiffre, (void *)chiffre_buffer);

  syncbuffer *lettre_buffer = syncbufferAlloc(512);
  pthread_t pth_lettre;
  pthread_create(&pth_lettre, NULL, lettre, (void *)lettre_buffer);

  /* Distribution des caractères. */
  int car;
  do {
    car = getchar();
    if (isalpha(car) || (car == EOF)) {
      syncbufferAppend(lettre_buffer, car);
    }
    if (isdigit(car) || (car == EOF)) {
      syncbufferAppend(chiffre_buffer, car);
    }
  } while (car != EOF);

  /* Attente des Threads Concurrents */
  int *somme_ptr;
  pthread_join(pth_chiffre, (void *)&somme_ptr);
  syncbufferFree(chiffre_buffer);
  int *frequence_ptr;
  pthread_join(pth_lettre, (void *)&frequence_ptr);
  syncbufferFree(lettre_buffer);

  /* Lecture et affichage de la somme. */
  printf("Somme : %d\n", *somme_ptr);

  /* Lecture et affichage des frequences. */
  for (int i = 'a'; i <= 'z'; i++) {
    printf(" ; %c : %d", i, frequence_ptr[i - 'a']);
  }
  printf("\n");

  return NULL;
}

/*
** Lancement de la fonction de distribution.
*/
int main(int argc, char *argv[]) {
  /* Lancement du travail. */
  pthread_t pth_distrib;
  pthread_create(&pth_distrib, NULL, distributeur, NULL);

  /* Attente du travail. */
  pthread_join(pth_distrib, NULL);

  /* Tout s'est bien terminé. */
  exit(0);
}


////////////////////////////////// clang-tidy output
```