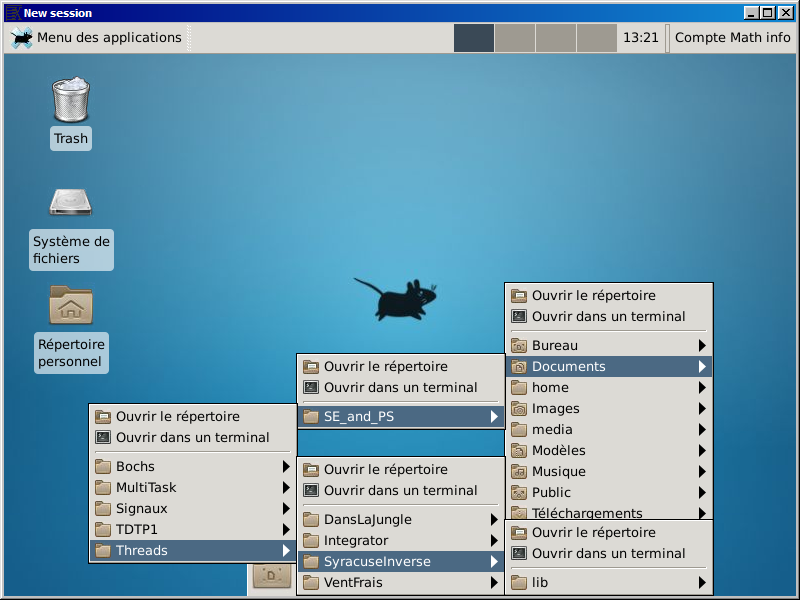
# 1516 L2 SEPS TDTP 5 – Threads Unix

## I] Connexion



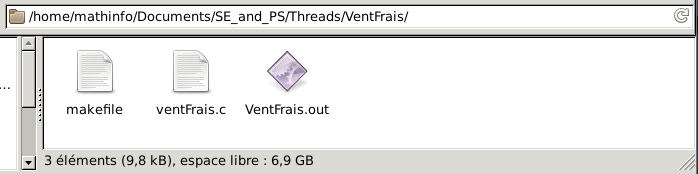
Nous pouvons voir sur l’image précédente, la connexion établie de la session.

## II] Le fond de l’air est frais

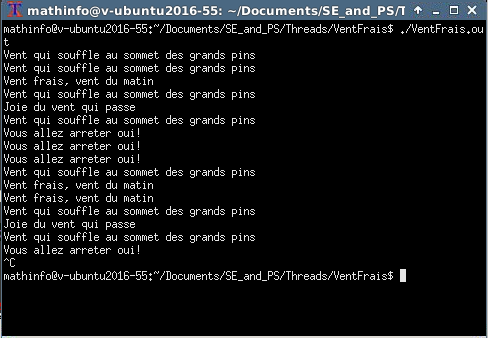
Dans le fichier "ventFrais.c", écrivez un programme en C qui crée un nombre len de threads à partir d’une fonction print, et ensuite effectue une boucle infinie en affichant un message toutes les 5 secondes. La fonction print tire en boucle un nombre au hasard entre 0 et 3, et affiche une phrase du canon « Vent frais, vent du matin » toutes les secondes.

### 1] Gestion des fichiers

Dans le dossier VentFrais/ :



### 2] Contenu du code et démonstration



Cet exercice se divise en deux partie, le programme principal et les threads.

Le programme principal :

* Initialise les threads Id.

// initialisation des threads Id

pthread\_t ThreadId[6];

* Boucle pour créer les six threads voulus.

// boucle de création des threads

for (i = 0; i < 6; ++i)

pthread\_create (&ThreadId[i], NULL, &print, NULL);

* Boucle indéfiniment en affichant.

// boucle infinie d'affichage

while (1){

sleep(1);

printf("Vous allez arreter oui!\n");

}

Les threads :

* Initialisation de liste des paroles.

// Initialisation des paroles.

char parole[ ][50] = {"Vent frais, vent du matin"

,"Vent qui souffle au sommet des grands pins"

,"Joie du vent qui passe"

,"Allons dans le grand ..."};

* Boucle indéfiniment en affichant aléatoirement un couplet.

// Boucle d'affichage des paroles choisie aléatoirement.

while (1){

printf("%s\n", parole[rand()%3]);

sleep(5);

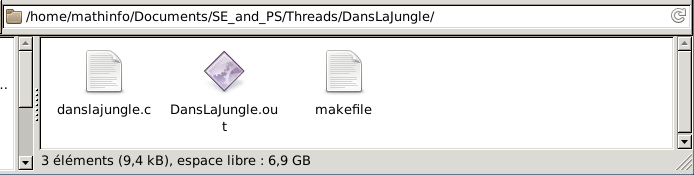
}

## III] Dans la jungle

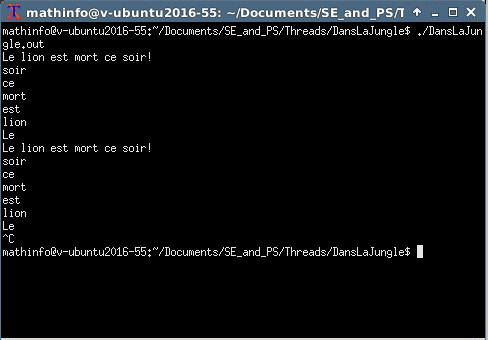
Dans le fichier "dansLaJungle.c", écrivez un programme en C qui crée 6 threads et leur passe à chacun une chaine de caractère issue de la liste ci-après : «Le», «lion», «est», «mort», «ce», «soir». Chaque thread est créé à partir d’une fonction print qui affiche la chaine de caractère reçue dans une boucle infinie.

### 1] Gestion des fichiers

Dans le dossier DansLaJungle/ :



### 2] Contenu du code et démonstration



Nous pouvons constater que les threads fonctionnent dans l’ordre inverse de leur création.

En effet, le code prévoit que les threads affichent la phrase « Le lion est mort ce soir » en tournant les uns après les autres. Chaque thread ayant reçu au préalable un mot de la phrase.

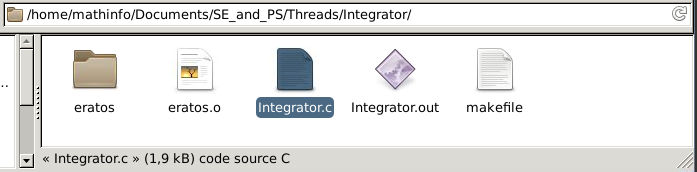
Or, comme dit plus haut, il affiche « soir ce mort est lion le ». Il affiche donc à l’envers.

IV] Integrator

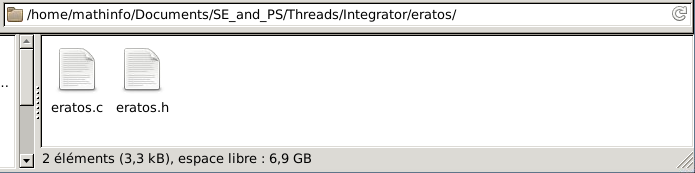
Dans le fichier "integrator.c", écrivez un programme en C qui crée un tableau dynamique de taille n des n premiers entiers naturels dans l’ordre croissant (par exemple n = 5000000). Le programme crée ensuite nb thread (par exemple nb=5)en leur passant comme argument une structure contenant le pointeur sur le tableau d’entiers, et deux nombres entiers start et end. Chaque thread est basé sur une fonction qui calcule la somme des nombres dans le tableau entre les start et end, et renvoie la valeur calculée au programme principal. Faites ensuite la somme des réponses des différents threads, et comparez la avec la somme obtenue en la calculant directement sur tout le tableau dans le ‘main’.

### 1] Gestion des fichiers

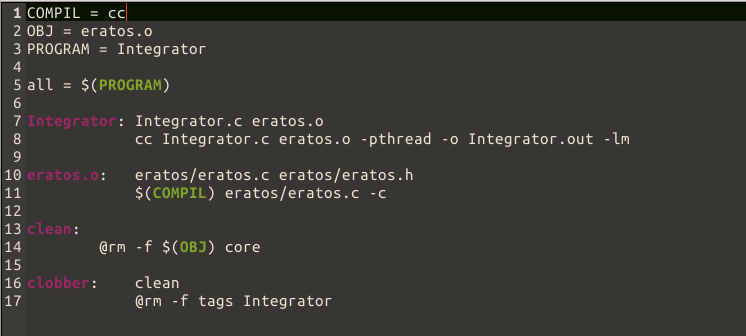
Dans le dossier Integrator/ :



Dans le dossier Integrator /eratos/ :

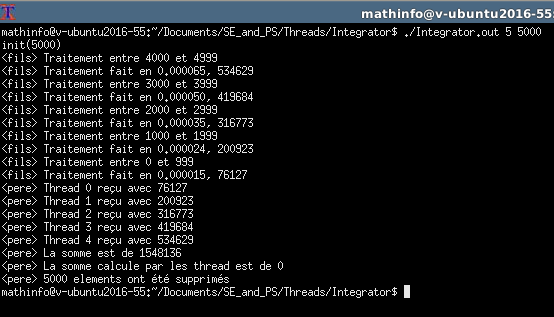


### 2] Contenu du makefile

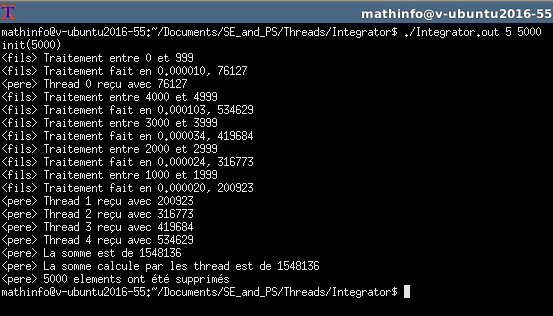


### 3] Contenu du code et démonstration

Situation sans le sleep()



Situation avec le sleep()



Cet exercice se divise en deux partie, le programme principal et les threads.

Le programme principal :

* Initialise la liste jusqu’à la borne donnée.

Eratos test = e\_initialiser(n);

e\_cribler(&test,n);

e\_ranger(&test);

* Initialise le tableau d'Id des threads.

pthread\_t ThreadId[nb];

* Initialise les paramètres des différents threads..

struct params paramsThread[nb];

paramsThread[0].eratos = &test;

paramsThread[0].start = 0;

paramsThread[0].end = prog-1;

for(i=1; i < nb ; ++i){

paramsThread[i].eratos = &test;

paramsThread[i].start = paramsThread[i-1].start + prog;

paramsThread[i].end = paramsThread[i].start+prog-1;

}

* Crée les threads.

for (i = 0; i < nb; ++i){

pthread\_create (&ThreadId[i], NULL, &integrator, (void\*)&paramsThread[i]);

usleep(400);

}

* Met en pause jusqu'à la fin de ses threads et calcule la somme des valeurs retournées par les threads.

unsigned long \*buffer = NULL,somme = 0;

for (i = 0; i < nb; ++i){

pthread\_join (ThreadId[i], (void \*\*) &buffer);

printf("<pere> Thread %d reçu avec %ld\n", i,\*buffer);

somme += \*buffer;

free(buffer);

}

* Affiche la somme calculée par le processus principal puis par les threads.

printf("<pere> La somme calcule par les thread est de

%ld\n",somme);

clock\_t t = clock();

somme = (e\_sommer(&test));

t = clock() - t;

printf("<pere> La somme est de %ld et faite en %f sec\n"

,somme,((float)t)/CLOCKS\_PER\_SEC);

* Affiche le nombre d'éléments supprimés.

printf("<pere> %d elements ont été supprimés\n",e\_supprimer (&test));

Les threads :

* Affiche un message au début du traitement.

printf ("<fils> Traitement entre %ld et %ld\n", p->start, p->end);

* Calcul de la somme entre les deux bornes.

t = clock();

(\*somme) = e\_sommer\_partie (p->eratos, p->start, p->end);

t = clock() - t;

* Affichage à la fin du traitement et du temps de calcul

printf ("<fils> Traitement fait en %f, %ld\n", ((float)t)/CLOCKS\_

PER\_SEC, (\*somme));

* Retour de la somme calculée.

return (void \*)somme;

Dans les images précédentes, on peut voir que le scheduller donne le temps à un des threads créés

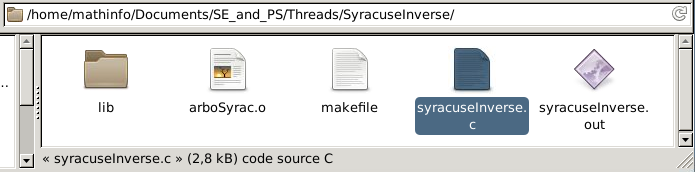
Voire le processus principal (ça dépend qui utilise la fonction sleep()) quand on place une pause avec sleep() . Par exemple dans l’exemple ci-dessus, vous pouvez voir qu’avec la pause, le premier thread récupère la main et se termine avant de lancer les autres threads.

## V] Syracuse Inverse

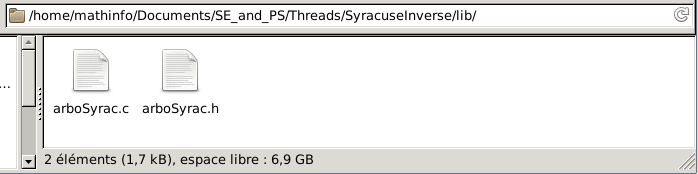
Dans le fichier "syracuseInverse.c", écrivez un programme en C qui calcule l’arborescence de la Conjecture Tchèque à partir de sa racine (1). Le programme principal crée un thread de calcul et lui passe le premier élément d’une liste chainée contenant l’élément racine. Ce Thread crée l’élément suivant de la liste chainée en multipliant le nombre valuereçu par deux, et si ce nombre moins un est un multiple de trois, il crée un nouveau thread en lui passant le nouvelélément avec (value-1)/3 pour calculer la nouvelle branche. Chaque thread qui crée lui-même des threads doit attendre que chacun de ses fils se termine avant de finir son exécution. L’exploration de l’arborescence de la conjecture se fait jusqu’à un degré level spécifié par l’utilisateur, et qui doit être stocké dans chaque élément de la liste chainée. Ces éléments sont donc composés d’une valeur entière, d’un niveau d’exploration level, et de deux élément suivant : un pair et parfois un impair (quand il existe). Commencez par écrire l’arborescence sur un papier en partant de 1.

### 1] Gestion des fichiers

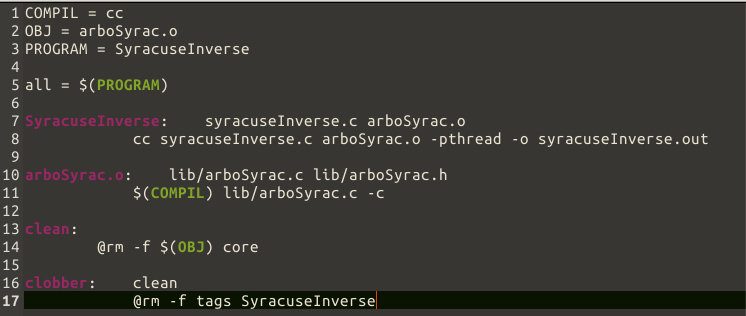
Dans le dossier SyracuseInverse/ :



Dans le dossier SyracuseInverse/lib/ :

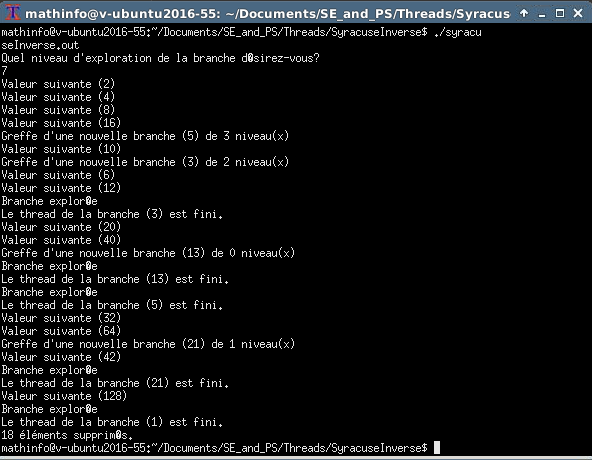


### 2] Contenu du makefile



Clean et clobber servent à nettoyer le dossier et SyracuseInverse nécessite de précompiler la librarie arboSyracuse.

### 3] Contenu du code et démonstration



Ce programme se compose de deux partie le programme principal et les threads.

Le programme principal :

* Commence par initialiser la variable Syracuse qui va contenir la racine de l’arborescence de l’inverse de syracuse ainsi que le niveau atteint.

typedef struct {

struct S\_Cellule \*racine;

int niveau;

}Syracuse;

* Demande le niveau d'exploration qui sera strictement supérieur à 0.

// Demande du niveau d'exploration à l'utilisateur

printf("Quel niveau d'exploration de la branche d%csirez-vous?\n",130);

do{

scanf("%d",&niveau);

}while(niveau < 0);

* Initialise la cellule racine et crée la cellule racine.

// Initialisation de la cellule racine.

syrac.racine = (S\_Cellule \*)malloc(sizeof(S\_Cellule));

s\_init\_cellule(syrac.racine,valeur,niveau);

* Crée le thread de calcul de base.

// Initialisation de l'id du thread

pthread\_t ThreadId;

// Création du thread de calcul de base.

pthread\_create (&ThreadId, NULL, &Thread, (void \*)syrac.racine);

* Se met en attente jusqu’à la fin du thread créé avec retour de la racine.

// variable de récupération vi le join

S\_Cellule \*temp;

// Mise en attente du créateur jusqu'à la fin du thread créé.

pthread\_join (ThreadId, (void \*\*) &temp);

// Affichage de la fin du thread créé..

printf("Le thread de la branche (%d) est fini.\n",temp->valeur);

* Supprime l'arborescence à partir de « l’instance » de Syracuse

// Suppression de l'arborescence

printf("%d éléments supprim%cs.\n",s\_supprimer(syrac),130);

Les threads :

* Initialise cur et base à partir de la racine.

// Initialise cur et base à partir de la racine.

struct S\_Cellule \*cur = parameter,\*racine = parameter;

* Modifie la valeur decrementation du niveau.

// Modification de la valeur decrementation du niveau.

valeur \*= 2; niveau --;

* Alloue la nouvelle cellule

// Alloction de la nouvelle cellule et ajout dans le chainage.

cur->pair = (S\_Cellule \*)malloc(sizeof(S\_Cellule));

cur = cur->pair;

* Initialise la cellule.

// Initialisation de la cellule.

s\_init\_cellule(cur,valeur,niveau);

* Effectue la condition suivante

// Condition: valeur - 1 multiple de 3 sauf valeur Ã  4.

if(valeur != 4 && 0 == (valeur-1)%3){

La condition étant que si la valeur moins un est divisible par 3, on fait la tâche sauf si la valeur est 3.

* Crée le nouveau thread qui va s’occuper de la nouvelle branche.

// Ajout de racine de la nouvelle branche au chainage.

cur->impair = (S\_Cellule \*)malloc(sizeof(S\_Cellule));

s\_init\_cellule(cur->impair,(valeur-1)/3,niveau);

// Affichage de la crÃ©ation de la nouvelle branche.

printf("Greffe d'une nouvelle branche (%d) de %d niveau(x)\n",(valeur-1)/3,niveau);

// CrÃ©ation du thread de calcul qui gÃ©nÃ¨re la branche.

pthread\_create (&ThreadId, NULL, &Thread, (void \*)cur->impair);

// variable crÃ©Ã©e temporairement pour rÃ©cupÃ©rer le retour du thread

S\_Cellule \*temp;

// Mise en attente du crÃ©ateur jusqu'Ã  la fin du thread crÃ©Ã©.

pthread\_join (ThreadId, (void \*\*) &temp);

// Affichage de la fin du thread crÃ©Ã©..

printf("Le thread de la branche (%d) est fini.\n",temp->valeur);