

Compte Rendu

Ing. sys. multi-coeurs et multiprocesseurs

Groupe 2.2

Réalisé par: Ahmed Gharbi

Année universitaire:

2020/2021

Table des matières

I. Tp1: Découverte.....	3
A. Objectif.....	3
B. Réalisation.....	3
1. Compilation et exécution :.....	5
II. TP2: Partage de données entre tâches.....	8
A. Objectif :.....	8
B. Conception du graphe de tâches :.....	8
C. Réalisation.....	10
III. TP3: Multiprocessing.....	15
A. Objectif.....	15
B. Conception du graphe de tâches :.....	16
C. Réalisation.....	17

I. Tp1: Découverte

A. Objectif

Comprendre la structure des fichiers

comment les compiler

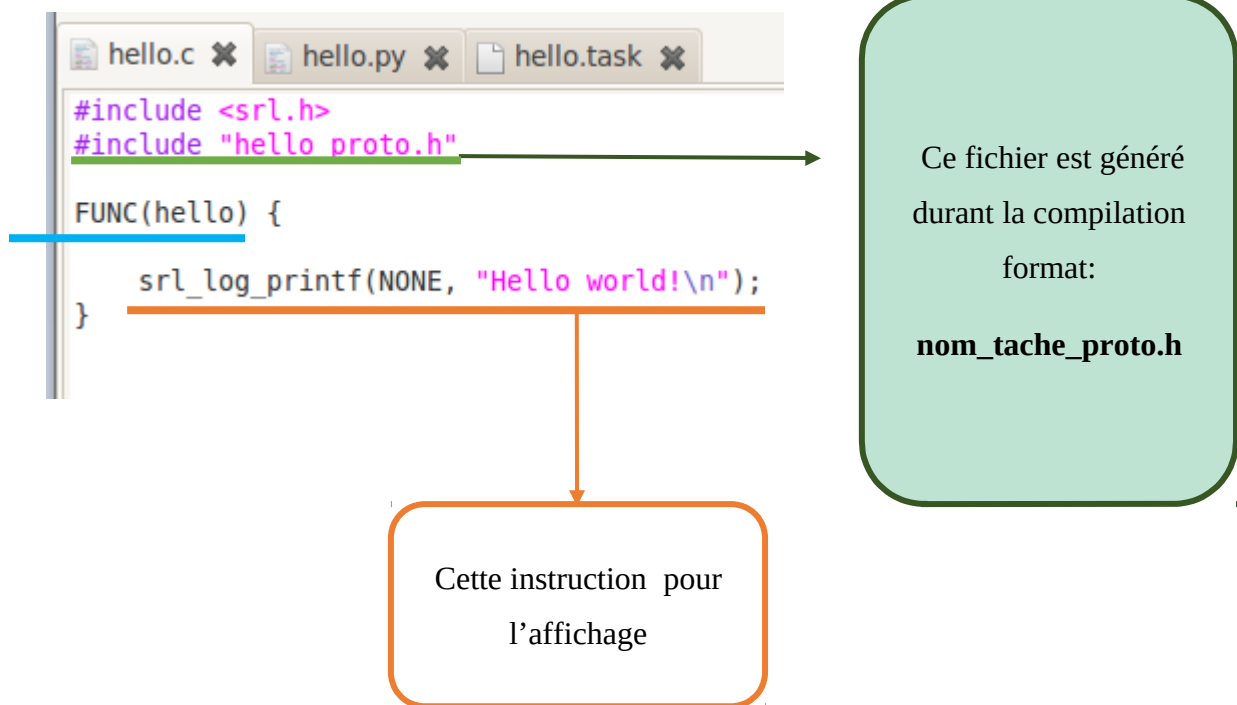
exécuter les tâches.

B. Réalisation

On va se focaliser généralement sur les fichiers .c , .task et .py

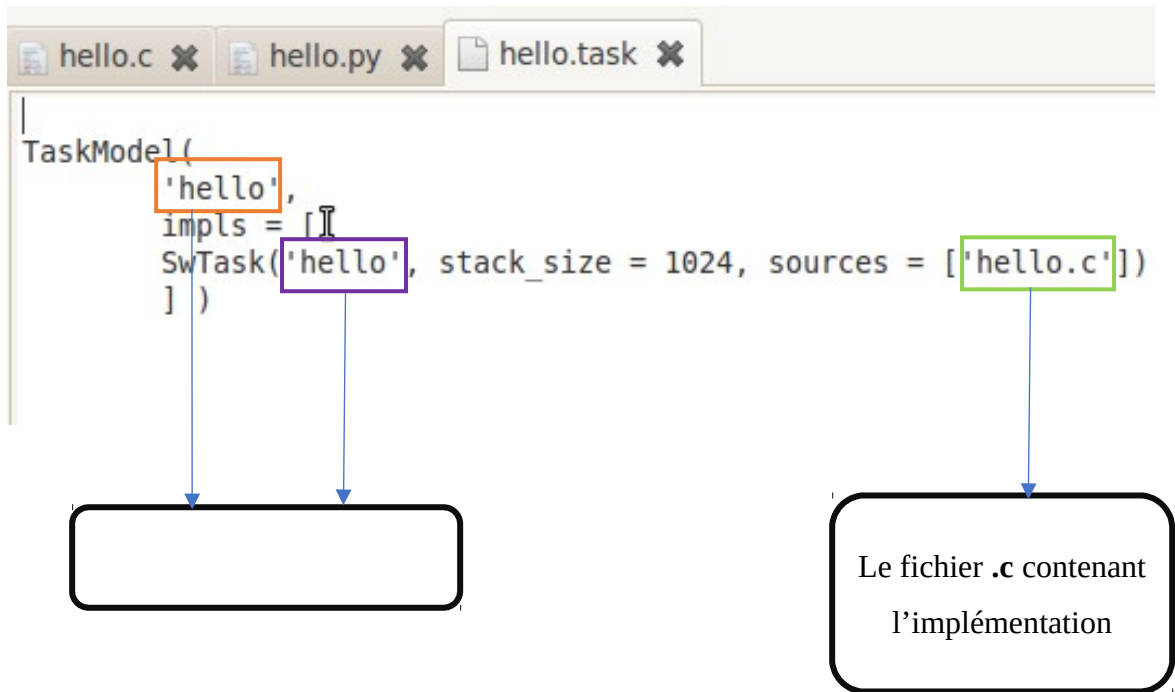
- Le fichier source **.C**

Dans le fichier C on va décrire l'implémentation de la tâche à exécuter.



- Le fichier **.task**

Définition de la structure de la tâche.



- Le fichier **.py**

1. Compilation et exécution :

Pour compiler notre code, nous devons compiler le code contenu dans notre fichier python(**.py**).

Pour ce faire, nous devons d'abord attribuer les droits d'exécution au fichier python en question avec la commande :

sudo chmod +x fichier.py

Puis, nous lançons la compilation en exécutant le fichier **./fichier.py**

```
user2@user2-desktop:~/Téléchargements/hello_world$ chmod +x hello.py
user2@user2-desktop:~/Téléchargements/hello_world$ ./hello.py
```

On génère alors dans le répertoire le fichier **exe.posix** et le noyau **posix** :



On exécute le fichier **exe.posix** :

```
user2@user2-desktop:~/Téléchargements/hello_world$ ./exe.posix
```

Le résultat obtenu est :

```
Hello world!  
H̃ello world!  
H̄ello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!
```

II. TP2: Partage de données entre tâches

A. Objectif :

L'objectif de ce tp est de :

- maîtriser les multitâches.
- la communication entre tâches.

Nous implémenterons donc trois tâches **hello**, **world** et **poly** qui se partageront des chaînes de caractères afin de former l'affichage suivant :

Hello

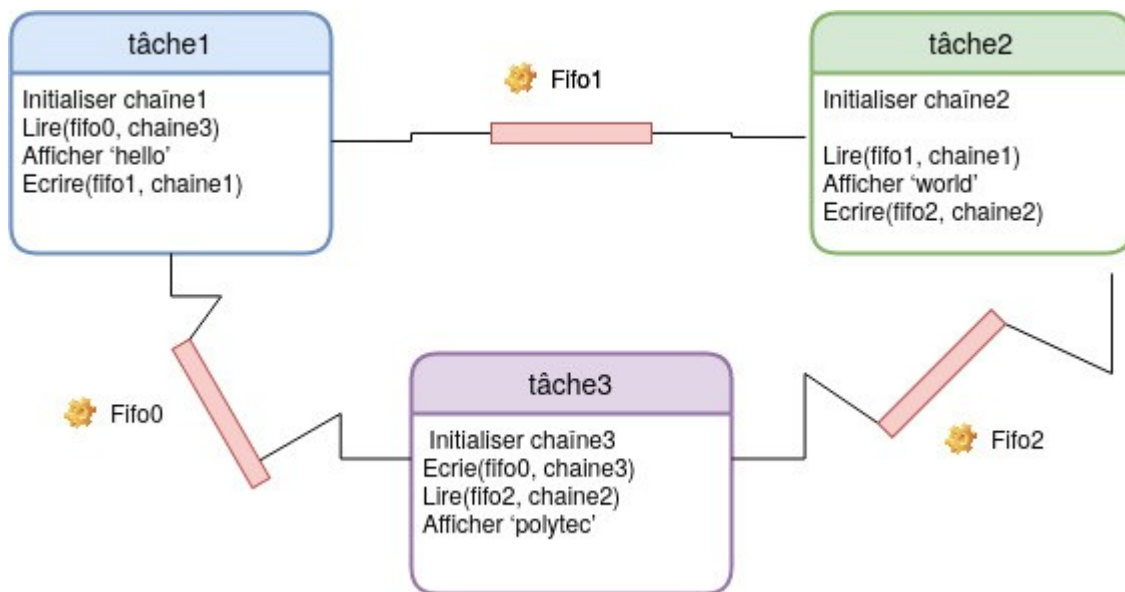
World

Polytec

C. Conception du graphe de tâches :

Il est souvent nécessaire d'effectuer une représentation graphique de notre programme. Cela nous permet d'avoir une idée plus claire de la manière dont nous obtiendrons le résultat escompté.

Nous allons donc concevoir notre graphe de tâches :



Le tableau suivant permet aussi de vérifier la logique derrière notre implémentation :

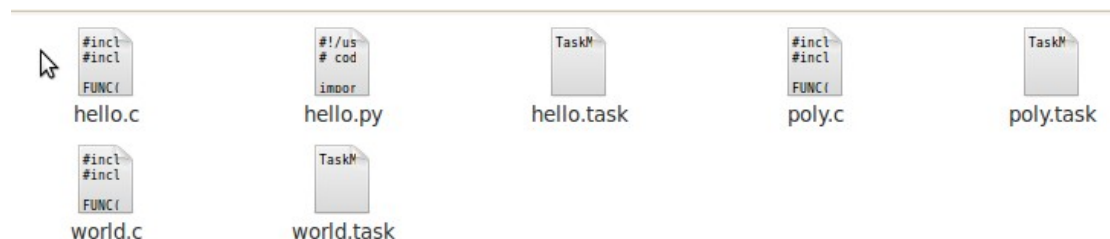
Instant	Tâche 1	Tâche 2	Tâche 3
T1	Initialiser chaîne1	Initialiser chaîne 2	Initialiser chaîne 3
T2	Bloquée	Bloquée	Ecrire (fifo0, chaîne3)
T3	Lire (fifo0, chaîne3)	Bloquée	Bloquée
T4	Afficher 'hello'	Bloquée	Bloquée
T5	Ecrire (fifo1, chaîne1)	Bloquée	Bloquée
T6	Bloquée	Lire (fifo1, chaîne1)	Bloquée
T7	Bloquée	Afficher 'world'	Bloquée
T8	Bloquée	Ecrire (fifo2, chaîne2)	Bloquée
T9	Bloquée	Bloquée	Lire (fifo2, chaîne2)
T10	Bloquée	Bloquée	Afficher 'polytec'

Tout au long de nos TP, nous avons eu à utiliser ce que l'on appelle **fifo (First In First Out)** plus particulièrement la **fifo bloquante**. La **fifo bloquante** est un algorithme de gestion de mémoire. Dans ce TP2, il nous permettra de gérer **séquentiellement** l'accès de chaque tâche à la mémoire.

Ainsi, chaque tâche ne pourra accéder à la mémoire qu'une fois que la tâche qui l'utilise à ce même instant ait arrêté son exploitation de la mémoire.

D. Réalisation

Comme présenté plus haut dans notre conception, notre programme comprendra trois tâche :



Les instructions présentées ci-dessus respectent toutes le graphe de tâche :

- Tâche 1 :

```
hello.c x hello.py x hello.task x poly.c x p
#include <srl.h>
#include "hello_proto.h"

FUNC(hello)
{
char *chaine1="world";
char *chaine2;
srl_mwmr_t fifo0 = GET_ARG(fifo0);
srl_mwmr_t fifo1 = GET_ARG(fifo1);

srl_mwmr_read(fifo0, &chaine2, 4);
srl_log_printf(NONE, "tache 1: %s\n", chaine2);

srl_mwmr_write(fifo1, &chaine1, 4);
}
```

Définition des
fifo à exploiter
dans cette tâche

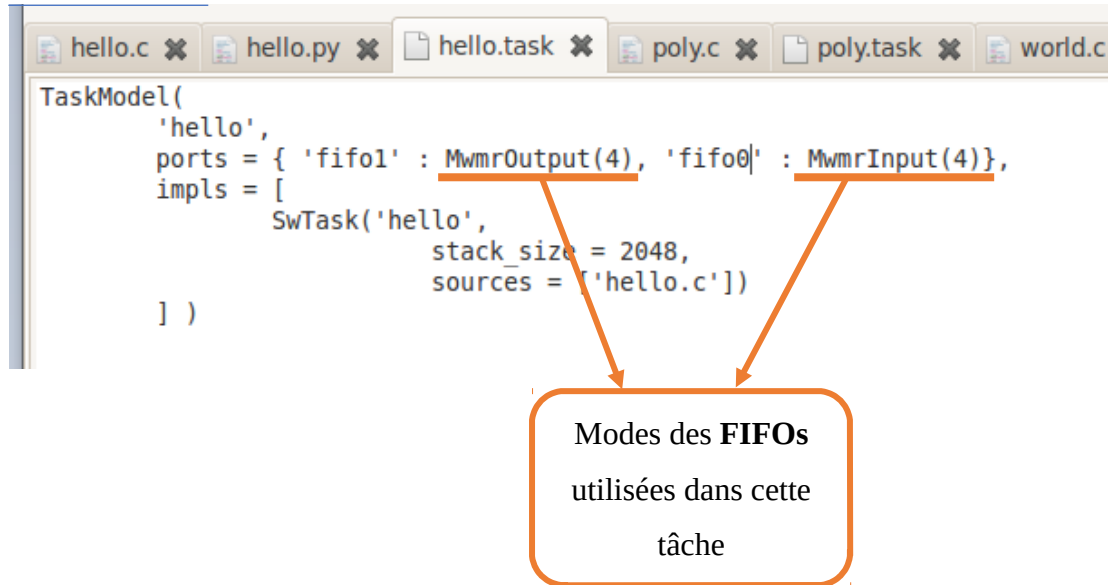
Instruction de lecture de la
fifo. Elle suit ce format:

```
srl_mwmr_read(
    Nom de la fifo,
    variable pour récupérer le
    contenu de la fifo,
    Taille du contenu
)
```

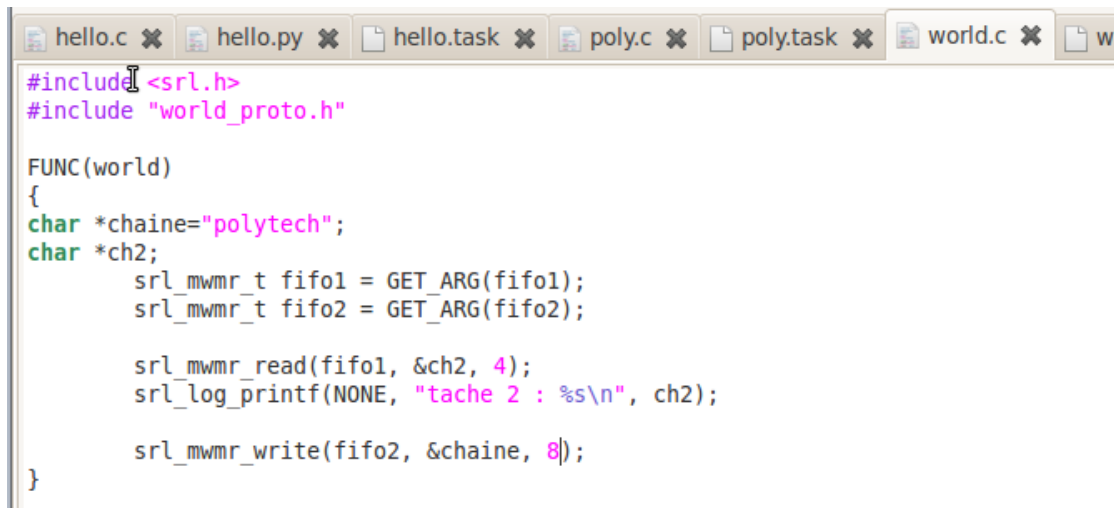
Instruction de lecture de la
fifo. Elle suit ce format:

```
srl_mwmr_write(
    Nom de la fifo,
    Valeur à écrire dans la
    fifo,
    Taille du contenu
)
```

Les **FIFOs** et leur mode(**output**, **input**) doivent également être spécifiées dans chaque fichier **.task**



- Tâche 2 :



```
hello.c x hello.py x hello.task x poly.c x poly.task x world.c x world.task x
TaskModel(
    'world',
    ports = { 'fifo1' : MwmrInput(4), 'fifo2' : MwmrOutput(8)},
    impls = [
        SwTask('world',
                stack_size = 2048,
                sources = ['world.c'])
    ] )
```

- Tâche 3

```
hello.c x hello.py x hello.task x poly.c x poly.task x
#include <srl.h>
#include "poly_proto.h"

FUNC(poly)
{
    char *ch="hello";
    char *ch2;
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    srl_mwmr_write(fifo0, &ch, 4);
    srl_mwmr_read(fifo2, &ch2, 8);

    srl_log_printf(NONE, "tache3 : %s\n", ch2);
}
```

```
hello.c x hello.py x hello.task x poly.c x poly.task x world.c x
TaskModel(
    'poly',
    ports = { 'fifo2' : MwmrInput(8), 'fifo0' : MwmrOutput(4)},
    impls = [
        SwTask('poly',
                stack_size = 2048,
                sources = ['poly.c'])
    ] )
```

- Hello.py

Puisque nous aurons à utiliser trois tâches, nous devons ajouter chacune d'elle à notre graphe tout en mentionnant les **FIFOs** qu'elles exploiteront :

```
hello.c x hello.py x hello.task x poly.c x poly.task x world.c x
import dsx

# Partie 1 : définition du TCG (Graphe des Tâches et des Communications)

fifo0 = dsx.Mwmr('fifo0', 4, 1)
fifo1 = dsx.Mwmr('fifo1', 4, 1)
fifo2 = dsx.Mwmr('fifo2', 8, 1)

tcg = dsx.Tcg(
    dsx.Task('hello', 'hello',
             {'fifo0':fifo0,'fifo1':fifo1} ),

    dsx.Task('world', 'world',
             {'fifo1':fifo1,'fifo2':fifo2} ),

    dsx.Task('poly', 'poly',
             {'fifo0':fifo0,'fifo2':fifo2} )
)

# Partie 2 : génération du code exécutable sur station de travail POSIX

tcg.generate( dsx.Posix() )
```

Après compilation et exécution, nous obtenons le résultat escompté :

```
tache 2 : world
tache3 : polytech
tache 1: hello
tache 2 : world
tache3 : polytech
tache 1: hello
tache 2 : world
tache3 : polytech
tache 1: hello
tache 2 : world
tache3 : polytech
```

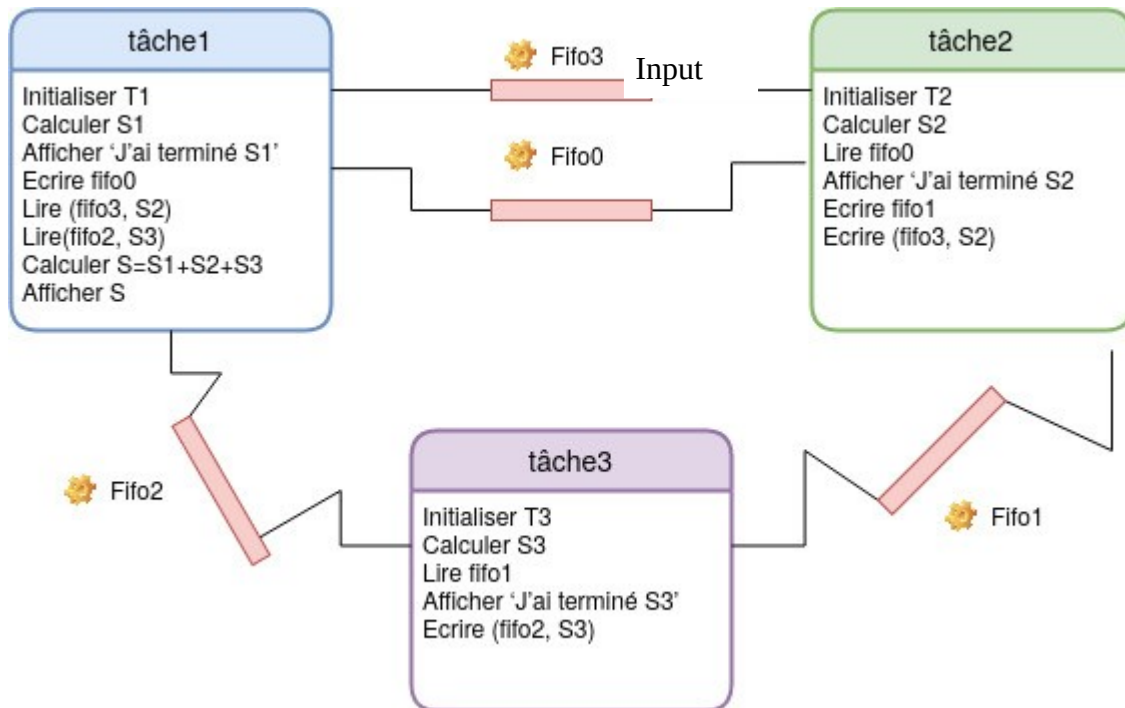
III. TP3: Multiprocessing

A. Objectif

Jusqu'à maintenant, nous n'avons réalisé que des tâches qui s'exécutent sur un seul et même **microprocesseur**. Cette fois-ci, par contre, nous avons utilisé simultanément plusieurs **microprocesseurs** en occurrence 3 puisque nous aurons 3 tâches.

Ainsi l'objectif visé dans ce TP est d'effectuer du **multiprocessing**.

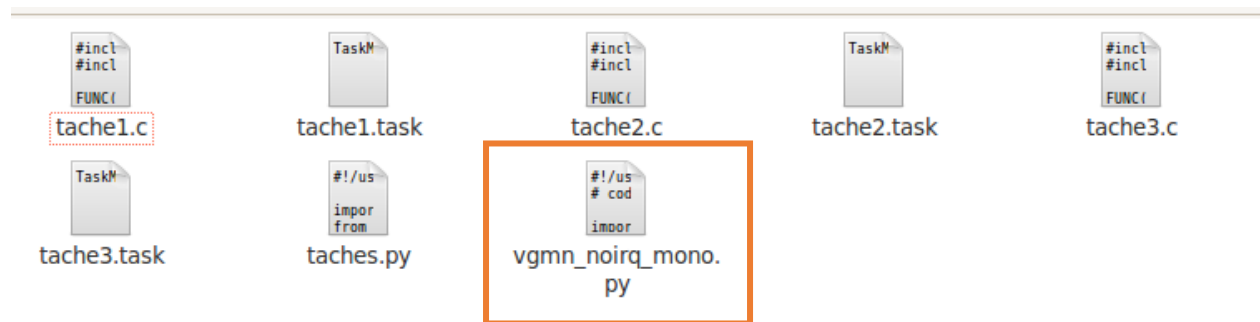
E. Conception du graphe de tâches :



Instant	Tâche 1	Tâche 2	Tâche 3
T1	Initialiser T1	Initialiser T2	Initialiser T3
T2	Calculer S1	Calculer S2	Calculer S3
T3	Afficher (J'ai terminé S1)	Bloquée	Bloquée
T4	Ecrire fifo0	Bloquée	Bloquée
T5	Bloquée	Lire fifo0	Bloquée
T6	Bloquée	Afficher (J'ai terminé S2)	Bloquée
T7	Bloquée	Ecrire fifo1	Bloquée
T8	Bloquée	Bloquée	Lire fifo1
T9	Bloquée	Bloquée	Afficher (J'ai terminé S3)
T10	Bloquée	Bloquée	Ecrire (fifo2, S3)
T11	Bloquée	Ecrire (fifo3, S2)	Bloquée
T12	Lire (fifo2, S3)	Bloquée	Bloquée
T13	Lire (fifo3, S2)	Bloquée	Bloquée
T14	Calculer S	Bloquée	Bloquée
T15	Calculer S	Bloquée	Bloquée

F. Réalisation

Au cours de cet Tp, nous aurons besoin d'un nouveau fichier :



Avant de discuter de son contenu, observons d'abord les changements dans les fichiers .c :

- Tache 1 :

```
FUNC(tache1)
{
    int a[1];
    char T1[10]={1, 2, 3, 4, 50, 6, 7, 8, 9, 10};
    int i=0, s1=0, s2=0, s3=0, s=0;
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);
    srl_mwmr_t fifo3 = GET_ARG(fifo3);

    for(i=0; i<10; i++){
        s1+=T1[i];
    }

    srl_log_printf	TRACE, "tache1 : J'ai termine S1: %d\n", s1);
    srl_mwmr_write(fifo0, &a[0], 4);

    srl_mwmr_read(fifo3, &s2, 4);
    srl_mwmr_read(fifo2, &s3, 4);

    s=s1+s2+s3;
    srl_log_printf	TRACE, "tache1 : s=%d\n", s);
}
```

Dans les tps précédents, il était inscrit NULL.

```
tache1.c x tache1.task x tache2.c x tache2.task x tache3.c x tache3.task
TaskModel(
    'tache1',
    ports = { 'fifo0' : MwmrOutput(4), 'fifo2' : MwmrInput(4), 'fifo3' : MwmrInput(4) },
    impls = [
        SwTask('tache1',
            stack_size = 2048,
            sources = ['tache1.c'])
    ] )
```

- Tâche 2

```
tache1.c x tache1.task x tache2.c x tache2.task x tache3.c x tache3.task
#include <srl.h>
#include "tache2_proto.h"

FUNC(tache2)
{
    int a[1];
    char T2[10]={1, 2, 3, 4, 5, 6, 7, 80, 9, 10};
    int i=0, s2=0;
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo3 = GET_ARG(fifo3);

    for(i=0; i<10; i++){
        s2+=T2[i];
    }

    srl_mwmr_read(fifo0, &a[0], 4);
    srl_log_printf	TRACE, "tache2 : J'ai termine S2: %d\n", s2);

    srl_mwmr_write(fifo1, &a[0], 4);
    srl_mwmr_write(fifo3, &s2, 4);
}
```



```
tache1.c x tache1.task x tache2.c x tache2.task x tache3.c x tache3.task
TaskModel(
    'tache2',
    ports = { 'fifo0' : MwmrInput(4), 'fifo1' : MwmrOutput(4), 'fifo3' : MwmrOutput(4)},
    impls = [
        SwTask('tache2',
                stack_size = 2048,
                sources = ['tache2.c'])
    ] )
```

- Tâche 3

```
tache1.c x tache1.task x tache2.c x tache2.task x tache3.c x
#include <srl.h>
#include "tache3_proto.h"

FUNC(tache3)
{
    int a[1];
    char T3[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i=0, s3=0;
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    for(i=0; i<10; i++){
        s3+=T3[i];
    }

    srl_mwmr_read(fifo1, &a[0], 4);
    srl_log_printf	TRACE, "tache3 : J'ai termine S3: %d\n", s3);

    srl_mwmr_write(fifo2, &s3, 4);
}
```

```
tache1.c  tache1.task  tache2.c  tache2.task  tache3.c  tache3.task
TaskModel(
    'tache3',
    ports = { 'fifo1' : MwmrInput(4), 'fifo2' : MwmrOutput(4)},
    impls = [
        SwTask('tache3',
                stack_size = 2048,
                sources = ['tache3.c'])
    ] )
```

Observons le contenu du fichier python **taches.py** :

```
fifo0 = dsx.Mwmr('fifo0', 4, 1)
fifo1 = dsx.Mwmr('fifo1', 4, 1)
fifo2 = dsx.Mwmr('fifo2', 4, 1)
fifo3 = dsx.Mwmr('fifo3', 4, 1)

tcg = dsx.Tcg(
    dsx.Task('tache1', 'tache1',
             {'fifo0':fifo0,'fifo2':fifo2,'fifo3':fifo3} ),

    dsx.Task('tache2', 'tache2',
             {'fifo0':fifo0,'fifo1':fifo1,'fifo3':fifo3} ),

    dsx.Task('tache3', 'tache3',
             {'fifo1':fifo1,'fifo2':fifo2} ),
)
```

```

archi = VgmnNoirqMono(ntty = 3)

#####
# Section C : Mapping
#
#####

m = Mapper(archi, tcg)

# mapping the MMR channel

m.map( "fifo0", buffer = "cram0", status = "cram0", desc = "cram0")
m.map( "fifo1", buffer = "cram0", status = "cram0", desc = "cram0")
m.map( "fifo2", buffer = "cram0", status = "cram0", desc = "cram0")
m.map( "fifo3", buffer = "cram0", status = "cram0", desc = "cram0")

```

```
# mapping tasks
```

```
m.map("tache1",  
      desc = 'cram0',  
      run = 'cpu0',  
      stack = 'cram0',  
      tty = 'tty',  
      tty_no = 0)
```

```
m.map("tache2",  
      desc = 'cram0',  
      run = 'cpu1',  
      stack = 'cram0',  
      tty = 'tty',  
      tty_no = 1)
```

```
m.map("tache3",  
      desc = 'cram0',  
      run = 'cpu2',  
      stack = 'cram0',  
      tty = 'tty',  
      tty_no = 2)
```

```

# mapping the software objects associated to a processor

m.map( 'cpu0',
      private = "cram0",
      shared  = "cram0")

m.map( 'cpu1',
      private = "cram0",
      shared  = "cram0")

m.map( 'cpu2',
      private = "cram0",
      shared  = "cram0")

# mapping the software objects used by the embedded OS

m.map(tcg,
      private = "cram0",
      shared  = "uram0",
      code    = "cram0",
      tty     = "tty",
      tty_no  = 0)

```

```

#####
# Section D : Code generation
#####

m.generate( dsx.MutekS() )
tcg.generate( dsx.Posix() )

```

Pour ce fichier, de nouvelles instructions ont été ajoutées comme le mapping des fifos et des tâches et leur allocation à un **CPU**.

Observons maintenant le contenu du fichier **vgmn_noirq_mono.py** :

```

def VgmnNoirqMono(ntty = 1):
    pf = soclib.Architecture(cell_size = 4,
                             plen_size = 8,
                             addr_size = 32,
                             rerror_size = 1,
                             clen_size = 1,
                             rflag_size = 1,
                             srcid_size = 8,
                             pktid_size = 1,
                             trdid_size = 1,
                             wrplen_size = 1
                             )

    pf.create('common:mapping_table',
              'mapping_table',
              addr_bits = [8],
              srcid_bits = [8],
              cacheability_mask = 0xc00000)
    pf.create('common:loader', 'loader')

    vgmn = pf.create('caba:vci_vgmn', 'vgmn0',
                     min_latency=10,

    for i in range(3):###nous avons besoin de 2 processeur donc 2 sinon si n
    processeurs, on mettra n
        cpu = pf.create('caba:vci_xcache_wrapper', 'cpu%d' % i,
                        iss_t = "common:mips32el",
                        ident = i,
                        icache_ways = 2,
                        icache_sets = 128,
                        icache_words = 32,
                        dcache_ways = 2,
                        dcache_sets = 128,
                        dcache_words = 32)

        vgmn.to_initiator.new() // cpu.vci

    for i in range(1):
        ram = pf.create('caba:vci_ram', 'ram%d'%i)
        base = 0x10000000*i
        ram.addSegment('cram%d'%i, base, 0x200000, True)
        ram.addSegment('uram%d'%i, base + 0x400000, 0x200000, False)
        ram.vci // vgmn.to_target.new()

```

```

        ram.addSegment('cram%d'%i, base, 0x200000, True)
        ram.addSegment('uram%d'%i, base + 0x400000, 0x200000, False)
        ram.vci // vgm.n.to_target.new()
    ram.addSegment('boot', 0xbfc00000, 0x1000, True) # Mips boot address, 0x1000 octets,
cacheable
    ram.addSegment('excep', 0x80000000, 0x1000, True) # Mips exception address, 0x1000
octets, cacheable

    tty = pf.create('caba:vci_multi_tty', 'tty', names = map(lambda x:'tty%d'%x, range
(ntty)))
    tty.addSegment('tty', 0x95400000, 0x20*ntty, False)
    tty.vci // vgm.n.to_target.new()

    return pf

# This is a python quirk to generate the platform
# if this file is directly called, but only export
# methods if imported from somewhere else

if __name__ == '__main__':
    VgmNoirqMono().generate(soclib.PfDriver())


```

Ce fichier contient les instructions nécessaires pour la création des terminaux et l'allocation de **CPU**.

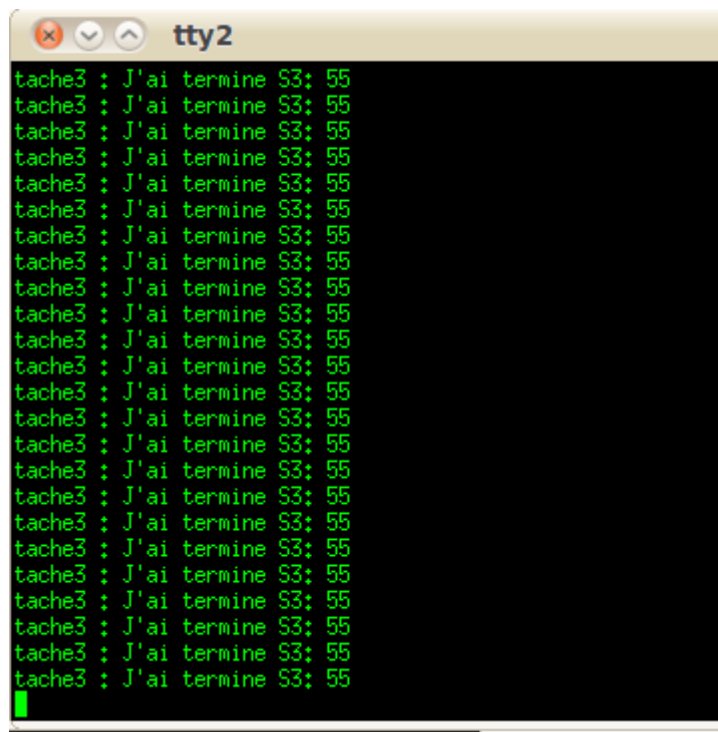
Pour compiler ce programme, nous utiliserons la méthode habituelle c'est-à-dire exécuter le fichier python **taches.py**.

Par contre, l'exécution ne se fera plus en exécutant le fichier **exe.posix** mais plutôt le fichier **exe.muteks_hard**.

Nous obtenons le résultat suivant après exécution :

[illegible]

The screenshot shows a terminal window with a title bar containing three icons (close, minimize, maximize) and the text 'tty1'. The terminal content consists of 20 identical lines: 'tache2 : J'ai termine S2: 127'. The text is green on a black background. A green cursor is visible at the end of the last line.



```
tty2
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
tache3 : J'ai termine S3: 55
```

Contrairement aux tps précédents, l’affichage ne se fait pas un seul et même terminal. Chacune de ces fenêtres affiche le résultat de l’exécution d’une tâche.