

Matière : Ingénierie des systèmes multicores et
multiprocesseurs

Année universitaire 2020/2021

COMPTE RENDU

Réalisé par : Marwa KHELIFI

Encadré par : Nidhameddine BELHADJ
4ème Génie Informatique G2.2

Introduction

En informatique, le parallélisme consiste à mettre en œuvre des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible.

Jusqu'au début des années 2000, le parallélisme était réservé aux serveurs et aux superordinateurs. Il est maintenant utilisé dans la grande majorité des architectures, des systèmes embarqués aux superordinateurs. Les monoprocesseurs sont remplacés par des processeurs multi-cœurs. Cet article introduit la notion de parallélisme et ses différents types. Il présente les grandes classes d'architectures parallèles avec leurs ressources et leurs organisations mémoire, en distinguant les architectures homogènes et hétérogènes.

Les architectures parallèles sont devenues le paradigme dominant pour tous les ordinateurs depuis les années 2000. En effet, la vitesse de traitement qui est liée à l'augmentation de la fréquence des processeurs connaît des limites. La création de processeurs multi-cœurs, traitant plusieurs instructions en même temps au sein du même composant, résout ce dilemme pour les machines de bureau depuis le milieu des années 2000.

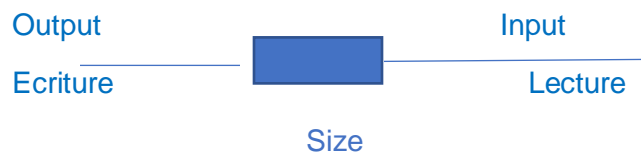
Les principes des techniques de programmation sont introduits avec les extensions parallèles des langages de programmation couramment utilisés et les modèles de programmation qui visent à rapprocher la programmation parallèle de la programmation séquentielle, tout en prenant en compte les spécificités des architectures.

Enfin, l'intérêt des architectures parallèles réside dans les performances qu'elles permettent d'atteindre. Il est nécessaire de modéliser d'une part le parallélisme existant dans une application et d'autre part les architectures parallèles. Nous examinons donc les métriques utilisées pour évaluer ou prévoir les performances et les grandes lois qui les gouvernent.

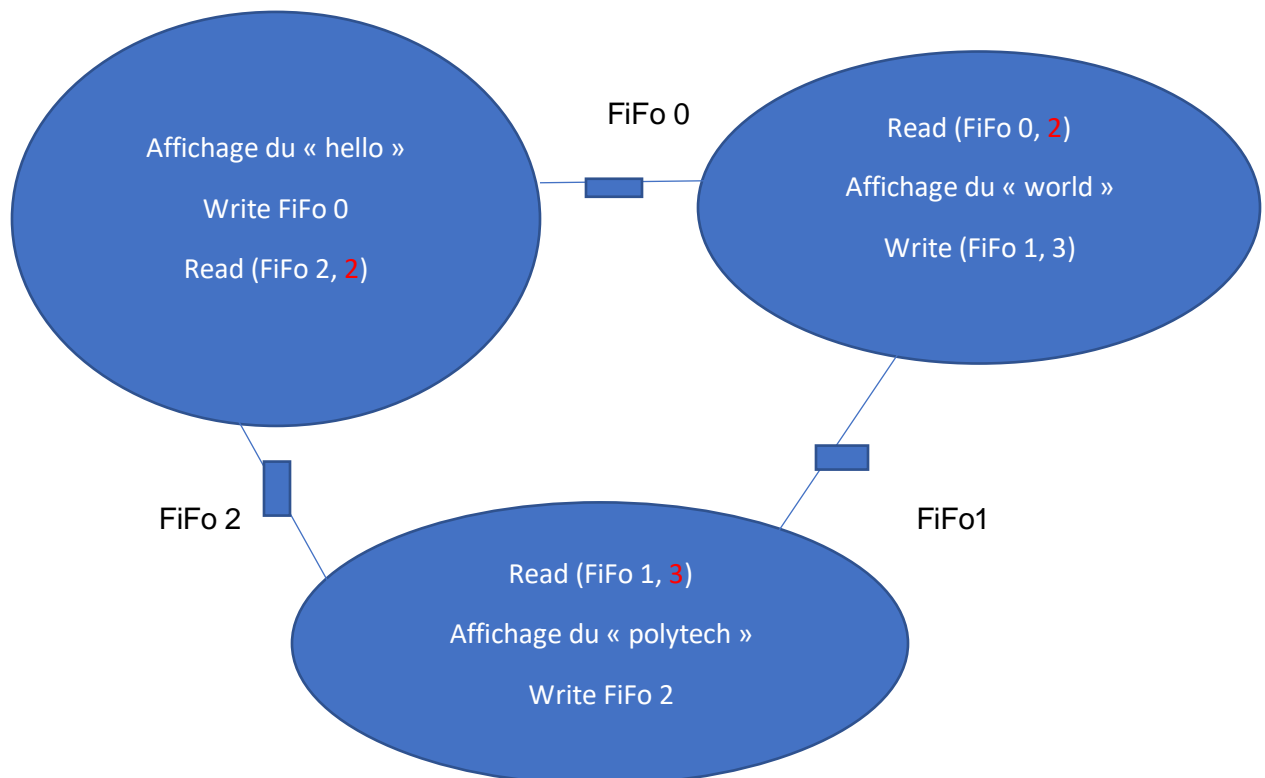
TP1 : MULTI-TACHES

FiFo bloquante avec synchronisation \longleftrightarrow Communication entre les tâches.

Principe de FiFo bloquante :



If count==0 \longleftrightarrow FiFo est vide
 Write_en = '1' et read_en = '0'
 Count++ **Après chaque opération d'écriture**
 If count==size \longleftrightarrow FiFo est pleine
 Write_en = '0' et read_en = '1'
 Count- - **Après chaque opération de lecture**



| T | Taches | Hello | World | Polytech |
|----|--------|----------------------|----------------------|-------------------------|
| T0 | | Affichage du "hello" | bloquée | bloquée |
| T1 | | Write fifo0 | bloquée | bloquée |
| T2 | | bloquée | Read fifo0 | bloquée |
| T3 | | bloquée | Affichage du "world" | bloquée |
| T4 | | bloquée | Write fifo1 | bloquée |
| T5 | | bloquée | bloquée | Read fifo1 |
| T6 | | bloquée | bloquée | Affichage du "polytech" |
| T7 | | bloquée | bloquée | Write fifo2 |
| T8 | | Read fifo2 | bloquée | bloquée |

Code du fichier "**hello.c**" :

```
#include <srl.h>
#include "hello_proto.h"

FUNC(hello)
{
    int a[1];

    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    srl_log_printf(NONE, "tâche1 : Hello...\n");
    srl_mwmr_write(fifo0, &a[0], 4);
    srl_mwmr_read(fifo2, &a[0], 4);
}
```

Code du fichier "**hello.task**" :

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

Code du fichier "**world.c**" :

```

hello.task x hello.py x polytech.c x polytech.task x srl_trace.txt x world.c x
#include <srl.h>
#include "world_proto.h"

FUNC(world)
{
    int a[1];
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);

    srl_mwmr_read(fifo0, &a[0], 4);
    srl_log_printf(NONE, "tâche2 : ... world\n");
    srl_mwmr_write(fifo1, &a[0], 4);
}

```

Code du fichier "**world.task**" :

```

hello.py x polytech.c x polytech.task x srl_trace.txt x world.c x world.task
TaskModel(
    'world',
    ports = { 'fifo0' : MwmrInput(4), 'fifo1' : MwmrOutput(4)},
    impls = [
        SwTask('world',
            stack size = 2048,
            sources = ['world.c'])
    ] )

```

Code du fichier "**polytech.c**" :

```

tache3.task x somme.py x hello.c x hello.task x hello.py x polytech.c x
#include <srl.h>
#include "polytech_proto.h"

FUNC(polytech)
{
    int a[1];
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    srl_mwmr_read(fifo1, &a[0], 4);
    srl_log_printf(NONE, "tâche3 : ... polytech\n");
    srl_mwmr_write(fifo2, &a[0], 4);
}

```

Code du fichier "**polytech.task**" :

```

somme.py x hello.c x hello.task x hello.py x polytech.c x polytech.task
TaskModel(
    'polytech',
    ports = { 'fifo1' : MwmrInput(4), 'fifo2' : MwmrOutput(4)},
    impls = [
        SwTask('polytech',
            stack size = 2048,
            sources = ['polytech.c'])
    ] )

```


Code du fichier python **"hello.py"** :

```
#!/usr/bin/env python
# coding: latin-1

import dsx

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

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

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

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

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

    .

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

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

Exécution

On fait l'exécution avec **"./exe.posix"**

La sortie du programme précédent est la suivante :

```
tâche1 : Hello...
tâche2 : ... world
tâche3 : ... polytech

tâche1 : Hello...
tâche2 : ... world
tâche3 : ... polytech

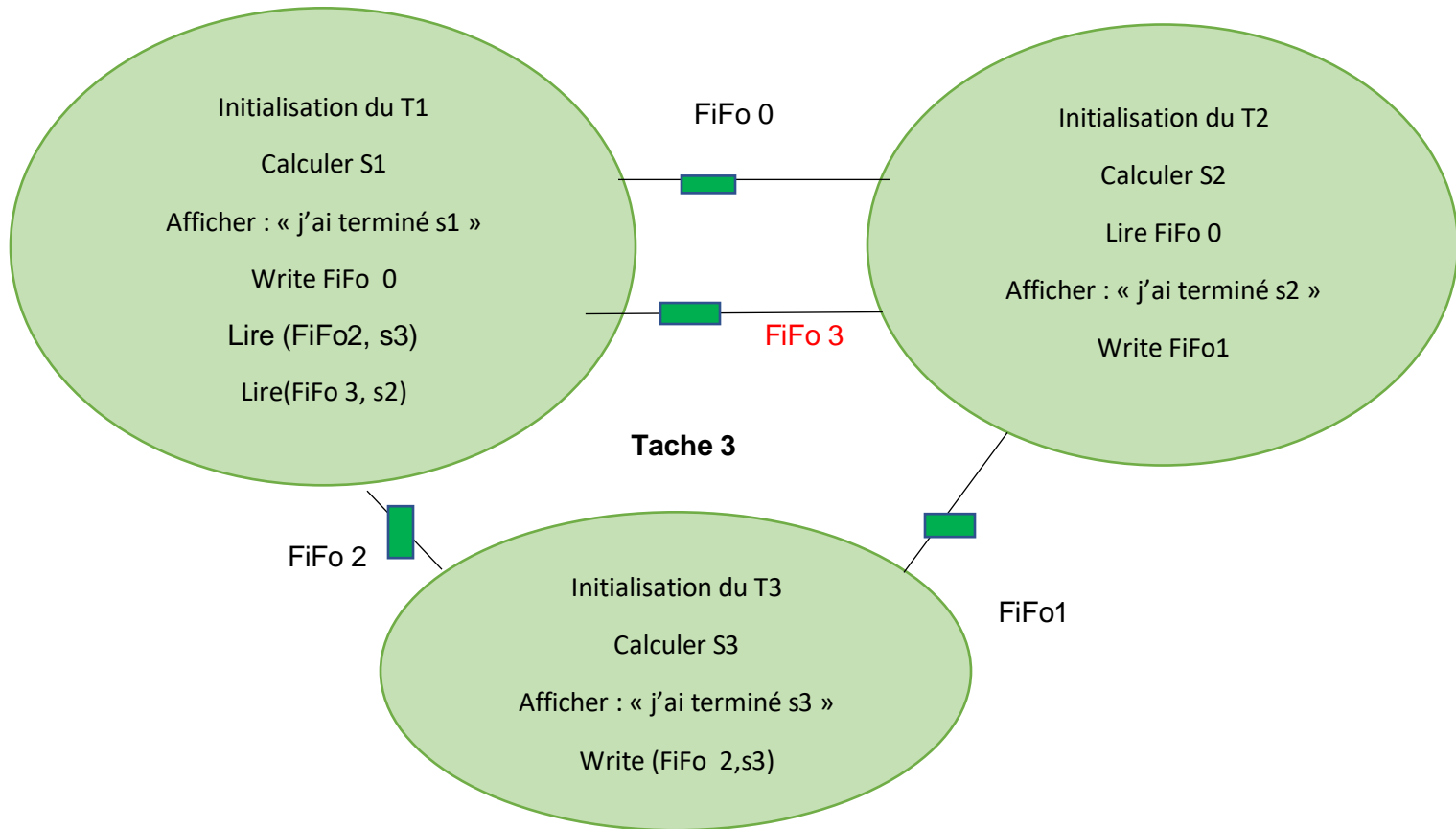
tâche1 : Hello...
tâche2 : ... world
tâche3 : ... polytech

tâche1 : Hello...
tâche2 : ... world
tâche3 : ... polytech^C
user2@user2-desktop:~/marwa/hello deux taches sync$
```

TP2: Calcul de Somme

Tache 1

Tache 2



| T | Taches | Tache 1 | Tache 2 | Tache 3 |
|----|--------|-------------------|--------------------|-------------------|
| T0 | | Initialisation T1 | Initialisation T2 | Initialisation T3 |
| T1 | | Calculer S1 | Calculer S2 | Calculer S3 |
| T2 | | Affichage “..S1” | bloquée | Bloquée |
| T3 | | Ecrire FiFo 0 | bloquée | Bloquée |
| T4 | | bloquée | Lire fifo 0 | Bloquée |
| T5 | | bloquée | Affichage “..S2” | bloquée |
| T6 | | bloquée | Ecrire FiFo 1 | bloquée |
| T7 | | bloquée | Ecrire(FiFo 3, s2) | bloquée |

| | | | |
|-----|------------------|---------|--------------------|
| T8 | bloquée | bloquée | Lire fifo 1 |
| T9 | bloquée | bloquée | Affichage “..S3” |
| T10 | bloquée | bloquée | Ecrire (FiFo 2,s3) |
| T11 | Lire (FiFo2, s3) | bloquée | bloquée |
| T12 | Lire (FiFo3, s2) | bloquée | bloquée |
| T13 | Calculer S | bloquée | bloquée |
| T14 | Affichage du S | bloquée | bloquée |

Code du fichier “**somme.py**” :

```
#!/usr/bin/env python
# coding: latin-1

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', 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, 'fifo2':fifo2} ),
    dsx.Task('tache3', 'tache3',
        {'fifo1':fifo1, 'fifo3':fifo3} ),
)
```

Code du fichier “**tache1.c**” :

```
FUNC(tache1)
{
    int a[10];
    int t1[10], s=0, s1[1], s2[1], s3[1], i;

    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);
    srl_mwmr_t fifo3 = GET_ARG(fifo3);

    t1[0]=8;
    s1[0]=t1[0];
    for(i=0; i<10; i++)
    {
        t1[i]=t1[i-1]+2;
        s1[i]=t1[i];
    }

    srl_log_printf(NONE, "s1 : j'ai termine s1...\n");
    srl_mwmr_write(fifo0, &a[0], 4);
    srl_mwmr_read(fifo2, &s2[0], 4);
    srl_mwmr_read(fifo3, &s3[0], 4);

    s=s1[0]+s2[0]+s3[0];
    srl_log_printf(NONE, "S=%d\n", s);
}
```


Code du fichier "**tache2.c**" :

```
#include <srl.h>
#include "tache2_proto.h"

FUNC(tache2)
{
    int a[1];
    int t2[10], s=0, s1[1], s2[1], s3[1], i;

    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);

    t2[0]=0;
    s2[0]=t2[0];
    for(i=0; i<10; i++)
        t2[i]=t2[i-1]+2;
    s2[0]+=t2[i];

    srl_mwmr_read(fifo0, &(s1[0]), 4);
    srl_log_printf(NONE, "s2 : j'ai termine s2...\n");
    srl_mwmr_write(fifo1, &(a[0]), 4);
    srl_mwmr_write(fifo2, &(s2[0]), 4);
}
```

Code du fichier "**tache2.task**" :

```
#include <srl.h>
#include "tache3_proto.h"

FUNC(tache3)
{
    int a[1];
    int t3[10], s=0, s1[1], s2[1], s3[1], i;

    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo3 = GET_ARG(fifo3);

    t3[0]=0;
    s3[0]=t3[0];
    for(i=0; i<10; i++)
        t3[i]=t3[i-1]+2;
    s3[0]+=t3[i];

    srl_mwmr_read(fifo1, &(s2[0]), 4);
    srl_log_printf(NONE, "s3 : j'ai termine s3...\n");
    srl_mwmr_write(fifo3, &(s3[0]), 4);
}
```

A screenshot of a terminal window. The title bar at the top reads "somme.py (~marwa/somme_tableaux) - gedit". Below the title bar is a menu bar with options: "dition Affichage Rechercher Outils Documents Aide". The main area of the terminal shows a command prompt "user2@user2-desktop: ~/marwa/somme_tableaux" followed by several lines of output. The output consists of repeating blocks of three lines: "s1 : j'ai terminé s1...", "s2 : j'ai terminé s2...", and "s3 : j'ai terminé s3 ...", each preceded by a variable assignment "s=403557564". At the bottom of the terminal, there is another command prompt "user2@user2-desktop:~/marwa/somme_tableaux\$". A mouse cursor is visible on the right side of the terminal window.

Tache 1

FiFo 1

Tache 2

FiFo 0

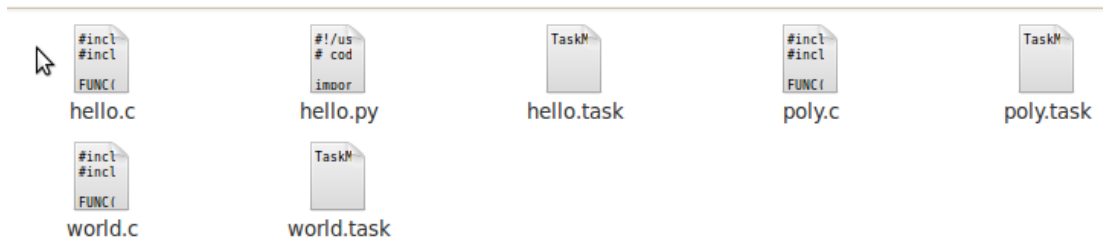
Tache 3

FiFo2

```
Déclaration de la chaîne3
Write(FiFo 0, chaîne3)
Read(FiFo2,chaîne2)
Affichage chaîne2
```

| T | Taches | Tache 1 | Tache 2 | Tache 3 |
|----|--------|---------------------------|---------------------------|---------------------------|
| T0 | | Déclaration de la chaîne1 | Déclaration de la chaîne2 | Déclaration de la chaîne3 |
| T1 | | bloquée | bloquée | Write(FiFo 0, chaîne3) |
| T2 | | Read (FiFo 0, chaîne 3) | bloquée | bloquée |
| T3 | | Affichage du chaîne3 | bloquée | bloquée |
| T4 | | Write(FiFo1,chaîne1) | bloquée | bloquée |
| T5 | | bloquée | Read(FiFo1,chaîne1) | bloquée |
| T6 | | bloquée | Affichage chaîne1 | bloquée |
| T7 | | bloquée | Write(FiFo3, chaîne2) | bloquée |
| T8 | | bloquée | bloquée | Read(FiFo2,chaîne2) |
| T9 | | bloquée | bloquée | Affichage du chaîne 2 |

Ce TP contient les fichiers suivants :



Code du fichier "**hello.c**" :

```

#include <srl.h>
#include "hello_proto.h"

FUNC(hello)
{
    char *chaîne1="world";
    char *chaîne2;
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);

    srl_mwmr_read(fifo0, &chaîne2, 4);
    srl_log_printf(NONE, "tache 1: %s\n", chaîne2);

    srl_mwmr_write(fifo1, &chaîne1, 4);
}

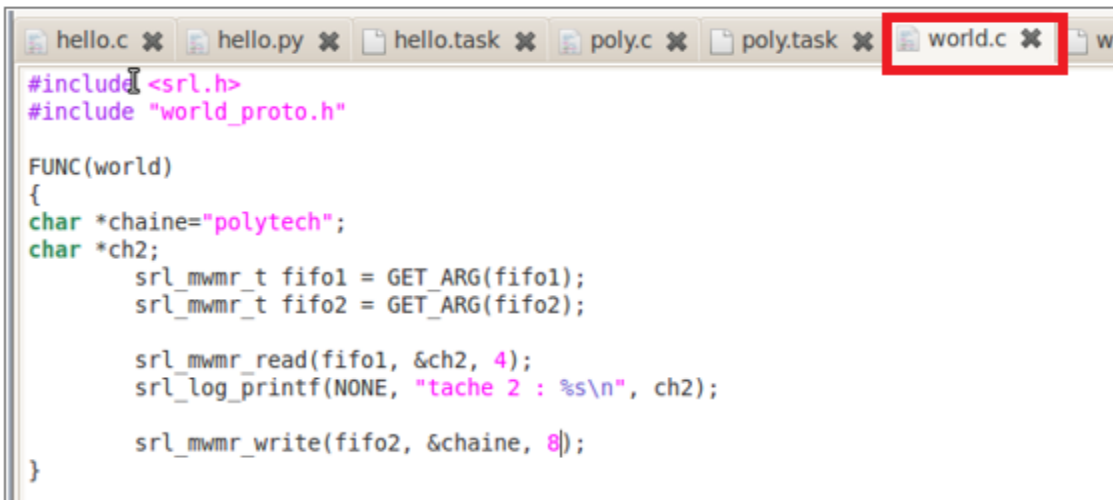
```

Code du fichier "**hello.task**" :



```
TaskModel(
    'hello',
    ports = { 'fifo1' : MwmrOutput(4), 'fifo0' : MwmrInput(4)},
    impls = [
        SwTask('hello',
            stack_size = 2048,
            sources = ['hello.c'])
    ] )
```

Code du fichier "**world.c**" :



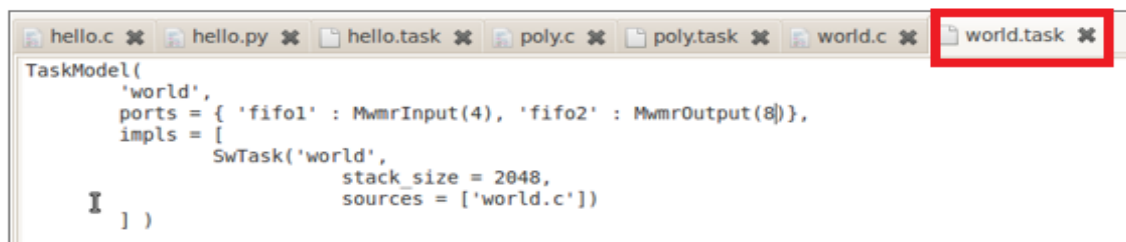
```
#include <srl.h>
#include "world_proto.h"

FUNC(world)
{
    char *chaine="polytech";
    char *ch2;
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    srl_mwmr_read(fifo1, &ch2, 4);
    srl_log_printf(NONE, "tache 2 : %s\n", ch2);

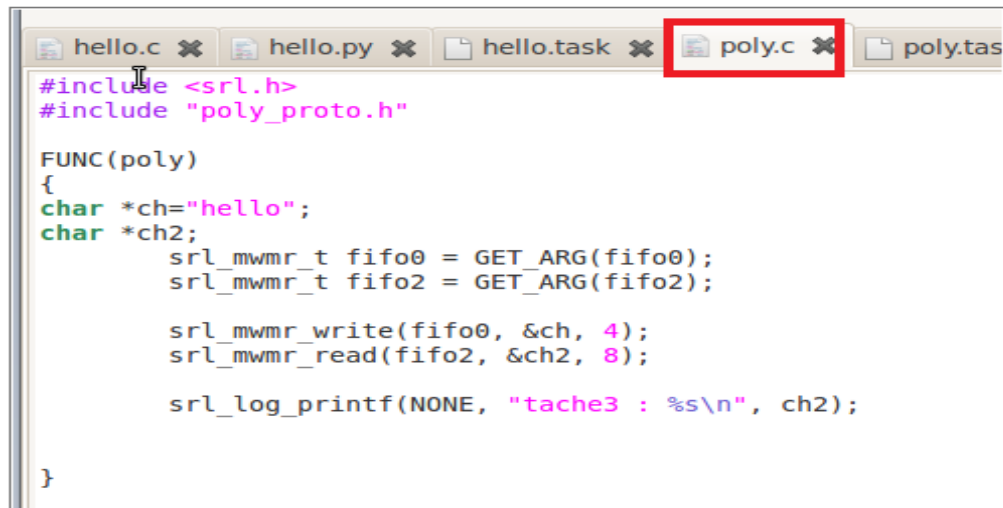
    srl_mwmr_write(fifo2, &chaine, 8);
}
```

Code du fichier "**world.task**" :



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

Code du fichier "**poly.c**" :



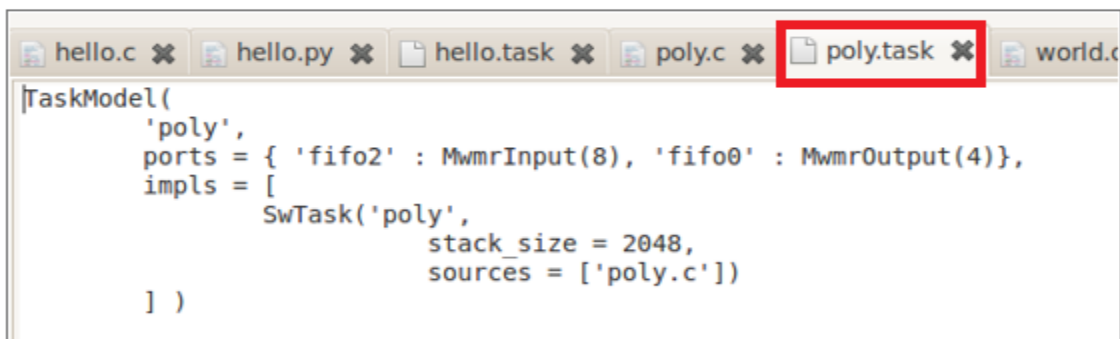
```
#include <srl.h>
#include "poly_proto.h"

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);

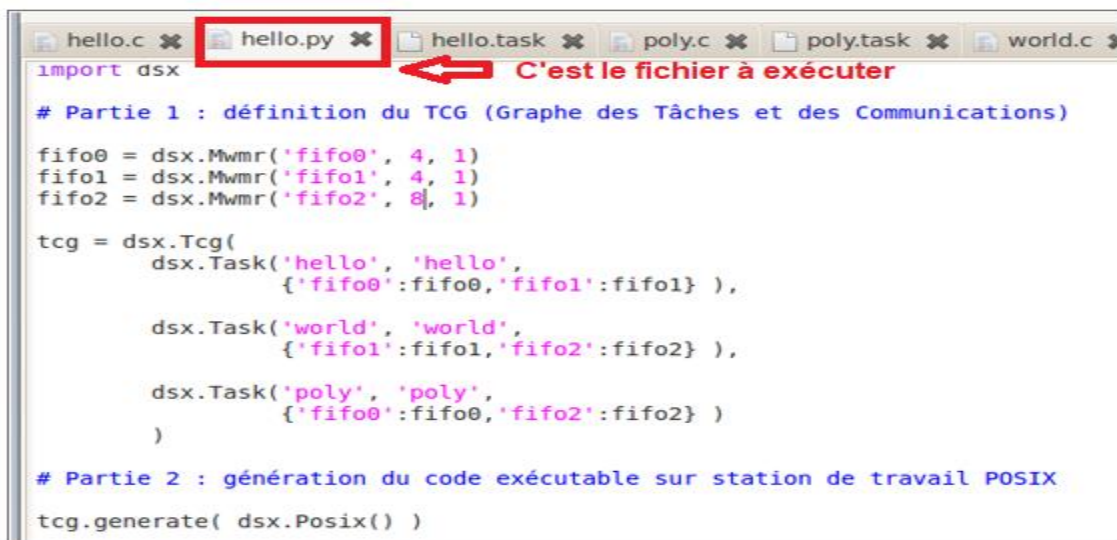
    srl_log_printf(NONE, "tache3 : %s\n", ch2);
}
```

Code du fichier “**poly.task**” :



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

Code du fichier “**hello.py**” :



```
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() )
```

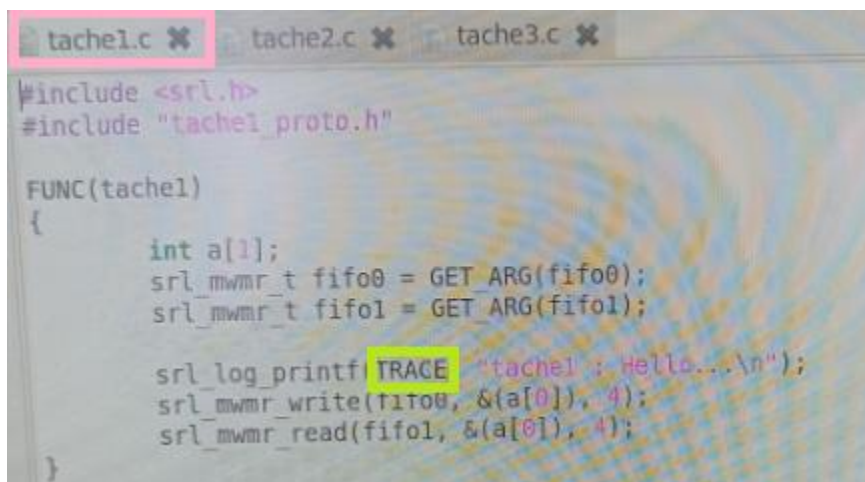

On fait l'exécution avec "`./exe.muteks_hard`"

Le résultat d'exécution est comme suit :

```
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
tache 1: hello
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^C
```

TP4 : Multiprocesseurs

Code du "**Tache1.c**" :



```
tache1.c ✕ tache2.c ✕ tache3.c ✕
#include <srl.h>
#include "tache1_proto.h"

FUNC(tache1)
{
    int a[1];
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);

    srl_log_printf(TRACE, "tache1 : Hello...\n");
    srl_mwmr_write(fifo0, &(a[0]), 4);
    srl_mwmr_read(fifo1, &(a[0]), 4);
}
```

Code du "**Tache1.task**" :

```

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

```

Code du "Tache2.c" :

```

tache1.c ✕  tache2.c ✕  tache3.c ✕
#include <srl.h>
#include "tache2_proto.h"

FUNC(tache2)
{
    int a[1];
    srl_mwmr_t fifo0 = GET_ARG(fifo0);
    srl_mwmr_t fifo1 = GET_ARG(fifo1);

    srl_mwmr_read(fifo0, &a[0], 4);
    srl_log_printf(TRACE, "tache2 : ... world\n\n");
    srl_mwmr_write(fifo1, &a[0], 4);
}

```

Code du "Tache2.task" :

```

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

```

Code du "Tache3.c" :

```

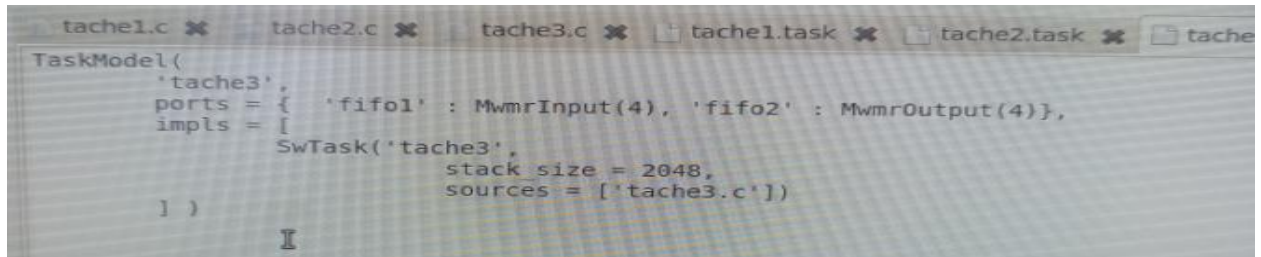
tache1.c ✕  tache2.c ✕  tache3.c ✕
#include <srl.h>
#include "tache3_proto.h"

FUNC(tache3)
{
    int a[1];
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    srl_mwmr_read(fifo1, &a[0], 4);
    srl_log_printf(TRACE, "tache3 : ... polytech\n\n");
    srl_mwmr_write(fifo2, &a[0], 4);
}

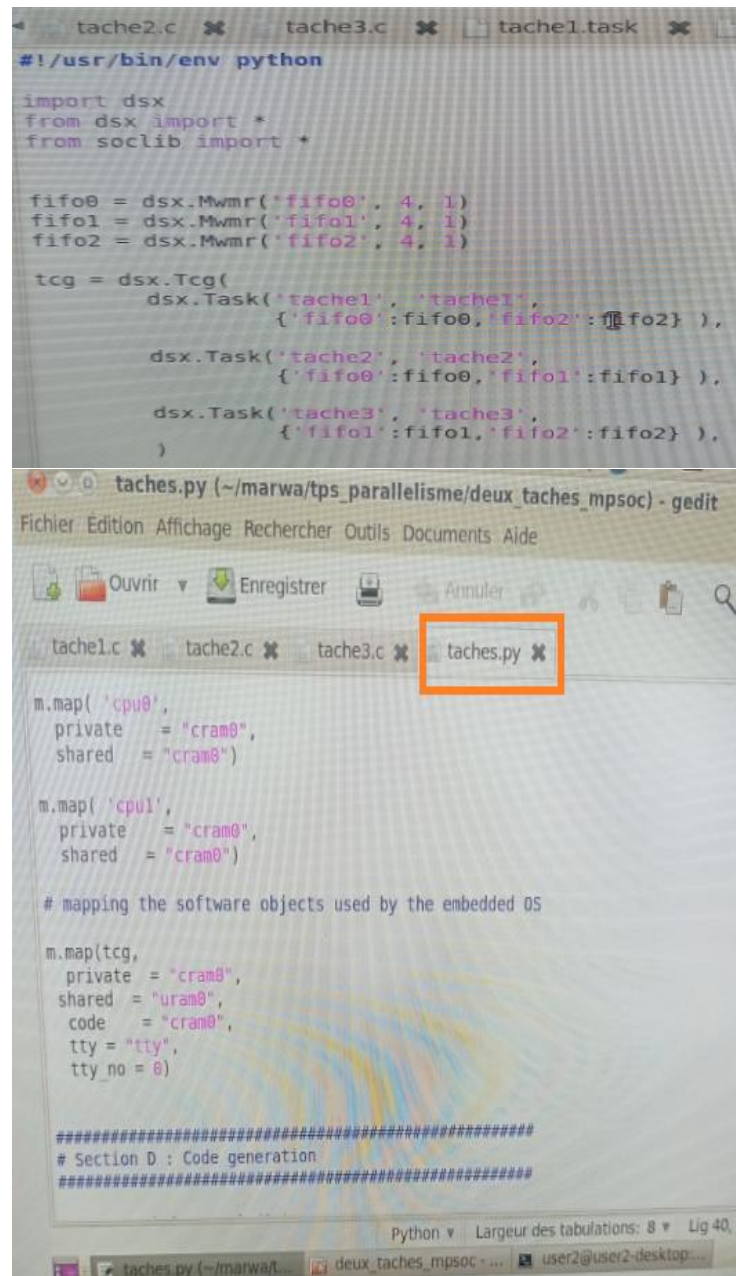
```

Code du “Tache3.task”



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

Code du fichier python “Taches.py” :



```
#!/usr/bin/env python

import dsx
from dsx import *
from soclib import *

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

tcg = dsx.Tcg(
    dsx.Task('tache1', 'tache1',
        {'fifo0':fifo0, 'fifo2':fifo2} ),
    dsx.Task('tache2', 'tache2',
        {'fifo0':fifo0, 'fifo1':fifo1} ),
    dsx.Task('tache3', 'tache3',
        {'fifo1':fifo1, 'fifo2':fifo2} ),
)

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

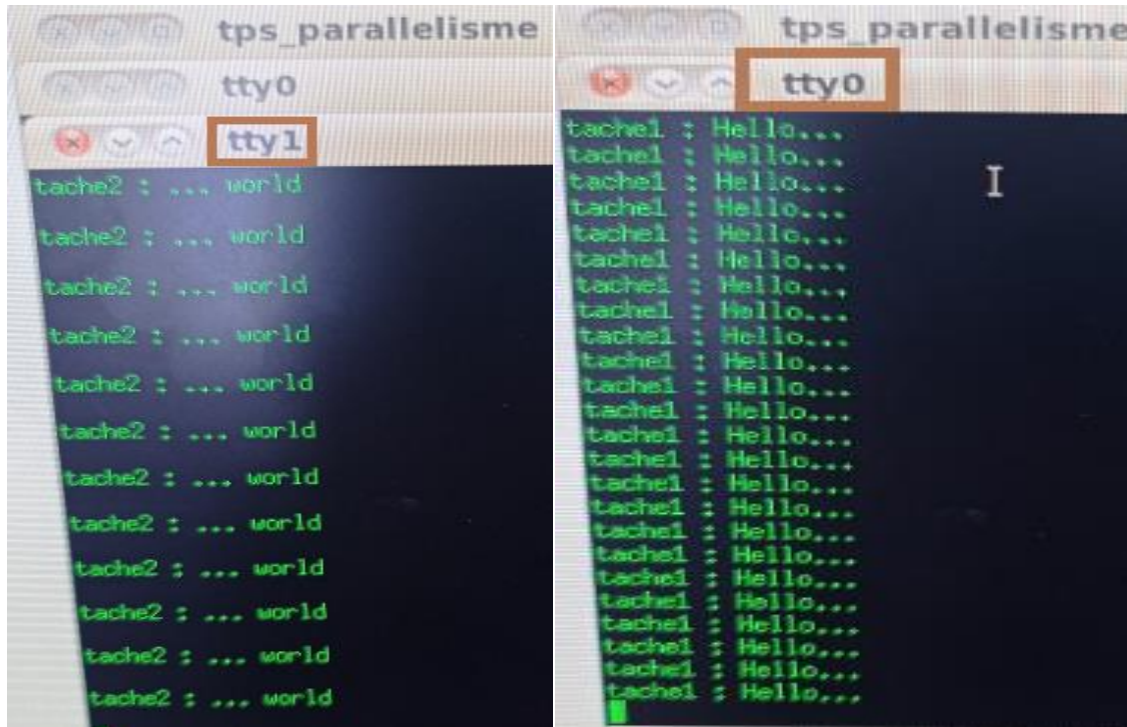
m.map( 'cpu1',
    private = "cram0",
    shared = "cram0")

# 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
#####
```

On fait l'exécution avec "`./exe.muteks_hard`"

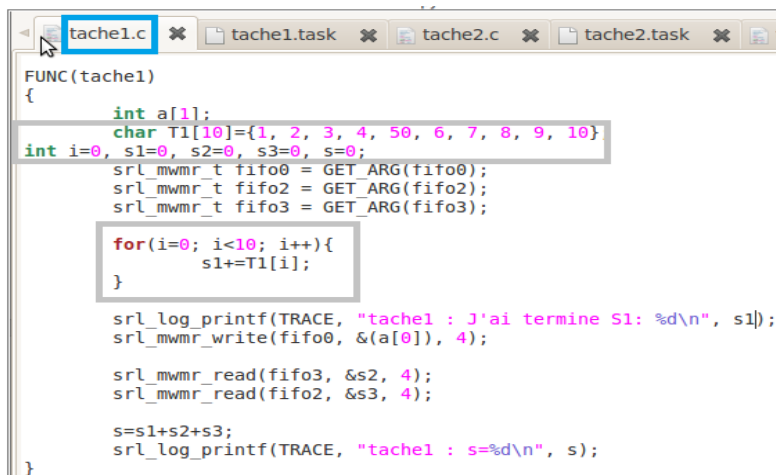


Calcul de somme avec multiprocesseur

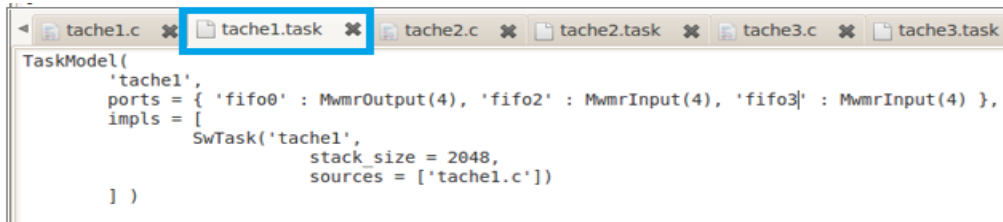
Notre TP contient les fichiers suivantes :



Code du « **tache1.c** »

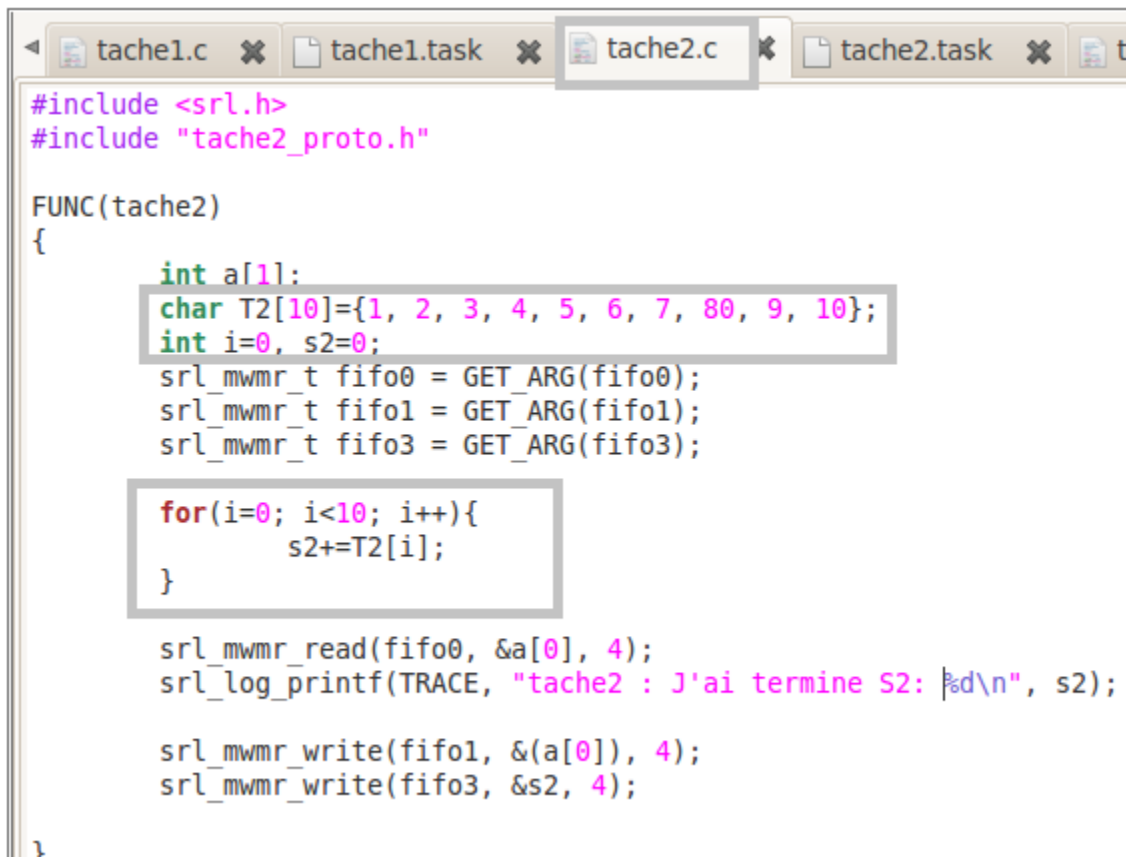


Code du « **tache1.task** » :



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

Code du « **tache2.c** » :



```
#include <srl.h>
#include "tache2_proto.h"


FUNC(tache2)
{
  int a[11];
  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);
}
```

Code du « **tache2.task** »



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


Code du « **tache3.c** » :



```
#include <srl.h>
#include "tache3_proto.h"

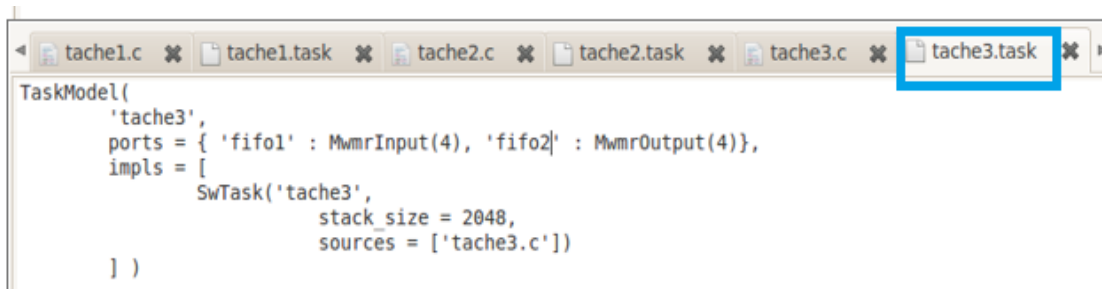
FUNC(tache3)
{
    int a[1];
    char T3[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i=0, s3=0;
    srl_mwmr_t fifo1 = GET_ARG(fifo1);
    srl_mwmr_t fifo2 = GET_ARG(fifo2);

    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);
}
```

Code du « **tache3.task** » :



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

Code du « **taches.python** » :

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

Code du « **vgmnoirq_mono.py** » :

```
archi = VgmnoirqMono(ntty = 3)

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

m = Mapper(archi, tcg)

# mapping the MWMR 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 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() )
```

```
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'%1, base, 0x200000, True)
ram.addSegment('uram%d'%1, base + 0x400000, 0x200000, False)
ram.vci // vgmn.to_target.new()
ram.addSegment('boot', 0xbfc00000, 0x1000, True) # Mips boot address, 0x1000 octets,
cacheable
ram.addSegment('except', 0x80000000, 0x1000, True) # Mips exception address, 0x1000
octets, cacheable

(ntty)))
tty = pf.create('caba:vci_multi_tty', 'tty', names = map(lambda x:'tty%d'%x, range
tty.addSegment('tty', 0x95400000, 0x20*ntty, False)
tty.vci // vgmn.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__':
    VgmnNoirqMono().generate(soclib.PfDriver())
```

On fait l'exécution avec `./exe.muteks_hard`

Le résultat nous donne la somme du trois taches :

[illegible]

Conclusion

Le parallélisme est dorénavant utilisé dans la majorité des architectures, des systèmes embarqués aux superordinateurs. Les monoprocesseurs sont remplacés par des processeurs multicœurs. Il présente les grandes classes d'architectures parallèles avec leurs ressources et organisations mémoire, en distinguant les architectures homogènes et hétérogènes.

Les principes des techniques de programmation sont introduits avec les extensions parallèles des langages de programmation couramment utilisés et les modèles de programmation visant à rapprocher la programmation parallèle de la programmation séquentielle, en incluant les spécificités des architectures.

Enfin, les modèles et métriques d'évaluation des performances sont examinés.