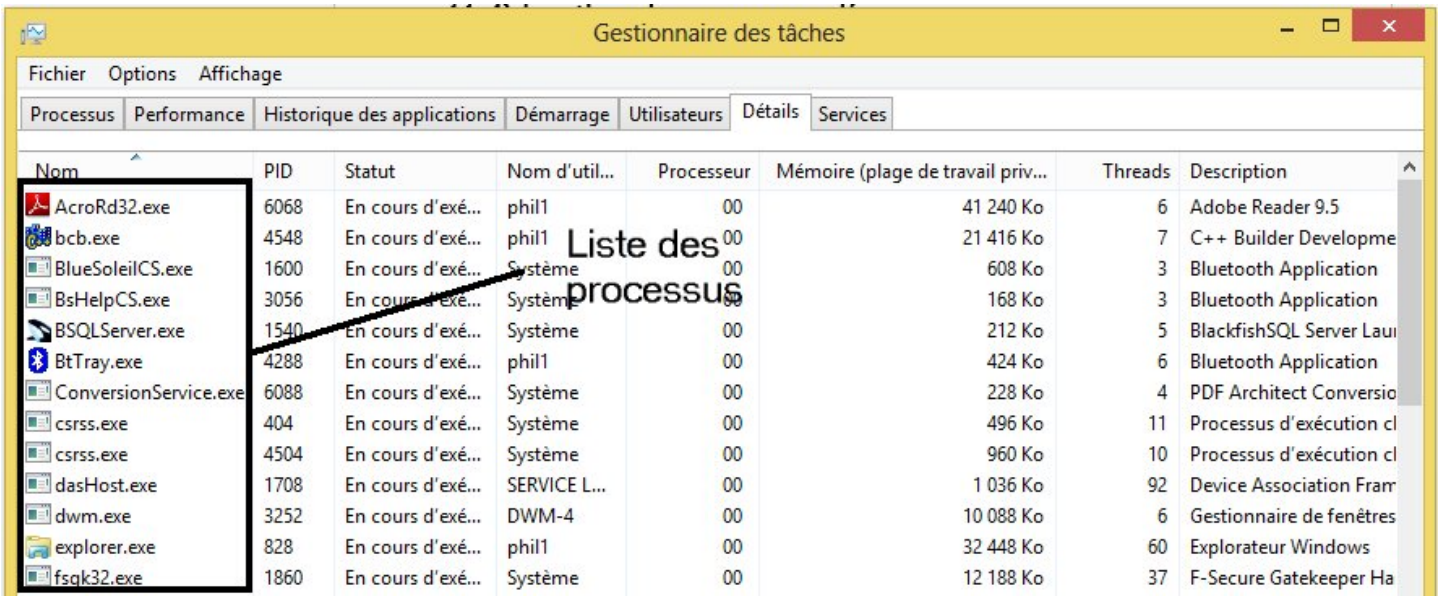
Les processus légers (Thread) sous LINUX

1. Rappel les processus lourds

Un processus lourd est un programme en cours d'exécution. Chaque processus lourd implique la gestion d'un espace d'adressage virtuel MMU et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).
Le principal avantage de ces processus est la protection mémoire entre les processus du même système.
Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

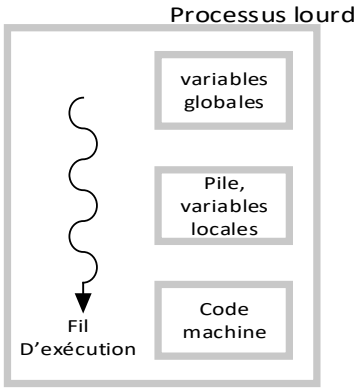
- Leur création nécessite des appels systèmes coûteux en temps,
- Le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- Le coût des mécanismes de protection associés au processus,
- L'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- Le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

La liste des processus lourd est donnée par le gestionnaire des taches sous windows:



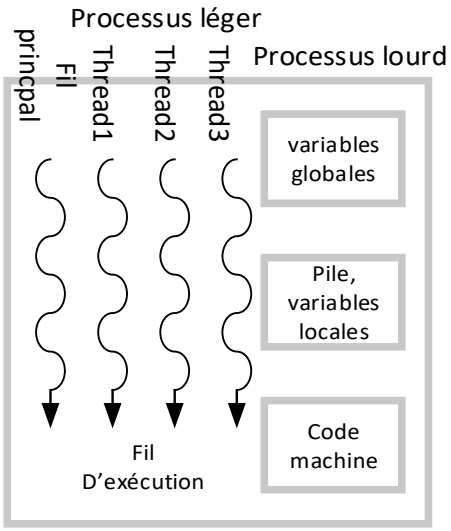
Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).

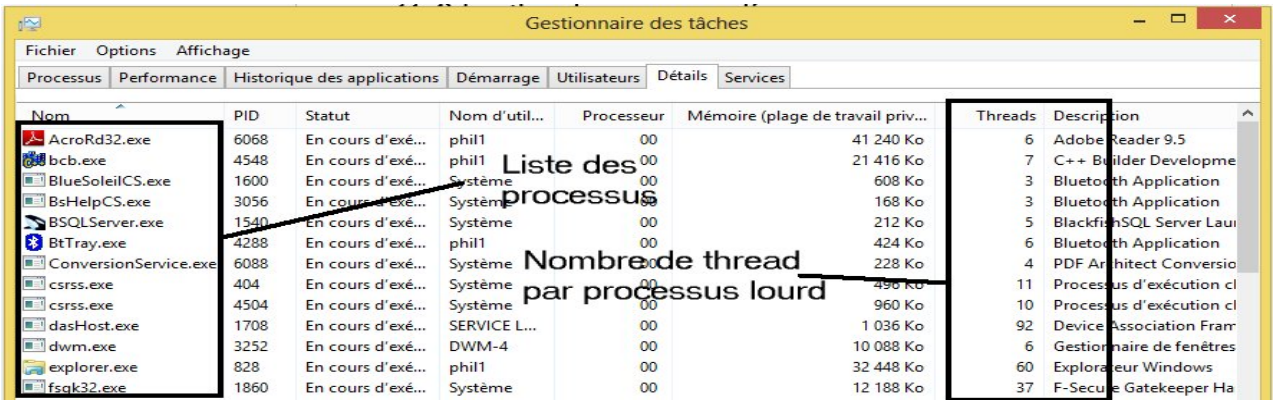


2) Les processus légers

Les threads ou processus léger permettent de dérouler plusieurs blocs d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction ou méthode. Chaque fonction associée à un thread va être exécutée de façon parallèle et indépendante.

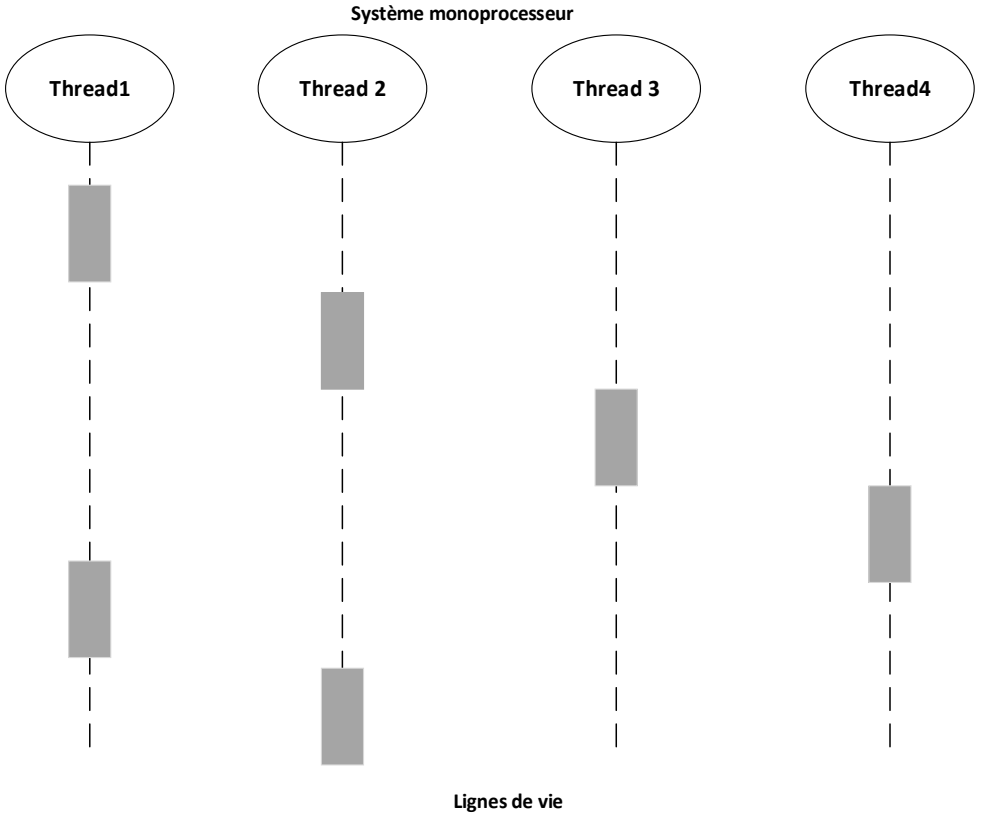


Le système d'exploitation gère donc des centaines de thread :



Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

Dans un système monoprocesseur l'illusion sera donnée que chaque thread est exécuté de façon parallèle ou concurrente. En réalité chaque thread sera exécuté à tour de rôle mais très rapidement.



Un programme comporte toujours un thread dit thread principal correspondant tout simplement à la méthode main()

Avantages :

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte de processus lourd

- **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads

Inconvénients :

Programmation utilisant des threads est toutefois plus difficile : obligation de mettre en place des **mécanismes de synchronisation**.

3) Création d'un thread sous LINUX

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);
```

•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.
•Le troisième argument est un pointeur vers la fonction à exécuter par 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.

Celle-ci donne naissance à un nouveau fil d'exécution, qui va démarrer en invoquant la fonction dont on passe le nom en argument.

Pour attendre la fin d'un thread

Voici le prototype de cette fonction :

```
#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).

Exemple d'utilisation :

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


void *thread_1(void *arg)
{
   printf("Nous sommes dans le thread.\n");
int monargument = *((int *)arg);// on récupère l'entier ma valeur
   /* Pour enlever le warning */
   (void) arg;
   pthread_exit(NULL);
}

int main(void)
{
   pthread_t thread1;
 int mavaleur=2 ;
   printf("Avant la création du thread.\n");

   if (pthread_create(&thread1, NULL, thread_1, mavaleur)) {
        perror("pthread_create");
        return EXIT_FAILURE;
   }

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

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

   return EXIT_SUCCESS;
}
```

4) L'attribut des threads et les techniques d'ordonnancement

Chaque thread est doté d'un certain nombre d'attributs, regroupés dans un type opaque phtread_attr_t.
Dans un premier temps, il faut invoquer la fonction pthread_attr_init( ) en lui transmettant un pointeur sur une variable de type pthread_attr_t.

```
int pthread_attr_init (pthread_attr_t * attributs);
```

Les fonctions permettant donc de modifier un attribut ou de lire la valeur de l'attribut d'un thread se nomment :
pthread_attr_setXXX( ) et pthread_attr_getXXX( ),

Les attributs XXX peuvent se nommer schedpolicy, schedparam,

Pour modifier la méthode d'ordonnancement employée pour exécuter le thread :

```
int pthread_attr_setschedpolicy (pthread_attr_t * attributs, int valeur);
```

Voir page 31 les techniques d'ordonnancement du type round robin ou FIFO

L'attribut schedparam contient principalement la priorité du thread.
La fonction setschedparam permet de modifier cette priorité :

```
int pthread_attr_setschedparam (pthread_attr_t * attributs,const struct sched_param * params);
```

Le programme suivant permet de mettre en place quatre taches élémentaires de priorité différentes en utilisant une technique d'ordonnancement du type FIFO

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
```

La fonction **pthread_attr_setinheritsched**() définit l'attribut d'héritage d'ordonnancement de l'objet d'attributs de thread auquel *attr* fait référence à la valeur indiquée par *inheritsched*. Cet attribut détermine si un thread créé en utilisant l'objet d'attributs de thread *attr* héritera les paramètres d'ordonnancement du thread appelant.

- **PTHREAD_INHERIT_SCHED** Les threads qui sont créés en utilisant *attr* héritent des attributs d'ordonnancement du thread appelant. Les attributs d'ordonnancement dans *attr* sont ignorés.
- **PTHREAD_EXPLICIT_SCHED** Les threads qui sont créés en utilisant *attr* prennent comme attributs d'ordonnancement les valeurs spécifiées dans l'objet d'attributs de thread.

La valeur par défaut de l'attribut d'ordonnancement pour un nouvel objet d'attributs de thread est **PTHREAD_INHERIT_SCHED**.

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#define NB_THREADS 4

void * fonction_thread(void * arg)
{
	int numero = (int) arg;
	int i, j;

	sleep(numero*2);

	fprintf(stderr, "Debut thread %d a %ld\n", numero, time(NULL));
	for (i = 0; i < 100000; i ++)
		for (j = 0; j < 10000; j ++)
			;
	fprintf(stderr, "Fin thread %d a %ld\n", numero, time(NULL));
	return NULL;
}

int main(void)
{
	int i;
	int err;
	pthread_attr_t attr;
	struct sched_param param;
	pthread_t thr[NB_THREADS];

	pthread_attr_init (& attr);
	if ((err = pthread_attr_setschedpolicy(& attr, SCHED_FIFO)) != 0) {
		fprintf(stderr, "setschedpolicy: %s\n", strerror(err));
		exit(EXIT_FAILURE);
	}

	// SCHED_RR pour le round robin
```

```
for (i = 0; i < NB_THREADS; i ++) {
        param.sched_priority = (i + 1) * 10;
        if ((err = pthread_attr_setschedparam(& attr, & param)) != 0) {
            fprintf(stderr, "setschedparam: %s\n", strerror(err));
            exit(EXIT_FAILURE);
        }
        if ((err = pthread_create(& (thr[i]), & attr, fonction_thread, (void *) (i+1))) != 0) {
            fprintf(stderr, "pthread_create: %s\n", strerror(err));
            exit(EXIT_FAILURE);
        }
    }
    for (i = 0; i < NB_THREADS; i ++)
        pthread_join(thr[i], NULL);

    return EXIT_SUCCESS;
}
```

5) L'attribution du processeur

Il est possible d'attribuer aux cœurs du processeur un fil d'exécution principal (main) ou secondaires (threads).
Pour le **fil d'exécution principal** (fonction main) on utilisera les lignes de code suivantes dans la fonction main() :

```
cpu_set_t cpuset;
// CPU_ZERO: cette fonction initialise le CPU.
CPU_ZERO(&cpuset);
 // CPU_SET: cette fonction additionne un cœur de CPU aux processeurs, c'est le premier
// paramètre d'entrée qui détermine le cœur de CPU choisi (dans le cas présent 0)
 CPU_SET(0, &cpuset);
 // sched_setaffinity positionne le masque d'affinité du CPO pour le fil d'exécution principal
if (sched_setaffinity(0, sizeof(cpuset), & cpuset) !=0)
{
        printf("sched_setaffinity \n");
        exit(0);
    }
```

Pour les fils d'exécution secondaires ou thread:

```
const pthread_t pid = pthread_self();// numéro de thread
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
// paramètre d'entrée qui détermine le cœur de CPU choisi (dans le cas présent 0)
CPU_SET(0, &cpuset);
const int set_result = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
if (set_result!=0)
  {
            printf("sched_setaffinity");
            exit(0);
      }
```

Pour connaitre le nombre de cœur CPU sous LINUX

cat /proc/cpuinfo





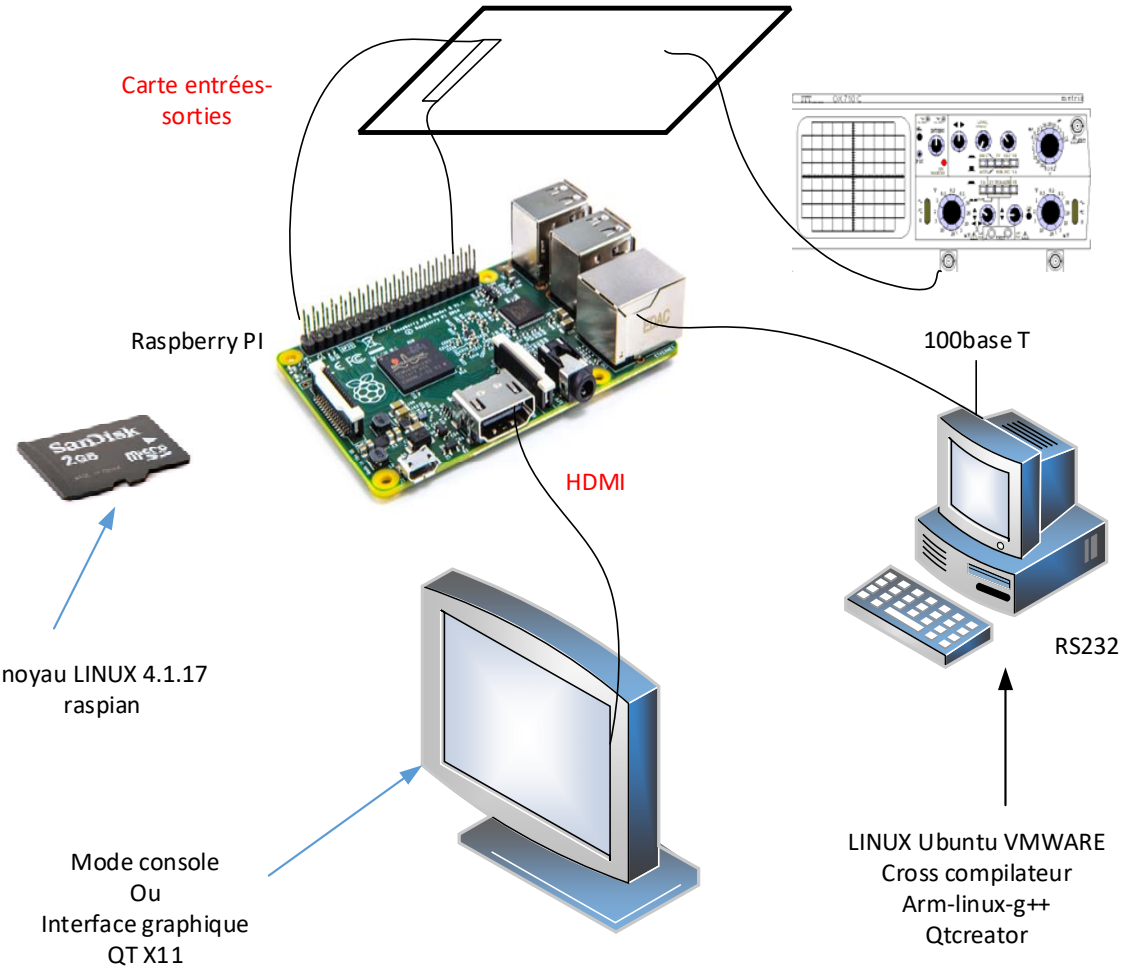
Les systèmes d'exploitation temps réel TD :Thread

TD : les Threads

1) Présentation :

On donne page N°5, le programme permettant de mettre en place les notions vues en cours sur les threads. Ce programme sera implanté sur un environnement LINUX avec la plateforme Raspberry P3

2) Schéma de câblage



3) Questionnaires

3.1) Branchez votre carte Raspberry PI3 et sur putty naviguez vers le répertoire /home/pi/rasp/thread. Vérifiez la présence du fichier source ThreadRR.c

3.2) Compilez votre applicatif

```
gcc -o ThreadRR ThreadRR.c -lpthread -D_GNU_SOURCE
```

Lancez votre applicatif et observez le résultat. Que proposez vous

✍

Sur putty naviguez vers le répertoire /home/pi/rasp/drivers/raspberry/module

Et insérez le module gpiosen.ko

```
sudo insmod gpiosen.ko
```

Vérifiez la présence du fichier de caractère spécial dans /dev/gpio_drv

Retournez dans le répertoire /home/pi/rasp/thread et relancez votre applicatif

Lancez une deuxième console sur putty et lancez les deux commandes suivantes :

Liste des processus

```
ps -aux
```

Threads en activité dans le processus lourd ThreadRR

```
top -H -p n°du processus ThreadRR
```

Observez les résultats obtenus sur les consoles à l'oscilloscope et conclure

✍

3.3) Modifiez votre programme deux façons à lancer les Threads (main thread1 thread2) sur le CPU 0, on utilisera winscp

3.4) Compilez ce nouveau programme et lancez-le :

Observez les résultats obtenus sur les consoles à l'oscilloscope et conclure

✍

3.5) Arrêtez le programme précédent et relancez ce processus lourd en mode temps réel mou

```
chrt -r 25 sudo ./ThreadRR
```

Observez les résultats obtenus sur les consoles à l'oscilloscope et conclure

✍

3.6) Arrêtez le programme précédent et relancez ce processus lourd en mode temps réel mou

```
chrt -r 40 sudo ./ThreadRR
```

Observez les résultats obtenus sur les consoles à l'oscilloscope et conclure

✍

```c
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <ctype.h>
#include <sys/time.h>
int fd=0;

void *thread_1(void *arg)
{
int errno=0;
char buff[256];
char choice[10];

/*************Reglage de l'attribution processeur pour le thread1*****/
/*************************************************/
const pthread_t pid = pthread_self();
cpu_set_t cpuset;
// CPU_ZERO: This macro initializes the CPU set set to be the empty set.
CPU_ZERO(&cpuset);
// CPU_SET: This macro adds cpu to the CPU set set.
CPU_SET(1, &cpuset);

// pthread_setaffinity_np: The pthread_setaffinity_np() function sets the CPU
affinity mask of the thread thread to the CPU set pointed to by cpuset. If the
call is successful, and the thread is not currently running on one of the CPUs in
cpuset, then it is migrated to one of those CPUs.
const int set_result = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
printf("CPU thread1 = %d \n",sched_getcpu());
while(1)
{
choice[0]=0x00;// LEDS à 0

write( fd, choice, 1 );
choice[0]=0x01;// LED1 à 1

write( fd, choice, 1 );
}

}



void *thread_2(void *arg)
{
int errno=0;
char buff[256];
char choice[10];
cpu_set_t cpuset;

/**Je recupere le param   tre d'entr   e de la fonction thread2 */
int var = *((int *)arg);
printf("VAR = %d \n",var);

/*************Reglage de l'attribution processeur pour le thread2*****/
```

```
60 /*******************************************************/
61
62 const pthread_t pid = pthread_self();
63
64 // CPU_ZERO: This macro initializes the CPU set set to be the empty set.
65 CPU_ZERO(&cpuset);
66 // CPU_SET: This macro adds cpu to the CPU set set.
67 CPU_SET(2, &cpuset);
68
69
70 const int set_result = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
71 printf("CPU thread2 = %d \n",sched_getcpu());
72 while(1)
73 {
74 choice[0]=0x00;// LEDS a    0
75
76 write( fd, choice, 1 );
77 choice[0]=0x02;// LED2 a    1
78
79 write( fd, choice, 1 );
80 }
81
82 }
83
84 int main(void)
85 {
86 int i;
87 int err;
88 pthread_attr_t attr;
89 struct sched_param param;
90 pthread_t thread1;
91 pthread_t thread2;
92 cpu_set_t cpuset;
93 int arg=2;
94 int choix=0;
95 setbuf(stdin,NULL);
96
/*****************************************************************************
****/
97 /* Ouverture du pilote pour la gestion du port entree sortie de la PI3 */
98 if ((fd = open( "/dev/gpio_drv", O_RDWR ))<0){
99 printf("Impossible d'ouvrir le pilote \n");
100 exit(0);
101 }
102
103
104 /*************Reglage de l'attribution processeur pour le fil d'execution
principal*****/
105 /*******************************************************/
106 CPU_ZERO(& cpuset); // coeur 0
107 CPU_SET(0, &cpuset);// coeur 0
108 if (sched_setaffinity(0, sizeof(cpuset), & cpuset) !=0) {
109 perror("sched_setaffinity");
110 exit(EXIT_FAILURE);
111 }
112 printf("CPU = %d fil d'execution principale\n",sched_getcpu());
113
114
/*****************************************************************************
****/
115 /* Reglage des techniques d'ordonnancement choisie pour tous les processus
```

```
legers et le fil d'execution principal*/
116 /*****************************************************/
117 pthread_attr_init (& attr); //initialisation de la structure pthread_attr_t
118
119 if ((err = pthread_attr_setschedpolicy(& attr, SCHED_RR)) != 0) {// reglage de
l'algorithme d'ordonnancement de round robin
120 fprintf(stderr, "setschedpolicy: %s\n", strerror(err));
121 exit(EXIT_FAILURE);
122 }
123 if ((err = pthread_attr_setinheritsched(& attr, PTHREAD_EXPLICIT_SCHED)) != 0) {
124 fprintf(stderr, "setinheritsched: %s\n", strerror(err));
125 exit(EXIT_FAILURE);
126 }
127 param.sched_priority = 40; // reglage de la
priorite du thread
128 if ((err = pthread_attr_setschedparam(& attr, & param)) != 0) {
129 fprintf(stderr, "setschedparam: %s\n", strerror(err));
130 exit(EXIT_FAILURE);
131 }
132
133
/*************************************************************************
****/
134 /****************creation des processus legers threads1 et thread
2*****************************/
135
136 if (pthread_create(&thread1,& attr, thread_1, NULL)) {
137 printf("pthread_create1 error");
138 return EXIT_FAILURE;
139 }
140
141 if (pthread_create(&thread2,& attr, thread_2, &arg)){
142 printf("pthread_create2 error");
143 return EXIT_FAILURE;
144 }
145 printf("Tapez 1 pour sortir du programme \n");
146
147 while(1){
148 scanf("%d",&choix);
149 if (choix==1)exit(0);
150 }
151
152 return 1;
153 }
154
155 //gcc -o ThreadRR ThreadRR.c -lpthread -D_GNU_SOURCE
156
157
158
```