POLYTECH NANTES

Spécialité Génie Electrique

SETR

D. DELFIEU

- Quatrième année -

Reproduction interdite sans autorisation de l'auteur et de l'école

POLYTECH
Premier réseau national
des écoles d'ingénieurs
polytechniques des universités

Site de la Chantrerie
Rue Christian Pauc - BP50609
44306 Nantes cedex 3 - France
Tél. +33 (0)2 40 68 32 00
Fax. +33 (0)2 40 68 32 32

Site de Gavy
Gavy Océanis - BP152
44603 St-Nazaire cedex - France
Tél. +33 (0)2 40 90 50 30
Fax. +33 (0)2 40 90 50 24

www.polytech.univ-nantes.fr

Systèmes Temps Réel

David Delfieu

Ecole Polytechnique de l'Université de Nantes - Dept GE - $4A$ - S_8

January 16, 2023

Sommaire

1 Introduction

2 Architecture Matérielle

3 Architecture Logicielle

Introduction

- Premier système temps Réel embarqué :
  Apollo Guidance Computer : Système de guidage de l'alunisseur des missions Apollo .
- Aujourd'hui :
  - Régulateur de vitesse ;
  - Voitures autonomes ;
  - Avionique : Pilote automatique ;
  - Astronautique : système de guidage de fusée ;
  - Grand public : smartphones, consoles de jeux, gps, électroménager.

Système Embarqué Temps Réel (*SETR*)

- Système :
  - microcontrôleur,microprocesseur FPGA/CPLD ...
  - Capteurs : Accéléromètres, lidars, température, pression...
- Embarqué : Alimentation propre ;
- Temps réel :
  - Puissance de calcul suffisante pour répondre au réel.
  - Pas surdimensionnée pour ménager l'autonomie.

Caractérisation d'un *SETR*

Caractéristiques d'un Système Embarqué Temps réel

- Sureté de fonctionnement : CEM, robustesse, redondance ;
- Temps d'exécution déterministe ou borné ;
- Consommation énergétique ajustée ;
- Encombrement minimal ;
- Espace mémoire ajusté.

Sommaire

1 Introduction

2 Architecture Matérielle
- Les principales puces de l'embarqué
- Les principales architectures
- Les nano-ordinateurs

3 Architecture Logicielle

Le marché des puces

- Marché des puces : 550,5 $\overline{M}$ \$ en 2021 ;
- Marché des smatrphones : 800 $\overline{M}$ \$ en 2021 ;
- Marché des *PC* est autour de 3 $\overline{M}$ \$;
- Electronique embarquée : 25 % du cout d'une automobile ! 10% d'un avion ;

Système embarqué grand public : Smartphone

Processeurs Apple :

- Iphone 12 : A14, 8 coeurs, 2020,
- Iphone 13,14 : A15, 16 coeurs, 2021,
- Iphone 14 pro : A16, 2022
  - 6 coeurs : 2 coeurs performance, 4 coeurs haute efficacité énergétique
  - 5 GPU
  - 16 coeurs neuronaux.

Processeurs Samsung :

- S10 : Exynos 9820, 8 coeurs, 2019,
- S21 : Exynos 990, 8 coeurs, 2021
- S22 : Ultra Exynos , 8 Coeurs, S22 Ultra, 4 nm
  - 1coeur 2.8 GHz Cortex-X2
  - 3 coeurs 2.50 GHz Cortex-A710
  - 4 coeurs1.8 GHz Cortex-A510

Autres microprocesseuret microcontrôleur

- SETR à base de microprocesseur :
  - Commande de radars fixes, routeur, calculateur embarqué.
  - Grande taille du code et de la mémoire, grande consommation et grosse capacité de calcul.
    - ARM ,
    - PowerPC (Apple , IBM, Freescale),
    - Intel Medfield , x86 ;
- *SETR* à base de microcontrôleur :
  - Automobile, domotique, électroménager.
  - Petite taille du code et de la mémoire, capacité de calcul limitée...
    - Atmega , PIC 16F (Microchip),
    - M68HC*** (NXP Semiconductors)
    - ST10F*** (STMicroelectronics)
    - microcontrôleurde traitement du signal :

      Texas Instrument : C5000,C6000
      Microchip : dsPIC30F et dsPIC33F

Système industriel à processeur : Rack

Système en Rack :

- Pas de problème de consommation et ni d'espace ;
- Processeur puissant, nombreuses cartes d'entrées sorties
- Tolérance aux pannes, sureté de fonctionnement.

Figure: Rack

Nano-ordinateur

System on Module (*SoM*) ou Single-board Computer (*SoM*) :

- Raspberry *PI_4* :
  - 4 coeurs ARM Cortex-A72 , 1,5 Ghz, 4GO, 2 HDMI
  - Debian (Linux) : Raspbian ou Windows 10
  - Wifi , bluetooth , 95 euros
- LattePanda 3 :
  - 4 coeurs Intel , 2 Ghz, 4GO, 1 HDMI
  - Windows 10
  - Wifi , bluetooth , 300 euros
- NVIDIA Jetson
  - 4 coeurs ARM - GPU 128 Coeurs, 1,4 Ghz, 2GO, 1 HDMI
  - Ubuntu (Linux),
  - Wifi , bluetooth , 80 euros

Raspberry PI4

- ARM Cortex A72 1,5GHZ
- 4 Go
- 2 HDMI , 2 USB 3.0, 2 USB 2.0
- Sortie son jack et sortie son 5.1 HDMI
- MicroSD
- 10/100/1000 Ethernet
- wifi/bluetooth 5.0
- 17 *GPIO* ; UART , I2C , . . .
- Consommation Maximale 885 mA max
- Raspbian OS mais :
  - Windows 10,
  - Ubuntu, Debian, Fedora, FreeBSD. . .

Sommaire

1 Introduction

2 Architecture Matérielle

3 Architecture Logicielle
- Programmation synchrone ou asynchrone ?
- Embedded LINUX
- La programmation multi-tâches
- La communication entre processus UNIX
- Les Threads Posix
- Accés aux variables blobales partagées

Operating System (*OS*)

Le rôle d'un *OS* est d'ordonnancer des tâches :

Ordonnancement

L'ordonnancement est réalisé suivant une politique d'ordonnancement :

- Fifo : Premier arrivé, premier servi
- Round Robin : Files d'attente sur des niveaux de priorité avec préemption
- User : Définie par l'utilisateur

Temps de réponse

Le choix de l'ordonnancement multi-taches se fait en fonction de l'application à réaliser. Cela permet de produire des temps de réponse adéquats avec la dynamique de l'application.

Catégorisation des Real Time Operating System (*RTOS*)

On peux classer les *RTOS* suivant leur aptitudes à respecter des contraintes sur les temps de réponse :

- *Absolu* : Les temps de réponse sont fixés
- *Strict certifiable* : Les temps de réponse sont fluctuant mais bornés
- *Strict non certifiable* : Les temps de réponse ne sont pas garantis mais indépendants de toute activité moins prioritaire du système.
- *Souple* : Les temps de réponse ne sont pas garantis mais indépendants des activités moins prioritaires en espace utilisateur.

RTOS tournant sur Raspberry

On peux les classer suivant leur niveau :

- Il existe des version strictes :
  - Stricte certifiable : RTEM
  - Stricte non certifiable : XENOMAI.
- La version que nous allons utiliser est dite souple.

Notion de tâches

Une application de contrôle commande est généralement décomposée en tâches.

Tâche

Une tâche correspond à une fonctionnalité logicielle et matérielle dont la granularité offre une complexité permettant une implémentation rapide.

Exemple : Régulation en courant d'un moteur

On peut décomposer cette application en trois tâches :

- Une tâche d'acquisition de la consigne en courant;
- Une tâche de calcul et d'application du correcteur ;
- Une tâche d'échantillonnage lecture du courant ;
- Boucle infinie du programme principal.

Programmation Synchrone

La programmation est dite synchrone car les tâches sont lancées de façon synchrone par rapport à des évènements correspondant généralement à des interruptions.

Implémentation :

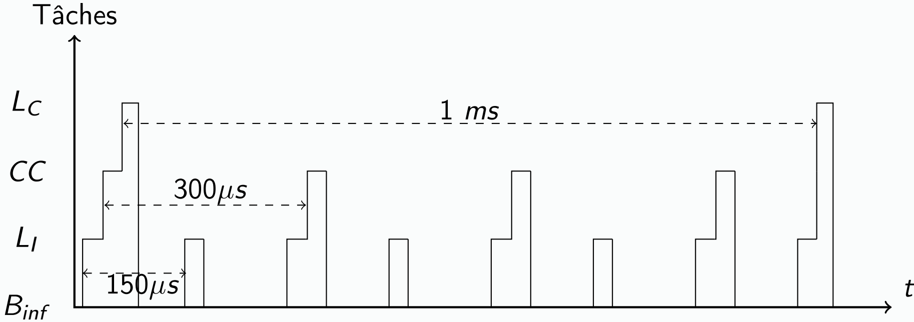
- L_C : Tâche d'acquisition de consigne : durée $= 10\mu s$, periode$= 1ms$;
- CC : Tâche de calcul du correcteur : durée $= 30\mu s$, periode$= 300\mu s$;
- L_I : Tâche d'acquisition du courant : durée $= 30\mu s$, periode$= 150\mu s$;
- B_{inf} : Boucle infinie du programme principal, durée $= 3\mu s$, periode$= 150\mu s$;

La tâche sera associée à un sous-programme d'interruption.

Programmation Synchrone

```
void LC(// Lecture de la consigne){}
void CC() {//Calcul du correcteur}
void LI(); {//Lecture du courant }

ISR(TIMER0_OVF_vect){LC(); }

ISR(TIMER1_OVF_vect){ CC(); }

ISR(TIMER2_OVF_vect){LI();}

int main(void){
  DDRB=0x1F;DDRC=0x00;
  TCCR0=0;TCCR0 = (1 << CS01)+(1 << CS00); // 1 sec
  TCCR1A=0;TCCR1B = (1 << CS11);  // 100 micros
  TCCR2=0;TCCR2 = (1 << CS11);  // 200 micros
  TIMSK = (1<<TOIE2)+(1<<TOIE1)+(1<<TOIE0);
  sei();
  while(1) {// boucle infinie}
  return(0);
}
```

Déroulement de tâches en Synchrone



Programmation Synchrone

Avantages : "La programmation est hyper-efficace car très proche du hard"

- Structure minimale : très performante ;
- Pas de code inutile ;
- Comportement déterministe : Chaque tâche est exécutée à un instant prédéterminé;

Programmation Synchrone

Inconvénients :

- Temps total des tâches : 73 μs, periode=150μs
- Si on veut rajouter des tâches :
  - Une Interruption supplémentaire pour chaque nouvelle tâche;
  - Exemple : La somme des temps d'exécution des tâches doit être $\leq$ à 77μs ;
  - Si on veut introduire la notion de priorité : On va devoir faire des modifications sur la partie Hardware.
- Les coûts de développement sont importants :
  - Codage en C : Connaître les registres, les bits...
  - Maîtrise complète des fonctionnalités du microcontrôleur
- La lisibilité du code, très lié au hard, est très faible, le code n'est pas portable.

Programmation Asynchrone

L'application est représentée par un système d'exploitation qui :

- lance des tâches ;
- qui attend des comptes rendus d'exécution ;
- Les tâches sont des sous-programmes particuliers "détachés du programme principal" ;

Ces sous-programmes peuvent être implémentés (sous UNIX/LINUX) par des processus ou des threads.

Programmation Asynchrone

Avantages : "la programmation est plus détachée du hard"

- Structure souple ;
- Ajout ou suppression de tâches simples ;
- Code lisible, détaché du hard : Communication avec le hard via des drivers ;
- Code portable, maintenance aisée ;
- Suivant les drivers portés dans le noyau on va pouvoir faire des tâches de très haut niveau : Protocole TCP/IP, telnet, serveur http, ...

Programmation Asynchrone

Inconvénients :

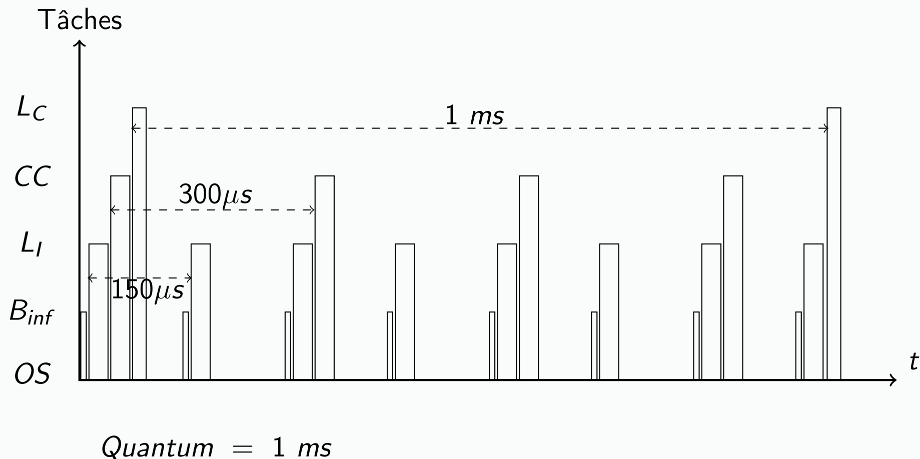
- L'image (code implanté dans le micro) intègre un mini système d'exploitation (qlqs Mo) en plus de l'application ;
- Un peu moins performant lors des changements de contextes ;
- Comportement temporel pas forcement déterministe ;
- Le déterminisme temporel peut être atteint pour un nombre très restreint de tâches de haute priorité.
- Pour atteindre un même niveau de déterminisme qu'en synchrone : il faut un processeur plus puissant donc plus consommateur d'énergie électrique.

Quantum

Inconvénients :

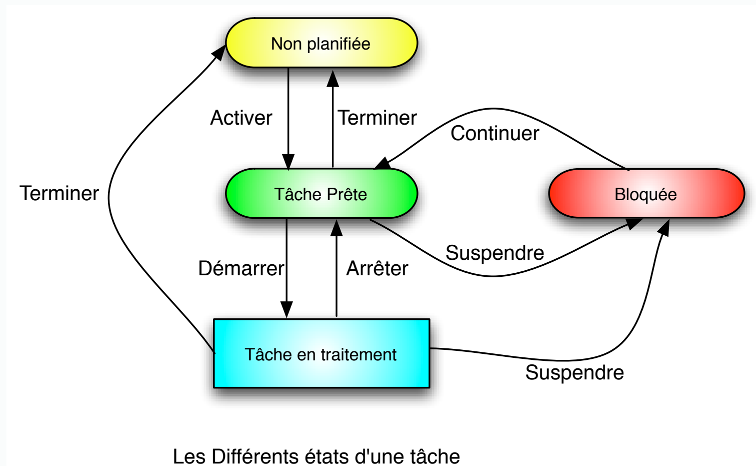
- Une tâche dispose d'un quantum de temps ;
- Une tâche qui est en traitement sort de son quantum :
  - Soit elle se termine normalement ;
  - Soit si elle a fini son quantum de temps ;
  - Soit quand elle a fait une demande d'une entrée-sortie qui la bloque ou la fait attendre ;
- Quantum : assure une redistribution régulière de la ressource *CPU*.

Déroulement de tâches en Asynchrone



$Quantum = 1\ ms$

Les états d'une tâche



Les Différents états d'une tâche

Les états d'un tâche

- Activer : La tâche entre dans un processus électif : File d'attente des tâches "Prêtes";
- Démarrer : Le tâche en tête de la file d'attente "Prête" est élue, elle a 100% du CPU ;
- Arrêter : Le tâche est interrompue, elle a terminée son quantum de temps ;
- Suspendre : Le tâche a demandé une ressource non encore disponible ou s'est endormi elle-même (sleep) ;
- Continuer : La ressource demandée est maintenant disponible ou le sommeil est terminé, le tâche rentre à nouveau dans la file d'attente "Prête".
- Terminer : Le tâche s'arrête définitivement par l'appel à exit ou return. Elle peut aussi recevoir un signal de fin émis par une tâche en traitement.

SETR et LINUX

Pourquoi LINUX

- LINUX est dérive d'UNIX, un OS le plus fiable du marché ;
- LINUX est libre : le source est disponible sans royalties à reverser ;
- LINUX est foisonnant : de nombreux contributeurs le font évoluer ;
- Les versions stables de LINUX sont FIABLES ;
- LINUX est entouré d'une communauté de développeurs fournissant une aide rapide ;
- LINUX offre une connectivité IP en standard.

SETR et LINUX

LINUX : sur quel système ?

- LINUX est décliné sur un grand nombre d'architecture : x86 , PowerPC , microprocesseur ARM , MIPS , 68000 , Coldfire ,
- Taille du noyau est modeste 500ko à 4 Mo ;
- différentes distributions sont proposées suivant les domaines applicatifs :
  - PC
  - routeur IP,
  - PDA, smartphone, ...
- Chargement de modules dynamiques : optimisation taille noyau ;
- Migration aisée de LINUX à LINUX Embarqué.

Historique de LINUX

- 1991 : la première version de LINUX est développé par Linus Torvalds;
- 1999 : Présentation à LINUX World d'un système compact PCI sous LINUX ;
- 2000 : Embedded LINUX Consortium : centralise des solutions LINUX embarqués ;
- Aujourd'hui : UBUNTU, Debian, Fedora, ...
- Raspbian : Issue de Debian

RTOS basés sur LINUX

Monde Libre :

- RTLINUX,
- RTAI.

Solutions propriétaires:

- VxWorks,
- pSOS,
- QNX.

Processus UNIX

Un processus UNIX est une implémentation en langage *C* de la notion de tâche. Un processus est une entité possédant :
Un contexte d'exécution :

- Un ensemble sauvegardé de valeurs des registres du processeur ;
- une zone mémoire associé au code exécutable ;
- une zone de mémoire dynamique privé ;
- une zone pour la pile.

Un environnement d'exécution :

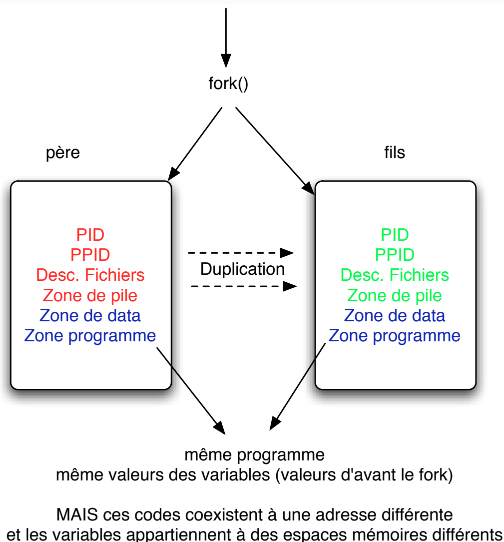
- Les fichiers ouverts,
- sa priorité d'exécution,
- un processus est identifié par un numéro : PID.
- ...

Création d'un processus UNIX : fork()

La primitive fork() fait partie du paquetage standard "stdio.h"

- Duplication quasi-complète du patrimoine du père ;
- Le "fils" se distingue de son père par :
  - Valeur retournée par fork() :
    - => 0 : chez le fils
    - => PID : chez le père (PID du fils)
  - son PID ;
  - son PPID ;
  - sa zone programme, data et pile ;
  - ses propres descripteurs de fichiers ouverts.

fork()



exemple

```
main() {
 int i,PID,res,CR;

 if ((PID=fork())==0){ /* Chez le fils */
   for(i=0;i<iteration;i++)
        printf("Fils, iter num %d \n",i);
   printf("je suis le fils, j'ai fini\n");
   exit(1);
 }
 else { /* PID != 0 : chez le pere */
     for(i=0;i<iteration;i++)
          printf("\t\t Pere, iter num %d \n",i);
     res=wait(&CR);
     printf("\t fils est termine, res= %d \n",res);
  }
}
```

exemple

Voir le code : processus.c

Terminaison d'un fils

Exit, Wait

Un processus fils se termine par la primitive exit qui permet de renvoyer un entier au père. Ce père récupère cet entier par l'appel à la procédure *wait* qui est un appel système bloquant qui renvoie le pid du fils qui à terminé.

TD1

Dans le répertoire TDSETR, ouvrir le squellete, éditer et exéuter le programme précédent.

TD2

- A partir du squellete suivant, réaliser trois processus :
- Fils1 : Affiche la date et l'heure toute les secondes, 1000 fois.
- Fils2 : Affiche son pid toutes les 200 millisecondes 2000 fois.
- Père : Attends la terminaison et affiche alors, le pid du fils qui se termine

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

extern int fork(); extern int getpid();
extern void usleep(int);extern void sleep(int);

int main(void){
  time_t date;
  time(&date);
  printf("Date et heure : %s", ctime(&date));
  sleep(1); // repos de 1 seconde
  usleep(200000); // repos de 200 millisecondes
  return 0;}
```

Boucle de création d'un ensemble de processus : entêtes

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#define repos 10000
#define iteration 20

extern void exit(int);
extern int fork();
extern int getpid();
extern void  usleep();
extern int wait(int *);
```

Boucle de création d'un ensemble de processus d'impression

```
void imprime(int i){
  printf("\t\t\t Je m'appelle %d mon travail suivant :",getpid());
  for(int j=0;j<i+5;j++) printf(".");
  printf("\n");usleep(repos);
}
```

main : Création des fils

```
int main() {
    int i,tab[nbfils],CR,PID,res;

    printf("\n\n\nJe suis le pere");
    printf(" je m'appelle %d\n",getpid());

    for(i=0;i<nbfils;i++) {
      if ((tab[i]=fork())==0) {imprime(i);exit(i);}
      else {printf("Pere : Je viens de creer %d \n",tab[i])
    }
...}
```

main : Attente fin des fils

```
  ...
  printf(" Pere : J'attends la terminaison de mes fils \n");
  for (int j=0;j<nbfils;j++) {
        res=wait(&CR);
        printf("\n\t\t Fils %d fini, il s'appelait %d ",CR>>8,res);
  }
}
```

TD3

Editer et exéuter le programme précédent.

Définition incorrecte d'un ensemble de processus

```
int main() {
      int i,CR,PID,res,nbfilscree=0;
      int tab[nbfils];
      printf("\n\n\nJe suis le pere et je m'appelle %d\r
      for(i=0;i<5;i++) {tab[i]=fork();}
      if (tab[i]==0) {printf("fils");imprime(i);exit(i)
      else printf("\n pere");
      printf("Je suis le pere j'ai fini, j'attends fin c
      for (int j=0;j<10;j++) {
            res=wait(&CR);
            printf("\n \t  Fils %d terminé, il renvoie %
      }
}
```

- Ce code produit 31 processus : Le pere produit 5 fils.
- Chaque fils en fait de même !!!!!
- Ecriture correcte : if (tab[i]=fork()==0) Code des fils!!!

Gestion de processus et mesure des temps d'éxécution

- ls : Liste les fichiers du répertoite
- cd Documents : Change de répertoire
- gcc fils.c -o file : compiler et produire un exécutable
- chmod 777 file : droit exécution ugo = rwx rwx rwx
- ./file : lancer ou
- ./file & ; lancer en tâche fille
- ps -aL; voir les processus
- kill -9 3214; tuer un processus
- time ./file : donne temps d'exécution

TD4

Ouvrir un shell. Tester les commandes précédentes avec le code c qui contient la boucle de création de process.

Commande Time

Time renvoie :le temps total écoulé, le temps du programme utilisateur, le temps système :

- elapsed time : temps total réel de l'exécution de la commande
- user time : temps CPU uniquement utilisateur
- sys time : temps utilisé par le système pour gérer l'exécution de tous les jobs

Elapsed time = Temps du programme utilisateur + temps système + temps utilisé par les autres tâches

La communication entre processus UNIX

- Pipes : Processus locaux
- Les sémaphores
- Socket : Processus distants

les pipes

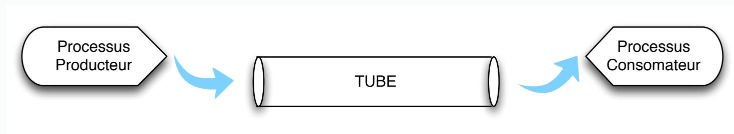


Figure: Information circulant dans un pipe

- Transport information unidirectionnel
- FIFO
- Producteur - Consommateur
- Reprogrammer le pipe par le producteur et le consommateur, pour fonctionner dans l'autre sens.

les pipes nommés

Si un pipe est nommé : il peut être partagé entre un père et ses fils. Soit p le nom du pipe :

- p[0] : sortie du pipe, c'est par la que l'on retire de l'information
- p[1] : entrée du pipe c'est par la que l'on dépose de l'information

Le producteur s'occupe de $p[1]$, tandis que le consommateur ne s'occupe que de $p[0]$

Exemple : Déclaration

Le main qui déclare le pipe

```
int main()  {
   int fils1, fils2, m, n, p[2], count;
   char lu[5];

   pipe(p);
```

Producteur

Premier fork() : Le fils producteur :

```
if ((fils1=fork())== 0){
       char c;
       close (p[0]);
       fprintf(stdout,"(%d) est le Producteur: \n",getpid());
       do {
              scanf("%c",&c);
             write(p[1],&c,1);
       } while (c!='T');
       close(p[1]);
       fprintf(stdout,"fin production \n");
       exit(2);
    }
```

Consommateur

Second fork() : Le fils consommateur

```
else if ((fils2=fork())==0){
     fprintf(stdout,"fils2 (%d) : lit ds pipe\n",getpid());
     close (p[1]);
     do{
       count = read(p[0],lu,1);
       printf("\t \t \t \t LU : %c\n",*lu);
     }while (*lu != 'T');
     fprintf(stdout,"fin de consommation \n");
     exit(3);
 }
```

Exemple de pipe nommé

Le père qui ferme le pipe et attend la terminaison de ses fils

```
else{
     fprintf(stdout,"processus pere %d\n",getpid());
     close(p[0]);
     close(p[1]);
     fprintf(stdout, "wait fils : %d\n",wait(&m));
     fprintf(stdout, "wait fils : %d\n",wait(&n));
     fprintf(stdout, "fin du pere\n");
     exit(0);
  }
```

TD5

Edition, compilation et exécution de pipe.c

Sémaphore

Definition (Semaphore)

Un sémaphore est un object permettant à N processus ou thread (tâche) d'accéder à une même région de code ou a une ressource critique.

Pour cela un sémaphore est un objet qui permet les actions suivantes :

- L'initialisation
- Le processus de réservation : $semP$
- Le processus de libération : $semV$

SemP

```
Masquer toutes les ITs;
S--;
if (S<0) {
Tache_courante --> File_Attente(S);
reordonnancement();
}
Demasquer toutes les ITs;
```

SemV

```
Masquer toutes les ITs;
S++;
if (S<=0) {
  Debloquer(Premier(File_Attente(S)));
  reordonnancement();
}
Demasquer toutes les ITs;
```

Exemple : les entêtes

Exemple deux imprimeurs veulent accéder à une unique imprimante.
L'imprimante est représentée par un affichage sur une ligne d'écran.
Entêtes :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "semaphore.h"
#define NB_IMPRIMEURS 40
#define NB 200

extern void exit(int);
extern void usleep();
extern int  getpid();
extern int fork();
extern void sleep();
extern int wait();
```

Exemple : la fonction imprime

Fonction imprime():

```
void imprime() {
  sem_P(ecran);
  for (int i=0;i<NB;i++)
    printf("Je suis suis %d, Je realise une impression  %d\n",getpi
  sem_V(ecran);
}
```

Exemple : le main

Main():

```
int main(){
  int i,tab[NB_IMPRIMEURS];
  sem_init(); //initialisation de(s) semaphore(s)
  ecran=sem_creer(1);
  printf(" Creation des imprimeurs \n");
  for(i=0;i<NB_IMPRIMEURS;i++) {
    tab[i]=fork();
    if (tab[i]==0) {imprime();exit(1);}
  }
  sleep(1);
  for(i=0;i<NB_IMPRIMEURS;i++) wait(NULL);
  sem_detruire();
  return(1);
}
```

Les imprimeurs

Compilation et exécution de procsem.c avec et sans sémaphore.

Les Threads Posix

Posix

POSIX définit une norme : IEEE 1003. Cette norme porte sur les interfaces de communication des logiciels : les API.

API

Une API : Application Programming Interface permet de définir la manière dont on peux utilser un logiciel pu composant informatique. C'est donc un ensemble de fonctions typées (décrites dans un *.h) qui indique comment on peut interagir avec le composant.

Les Threads Posix

Les threads

Les threads qui implémentent la norme POSIX sont des processus légers car ils ont été créés par un processus père avec qui ils partagent avec lui le même espace mémoire.

Ils ont les caractéristiques suivantes :

- Ils peuvent avoir des variables communes (globales) et des variables locales ;
- Ils partagent les mêmes fichiers et les mêmes ressources ;
- Ils ont leur propre pile d'exécution ;
- Ils sont ordonnancés ;

exemple

Voir le code : API "pthread.h"

Création d'un thread

```
int        pthread_create(pthread_t *thread,
                          const pthread_attr_t *attr,
                          void *(*start_routine)(void *),
                          void *arg);
```

- thread : identifiant, descripteur du thread ;
- attr : attribut de création ;
- start_routine : fonction constituant le corps du thread ;
- arg : argument de la fonction.

Teminaison et synchronisation d'un thread

```
void      pthread_exit(void *value_ptr) ;

int       pthread_join(pthread_t thread,
       void **value_ptr) ;
```

value_ptr : valeur de retour au père. thread : identifiant

exemple

Voir le code : thread2.c

Accès aux variables globales partagées

Deux threads qui partagent et modifient une même variable globale peuvent provoquer un comportement non déterministe dse l'application.

```
#include <pthread.h>
#define NBITER 20000
int main(){
  int res; pthread_t l1, l2;
  res=pthread_create(&l1, NULL, ligne1, NULL);
  res=pthread_create(&l2, NULL, ligne2, NULL);
  pthread_join(l1,NULL);
  pthread_join(l2,NULL);
  printf("Fin pere, valeur finale de la variable globale = %d\n",glo
  exit(EXIT_SUCCESS);
}
void *ligne1(void *arg) {
  for(i=0;i<NBITER;i++) {globale=globale+1;}
  pthread_exit(NULL); }
void *ligne2(void *arg) {
  for(i=0;i<NBITER;i++)  {globale=globale*10;}
  pthread_exit(NULL); }
```

Accès multiple à une variable globale

Compilation et exécution de globale.c.

Solution

On crée deux sémaphores pour alterner les deux threads de façon déterministe Entêtes

```
#include <pthread.h>
#include "semaphore.h"
#define NBITER 11000

unsigned int globale=0;
int ping, pong;

void *ligne1(void *arg);
void *ligne2(void *arg);
```

Solution

Création et initialisation des sémaphores. Création des threads.

```
int main(){
  int res;
  pthread_t l1, l2;
  char c;

  sem_init();

  ping=sem_creer(1);
  pong=sem_creer(0);

  res=pthread_create(&l1, NULL, ligne1, NULL);
  if (res == -1) {perror("Err. creation thread\n");exit (-1);}
  res=pthread_create(&l2, NULL, ligne2, NULL);
  if (res == -1) {perror("Err. creation thread\n");exit (-1);}
```

Solution

Les codes des threads :

```
void *ligne1(void *arg) {
  for(int i=0;i<NBITER;i++) {
    sem_P(ping);
     globale=globale+1;printf("\n ping -");
    sem_V(pong);
  }
  pthread_exit(NULL);
}
void *ligne2(void *arg) {
  for(int i=0;i<NBITER;i++) {
    sem_P(pong);
     globale=globale+10;printf(" pong");
    sem_V(ping);
  }
  pthread_exit(NULL);
}
```

Solution

Le père attend la terminaison des fils, il détruit les séamphores et se termine proprement.

```
  ...
  printf("attente des fils\n");
  pthread_join(l1,NULL);
  pthread_join(l2,NULL);
  sem_detruire();
  printf("Fin du père, variable globale = %d\n",globale);
  exit(EXIT_SUCCESS);
}
```

Accés controlé à une variable globale partagée par des threads

Compiler et exécuter globalesem.c

Tableau de threads et passage de paramètres

Boucle de création de thread. On va faire un travail à chaque thread qui est différent grâce à la variable i. On utilise un sémaphore pour que chaque thread écrive sur une seule et même ligne.

```
int main(){
  int i;
  pthread_t pt[N];
  sem_init();
  ligne=sem_creer(1);

  for(i=0;i<N;i++){
    pthread_create(&pt[i],NULL,imprime,(void *)(size_t) i);
  }

  for(i=N;i>0;i--) pthread_join(pt[i],NULL);
  sem_detruire();
  return 0;
}
```

Tableaux de threads et passage de paramètres

Entêtes et la fonction imprime qui a appelée par les N threads

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "semaphore.h"
#define N 5  // nombre de threads
int ligne;

void *imprime(void *k){
  int p=(int) k+5,j=0;
  sem_P(ligne);
  for(j=1;j<=p;j++) printf("%d ",j);
  sem_V(ligne);
  printf("\n");
  pthread_exit()
}
```

Passage de paramètres à un main (1/2)

On peut passer des paramêtres à un main par deux variables :

```
int main(int argc, char *param[]) {}
```

- argc compte le nombre de paramètre passé
- param[] : Tableau de chaines de caractères
- param[0] : Nom du programme auquel est attaché le main
- param[1] : Chaine qui représente le premier paramêtre
  Exemple si ce paramètre vaut 12 alors param[1] vaut "12". Il faut utiliser alors la fonction atoi() ou atof() pour la récupérer.
- param[2] : Chaine qui représente le second paramêtre
- ...

Passage de paramètres à un main (2/2)

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

extern int atoi();
extern double atof();

int main(int argc, char *param[]){
  int a= atoi(param[1]);
  double f=atof(param[2]);

  printf("\n Le nom de votre executable est  %s ",param[0]);
  printf("\n Vous avez entré %d parametres \n",argc);
  printf("\n Arguments de la commande :  %d, %f\n", a, f);
  return 0;
}
```

confère paramMain.c

Passage de type structuré à un thread

Un type structuré agrège plusieurs types en un seul. Il y a trois façon de déclarer un type structuré :

- Mode simple

```
struct point_t {
  int a; int b;
};
struct point p1, p2;
p1.a =2 ; p1.b =3 ; p2.a = 43 ; p2.b =27;
```

- Mode avec définition de type et champ suivant :

```
typedef struct point {
  int a ; int b ;
  struct point *suivant;
} point_t;
 point_t p1,p2;
 p1.a = 2 ;   p1.b = 3 ;  p1.suivant = &p2;
 p2.a = 43 ;  p2.b = 27 ; p2.suivant = NULL;
```

Passage de type structuré à un thread

Mode avec définition de type simple.

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

typedef struct{
  int a;
  int b;
} point_t;
```

C'est le mode que l'on va utiliser dans le paramètrage d'un thread dans l'example suivant.

Passage de type structuré à un thread

Voici un main simplifié à l'extrême qui crée 2 thread fils et qui leur passent les objects structurés p1 et p2 :

```
int main() {
      point_t p1,p2;
      p1.a = 2 ; p1.b =3;
      p2.a = 43 ; p2.b =27;

      pthread_t T1, T2;
      pthread_create(&T1, NULL, traite, (void *)(size_t) &p1);
      pthread_create(&T2, NULL, traite, (void *)(size_t) &p2);
      pthread_join(T1, NULL);
      pthread_join(T2, NULL);
      return 0;
}
```

- (size_t) permet d'évaluer le nombre d'octet d'un système
- (void *) transtype alors l'objet en pointeur vers "tout objet" qui est un type abstrait.

Passage de type structuré à un thread

```
void *traite(void *parametre){
  point_t *p=(point_t *) parametre;
  printf("\n a = %d, b = %d\n",p->a, p->b);
  pthread_exit(0);
}
```

- traite : Fonction qui est associée à chaque thread.
- *k* est le paramètre d'appel qui est lié au paramètre formel *parametre*
- parametre : (void *) est un pointeur sur "tout objet". C'est lui qui a reçu le type structuré.
- parametre est alors "re-transtypé", en sens inverse, c'est à dire vers un type pointeur sur *point_t* pour être enfin utilisé.

Passage de type structuré à un thread

Voici une seconde version avec Tableau de threads :

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#define N 10
extern void usleep();
typedef struct{
  int a; int b; int indice;
} point_t;
void *traite(void *parametre){
  point_t *p=(point_t *) parametre;
  printf("\n Tread %d",p->indice);
  printf(" a = %d, b = %d",p->a, p->b);
  pthread_exit(0);
}
```

- On rajoute une indice dans le type structuré
- *N* représente le nb de threads crées.

Passage de type structuré à un thread

```
int main() {
  point_t TabP[N];
  for(int i=0;i<N;i++) {
    TabP[i].a = 3*i+15;
    TabP[i].b = 7*i+3;
    TabP[i].indice = i;
  }

  pthread_t Tp[N];
  for(int i=0;i<N;i++) {
    pthread_create(&Tp[i],NULL,traite,(void *)(size_t) &TabP[i]);
  }
  for(int i=0;i<N;i++) {
    pthread_join(Tp[i], NULL);
  }
  printf("\n\n Pere : Tous mes fils ont fini \n");
  return 0;
}
```

1 Processus unix : Fork()

1.1 Un fils

Dans le répertoire TDSETR dupliquer le squelette sous un autre nom, l'éditer sous Codebloc. Récupérer sur Madoc "unfils.c" et copier ce code dans le nouveau projet ouvert, puis l'exécuter, observer les alternances père-fils.

- Remplacer le delai du fils par un nice(), observer les changements sur l'exécution.
- Remplacer le nice() du père par un usleep(repos) , puis modifier le repos pour avoir une meilleur alternance.

1.2 Deux fils

Dans le répertoire TDSETR dupliquer le squelette sous un autre nom, l'éditer sous Codebloc. Récupérer sur Madoc "gestime.c" et toujours à partir du nouveau projet ouvert, réaliser trois processus :

- Fils1 : Affiche la date et l'heure toutes les secondes, 1000 fois.
- Fils2 : Affiche son pid toutes les 200 millisecondes 2000 fois.
- Père : Attends la terminaison et affiche alors, le pid du fils qui se termine

Dans le répertoire TDSETR dupliquer le squelette sous un autre nom, l'éditer sous Codebloc. Récupérer sur Madoc "bouclefilsbug.c" et copier ce code dans le nouveau projet ouvert, puis l'exécuter, observer l'exécution, analyser ce qui se passe.

2 Les pipes

Dans le répertoire TDSETR dupliquer le squelette sous un autre nom, l'éditer sous Codebloc.

- Récupérer sur Madoc "pipe.c" et copier ce code dans le nouveau projet ouvert, puis l'exécuter en jouant le rôle du producteur.
- Rajouter un pipe pour que les deux fils communiquent dans les 2 sens.

Le but de ce TP sera l'implémentation d'une application temps réel embarquée multi-threadée commandant un moteur pas à pas sur une carte Raspberry.

1 Préambule

a A partir du fichier "squeletteTPRaspberry.c"

b Brancher les deux cartes de connexion permettant l'accés aux ports de la raspberry.

c Brancher les emplacements de la carte d'extention JB_1 sur la bobine 0, JB_2 sur la bobine 1, JB_3 sur la bobine 2 et JB_4 sur la bobine 3 de la maquette du MPP.

d Connecter l'alimentation externe ($5V$) et sa masse sur les entrées d'alimentation de la carte moteur. Ne PAS connecter l'alimentation externe au Vcc de la carte d'extention !

e Connectez la masse de la carte moteur et la masse Gnd (du haut) de la carte d'extention.

f Ouvrir un terminal sur la Raspberry et lancer code blocks avec un *sudo* pour être autorisé à écrire sur le port $GPIO$

g Taper : $ sudo codeblocks

2 Enoncé

1. Sans modifier le code des fonctions *init*() et *ecrit*(), écrire un sous-programme permettant de faire tourner le moteur pas à pas de n tour(s) dans le sens s et avec la vitesse v.
2. Écrire une fonction *menu*() qui interroge l'utilisateur sur le sens, et la Vitesse (tant que la touche 'q' pour quitter n'a pas été choisie).

   ```
   - Tapez '0' pour obtenir un sens trigo ou '1' pour un sens horaire
   - Entrez un entier [0,100] pour définir une image de la vitesse.
   - Tapez 'q' pour Quitter.
   ```

   Ce menu initialisera les variables globales S, V et Q.
3. En vous inspirant des fonctions pere et fils (un seul fils) présentes dans "squeletteTP1.c", créer une application multithread :
   - Le main (père) lance le menu précédent, puis il crée un fils qui initie la rotation du moteur selon les variables globales S et V précédement initialisées.
   - Puis le père propose alors (en mode boucle infinie), un menu de gestion de la vitesse du *mpp* grâce aux interactions '+' et '-' (variation de la vitesse) ou 'q' (pour quitter) avec l'utilisateur.
   - Le père contrôle uniquement la vitesse, tandis que le fils agit sur la commande du moteur. Pendant la rotation du moteur gérée par le fils, le père peut accélérer la rotation du moteur en tapant par '+' et '-' pour décélérer ou bien 'q' pour arrêter.
   - Le fils, lui, tourne désormais en boucle infinie, jusqu'à la frappe de la touche 'q' que le père transmet au fils par la variable globale Q. Si le fils détecte que Q a été validé alors il s'arrête (*pthread_exit*). Le père fait un *pthread_join* et se termine alors aussi.

3 Conclusion

Trouver des exemples applicatifs pour l'usage multi-tâches avec l'usage de processus créer par l'instruction fork() d'une part, et d'autres part, avec des threads.

Compte rendu

De plus, dans votre code, pour chaque sous-programme vous donnerez les informations en commentaires suivant :

```
/*---------------------------------------------------------------
Fonction : nom de la fonction et des paramètres avec leurs types ;
Entrées : Paramètres en entrée de la fonction ;
Sorties : paramètres modifiés ou de sortie ;
Variables globales utilisées ;
Sous-programmes appelés ;
Fonctionnement : description rapide/sommaire du fonctionnement.
----------------------------------------------------------------*/
```

- Ce qui est intéressant :
  - Limiter le code dans les explications en se concentrant sur quelques points clefs
  - Résultats d'exécution et analyse événtuelle des résultats
  - Copie d'écran, photos (gifs) des montages, de l'oscilloscope.
  - Videos accessibles sur un drive
  - Code complet en annexe.
- Ce qui N'EST PAS intéressant :
  - La reprise de l'énoncé
  - Copié collé internet

Commande et Calcul massivement Parallèle

D. Delfieu - Polytech'Nantes - GE_4

Le but est d'insérer des threads de calcul à une commande de moteurs.

La conjecture [1] de Syracuse

Lothar Collatz (1910-1990) était un mathématicien allemand, en 1937, il énonce une conjecture (dite de Collatz) connue également sous le nom de conjecture de Syracuse :

La suite de Syracuse de n'importe quel entier strictement positif atteint 1

La suite dite de Syracuse se définit de la façon suivante : $\begin{cases} U_{n+1} = \frac{U_n}{2} & \textbf{Si } U_n \textbf{ est pair} \\ U_{n+1} = 3\,U_n + 1 & \textbf{Si } U_n \textbf{ est impair} \\ U_0 = N \end{cases}$

Exemple : Syracuse(15) = 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1

Tuer une tâche

Si votre programme met trop de temps à se terminer et que vous souhaitez le terminer plus tôt. Vous pouvez soit l'arrêter en tapant au clavier dans l'invite de commande lancé par code blocks $CTRL + C$ ou si cela ne marche pas immédiatement, suivre ces trois étapes.

1. Dans une invite de commande tapez top (ou ps -ac). Vous devez voir une fenêtre comme ci-dessous.
2. Sur cette dernière, cherchez votre exécutable, dans notre cas TP2, et repérez le numéro PID qui lui a été attribué (1742 ici).
3. Dans une nouvelle invite de commande, tapez sudo kill 1742 (PID de la tâche à supprimer).
4. Votre programme s'arrête et vous pouvez reprendre.
5. Une méthode plus rapide mais qui ne vous permet pas de voir la saturation du CPU est de taper dans une invite de commande sudo pkill TP2 (le nom de votre exécutable).

Attention à ne pas arrêter un autre processus. Par exemple, Xorg qui prend une grande partie du CPU, va arrêter toute la carte Raspberry.

1 Questions

1.1 Comprendre

1. Calculer la suite de Syracuse pour U_0=17.
2. Coder la fonction syracuse_rec à l'aide de l'algorithme récursif suivant qui calcule la suite pour Un :

[1] Une conjoncture est une proposition qui n'a été ni démontrée ni réfutée. Un théoréme est une conjoncture qui a été démontrée.

Syracuse(Un) **begin**
  Afficher U_n ;
  si $U_n == 1$ **alors**
    retourner 1
  fin
  si (U_n *est impair)* **alors**
    retourner $Syracuse(3 * U_n + 1)$
  fin
  sinon
    retourner $Syracuse(U_n/2)$
  fin
end

1.2 Passage de paramêtres

1. Passage de paramètres à un main :
   On va maintenant produire un exécutable en dehors de codeblock. La commande de compilation est la suivante :

   ```
   $sudo gcc tp2.c l wiringPi -l prthread -o syracuse_Rec
   ```

   On veut en effet passer un paramètre à notre programme. On peut en effet passer un ou plusieurs paramètres à un éxécutable. Pour cela, dans les parenthèse d'un *main*, on peut faire apparaître, le nombre de paramêtres *nbparametre* et un tableau de paramêtres *parametre* :

   ```
   int main(int nbparametre, char * parametre[]) {
     syracuse((int) parametre[1]);  /* forçage au type int$/
   }
   ```

2. Coder en C la fonction $NSyracuses_Rec$ qui calcule toutes les suites de Syracuse de 1 à un nombre n, qui est passé en paramètre au main. Créer alors l'exécutable $Nsyracuse_Rec$ et relever, à l'aide de la commande time, les temps d'exécution pour de grands nombres (10000) :

   ```
   #time sudo ./syracuse_Rec 10000
   ```

2 Parallélisation du calcul des suites de Syracuse

On veut paralléliser le calcul des suites de tous les entiers compris entre 2 à $nbSuitesTotal$ ou $nbSuitesTotal$ est un grand nombre saisi par l'utilisateur en ligne de commande.

On pourra alors calculer $nbSuitesparThread$ suites de Syracuse en partageant ce calcul entre plusieurs threads à l'aide de $nbThread$.

- $nbSuitesTotal$ est récupéré au sein des paramètres du main (son premier paramètre).
- $nbThread$ est lui définie comme une étiquette à l'aide d'un $\#define$.

Modifier la fonction *pSyracuses* de façon à ce que en fonction d'un indice i, elle calcule :

```
Pour i = 0 elle calcule de bi = 2 à bs = bi + nbSuitesparThread,
Pour i = 1 elle calcule de bi = nbSuitesparThread+1 à bs = bi + 2*nbSuitesparThread,
....
```

2.1 Création de multiples threads

Compléter la fonction lesthreads() qui fait la création de nbThread threads. Ces threads vont produire chacun nbSuites suites. On utilisera donc la boucle de création de threads suivante :

```
#define N 200   // definition statique du nombre de threads

pthread_t ???;

for ( i =0; i < N ; i ++)
   pthread_create (???, NULL, pSyracuses, ( void *) & i );
```

Comparer les temps d'exécutions entre le calcul parallèle et non parallèle.

3 Mise en parallèle de commande moteur et d'un calcul de suites de Syracuse

3.1 Machine pas à pas

Insérer, sous la forme d'un thread, la commande du moteur pas à pas. Ce thread commande en pas entier, à vitesse rapide prédéterminé, dans un sens fixé, et en boucle infinie le moteur pas à pas.

3.2 Machine à courant continu

Tester le code suivant

```
#include <stdio.h>
#include <wiringPi.h>
#include <stdlib.h>

int main (void) {
  int var=2700;
  int MLI=2000;

  if(wiringPiSetup() ==-1) exit(1);
  pinMode(1,PWM_OUTPUT);   //JB9
  pwmSetMode(0);           //0:Fréquence fixe, 1:Fréquence variable
  pwmSetRange(var);        //Fréquence de la MLI
  pwmSetClock(2);          //Diviseur d'horloge (2 à 0x20)
  pwmWrite(1,MLI);       //Rapport cyclique : [0, var]
  return 0;
}
```

- Cabler JB_9 sur l'entrée numérique de la MCC et connecter les masses.
- Insérer, sous la forme d'un thread, la commande de la MCC sous forme de thread à votre application.

3.3 Parallélisation

Tester l'influence du calcul sur la commande des deux moteurs en fonction de l'augmentation de $nbThread$ et $nbSuites$.

4 Conclusion

En conclusion, chercher sur internet, une application de calcul scientifique qui peut être optimisée par un parallèlisation multi-thread.

Compte rendu

De plus, dans votre code, pour chaque sous-programme vous donnerez les informations en commentaires suivant :

```
/*----------------------------------------------------------------
Fonction : nom de la fonction et des paramètres avec leurs types ;
Entrées : Paramètres en entrée de la fonction ;
Sorties : paramètres modifiés ou de sortie ;
Variables globales utilisées ;
Sous-programmes appelés ;
Fonctionnement : description rapide/sommaire du fonctionnement.
---------------------------------------------------------------*/
```

- Ce qui est intéressant :
  - Résultats d'exécution et analyse des résultats
  - Copie d'écran, photos (gifs animés) des montages, de l'oscilloscope.
  - Limiter le code : se concentrer sur les points clefs du code : Le code complet doit être relégué en annexe.
- Ce qui N'EST PAS intéressant :
  - La reprise de l'énoncé
  - Copié collé internet