Systèmes temps réel

Dr ABDELLAOUI Ghouti

École Supérieur des Sciences Appliquées Tlemcen

22 octobre 2023

Plan du cour

Information sur le module

Charge horaire : 1,5h cours + 1,5h TD + 3h TP
Calcule moyenne :

$$Test = \frac{Test1 + Test2 + MiniProjet + assiduite}{4}$$

$$ControleContinue = \frac{Test + Moy(TP)}{2}$$

$$MoyenneGenerale = \frac{ControleContinue + Examen}{2}$$

Information sur les travaux pratiques

Système Hardware utilisé : Raspberry Pi 3B+
Système d'exploitation utilisé : Raspbian (Linux) + Xenomai (real time)
Langage de programmation utilisé : Langage C et ou C++

Introduction aux systèmes temps réel

Introduction

Figure – Bras robot automatique

Introduction

Figure – Application d'un bras robot automatique

Introduction

Figure – Exemple d'un système embarqué d'une voiture

Introduction

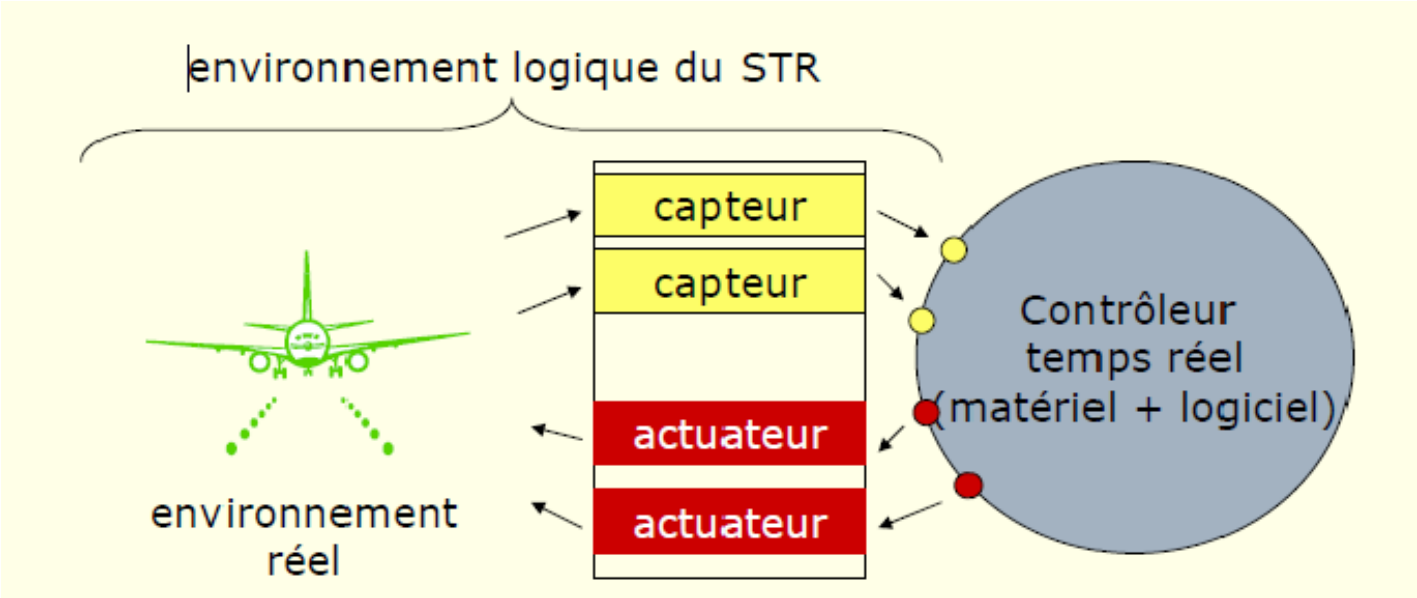


Figure – Exemple d'un système embarqué d'un avion

Introduction

Un logiciel "avionique" est :

- Embarqué
- Temps réel
- Critique

Définitions et paradigmes des systèmes temps réel

Définitions

- Temps réel signifie l'aptitude d'un système d'exploitation de fournir le niveau de service requis au bout d'un temps de réponse borné.
- Un système temps réel est un système informatique qui doit répondre à des stimuli fournis par un environnement externe afin de le contrôler.
- Un STR est un système dont la correction dépend non seulement de la justesse des calculs mais aussi du temps auquel est fourni la réponse

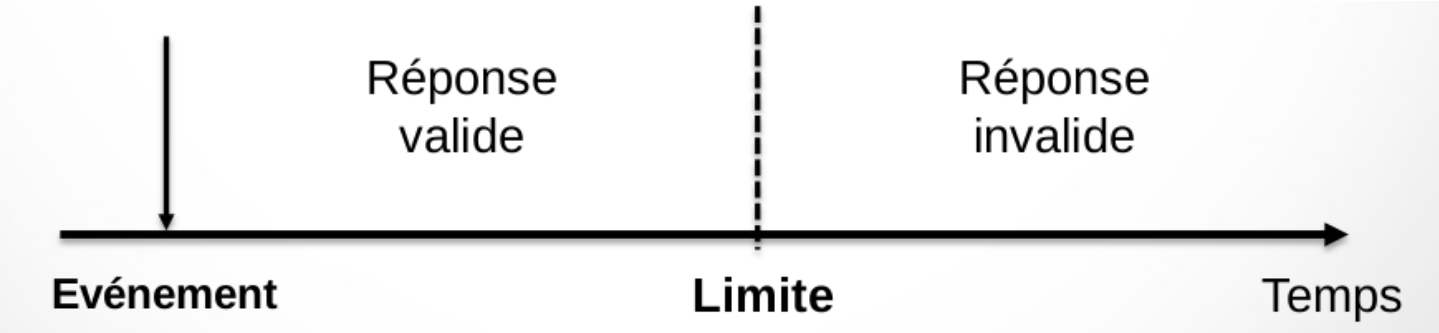


Figure – La correction d'un système temps réel

Caractéristiques des systèmes temps réel

- Large, complexe et fiable :
  - Un système temps réel interagit avec un environnement extérieur souvent complexe et en évolution,
  - Il doit garantir une fiabilité importante
- Utilisation du temps concret :
  - Au sein d'une application ou d'un système temps-réel il faut pouvoir manipuler le temps concret (horloge) pour limiter la **date début** et la **date fin**,
  - Il peut être nécessaire de pouvoir modifier ces paramètres en cours d'exécution et de pouvoir préciser les actions à prendre en cas de faute temporelle

Caractéristiques des systèmes temps réel

- Découpé en tâches ou en processus Concurrents :
  - Dans le monde réel les périphériques et l'environnement du système évoluent simultanément (en parallèle ou concurrence),
  - Le modèle utilisé en programmation des systèmes temps réel est un modèle basé sur la concurrence (applications concurrentes)
- Respect des échéances temporelles :
  - Vue la limitation des ressources et afin de respecter en permanence les échéances, il faut gérer efficacement la pénurie et tenter de favoriser les processus (Ordonnancement) dont l'avancement est le plus "urgent",

Programmation concurrente

Programmation concurrente

- La concurrence dans les L.P et le parallélisme au niveau Matériel sont deux concepts indépendants :
- Les opérations Matérielles se produisent en parallèle si elles se superposent dans le temps ;
- Les opérations dans le code d'un programme sont concurrentes si elles peuvent être, et pas nécessairement, exécutées en parallèles.

Exemple :

Non Concurrent	Concurrent
X = 5 Y = 2*X+3	x= A+B+C Y= 2+5*A

Processus

- Un processus comporte du code machine exécutable, une zone mémoire (données allouées par le processus), une pile ou *stack* (pour les variables locales des fonctions et la gestion des appels et retour des fonctions) et un tas ou heap pour les allocations dynamiques.
- Ce processus est une entité qui, de sa création à sa mort, est identifié par une valeur numérique : le PID (Process IDentifier).
- Tous les processus sont donc associés à une entrée dans la table des processus qui est interne au noyau.
- Chaque processus a un utilisateur propriétaire, qui est utilisé par le système pour déterminer ses permissions d'accès aux fichiers.
- ***Remarques*** *: Les commandes* ***ps*** *et* ***top*** *listent les processus sous UNIX/Linux et, sous Windows on utilisera le gestionnaire de tâches (****taskmgr.exe****).*

Processus : Contexte d'un processus

- Le contexte d'un processus (Process Control Block) est l'ensemble de :
  1. son état
  2. son mot d'état : en particulier la valeur des registres actifs et le compteur ordinal
  3. les valeurs des variables globales statiques ou dynamiques
  4. son entrée dans la table des processus
  5. les données privées du processus
  6. Les piles user et system
  7. les zones de code et de données.
- L'exécution d'un processus se fait dans son contexte.Quand il y a changement de processus courant, il y a **une commutation ou changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.
- En raison de ce contexte, on parle de processus lourd, en opposition aux processus légers que sont les threads.

Processus :Généalogie des processus

- La création d'un processus étant réalisée par **un appel système** (**fork** sous UNIX/Linux). Chaque processus est identifié par un numéro unique, le **PID** (Processus IDentification).
- Un processus est donc forcément créé par un autre processus (notion **père-fils**).Le **PPID** (Parent PID) d'un processus correspond au PID du processus qui l'a créé (son père)
- Exemple sous UNIX/Linux :

Processus :Généalogie des processus

Terminal

Fichier Édition Affichage Rechercher Terminal Aide

```
top - 22:41:27 up  2:45,  2 users,  load average: 1,08, 0,54, 0,35
Tâches: 198 total,   1 en cours, 197 en veille,   0 arrêté,   0 zombie
%Cpu(s):  4,4 ut,  1,4 sy,  0,0 ni, 83,1 id, 11,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   1871916 total,  1736176 used,   135740 free,    81284 buffers
KiB Swap:  4111352 total,   235988 used,  3875364 free.   584484 cached Mem

  PID UTIL.      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TEMPS+ COM.
 2369 ghouti     20   0 1896728 227044  37588 S   8,3 12,1   4:46.71 cinnamon
 2644 ghouti     20   0 3212680 144380  63396 S   8,3  7,7   4:33.54 chromium-b+
 2725 ghouti     20   0  929588 111012  59936 S   6,0  5,9   3:37.89 chromium-b+
 1464 root       20   0  652832 107940  90204 S   5,3  5,8   2:48.81 Xorg
 5294 ghouti     20   0  476356  26500  21384 S   1,0  1,4   0:00.19 gnome-scre+
    7 root       20   0       0      0      0 S   0,3  0,0   0:05.09 rcu_sched
 2150 ghouti     20   0  124924   3248   3092 S   0,3  0,2   0:02.14 at-spi2-re+
 4703 ghouti     20   0 1517724 182156  84680 S   0,3  9,7   0:34.37 chromium-b+
 5100 ghouti     20   0  626936  31116  25084 S   0,3  1,7   0:00.51 gnome-term+
 5124 ghouti     20   0   25084   3028   2468 R   0,3  0,2   0:00.48 top
    1 root       20   0   33900   3112   2092 S   0,0  0,2   0:00.84 init
    2 root       20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
    3 root       20   0       0      0      0 S   0,0  0,0   0:00.06 ksoftirqd/0
    5 root        0 -20       0      0      0 S   0,0  0,0   0:00.00 kworker/0:+
    8 root       20   0       0      0      0 S   0,0  0,0   0:00.00 rcu_bh
    9 root       rt   0       0      0      0 S   0,0  0,0   0:00.00 migration/0
   10 root       rt   0       0      0      0 S   0,0  0,0   0:00.02 watchdog/0
```

Figure – Exemple de la commande top sous linux

Processus :Généalogie des processus

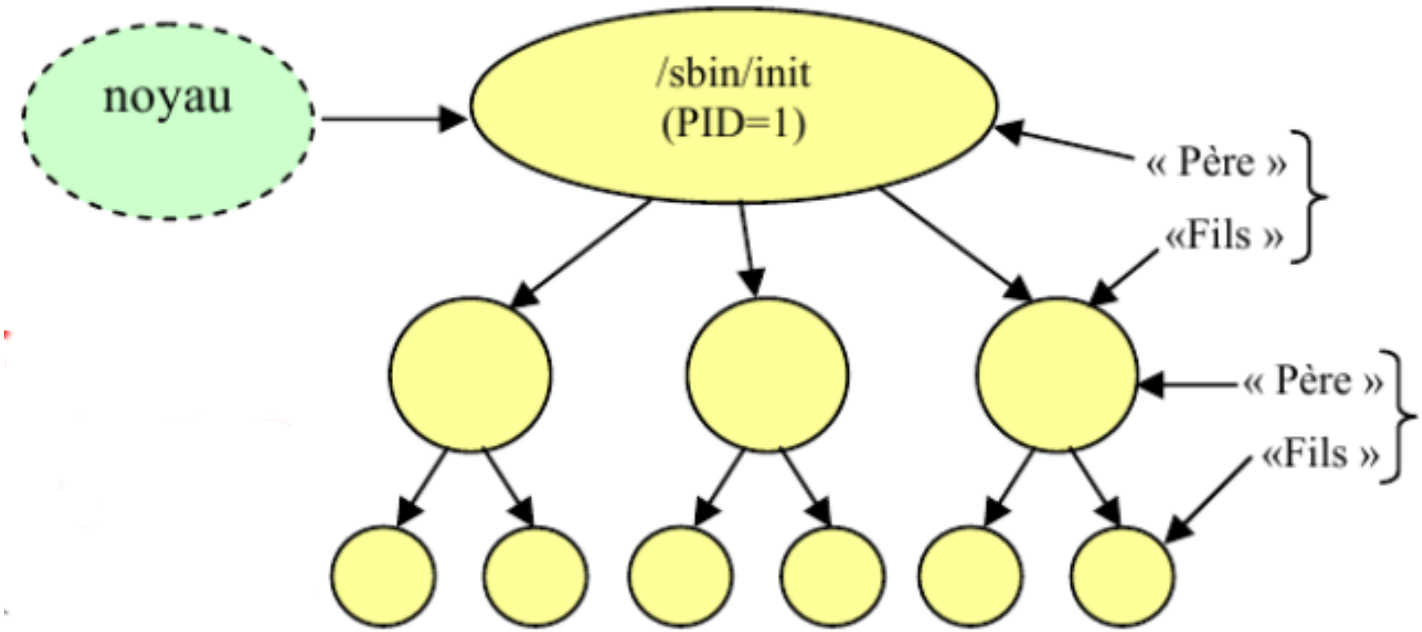


Figure – Arbre genéaologique des processus

Processus :Opérations réalisables sur un processus

1. Création : **fork**
2. Exécution : **exec**
3. Destruction :
   - terminaison normale
   - auto destruction exit
   - meurtre kill, **^C**
4. Mise en attente/réveil : **sleep, wait**
5. Suspension/reprise : **^Z/fg, bg**
6. Changement de priorité : **nice**

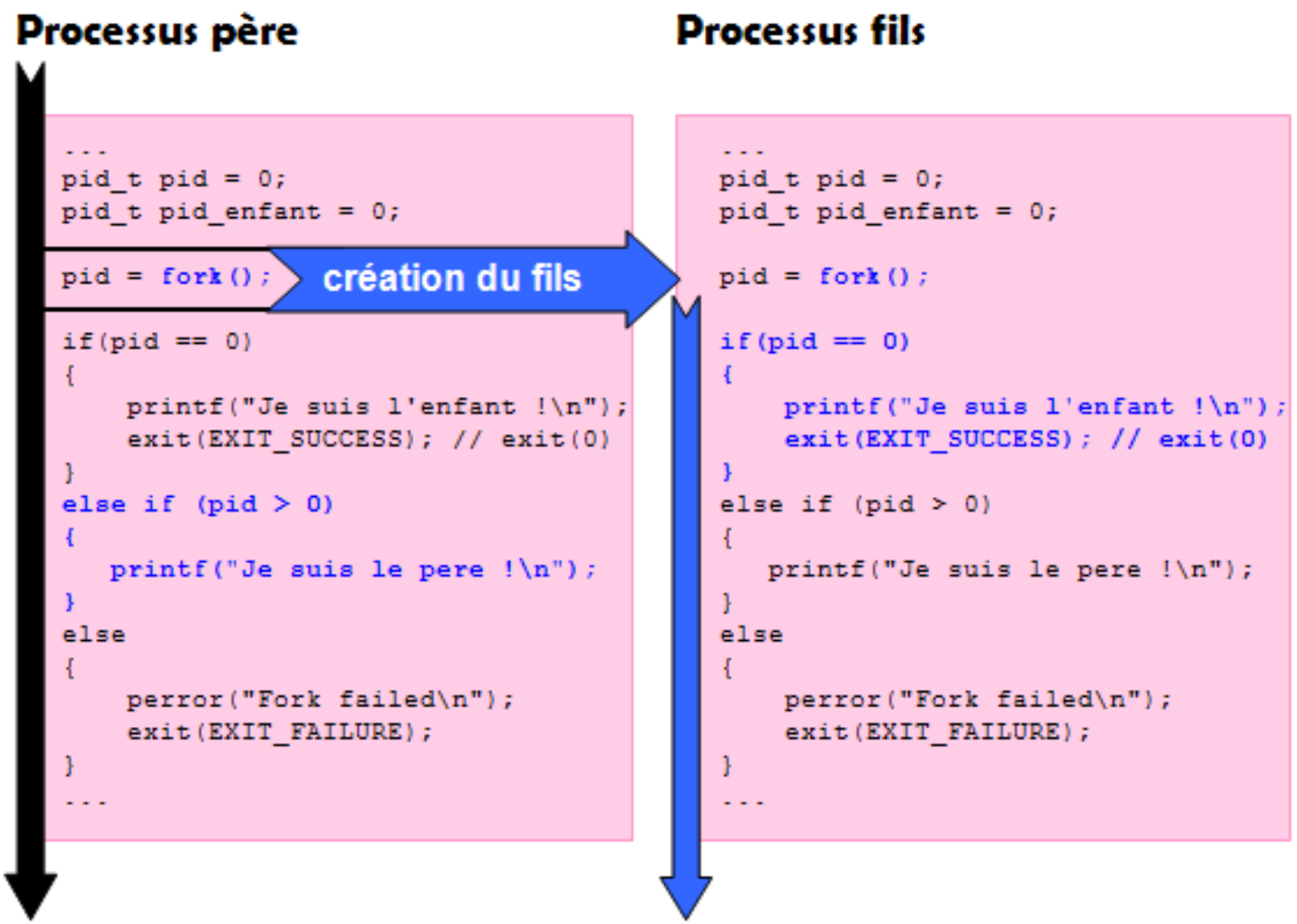
Processus :fork

- Lors d'une opération **fork**, le noyau Unix crée un nouveau processus qui est une copie conforme du processus père. Le code, les données et la pile sont copiés et tous les fichiers ouverts par le père sont ainsi hérités par le processus fils.

fork

```
pid_t fork(void);
```

Processus :fork



Processus :Exemple programme fork

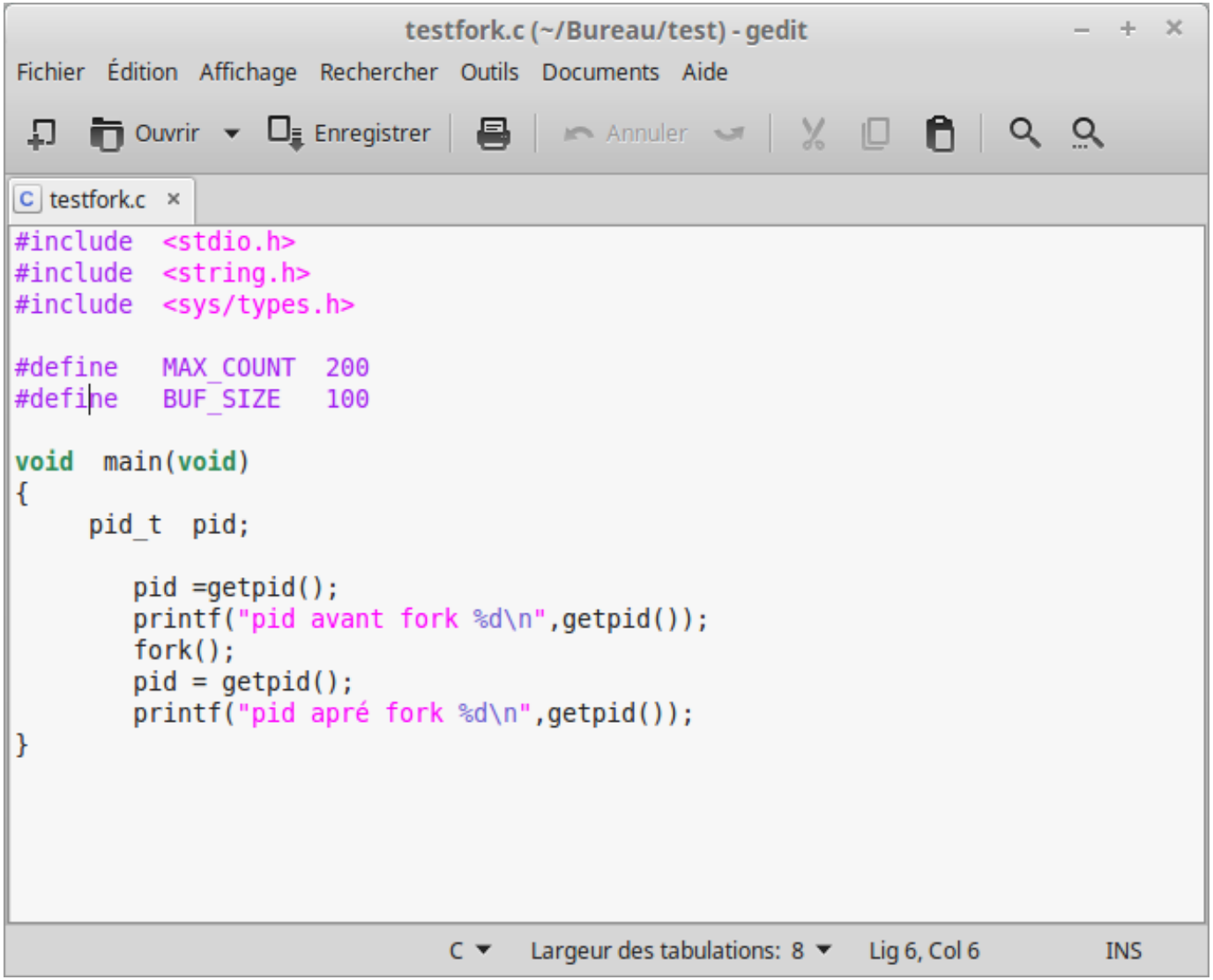


Figure – Exemple d'execution du fork

Processus :Exemple programme fork

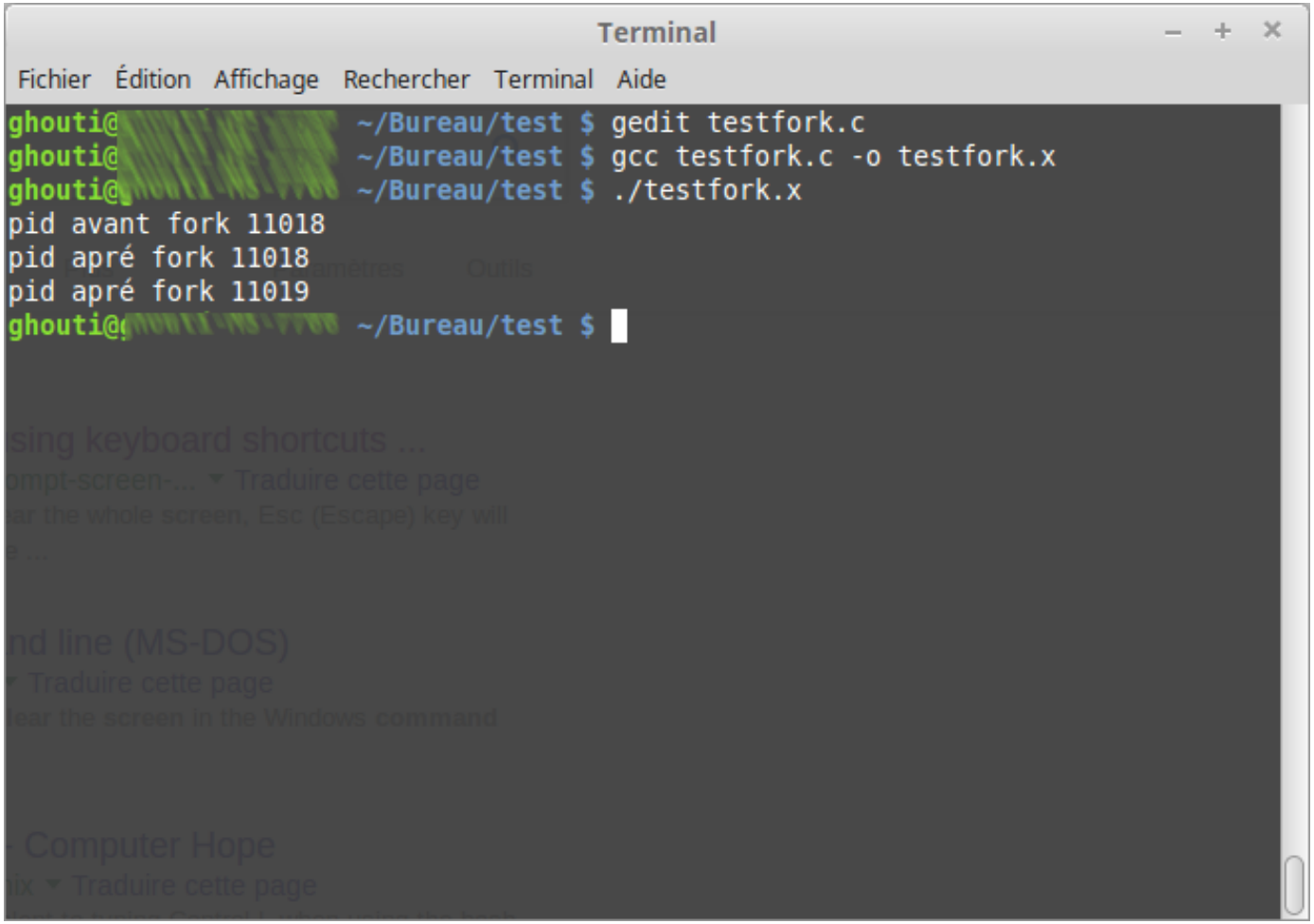


Figure – Exemple d'execution du fork

Processus :Exemple programme fork

La fonction **fork** retourne une valeur de type pid_t ;Il s'agit généralement d'un int ; il est déclaré dans <sys/types.h>.
La valeur renvoyée par **fork** est de :

- -1 si il y a eu une erreur ;
- 0 si on est dans le processus fils ;
- Le PID du fils si on est dans le processus père. Cela permet ainsi au père de connaître le PID de son fils.

Processus :résultats de la fonction fork()

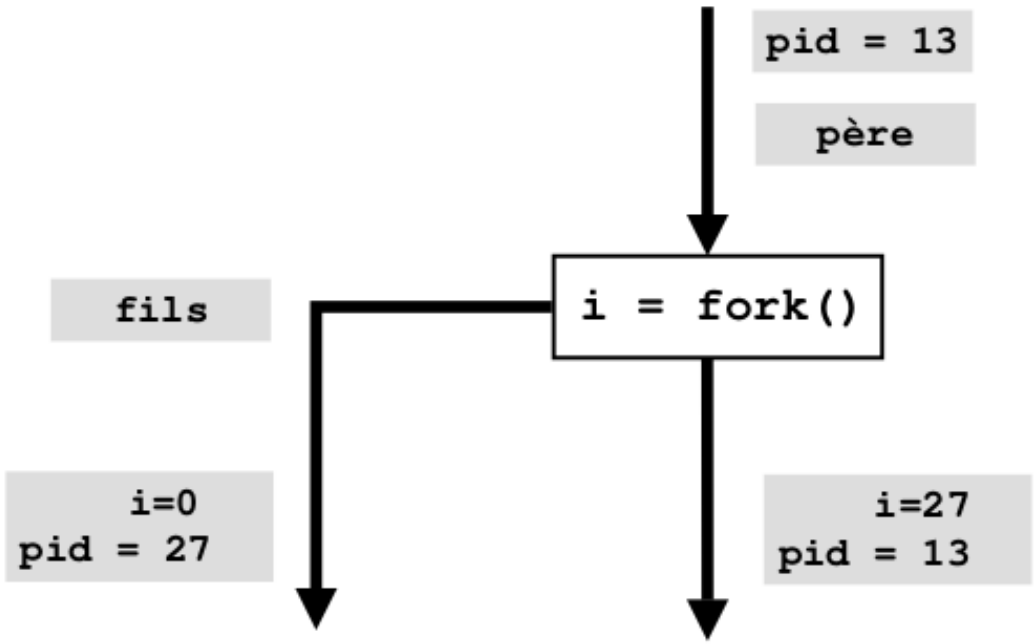


Figure – Exemple d'exécution du fork

Processus :Exemple programme fork

- La fonction **getpid()** retourne le PID du processus appelant.
- La fonction **getppid()** retourne le PPID du processus appelant.

getpid()

```
pid_t getpid(void);
```

getppid()

```
pid_t getppid(void);
```

Processus :Exemple programme fork

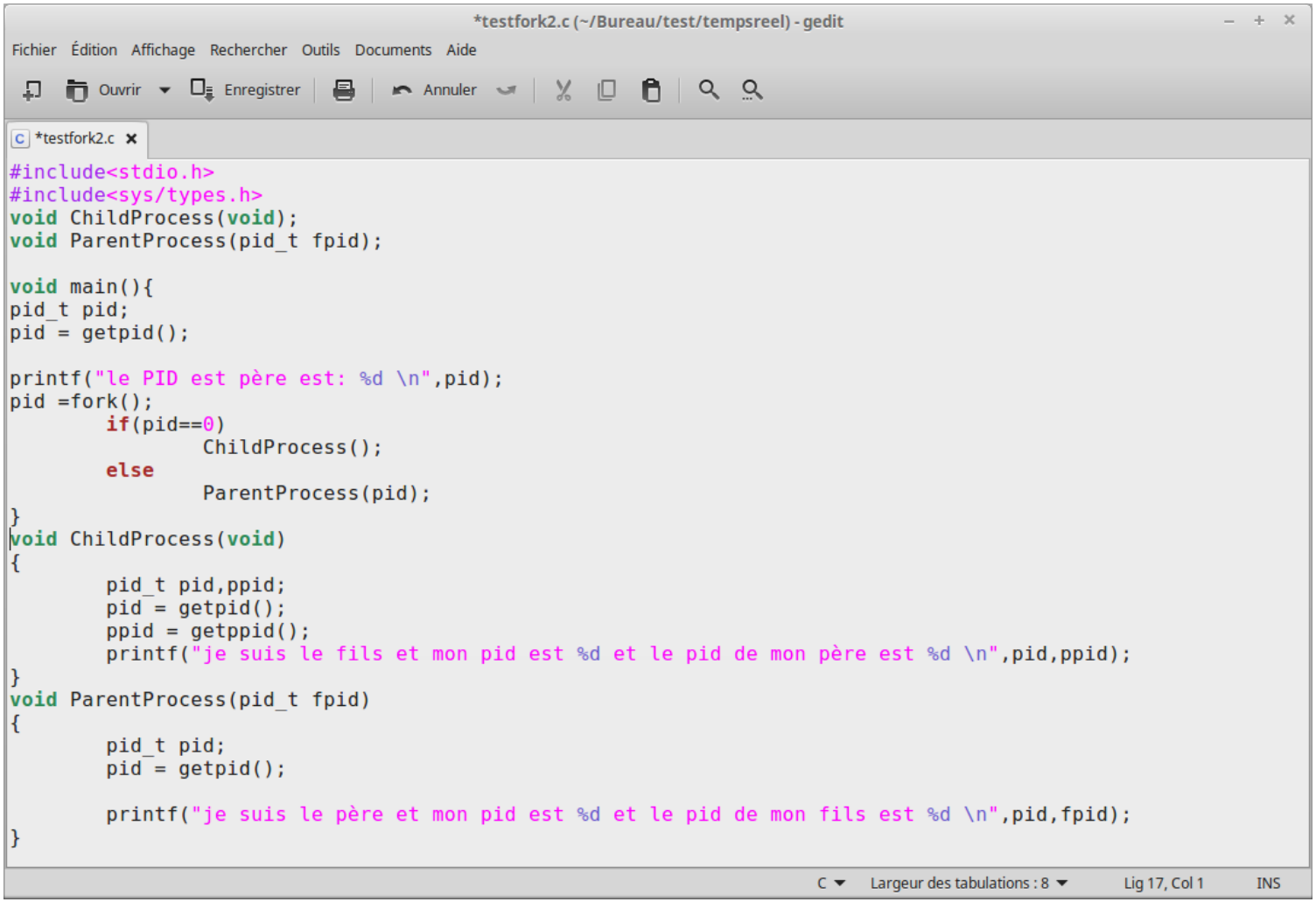


Figure – Exemple d'exécution du fork avec séparation du traitement

Processus :Exemple programme fork

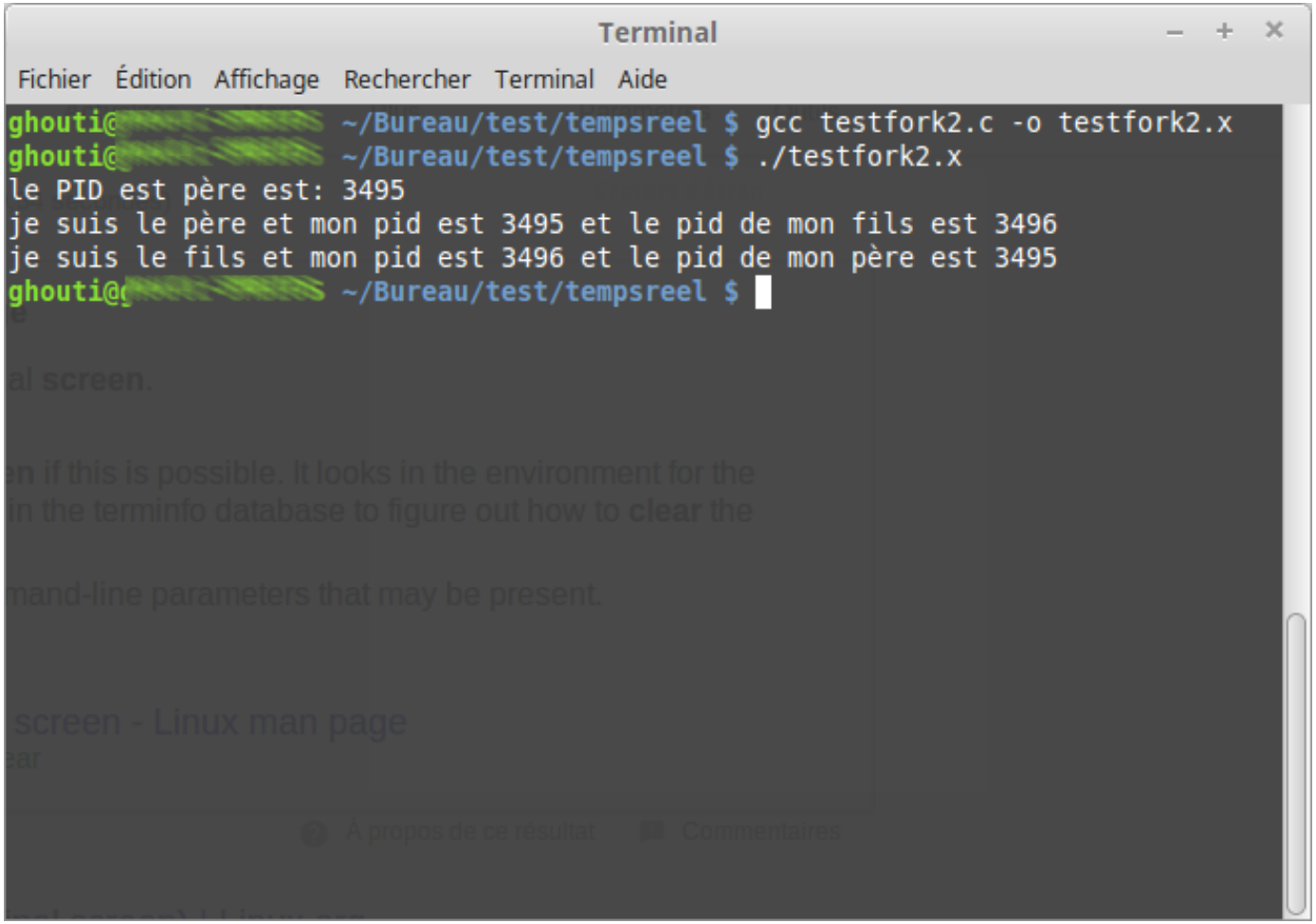


Figure – Exemple d'exécution du fork avec séparation du traitement

Processus :La famille exec

- Si l'on désire exécuter du code à l'intérieur de ce nouveau processus, on utilisera un appel de type **exec : execl, execlp, execle, execv ou execvp.**
- La famille de fonctions **exec** remplace l'image mémoire du processus en cours par un nouveau processus.

Processus :Exemple programme fork

testfork3.c (~/Bureau/test) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

testfork3.c

```c
#include  <stdio.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);               /* child process prototype  */
void  ParentProcess(void);              /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   Le fils dit , valeur = %d\n", i);
     printf("   *** Le fils a terminé ***\n");
}

void  ParentProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("Le père dit : valeur = %d\n", i);
     printf("*** Le père a terminé ***\n");
}
```

C Largeur des tabulations: 8 Lig 6, Col 1 INS

Figure – Exemple d'execution du fork

Processus :Exemple programme fork

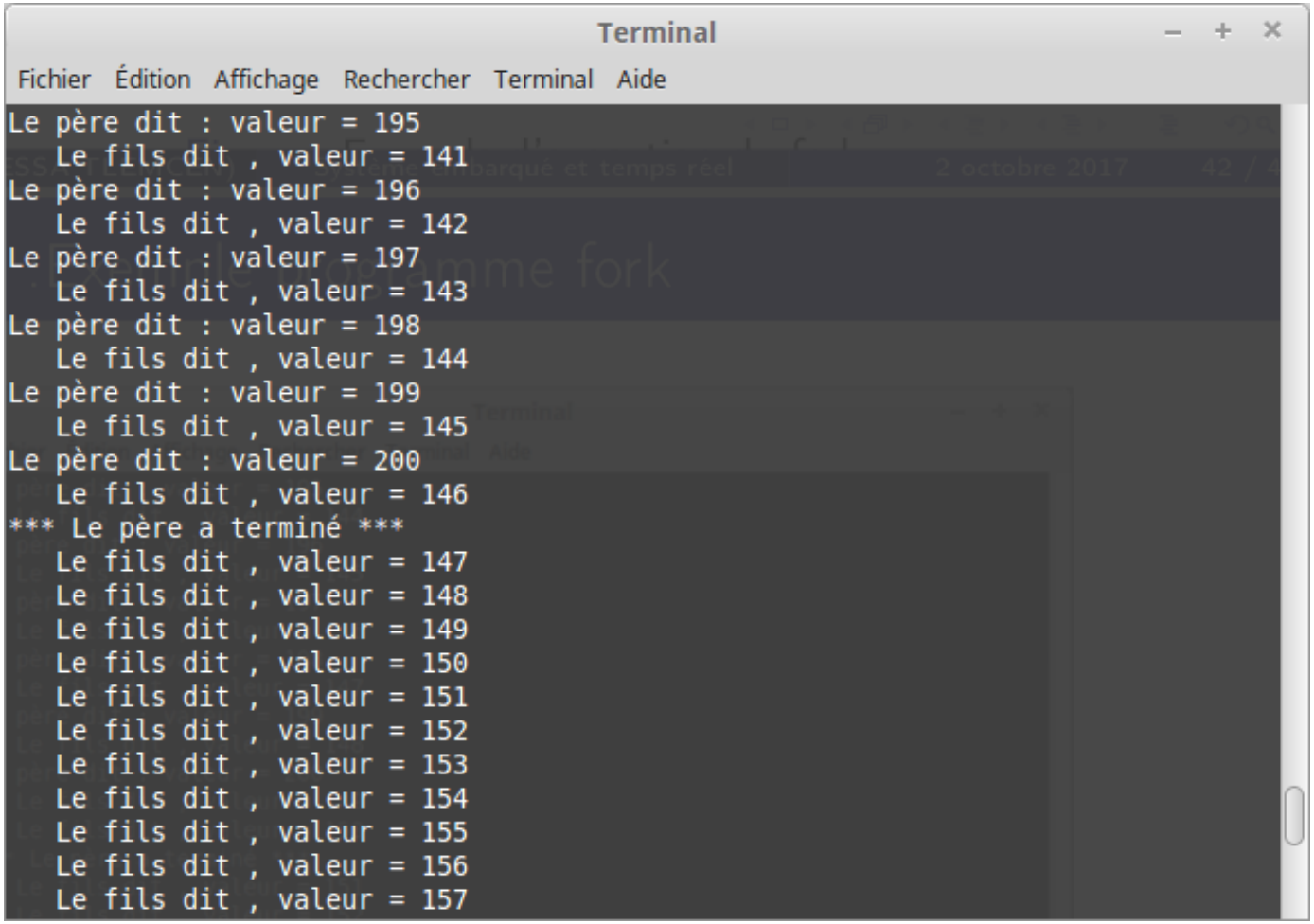


Figure – Exemple d'execution du fork

Processus :wait()

- Rappel : les processus créés par des **fork** s'exécutent de façon concurrente avec leur père. On ne peut présumer l'ordre d'exécution de ces processus (cf. politique de l'ordonnanceur).
- Il sera donc impossible de déterminer quels processus se termineront avant tels autres (y compris leur père). D'où l'existence, dans certains cas, d'un problème de synchronisation.
- La primitive **wait** permet l'élimination de ce problème en provoquant la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.

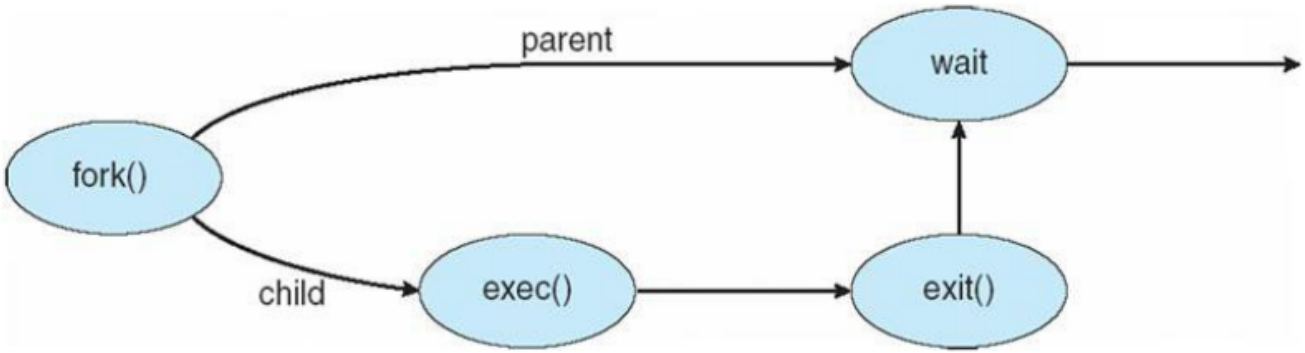


Figure – Synchronisation processus père/fils

Processus :Exemple programme fork et wait

testfork3.c (~/Bureau/test) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

testfork3.c

```c
#include  <stdio.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);               /* child process prototype  */
void  ParentProcess(void);              /* parent process prototype */
pid_t child_pid;
void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0) {
        child_pid=getpid();
          ChildProcess();}
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   Le fils dit , valeur = %d\n", i);
     printf("   *** Le fils a terminé ***\n");
}

void  ParentProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("Le père dit : valeur = %d\n", i);
        wait(NULL);//<--------|---------------------- suspendre l'execution du processus père
     printf("*** Le père a terminé ***\n");
}
```

C Largeur des tabulations: 8 Lig 32, Col 31 INS

Figure – Exemple d'execution du fork

Processus :Exemple programme fork et wait

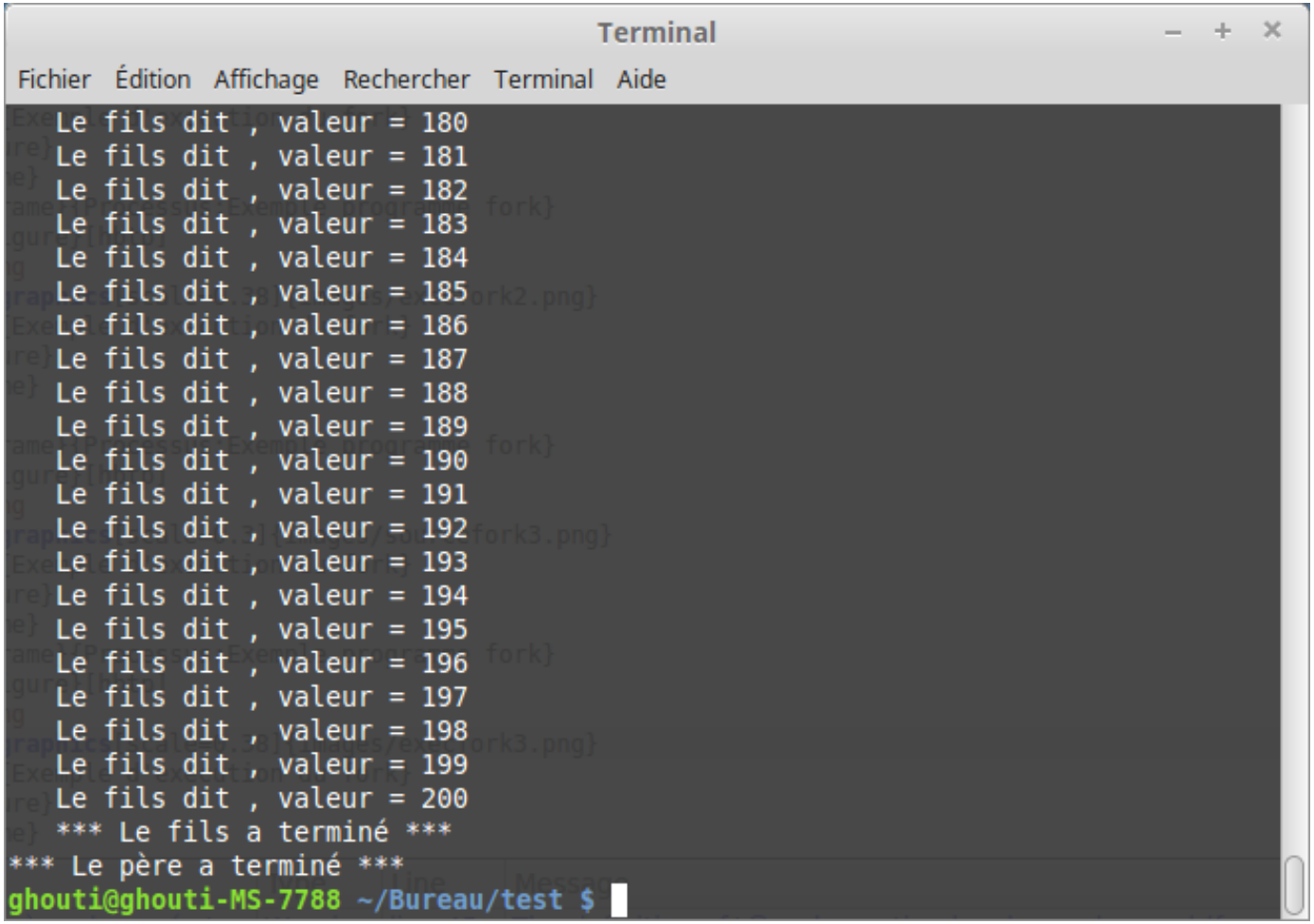


Figure – Exemple d'execution du fork

Processus :waitpid() ;

waitpid()

pid_t waitpid(pid_t pid,int* status,int options) ;

Permet à un processus père d'attendre jusqu'à ce que le processus fils numéro pid termine.
Il retourne l'identifiant du processus fils et son état de terminaison dans &status.

Processus :waitpid() ;

options permet de préciser le comportement de waitpid. On peut utiliser deux constantes :

- WNOHANG : ne pas bloquer si aucun fils ne s'est terminé.
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.
- 0 : Dans le cas où cela ne vous intéresse pas

Processus :waitpid() ;

Le paramètre **status** correspond au code de retour du processus. Autrement dit, la variable que l'on y passera aura la valeur du code de retour du processus (ce code de retour est généralement indiquée avec la fonction exit).

int WIFEXITED(int status)

Elle renvoie vrai si le statut provient d'un processus fils qui s'est terminé en quittant le main avec **return** ou avec un appel à **exit** ;

int WEXITSTATUS(int status)

Elle renvoie le code de retour du processus fils passé à **exit** ou à **return** ;

Processus :waitpid() ;

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
int main(int argc, char *argv[]){
pid_t pid_fils;
int status;
char *args[] = {"ls","-l",argv[1],NULL};
//Dublique le processus
pid_fils = fork();
if(pid_fils!=0){

  // processus père
  while(waitpid(pid_fils,&status,0)<=0);
 if (WIFEXITED(status)) {
    printf(" Terminaison normale du processus fils.\n"
           " Code de retour : %d\n", WEXITSTATUS(status));

    }
  printf("programme principal termine\n");
}else{
  // processus fils
  execvp("ls",args);
  //retourner en cad d'erreur
  perror("erreur dans execvp\n");
  exit(1);
}
return 0;
}
```

Code source 1 – WaitPid exemple

Les threads
Qu'est ce qu'un thread ?

Thread

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un thread de contrôle unique, le thread principal. Du point de vue programmation, ce dernier exécute le main.

Compilation

Toutes les fonctions relatives aux threads sont incluses dans le fichier d'en-tête <pthread.h> et dans la bibliothèque libpthread.a (soit -lpthread à la compilation).

Exemple

gcc monProgramme.c -o monProgramme.x **-lpthread**

Les threads

Créer un thread

pthread_create()

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

- La fonction renvoie 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 : joignable (par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...).
- Le troisième argument est un pointeur vers la fonction à exécuter dans le thread.
- le quatrième et dernier argument est l'argument à passer au thread.

Les threads
Supprimer un thread

pthread_exit()

```
void pthread_exit(void *ret);
```

Les threads

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

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");
    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;
    printf("Avant la création du thread.\n");
    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
  perror("pthread_create");
  return EXIT_FAILURE;
    }
    printf("Après la création du thread.\n");
    return EXIT_SUCCESS;
}
```

Code source 2 – Exemple de création d'un thread

Les threads

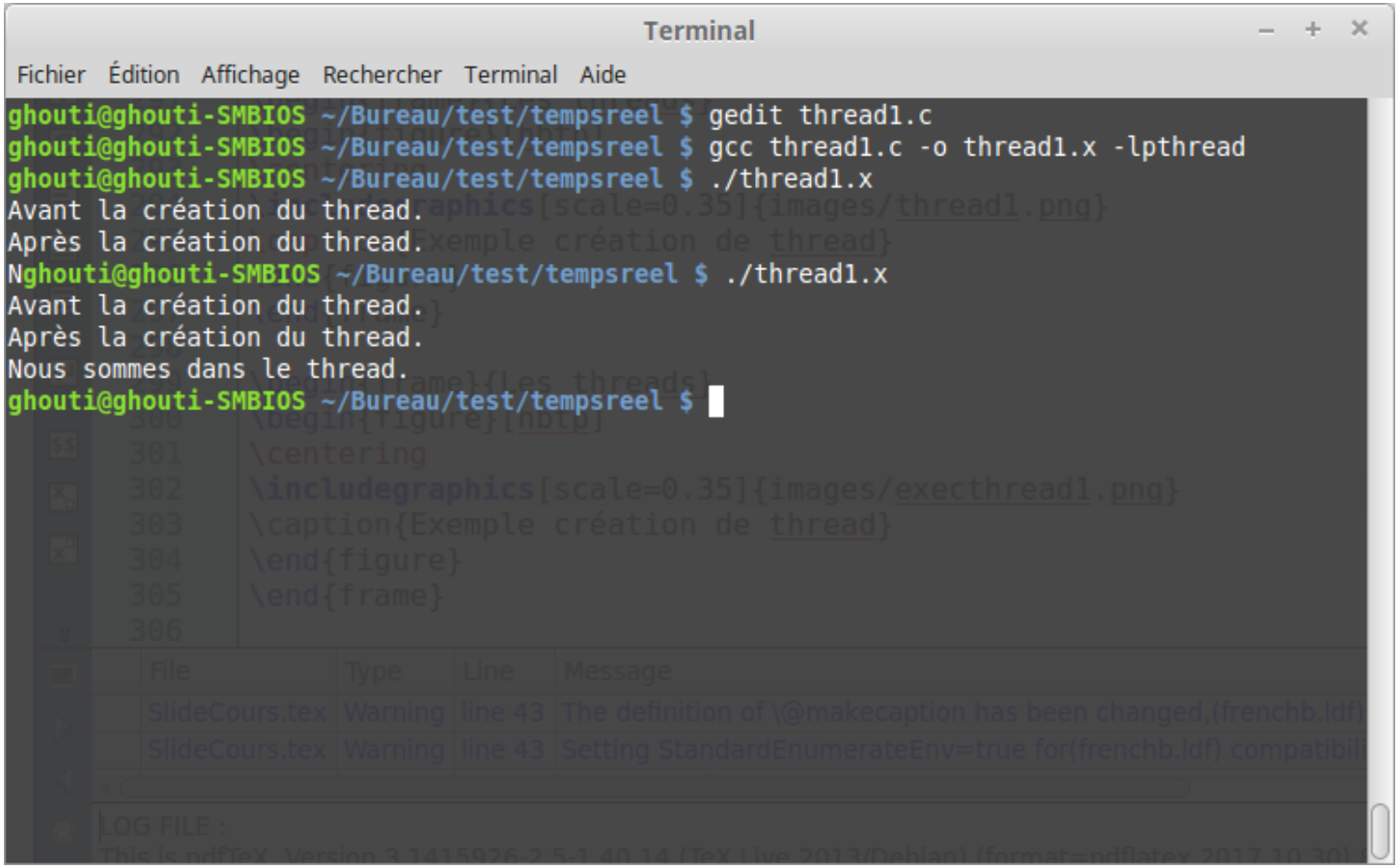


Figure – Exemple création de thread

Les threads

pthread_join()

```
int pthread_join(pthread_t th, void **thread_return);
```

Les threads

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

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");
    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;
    printf("Avant la création du thread.\n");
    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
  perror("pthread_create");
  return EXIT_FAILURE;
    }
if (pthread_join(thread1, NULL)) {//<---------- Attendre la fin du
     thread1
    perror("pthread_join");
    return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");
    return EXIT_SUCCESS;
}
```

Code source 3 – Exemple de synchronisation entre deux threads

Les threads

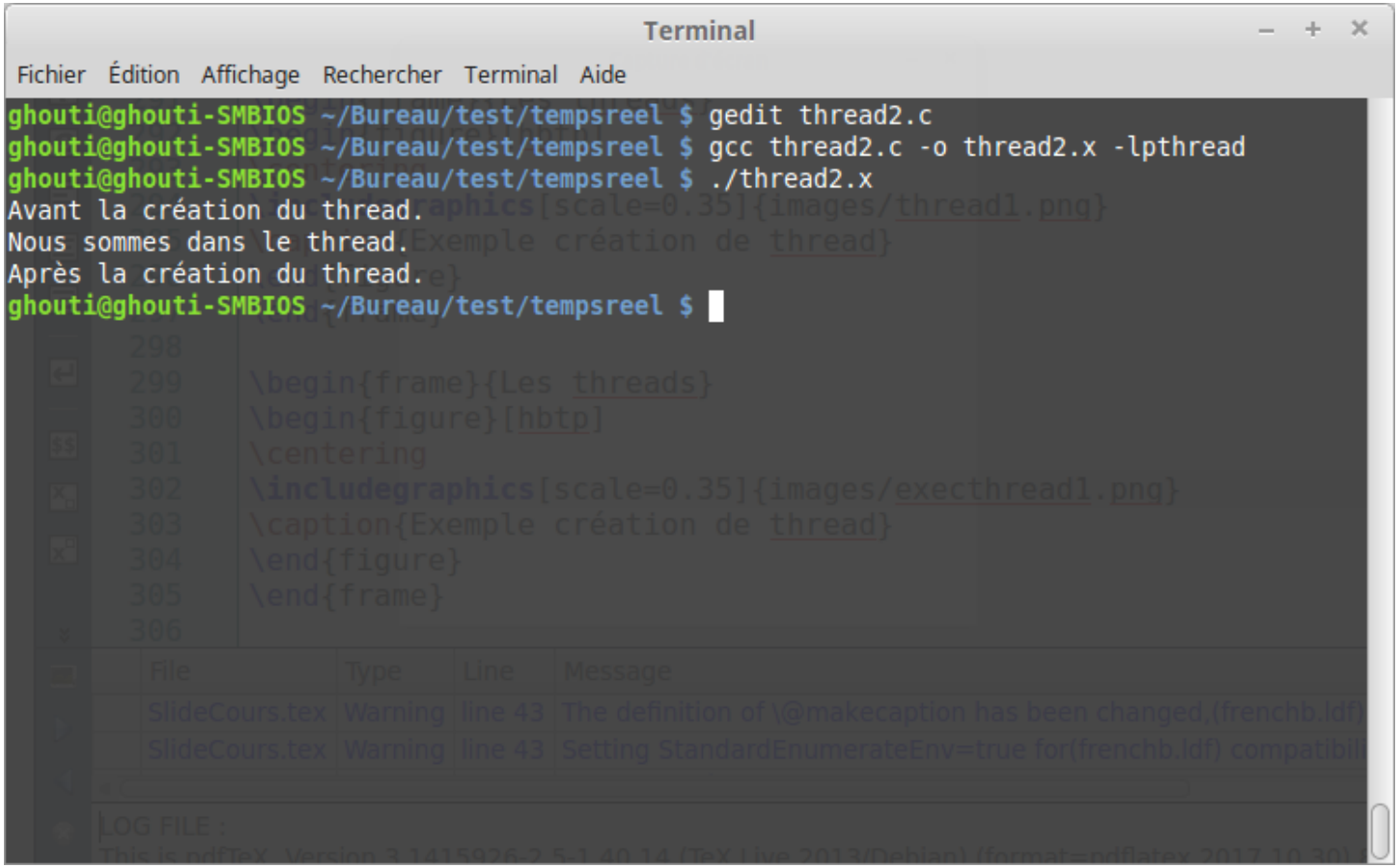


Figure – Exemple création de thread

Synchronisation

La synchronisation des processus est un problème plus large que celui de la concurrence :

- concerne les méthodes mises en œuvre afin que les processus coordonnent leurs activités ;
- attente de la réalisation d'un calcul (fork/join) ;
- utilisation de ressources (modèles Lecteurs/Rédacteurs et Producteurs/consommateurs, etc.) ;
- file d'attente de traitements ;
- barrière de synchronisation etc.

Synchronisation

La synchronisation lie le thread (de contrôle) d'un processus avec celui d'un autre processus ;
Elle implique l'échange d'information de contrôle entre processus. Exemples :

- un processus dépend des données produites par un autre ; si les données ne sont pas disponibles, le processus doit attendre jusqu'à ce qu'elles soient disponibles.
- un processus peut changer son état à "Blocked"(Wait(Pi)), et peut signaler aux Processus "Blocked" qu'ils peuvent continuer (signal(Pi)).

Synchronisation

Les primitives de manipulation des processus sous Unix/Linux sont :

- signal() : gestion des signaux
- fork() : création d'un processus
- pause() : mise en sommeil sur l'arrivée d'un signal
- wait() : mise en sommeil sur la terminaison d'un fils
- sleep() : mise en sommeil sur une durée déterminée (argument)
- kill() : envoi de signal à un processus
- exit() : terminaison d'un processus

Synchronisation :États d'un processus

- Le processus est une activité dynamique et il possède un **état** qui évolue au cours du temps. Ce processus transitera par différents états selon que :
  - il s'exécute (ACTIF ou Élu)
  - il attend que le noyau lui alloue le processeur (PRET)
  - il attend qu'un événement se produise (ATTENTE ou Bloqué)
- C'est l'**ordonnanceur** (*scheduler*) qui contrôle l'exécution et les états des processus.

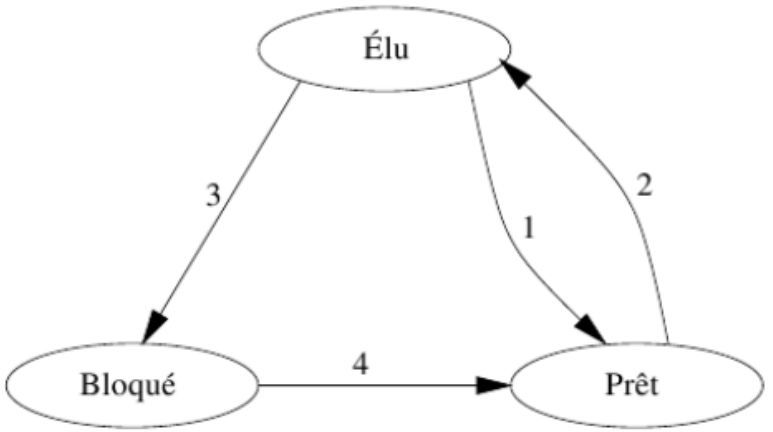


Figure – États d'un processus

Synchronisation

WIFSIGNALED(status)

Elle renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause d'un signal.

WTERMSIG(status)

Elle renvoie la valeur du signal qui a provoqué la terminaison du processus fils.

Exercices

Exercice 1 :
Donner un programme qui affiche, dans l'ordre, les entiers de 0 à 3 par 4 processus à partir de la structure suivante ;

```
for (i=0 ; i < 4 ; i++) {
    switch (fork ()) {
        case -1 :
            ...
        case 0 :
            ...
        case -1 :
            ...
        default :
            ...
    }
```

Primitives dans le win32

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Synchronisation et communication

Synchronisation et communication

Communication

Échange de données

Synchronisation

Échange d'information de contrôle.

Communication

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main(){
int p[2];
char message[50];
pipe(p);
if(fork()==0){
printf("Message du fils a son père\n");
sprintf(message,"Bonjour papa\n");
write(p[1],message,sizeof(message));
printf("le fils a dit: %s\n",message);
}else {
read(p[0],message,sizeof(message));
printf("lecture du message de fils par le père\n");
printf("Mon fils ma dit: %s\n",message);
}
}
```

Code source 4 – Exemple de message entre processus

Communication

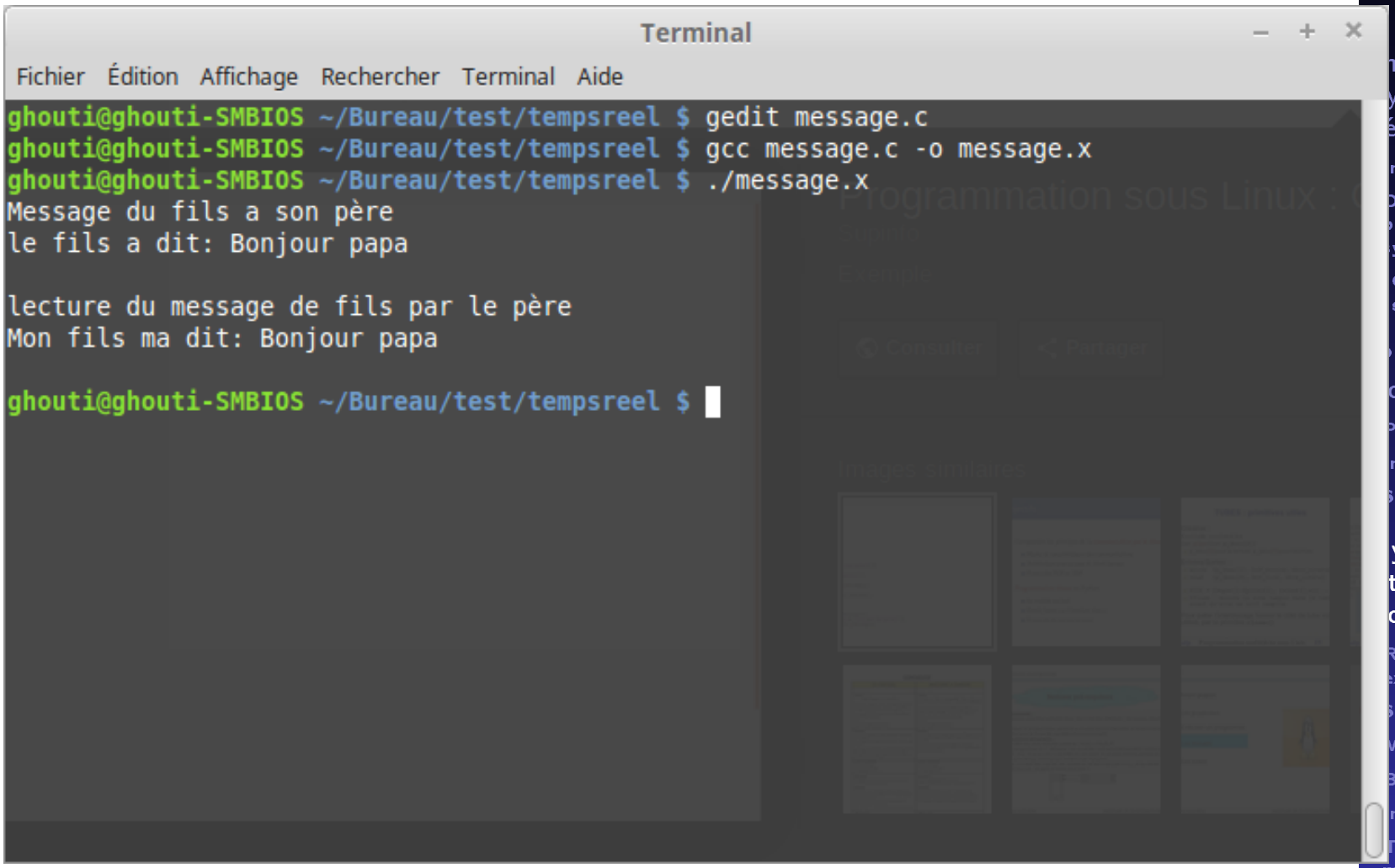


Figure – Exemple de message entre processus

Communication

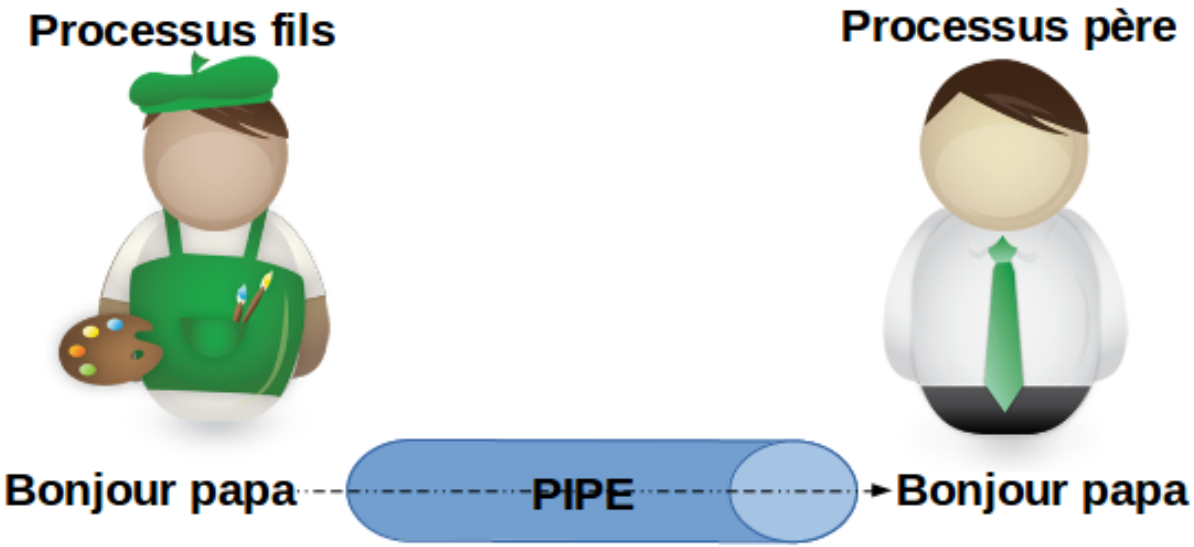


Figure – principe de pipe entre processus

Ressources communes exclusion mutuelle
La notion de ressource

Définition

Ressource La notion de ressource dans les SE désigne tout élément qui est utile au déroulement d'un processus.

Une ressource peut être :

- Physique (si on se place du point de vue du système d'exploitation) : processeur, mémoire, périphérique
- Logique (si on se place du point de vue du programmeur) : fichier, variable, objet

La notion de ressource

Plusieurs processus peuvent avoir besoin des mêmes ressources.

Définition

Ressource une ressource est dite partageable si elle peut être utilisée en même temps à plusieurs processus. Une ressource est dite non partageable si elle doit être à utilisée exclusivement par un seul processus.

Dans ce cas on dit que les processus doivent être exclusion mutuelle pour l'accès à la ressource.

Ressources communes exclusion mutuelle
Exposé du problème

- Les entités (**processus** ou **threads**) en cours d'exécution sont généralement (Indépendantes et Asynchrones) :
  - Leur fonctionnement ne dépend pas a priori du travail réalisé par les autres entités
  - Elles peuvent a priori progresser à leur rythme sans se soucier les unes des autres : elles pourraient s'exécuter en parallèle, même si sur un seul CPU, on parlera de « pseudo » parallélisme
- Pourtant, ces entités peuvent être en concurrence pour l'utilisation de ressources
- Avoir besoin de se synchroniser et communiquer : dans ce cas, elles seront au contraire dépendantes les unes des autres

Ressources communes exclusion mutuelle
But : Contrôler la concurrence

- Organiser la compétition :
  - Fournir des services de synchronisation indirecte par exclusion mutuelle : « Arbitrage », rôle du système
  - Ou au contraire, inclure la partie contrôle de concurrence au sein des programmes : « sans arbitrage » par le système
- Coordonner l'utilisation des ressources :
  - Empêcher ou réparer des blocages, garantir l'équité ou absence de famine

Ressources communes exclusion mutuelle
Solution au Problème

- Imposer des sections critiques qui doivent être exécutées de manière non entrelacée pour garantir une utilisation correcte des ressources
- On dit aussi qu'une section critique doit être indivisible, atomique

Ressources communes exclusion mutuelle
Section critique

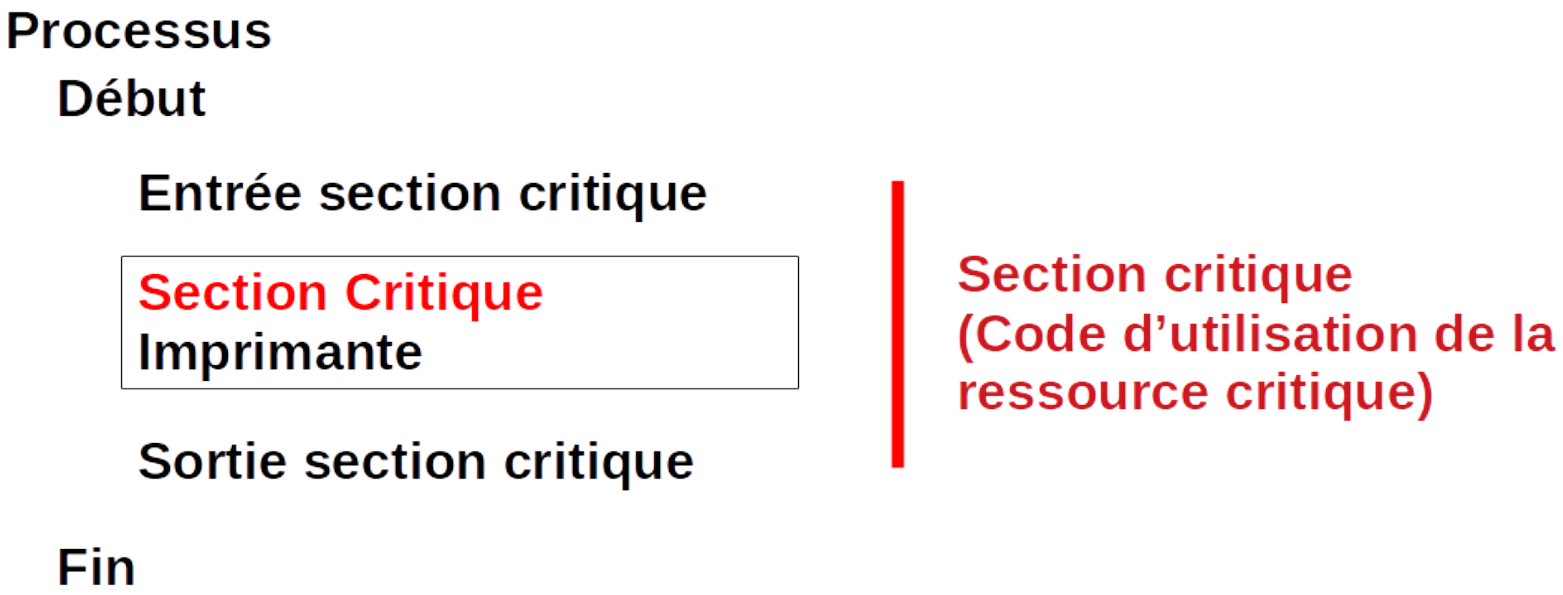


Figure – Processus avec une section critique

Ressources communes exclusion mutuelle
La notion verrouillage

Lorsqu'une ressource est partagée, si plusieurs processus accèdent à la ressource, des incohérences peuvent en résulter : il faut un arbitre

Exemple

des guichets automatiques de retrait d'argent : simulation du comportement

Comment éviter les situation de compétition sur des ressources non partageable ?

- interdire l'accès partagé à la ressource :exclusion mutuelle
- interdire l'exécution simultanée du codé accédant à la ressource : section critique

Ressources communes exclusion mutuelle
La notion verrouillage

Pour résoudre ce problème on utilise la notion de verrou (lock)
Deux opérations sont proposées :

- Verrouiller(v) permet à un processus d'acquérir un verrou,s'il n'est pas disponible, le processus est bloqué en attente du verrou
- Déverrouiller(v) permet de libérer un verrou.

Ressources communes exclusion mutuelle
Section critique

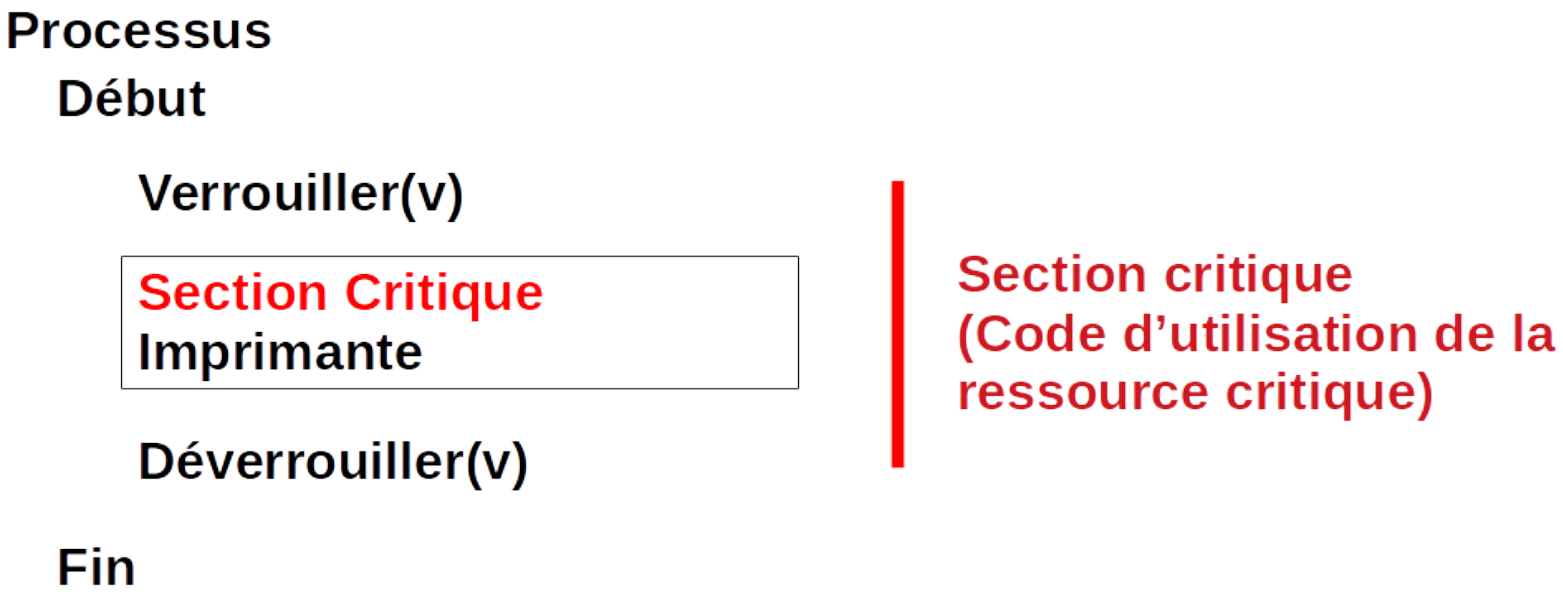


Figure – Processus avec une section critique

Sémaphores

Les sémaphores (Dijkstra 1965) proposent un mécanisme de contrôle de la concurrence sous la forme d'un distributeur de jeton (n jetons).
Un sémaphore S est une variable entière qui n'est accessible qu'au travers de 3 opérations Init, P et V.
Deux opérations sont définies :

- Init(S, valeur) est seulement utilisé pour initialiser le sémaphore S avec une valeur.
- P(s) pour obtenir un jeton : l'opération P est mise en attente jusqu'à ce qu'une ressource soit disponible
- V(s) pour restituer un jeton.

Remarque

Les opérations P et V doivent être indivisibles : le SE leur garanti une exécution atomique (pas de commutation de contexte).

Sémaphores

P()

P(){
Si S > 0 **alors** S = S -1
sinon
ajouter tâche à la file d'attente
}

V()

V(){
S = S + 1
Si file d'attente n'est pas vide **alors**
transfert de contrôle a une autre tache dans la fille t'attente(réveil)
}

Sémaphores
Sémaphore binaire

- Un sémaphore binaire est un sémaphore qui est initialisé avec la valeur 1.
- Ceci a pour effet de contrôler l'accès une ressource unique.
- Le sémaphore binaire permet l'exclusion mutuelle (mutex).

Sémaphores
Accès à une section critique

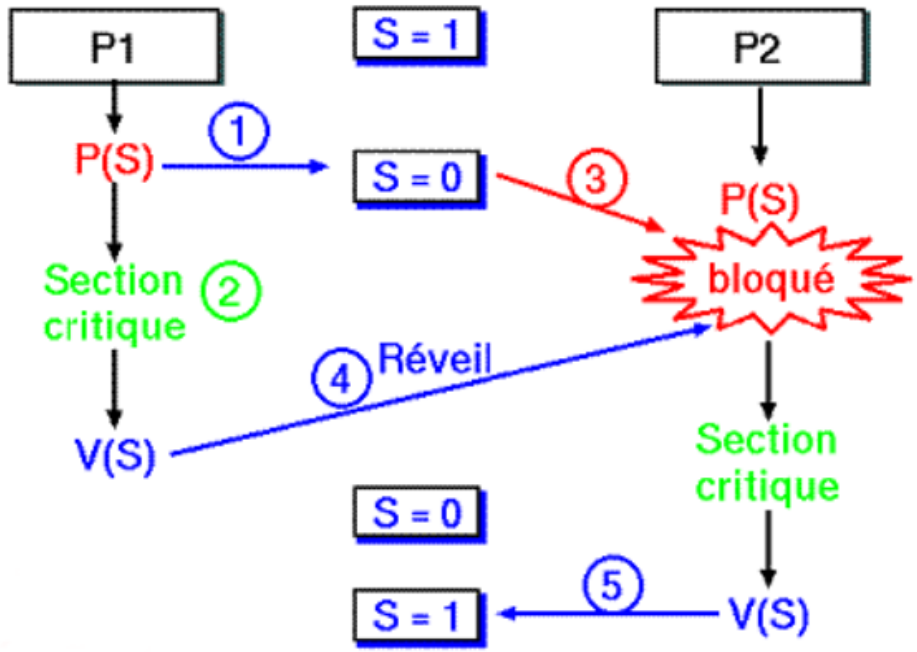


Figure – Principe des sémaphores

Sémaphores
Interblocage

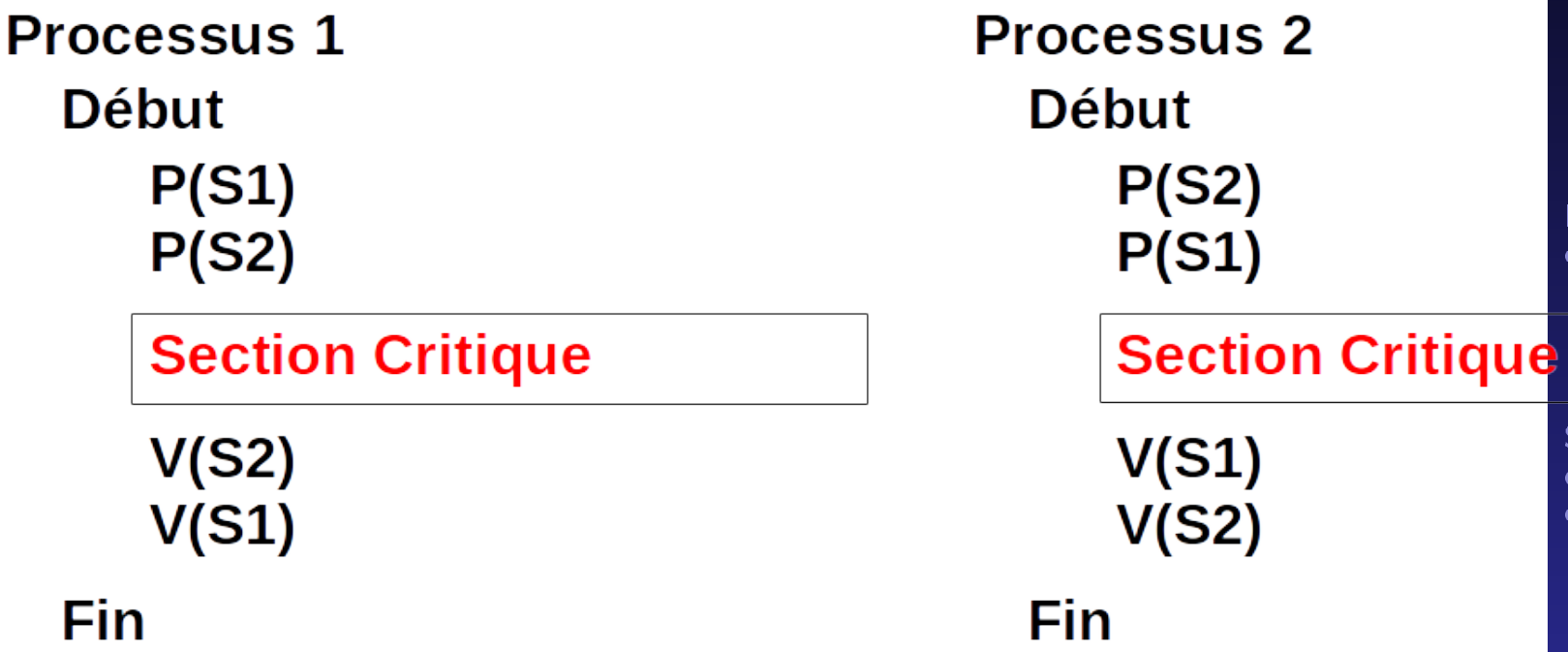


Figure – Cas d'interblocage entre deux processus

Sémaphores
Interblocage

Un interblocage est possible si :

- Le processus 1 obtient S1.
- Le processus 2 obtient S2.
- Le processus 1 attend pour obtenir S2 (qui est entre les mains du processus 2).
- Le processus 2 attend pour obtenir S1 (qui est entre les mains du processus 1).

Dans cette situation, les deux tâches sont définitivement bloquées.

Prévention : Une méthode consiste à toujours acquérir les mutex dans le même ordre.

Sémaphores
Sémaphore bloquant

- Un sémaphore **bloquant** est un sémaphore qui est initialisé avec la valeur **0**
- Ceci a pour effet de bloquer n'importe quel thread qui effectue P(S) tant qu'un autre thread n'aura pas fait un V(S). Ce type d'utilisation est très utile lorsque l'on a besoin de contrôler l'ordre d'exécution entre threads.

Sémaphores
Producteur/Consommateur

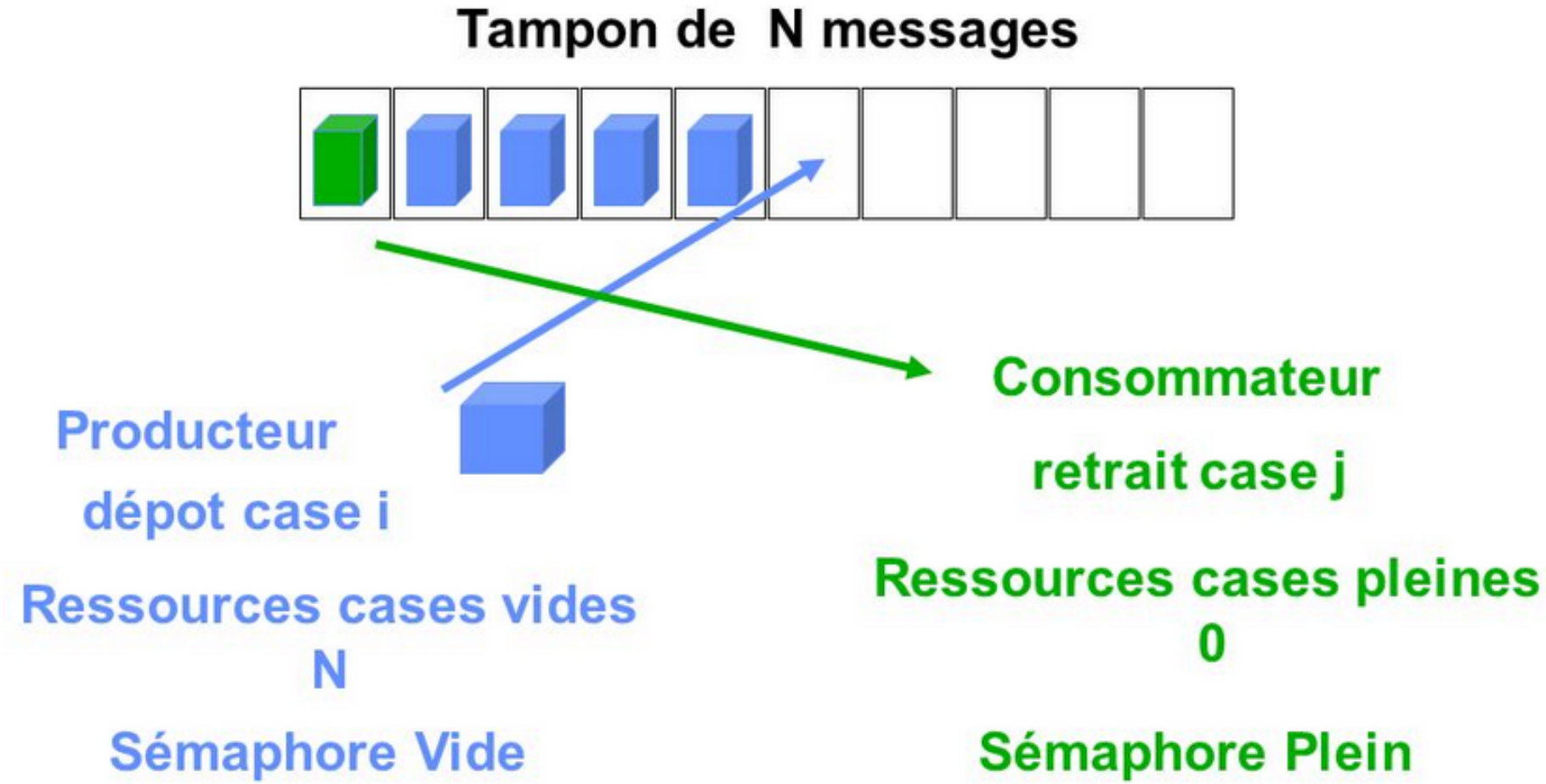


Figure – Problème producteur/consommateur

Sémaphores
Producteur/Consommateur

```
plein = 0
vide = N
Producteur
   Début
     Tant que (Vrai)
        Produire_objet()
        P(vide)
        Section Critique
        V(plein)
     Fin Tant que
   Fin
```

```
Consommateur
   Début
     Tant que (Vrai)
        P(plein)
        Section Critique
        V(vide)
        Consommer_objet()
     Fin Tant que
   Fin
```

Figure – Problème producteur/consommateur

Sémaphores et Le langage C

#include <semaphore.h>

- int **sem_init**(**sem_t** *semaphore, int pshared, unsigned int valeur) ;
  Création d'un sémaphore et préparation d'une valeur initiale.
- int **sem_wait**(**sem_t** * semaphore) ;
  Opération P sur un sémaphore.
- int **sem_trywait**(**sem_t** * semaphore) ;
  Version non bloquante de l'opération P sur un sémaphore.
- int **sem_post**(**sem_t** * semaphore) ;
  Opération V sur un sémaphore.
- int **sem_getvalue**(**sem_t** * semaphore, int * sval) ;
  Récupérer le compteur d'un sémaphore.
- int **sem_destroy**(**sem_t** * semaphore) ;
  Destruction d'un sémaphore.

Sémaphore : exemple

semaphore.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

semaphore.c

```c
void* affichage (void* name) {
int i, j;
for(i = 0; i < 20; i++) {
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
printf("\n ");
}
return NULL;
}

int main (void) {
pthread_t filsA, filsB;

if (pthread_create(&filsA, NULL, affichage, "AA")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
if (pthread_create(&filsB, NULL, affichage, "BB")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
if (pthread_join(filsA, NULL))
perror("pthread_join");
if (pthread_join(filsB, NULL))
perror("pthread_join");
printf("Fin du pere\n") ;
return (EXIT_SUCCESS);
}
```

Sémaphore : exemple

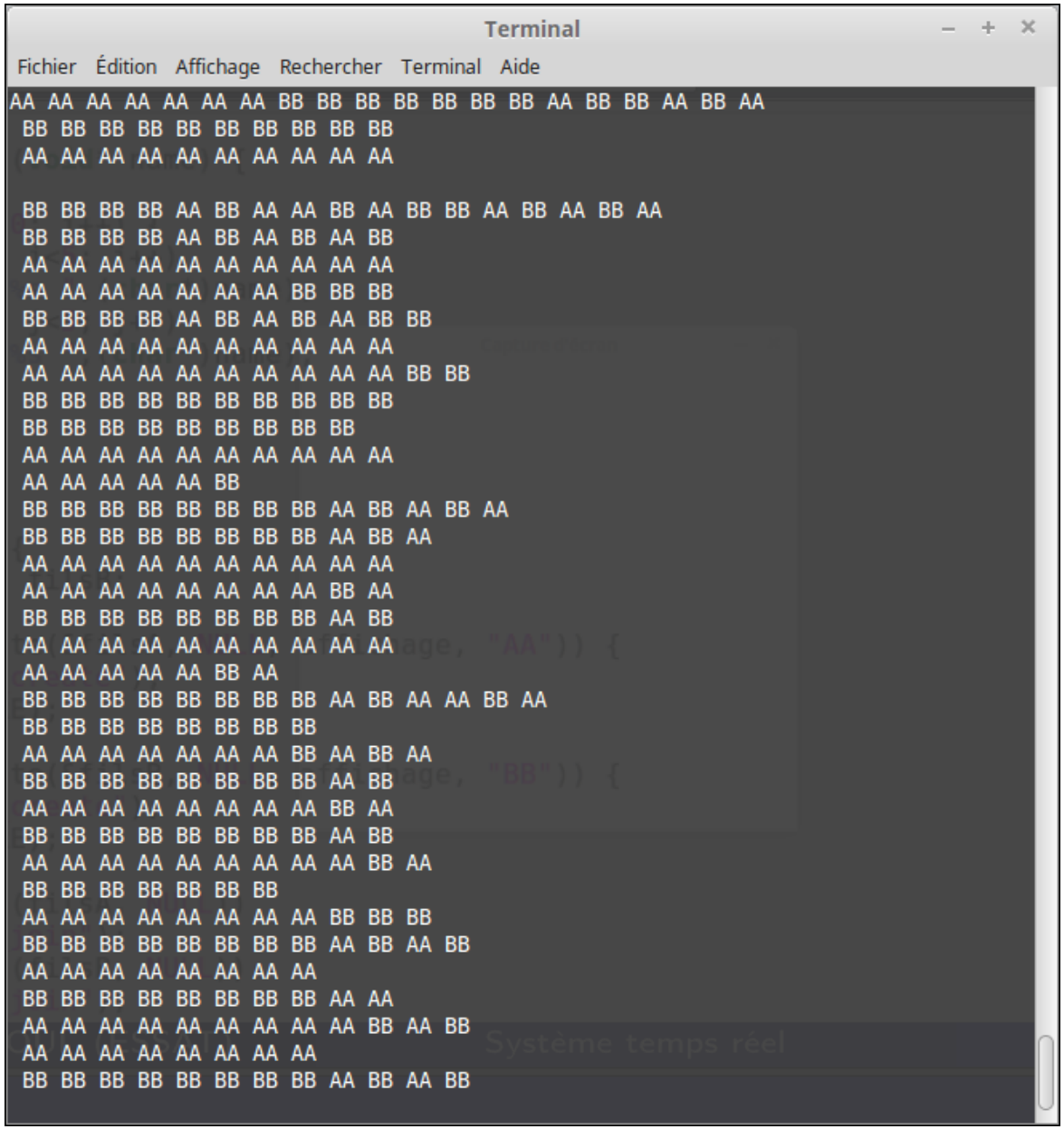


Figure – Exemple de deux threads sans sémaphores

Sémaphore : exemple

semaphore.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

semaphore.c

```c
sem_t mutex;//<----------------------------Sémaphore mutex
void* affichage (void* name) {
int i, j;
for(i = 0; i < 20; i++) {
        sem_wait(&mutex); //<---------------P(mutex)
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
printf("\n ");
sem_post(&mutex);//<-----------------------V(mutex)
}
return NULL;
}
int main (void) {
pthread_t filsA, filsB;
sem_init(&mutex, 0, 1);//<-----------------init(s,1)
if (pthread_create(&filsA, NULL, affichage, "AA")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
```

C Largeur des tabulations : 8 Lig 6, Col 60 IN

Figure – Exemple de deux threads avec sémaphore

Sémaphore : exemple

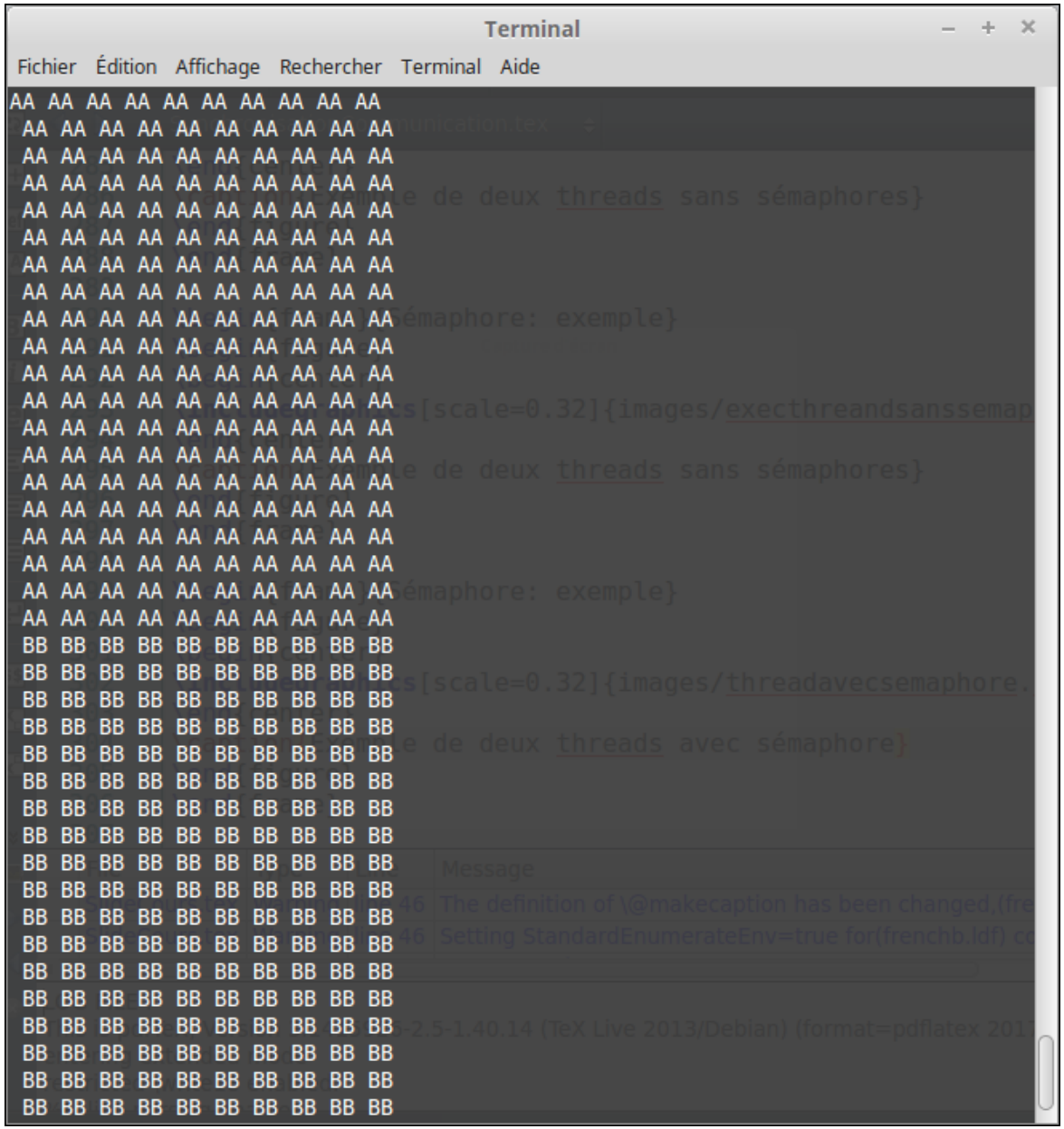


Figure – Exemple de deux threads avec sémaphore

Moniteurs

- Les moniteurs proposent une solution de "haut-niveau" pour la protection de données partagées (Hoare 1974)
- Ils simplifient la mise en place de sections critiques
- Ils sont définis par :
  1. des données internes (appelées aussi variables d'état)
  2. des primitives d'accès aux moniteurs (points d'entrée)
  3. des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
  4. une ou plusieurs files d'attentes

Les moniteurs
Structure d'un d'un moniteur

Type m = **moniteur**
Début
Déclaration des variables locales (ressources partagées) ;
Déclaration et corps des procédures du moniteur (points d'entrée) ;
Initialisation des variables locales ;
Fin

Les Moniteurs
Principe de fonctionnement

- Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur
- La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur

Remarque : L'accès à un moniteur construit donc implicitement une exclusion mutuelle

Les Moniteurs

- Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse), il libère l'accès au moniteur avant de se bloquer.
- Lorsque des variables internes du moniteur ont changé, le moniteur doit pouvoir « réveiller » un processus bloqué.
- Pour cela, il existe deux types de primitives :
  1. **wait** : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition
  2. **signal** : qui réveille sur une condition un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un **wait** sur la même condition)

Les Moniteurs
Les variables condition

- Une variable condition : est une variable
  - qui est définie à l'aide du type condition ;
  - qui a un identificateur mais,
  - qui n'a pas de valeur (contrairement à un sémaphore).
- Une condition :
  - ne doit pas être initialisée
  - ne peut être manipulée que par les primitives Wait et Signal.
  - est représentée par une file d'attente de processus bloqués sur la même cause ;
  - est donc assimilée à sa file d'attente.

Les Moniteurs
Les primitives Wait et Signal

- La primitive **Wait** bloque systématiquement le processus qui l'exécute
- La primitive **Signal** réveille un processus de la file d'attente de la condition spécifiée, si cette file d'attente n'est pas vide ; sinon elle ne fait absolument rien.

Les Moniteurs
Syntaxe

1. Syntaxe :

   cond.**Wait** ;

   cond.**Signal** ;

   /* cond est la variable de type condition déclarée comme variable locale */

2. Autre syntaxe :

   Wait(*cond*) ;

   Signal(*cond*) ;

Un processus réveillé par *Signal* continue son exécution à l'instruction qui suit le *Wait* qui l'a bloqué.

Exemple RDV entre N processus à l'aide d'un moniteur

```
Type Rendez_vous = moniteur
        {variables locales }
        Var Nb_arrivés : entier ; Tous_Arrivés : condition ;
        {procédure accessible aux programmes utilisateurs }
        Procedure Entry Arriver ;
        Début
                Nb_arrivés = Nb_arrivés + 1 ;
                Si Nb_arrivés < N Alors Tous_Arrivés.Wait ;
                Tous_Arrivés.Signal ;
        Fin
        Début  {Initialisations }
                Nb_arrivés = 0;
        Fin.
```

Figure – Exemple de RDV entre N processus

Exemple RDV entre N processus à l'aide d'un moniteur

```
Type Rendez_vous = moniteur
        {variables locales }
        Var Nb_arrivés : entier ; Tous_Arrivés : condition ;
        {procédure accessible aux programmes utilisateurs }
        Procedure Entry Arriver ;
        Début
                Nb_arrivés = Nb_arrivés + 1 ;
                Si Nb_arrivés < N Alors Tous_Arrivés.Wait ;
                Tous_Arrivés.Signal ;
        Fin
        Début  {Initialisations }
                Nb_arrivés = 0;
        Fin.
```

Figure – Exemple de RDV entre N processus

Les Moniteurs

Exemple RDV entre N processus à l'aide d'un moniteur Les programmes des processus s'écrivent alors : **Processus Pi**

.....

Rendez_vous.Arriver ; {*Point de rendez-vous : sera bloquant si au moins un processus n'est pas arrivé au point de rendez-vous*}

.....

Les Moniteurs
Producteur-Consommateur à l'aide des moniteurs

Type ProducteurConsomateur = **Moniteur**
{variables locales}
Var Compte : **entier** ; Plein, Vide : **condition** ;
{procédures accessibles aux programmes utilisateurs }
Procedure Entry Déposer(message M) ;
Début
    si Compte=N alors Plein.Wait ;
    dépôt(M) ;
    Compte=Compte+1 ;
    si Compte==1 alors Vide.Signal ;
Fin

Les Moniteurs
Producteur-Consommateur à l'aide des moniteurs(Suite)

Procedure Entry Retirer(message M) ;
Début
    si Compte=0 alors Vide.Wait ;
    retrait(M) ;
    Compte=Compte-1 ;
    si Compte==N-1 alors Plein.Signal ;
Fin
Début {*Initialisations*}Compte= 0 ; **Fin**.

Les Moniteurs
Producteur-Consommateur à l'aide des moniteurs(Suite)

- **Processus Producteur**

  message M ;
  Début
  tant que vrai faire
  Produire(M) ;
  ProducteurConsommateur.déposer(M) ;
  Fin

- **Processus Consommateur**

  message M ;
  Début
  tant que vrai faire
  ProducteurConsommateur.retirer(M) ;
  Consommer(M) ;
  Fin

Les Moniteurs
Les moniteurs en langage C

- d'un mutex (type **pthread_mutex_t**) qui sert à protéger la partie de code où l'on teste les conditions de progression
- et d'une variable condition (type **pthread_cond_t**) qui sert de point de signalisation :
- on se met en attente sur cette variable par la primitive : Libération du mutex + Blocage systématique de l'appelant de manière atomique **pthread_cond_wait**(&*laVariableCondition*,&*leMutex*) ;
- on est réveillé sur cette variable avec la primitive : Réveil d'une thread en attente qui acquiert à nouveau le mutex **pthread_cond_signal**(&*laVariableCondition*) ;

Les moniteurs en langage C

- Un mutex peut être verrouillé par la primitive
  int pthread_mutex_lock(pthread_mutex_t *mutex) ;
- Un mutex peut être déverrouillé par la primitive
  int pthread_mutex_unlock(pthread_mutex_t *mutex) ;

Les Moniteurs

moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

moniteur.c

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

/* définition du tampon */
#define N 10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */

int NbPleins=0;
int tete=0, queue=0;
int tampon[N];

/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;
pthread_t producteur;
pthread_t consomateur;
```

C Largeur des tabulations : 8 Lig 5, Col 1 INS

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```
void Deposer(int m){
pthread_mutex_lock(&mutex);
        if(NbPleins == N) pthread_cond_wait(&plein, &mutex);
        tampon[queue]=m;
        queue=(queue+1)%N;
        NbPleins++;
pthread_cond_signal(&vide);
pthread_mutex_unlock(&mutex);
}
int Prelever(void){
int m;
pthread_mutex_lock(&mutex);
if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);
m=tampon[tete];
tete=(tete+1)%N;
NbPleins--;
pthread_cond_signal(&plein);
pthread_mutex_unlock(&mutex);
return m;
}
```

C Largeur des tabulations : 8 Lig 29, Col 1 INS

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```
void * Prod(void * k)
/********** PRODUCTEUR */
{
int i;
int mess;
srand(pthread_self());
for(i=0;i<=NbMess; i++){
usleep(rand()%10000); /* fabrication du message */
mess=rand()%1000;
Deposer(mess);
printf("Mess depose: %d\n",mess);
}
}
```

C Largeur des tabulations : 8 Lig 66, Col 1 IN

Figure – Exemple de producteur consommateur

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```c
void * Cons(void * k)
/********** CONSOMMATEUR */
{
int i;
int mess;
srand(pthread_self());
for(i=0;i<=NbMess; i++){
mess=Prelever();
printf("\tMess preleve: %d\n",mess);
usleep(rand()%1000000); /* traitement du message */
}
}
```

C Largeur des tabulations : 8 Lig 80, Col 1

Figure – Exemple de producteur consommateur

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```c
void main()
/* M A I N */
{
int i, num;
pthread_mutex_init(&mutex,0);
pthread_cond_init(&vide,0);
pthread_cond_init(&plein,0);
/* creation des threads */
pthread_create(&producteur, 0, (void * (*)()) Prod, NULL);
pthread_create(&consomateur, 0, (void * (*)()) Cons, NULL);
// attente de la fin des threads
pthread_join(producteur,NULL);
pthread_join(consomateur,NULL);
// libération des ressources
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&vide);
pthread_cond_destroy(&plein);
exit(0);
}
```

C Largeur des tabulations : 8 Lig 80, Col 1 INS

Blocage

L'exécution d'un processus nécessite un ensemble de ressources (mémoire principale, disques, fichiers, périphériques, etc.) qui lui sont attribuées par le système d'exploitation. L'utilisation d'une ressource passe par les étapes suivantes :

- Demande de la ressource : Si l'on ne peut pas satisfaire la demande il faut attendre. La demande sera mise dans une table d'attente des ressources.
- Utilisation de la ressource : Le processus peut utiliser la ressource.
- Libération de la ressource : Le processus libère la ressource demandée et allouée.

Inter-blocage

Des problèmes peuvent survenir, lorsque les processus obtiennent des accès exclusifs aux ressources. Par exemple, un processus P1 détient une ressource R1 et attend une autre ressource R2 qui est utilisée par un autre processus P2 ; le processus P2 détient la ressource R2 et attend la ressource R1. On a une situation d'interblocage (deadlock) car P1 attend P2 et P2 attend P1. Les deux processus vont attendre indéfiniement comme le montre la figure suivante :

Inter-blocage

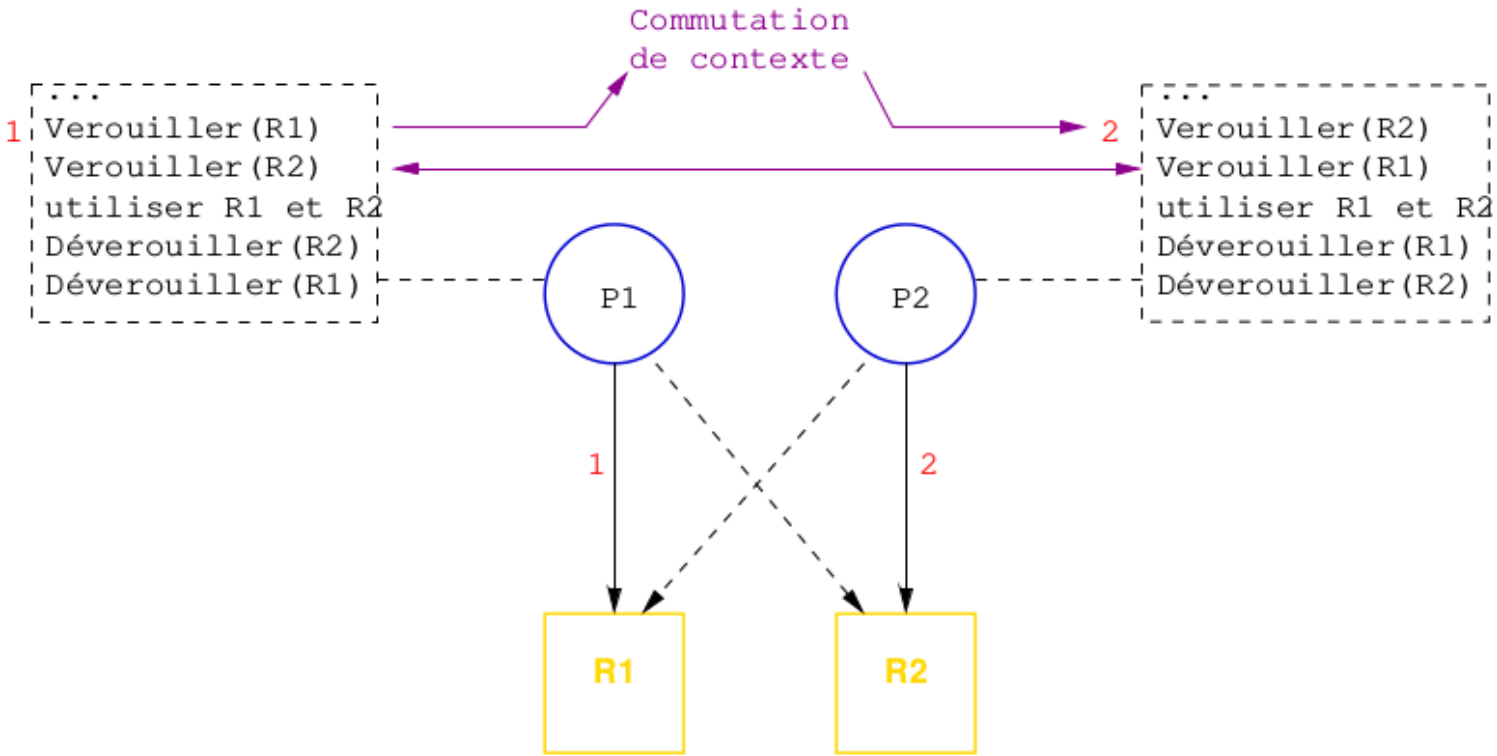


Figure – Exemple d'interblocage

Inter-blocage
Conditions nécessaires pour l'inter-blocage

Pour qu'une situation d'interblocage ait lieu, les quatre conditions suivantes doivent être remplies (Conditions de *Coffman*) :

- **L'exclusion mutuelle**. A un instant précis, une ressource est allouée à un seul processus.
- **La détention et l'attente**. Les processus qui détiennent des ressources peuvent en demander d'autres.
- **Pas de préemption**. Les ressources allouées à un processus sont libérées uniquement par le processus.
- **L'attente circulaire**. Il existe une chaîne de deux ou plus processus de telle manière que chaque processus dans la chaîne requiert une ressource allouée au processus suivant dans la chaîne.

Inter-blocage
Graphe d'allocation des ressources

Le graphe d'allocation des ressources est un graphe biparti composé de deux types de nœuds et d'un ensemble d'arcs :

- **Les processus** qui sont représentés par des cercles.
- **Les ressources** qui sont représentées par des rectangles. Chaque rectangle contient autant de points qu'il y a d'exemplaires de la ressource représentée.
- **Un arc** orienté d'une **ressource vers un processus** signifie que la ressource est allouée au processus.
- **Un arc** orienté d'un **processus vers une ressource** signifie que le processus est bloqué en attente de la ressource.

Inter-blocage
Graphe d'allocation des ressources

Exemple

Soient trois processus A, B et C qui utilisent trois ressources R, S et T selon le tableau suivant :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

Inversion de priorité

1. A demande R
2. B demande S
3. C demande T
4. A demande S
5. B demande T
6. C demande R

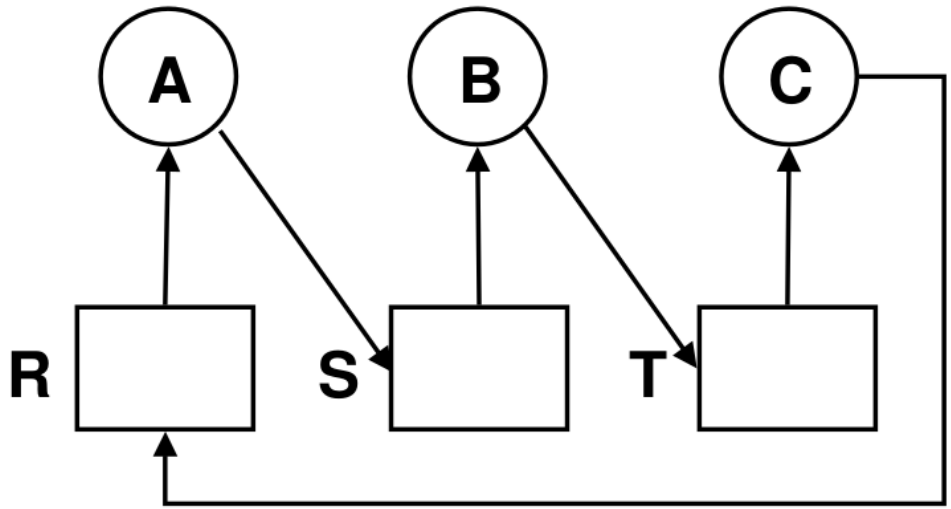


Figure – Situation d'interblocage de trois processus

Inter-blocage

Réduction du graphe d'allocation des ressources

Un graphe réduit peut-être utilisé pour déterminer s'il existe ou non un interblocage. Pour la réduction d'un graphe d'allocation des ressources, les flèches associées à chaque processus et à chaque ressource doivent être vérifiées.

- Si une ressource possède seulement des flèches qui sortent (il n'y a pas des requêtes), on les efface.
- Si un processus possède seulement des flèches qui pointent vers lui, on les efface.
- Si une ressource a des flèches qui sortent, mais pour chaque flèche de requête il y a une ressource disponible dans le bloc de ressources où la flèche pointe, il faut les effacer.

Inter-blocage

Réduction du graphe d'allocation des ressources

Exemple

Considérez quatre processus P1,...P4 qui utilisent des ressources du type R1 , R2 et R3 . Le tableau suivant montre l'allocation courante et le nombre maximaux d'unités de ressources nécessaires pour l'exécution des processus. Le nombre de ressources disponibles est A=[0,0,0].

Processus	R1	R2	R3	R1	R2	R3
P1	3	0	0	0	0	0
P2	1	1	0	1	0	0
P3	0	2	0	1	0	1
P4	1	0	1	0	2	0

Réduction du graphe d'allocation des ressources

Exemple (suite)

Le graphe d'allocation des ressources pour l'état courant est représenté par la figure suivante :

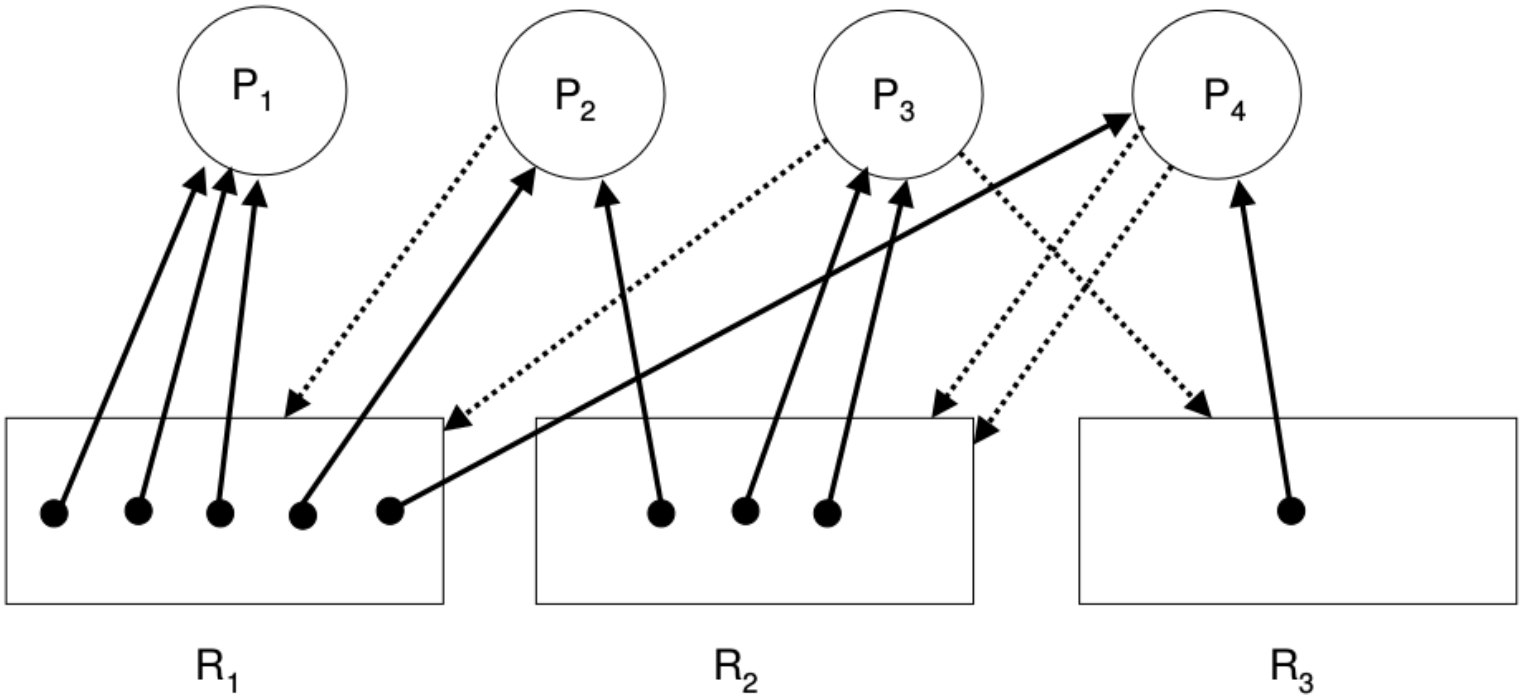


Figure – Graphe d'allocation des ressources

Réduction du graphe d'allocation des ressources

Exemple (suite)

Le graphe d'allocation des ressources pour l'état courant est représenté par la figure suivante :

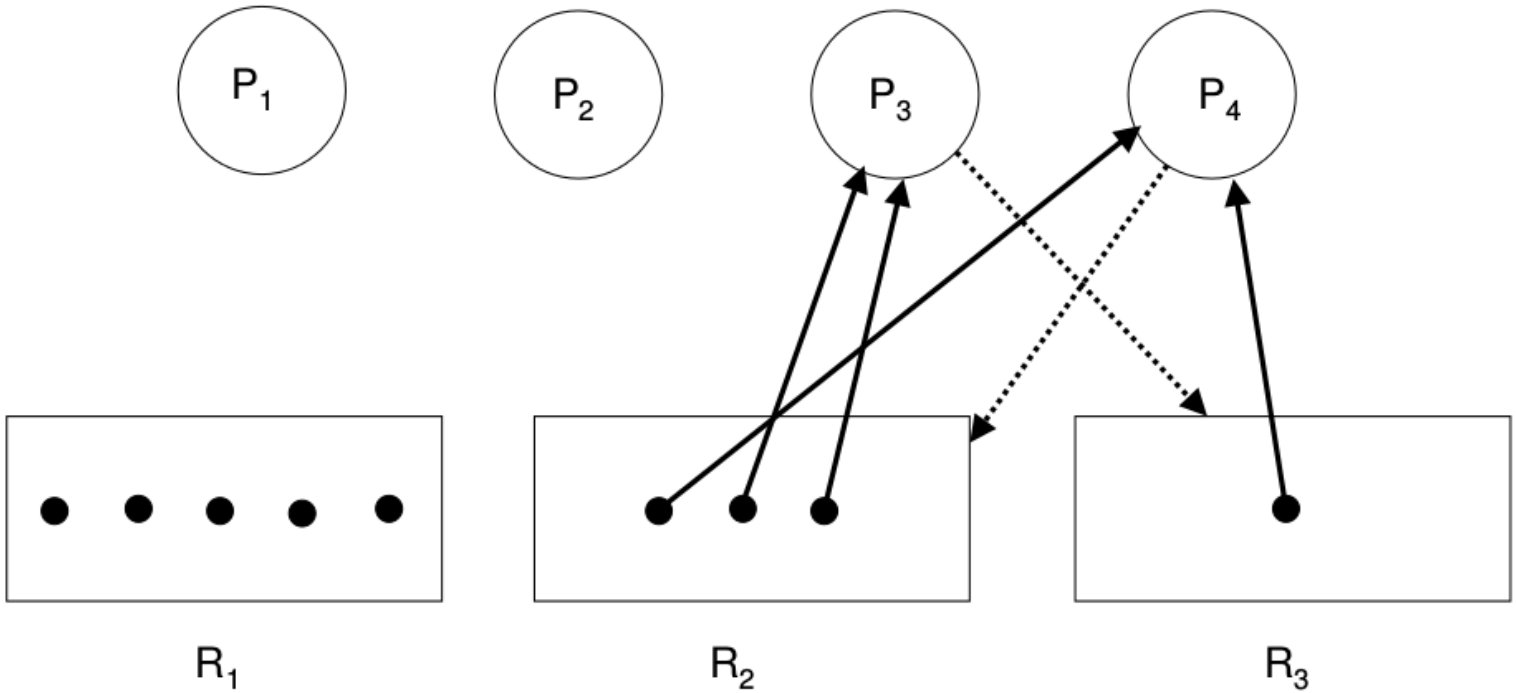


Figure – Graphe réduit d'allocation

Traitement des interblocages

Comme nous l'avons mentionné, les situations d'interblocage peuvent se produire dans un système. La question qui se pose est donc : doit-il prendre en compte ce problème ou l'ignorer ?

- **Ignorer complètement les problèmes** : On peut faire l'autruche et ignorer les possibilités d'interblocages. Cette « stratégie » est celle de la plupart des systèmes d'exploitation courants car le prix à payer pour les éviter est élevé.
- **Les détecter et y remédier** : On tente de traiter les interblocages, en détectant les processus inter-bloqués et en les éliminant.
- **Les éviter** : En allouant dynamiquement les ressources avec précaution.
- **Les prévenir** : En empêchant l'apparition de l'une des quatre conditions de leur existence.

Inter-blocage
Traitement des interblocages

La détection et la reprise

La démarche pour détecter les interblocages :

1. construit dynamiquement le graphe d'allocation des ressources,
2. Verifier les inter-blocage dans les deux cas :
   - des ressources à exemplaire unique : Si le graphe contient au moins un cycle
   - des ressources à exemplaires multiples : si le graphe contient au moins un cycle terminal (aucun arc ne permet de le quitter)

Inter-blocage
Traitement des interblocages

Algorithme de détection des interblocages

Soient n le nombre de processus P[1],...P[n] et m le nombre de type de ressources E[1],...,E[m] qui existent dans le système.

Traitement des interblocages

Algorithme de détection des interblocages

L'algorithme de détection des interblocages utilise les matrices et vecteurs suivants :

- Matrice C des allocations courantes d'ordre (n x m). L'élément C[i,j] désigne le nombre de ressources de type E[j] détenues par le processus P[i].
- Matrice R des demandes de ressources d'ordre (n x m). L'élément R[i,j] désigne e nombre de ressource de type E[j] qui manquent au processus P[i] pour pouvoir poursuivre son exécution.
- Vecteur A d'ordre m . L'élément A[j] est le nombre de ressources de type E[j] disponible (non allouées).
- Vecteur E d'ordre m . L'élément E[j] est le nombre total de ressources de type j.

Inter-blocage
Traitement des interblocages

Algorithme de détection des interblocages

1. Rechercher un processus P[i], non marqué dont la rangée R[i] est inférieur a A.
2. Si ce processus existe, ajouter la rangée C[i] à A, marquer le processus i et revenir a l'étape 1.
3. Si ce processus n'existe pas, les processus non marqués sont en inter- blocage. L'algorithme se termine.

Traitement des interblocages

Exemple

Considérons un système où nous avons trois processus en cours et qui dispose de 4 types de ressources : 4 dérouleurs de bandes, 2 scanners, 3 imprimantes et un lecteur de CD ROM. Les ressources détenues et demandées par les processus sont indiquées par les matrices C et R.

E=[4,2,3,1] ; A=[2,1,0,0]

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} ; R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Traitement des interblocages

Exemple (suite)

Seul le processus P3 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin puis libère ses ressources, ce qui donne : A=[2,2,2,0].

Ensuite P2 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin et libère ses ressources, ce qui donne : A=[4,2,2,1] . Enfin, P1 peut obtenir les ressources dont il a besoin. Il n'y a pas d'interblocage.

Transmission de messages

La transmission de messages est un mécanisme de communication entre les processus ; Chaque message véhicule des données. Un processus peut envoyer un message à un autre processus se trouvant sur la même machine ou sur des machines différentes.
Unix offre plusieurs mécanismes de communication pour l'envoie de messages : les tubes de communication sans nom, nommés et les sockets.

Figure – Processus en parallèle reliés par un tube de communication.

Transmission de messages
Tubes de communication (PIPE)

Les tubes de communication ou pipes permettent à deux ou plusieurs processus d'échanger des informations. On distingue deux types de tubes :

- les tubes sans nom (unamed pipe),
- les tubes nommés (named pipe).

Transmission de messages
Tubes de communication (PIPE)

Tubes sans nom Les tubes sans nom sont des liaisons unidirectionnelles de communication. La taille maximale d'un tube sans nom varie d'une version à une autre d'Unix, mais elle est approximativement égale à 4 Ko. Les tubes sans nom peuvent être créés par le shell ou par l'appel **système** *pipe()*.

Transmission de messages

Tubes de communication (PIPE)

Tubes sans nom

Création d'un tube sans nom

Un tube de communication sans nom est créé par l'appel système pipe, auquel on passe un tableau de deux entiers :

```
int pipe(int descripteur[2]) ;
```

Au retour de l'appel système pipe(), un tube aura été créé, et les deux positions du tableau passé en paramètre contiennent deux descripteurs de fichiers.

- le descripteur pour **les lectures** du tube, position **0**.
- le descripteur pour **les écritures** dans le tube, position **1**.

Remarque

Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube.

Transmission de messages

Tubes de communication (PIPE)

Tubes sans nom

Communication bidirectionnelle

La communication bidirectionnelle entre processus est possible en utilisant deux tubes : un pour chaque sens de communication.

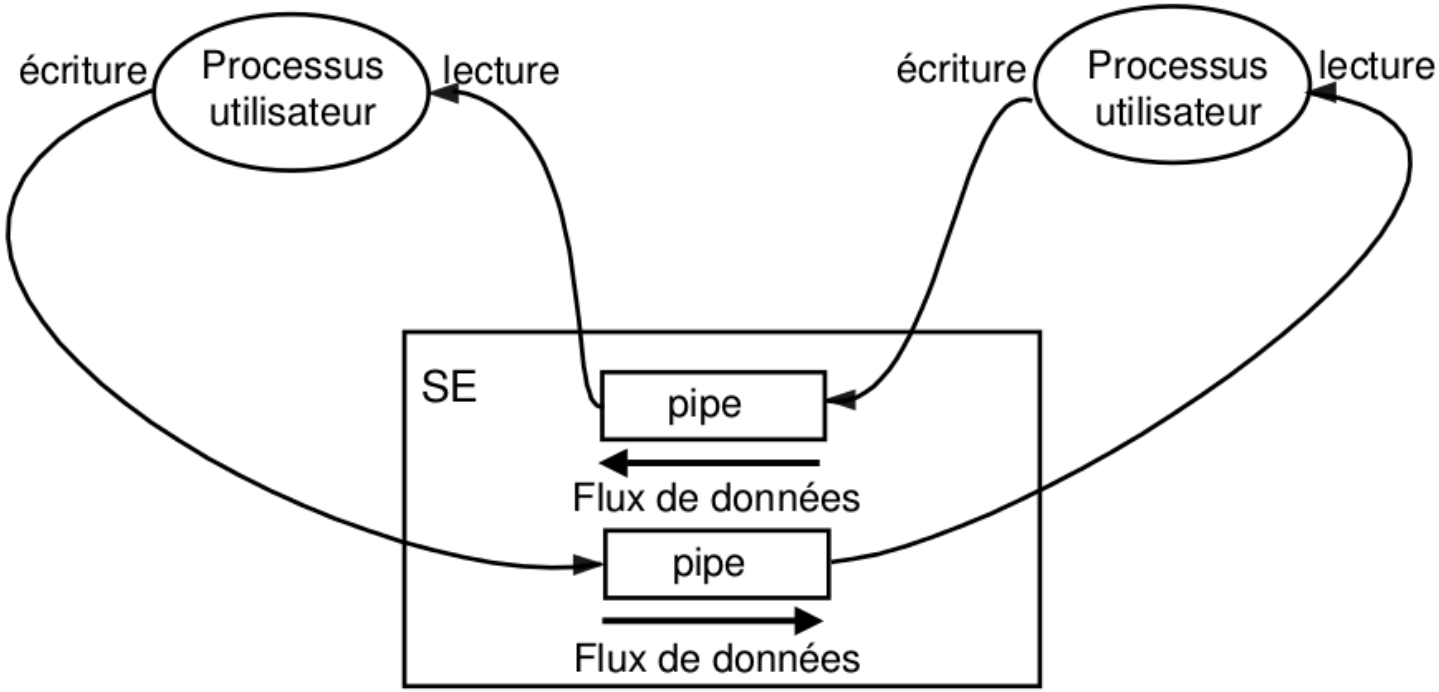


Figure – Création d'une communication bidirectionnelle entre deux

Transmission de messages
Tubes de communication (PIPE)

Tubes de communication nommés

Linux supporte un autre type de tubes de communication, beaucoup plus performants. Ils s'agit des tubes nommés (named pipes). Les tubes de communication nommés fonctionnent aussi comme des files du type FIFO (First In First Out). Ils sont plus intéressants que les tubes sans nom car ils offrent, en plus, les avantages suivants :

Transmission de messages
Tubes de communication (PIPE)

Tubes de communication nommés

1. Ils ont chacun un nom qui existe dans le système de fichiers (une entrée dans la Table des fichiers).
2. Ils sont considérés comme des fichiers spéciaux.
3. Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
4. Ils existeront jusqu'à ce qu'ils soient supprimés explicitement.
5. Ils sont de capacité plus grande, d'environ 40 Ko.

Transmission de messages
Tubes de communication (PIPE)

Tubes de communication nommés

Création d'un tube de communication nommé

Les tubes de communication nommés sont créés par la commande shell mkfifo ou par l'appel système mkfifo() :

```
int mkfifo(char *nom, mode_t mode);
```

où **nom** contient le nom du tube et **mode** indique les permissions à accorder au tube.

Transmission de messages

Tubes de communication (PIPE)

Tubes de communication nommés

Exemple de création d'un tube de communication nommé

writer.c (~/Bureau/test/tempsreel) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

writer.c

```c
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
int main(void){
int fd ;
char message[26];
sprintf(message,"bonjour du writer %d\n",getpid());
//ouverture du pipe "mypipe" en mode écriture
fd=open("mypipe", O_WRONLY);
printf("ici writer %d\n",getpid());
if(fd!=-1){
        write(fd,message,strlen(message)) ;
}else printf("désolé, le tube n'est pas disponible\n");
//Fermeture du tube
close(fd);
return 0;}
```

C Largeur des tabulations : 8 Lig 17, Col 11 INS

Transmission de messages

Tubes de communication (PIPE)

Tubes de communication nommés

Exemple de création d'un tube de communication nommé

reader.c (~/Bureau/test/tempsreel) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

reader.c

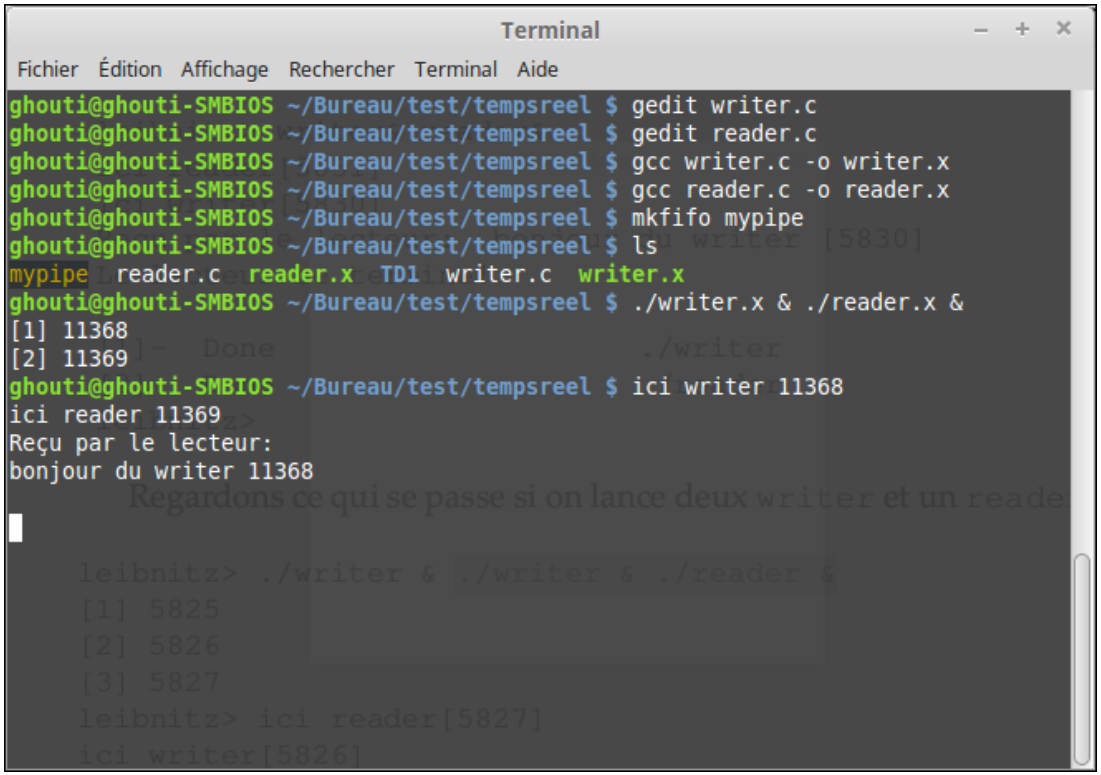
```c
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
int main(void){
int fd,n ;
char input;
//ouverture du pipe "mypipe" en mode lecture
fd=open("mypipe", O_RDONLY);
printf("ici reader %d\n",getpid());
if(fd!=-1){
        printf("Reçu par le lecteur:\n");
        while((n=read(fd,&input,1))>0){
        printf("%c",input);
        }
        printf("\n");
}else printf("désolé, le tube n'est pas disponible\n");
//Fermeture du tube
close(fd);
return 0;}
```

C Largeur des tabulations : 8 Lig 19, Col 11 INS

Transmission de messages
Tubes de communication (PIPE)

Tubes de communication nommés

Exemple de création d'un tube de communication nommé



Transmission de messages
Sockets

Les tubes de communication permettent d'établir une communication unidirectionnelle entre deux processus d'une même machine. Le système Unix/Linux permet la communication entre processus s'exécutant sur des machines différentes au moyen de **sockets** ou **prises**.

Interruptions

Interruptions

Une interruption est une commutation de mot d'état provoquée par un signal produit par le matériel.
Ce signal étant la conséquence d'un événement extérieur ou intérieur, il modifie l'état d'un indicateur qui est régulièrement testé par l'unité centrale. Une fois que le signal est détecté, il faut déterminer la cause de l'interruption. Pour cela on utilise un indicateur, pour les différentes causes, On parle alors du **vecteur d'interruptions.**

Interruptions

Trois grands types d'interruptions :

- **externes** (indépendantes du processus) interventions de l'opérateur, pannes,etc
- **déroutements** erreur interne du processeur, débordement, division par zéro, page fault etc (causes qui entraine la réalisation d'une sauvegarde sur disque de l'image mémoire "core dumped" en général)
- **appels systèmes** demande d'entrée-sortie par exemple.

Interruptions

Chaque machine possède un nombre de niveaux d'interruption, l'UNIX possède 6 niveaux d'interruptions représentés par le tableau 1

	Nature d'interruption	Fonction de traitement
0	horloge	clockintr
1	disques	diskintr
2	console	ttyintr
3	autres périphériques	devintr
4	appels système	sottintr
5	autres interruptions	otherintr

Table – Les niveaux d'interruption sous l'UNIX

Interruption d'horloge(Timer)
RTC et horloge système

Dans un système temps réel nous distinguons des mécanismes de gestion de temps :

- Real Time Clock (RTC) : La RTC garde la trace du temps quand le système est arrêté à l'aide d'une petite batterie située sur la carte mère, elle sert a initialiser une variable définie dans le système xtime (sched.h).
- Horloge système.

Interruption d'horloge(Timer)
Interruption du timer

Le noyau garde généralement la trace des intervalles de temps (habituellement pour entretenir divers timers appelés par des applications) à l'aide d'une interruption de timer produite par le timer matériel du système à des périodes préétablies, qui est définie par une variable HZ. Cette variable est définie en fonction de la valeur de HZ (qui peut être fixée pendant la durée de construction du noyau), la valeur de timer minimale de base (fréquence de base d'interruption d'horloge) prise en charge par le système peut changer (la valeur par défaut est 10 millisecondes).

Primitives temporelles
jiffies

jiffies est une variable système interne maintenue par le noyau, qui stocke le nombre de ticks d'horloge depuis la mise sous tension (démarrage) du système. Elle est définie dans <linux/-schedule.h>. Cependant, bien qu'elle soit définie comme volatile et longue, elle peut déborder en raison d'un temps utilisable continu (environ un an et demi).

Primitives temporelles
TimeStamp Counter (TSC)

La plupart des plates-formes prennent en charge quelques mécanismes de timer propres à l'architecture, ce qui aide les développeurs à avoir des intervalles de timer très courts. Une telle option est un registre de compteur de timer qui incrémente sa valeur après chaque cycle d'horloge. Ce peut être 32 bits ou 64 bits, selon la plate-forme. Le registre compteur le plus actualisé est le TSC (TimeStamp Counter, compteur d'estampilles temporelles) qui est maintenant disponible sur toutes les plates-formes majeures et peut être lu à la fois depuis l'espace utilisateur et l'espace du noyau. Les macros suivantes sont employées pour lire ce registre (#include <asm/msr.h>)

rtdsc(low,high) ; // Capture la valeur sur 64 bits en deux variables 32 bits
rtdscl(low) ; // Capture la valeur 32 bits inférieure

Primitives temporelles
Heure actuelle

La valeur de **jiffies** peut toujours donner le module, la valeur correcte de l'heure actuelle. les programmes système comme les pilotes de périphériques ou les modules du noyau exigent l'horloge système et non l'horloge murale. Il existe plusieurs fonctions qui permet d'avoir l'heure actuelle :

Primitives temporelles
Heure actuelle

void gettimeofday(struct timeval *tv) ;

Elle remplit une structure *timeval* avec l'heure actuelle en une valeur en secondes et micro-secondes.Elle est définie dans (#include <sys/time.h>).

```
1  #include <sys/time.h>
2  #include <stdio.h>
3
4  int main() {
5    struct timeval current_time;
6    gettimeofday(&current_time, NULL);
7    printf("seconds : %ld\nmicro seconds : %ld",
8      current_time.tv_sec, current_time.tv_usec);
9
10   return 0;
11 }
```

Code source 5 – Exemple de la fonction gettimeofday

void get_fast_time (struct timeval *tv) ;

Une autre option pour obtenir l'heure actuelle d'une manière rapide et efficace.

Primitives temporelles

Timer

Le timer est un mécanisme permettant d'assurer la fonctionnalité temporelle vitale dans tout module. La structure de données d'un timer ressemble à cela (#<linux/timer.h>) :

```
struct timer_list
{
        struct timer_list *next; // pointeur vers le prochain timer de la liste
        struct timer_list *prev; // pointeur sur le précédent timer de la liste
        unsigned long expires; // valeurs de temps imparti en jiffies
        unsigned long data; // argument pour le gestionnaire
        void (*function) (unsigned long); // fonction du gestionnaire
        volatile int running; // drapeau désignant l'état du nœud
};
```

Primitives temporelles
Timer

void init_timer (struct timer_list* timer) ;

Cette fonction initialise un timer nouvellement alloué.

void add_timer (struct timer_list* timer) ;

Cette fonction insère le timer nouvellement allouée dans la liste globale des timers actifs.

void mod_timer (struct timer_list* timer, unsigned long expires) ;

Cette fonction sert à modifier la valeur d'un timer actif.

int del_timer (struct timer_list* timer) ;

Cette fonction sert à supprimer le timer spécifié de la liste des timers actifs.

int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid);

clokid :

- **CLOCK_REALTIME** : Une horloge temps réel configurable à l'échelle du système.
- **CLOCK_MONOTONIC** : Une horloge non configurable, toujours croissante qui mesure le temps depuis un instant non spécifié dans le passé et qui ne change pas après le démarrage du système.
- **CLOCK_PROCESS_CPUTIME_ID** : Une horloge qui mesure le temps CPU (utilisateur et système) consommé par le processus appelant (et tous ses threads).
- **CLOCK_THREAD_CPUTIME_ID** : Une horloge qui mesure le temps CPU (utilisateur et système) consommé par le processus appelant.

Tâches périodiques

Lors du développement d'une application temps réel (en mode utilisateur), il est parfois nécessaire de déclencher une action avec une période précise. L'usage d'**usleep()** ou **nanosleep()** ne permet pas de respecter le temps réel et il faudra donc utiliser un timer **POSIX**. Si l'application est *multithread*, il faudra prêter une attention particulière à la configuration des signaux sous peine de conséquences sur l'ordonnancement.

Tâches périodiques
usleep et nanosleep

Supposons que nous souhaitons effectuer une action toutes les millisecondes. Il pourrait sembler évident d'écrire un code comme celui-ci :

```c
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main() {
   struct itimerspec tv,time;
gettimeofday(&tv, NULL);
while (1)
{
   tv.nsec += 1000000; // calcul de l'heure à attendre
   gettimeofday(&time);
   nanosleep(tv.nsec - time.nsec); // attente de l'échéance
   /* Exécution du code périodique */
}

   return 0;
}
```

Code source 6 – Exemple de la fonction nanosleep

Tâches périodiques
usleep et nanosleep

Pour une application temps réel, ce code n'est cependant pas correct. En effet, celle-ci pourrait être ordonnancée entre les appels à gettimeofday() et nanosleep() et reprendrait son exécution après une durée indéterminée pour appeler nanosleep() avec un temps d'attente trop important.

Tâches périodiques
Timer POSIX

Afin de respecter la période, il est donc nécessaire d'utiliser les timers POSIX. Contrairement aux fonctions du type sleep(), un timer va émettre un signal à l'application elle-même qui peut le traiter de deux façons :

- en déclarant un handler pour le signal concerné (man sigaction)
- en attendant la réception du signal (fonctions sigwaitinfo ou sigtimedwait)

Tâches périodiques
La norme POSIX

La norme POSIX introduit les ensembles de signaux de type sigset_t et sont manipulables grâce aux fonctions suivantes :

```
int sigemptyset(sigset_t *ens) /* raz */
int sigfillset(sigset_t *ens) /* ens = { 1,2,...,NSIG} */
int sigaddset(sigset_t *ens, int sig) /* ens = ens + {sig} */
int sigdelset(sigset_t *ens, int sig) /* ens = ens - {sig} */
```

Ces fonctions retournent -1 en cas d'échec et 0 sinon.

```
int sigismember(sigset_t *ens, int sig) ;/* sig appartient à ens ?*/
```

retourne vrai si le signal appartient 'a l'ensemble.

Tâches périodiques
La norme POSIX

Le blocage des signaux

La fonction suivante permet de manipuler le masque de signaux du processus :

```
#include <signal.h>
int sigprocmask(int op, const sigset_t* nouv, sigset_t
*anc) ;
```

L'opération op

- **SIG_SETMASK** affectation du nouveau masque, récupération de la valeur de l'ancien masque.
- **SIG_BLOCK** union des deux ensembles nouv et anc
- **SIG_UNBLOCK** soustraction anc - nouv

Tâches périodiques

Timer POSIX (Exemple *sigwaitinfo*)

```c
#include <signal.h>
#include <time.h>
#include <stdio.h>

#define PERIOD_SIG      SIGRTMIN
#define PERIOD_SEC      0
#define PERIOD_NSEC     1000000 // 1ms

#define TIMER_SIG SIGRTMAX //Last real-time signal (see kill -l)

int main(int argc, char *argv[])
{
  sigset_t mask;  //signal set
  timer_t timerid; //timer
  struct itimerspec its; //timer specification interval
  struct sigevent evp = { //signal event
    .sigev_notify = SIGEV_SIGNAL, //Notification method: Notify the process by sending the signal specified in sigev_signo.
    .sigev_signo = TIMER_SIG, //Notification signal: Last real-time signal
  };
  siginfo_t si;
  int sig;

  sigemptyset(&mask);
  sigaddset(&mask, TIMER_SIG);
  sigprocmask(SIG_BLOCK, &mask, NULL); //Used to fetch and/or change the signal mask of the calling thread.

  timer_create(CLOCK_REALTIME, &evp, &timerid); //creates a new per-process interval timer.
```

```
1
2     its.it_value.tv_sec = PERIOD_SEC;
3     its.it_value.tv_nsec = PERIOD_NSEC;
4     its.it_interval.tv_sec = PERIOD_SEC;
5     its.it_interval.tv_nsec = PERIOD_NSEC;
6     timer_settime(timerid, 0, &its, NULL);
7
8     while (1)
9     {
10      sig = sigwaitinfo(&mask, &si);
11      if (sig != TIMER_SIG)
12        continue;
13      /* Periodic task... */
14    printf("Periodic task...\n");
15    }
16
17    return 0;
18  }
```

Code source 7 – Exemple de message entre processus

Commande par ordinateur

Commande par ordinateur

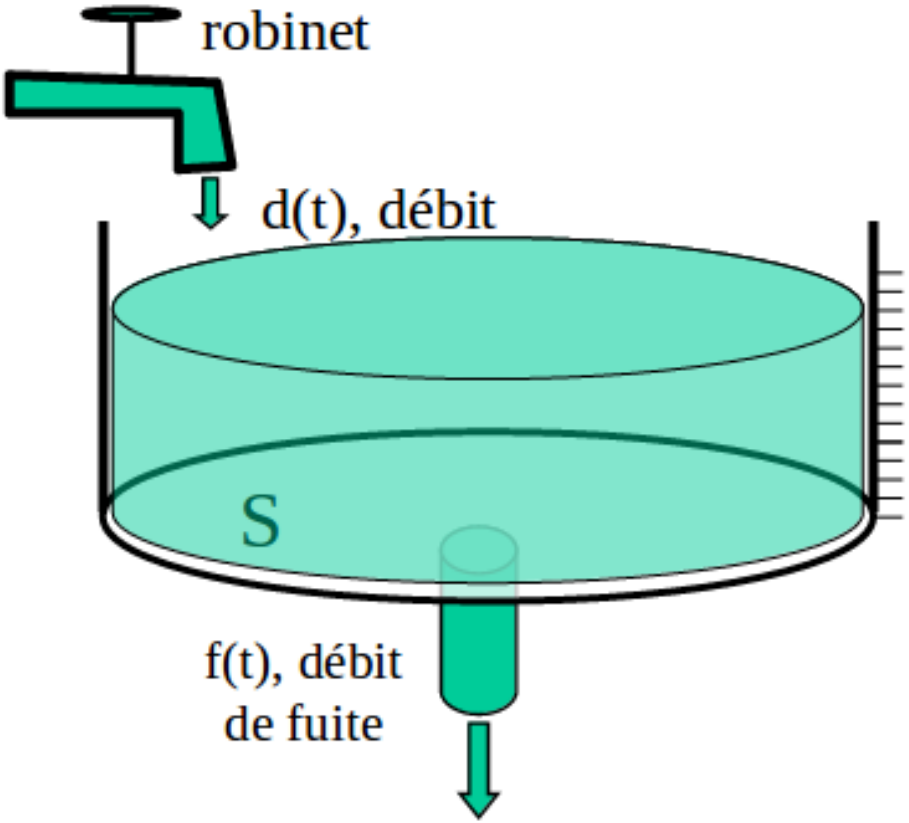
comment un ordinateur embarqué peut-il prendre les commandes d'un processus physique ?

Commande par ordinateur

Les notions de base pour la commande par ordinateur :

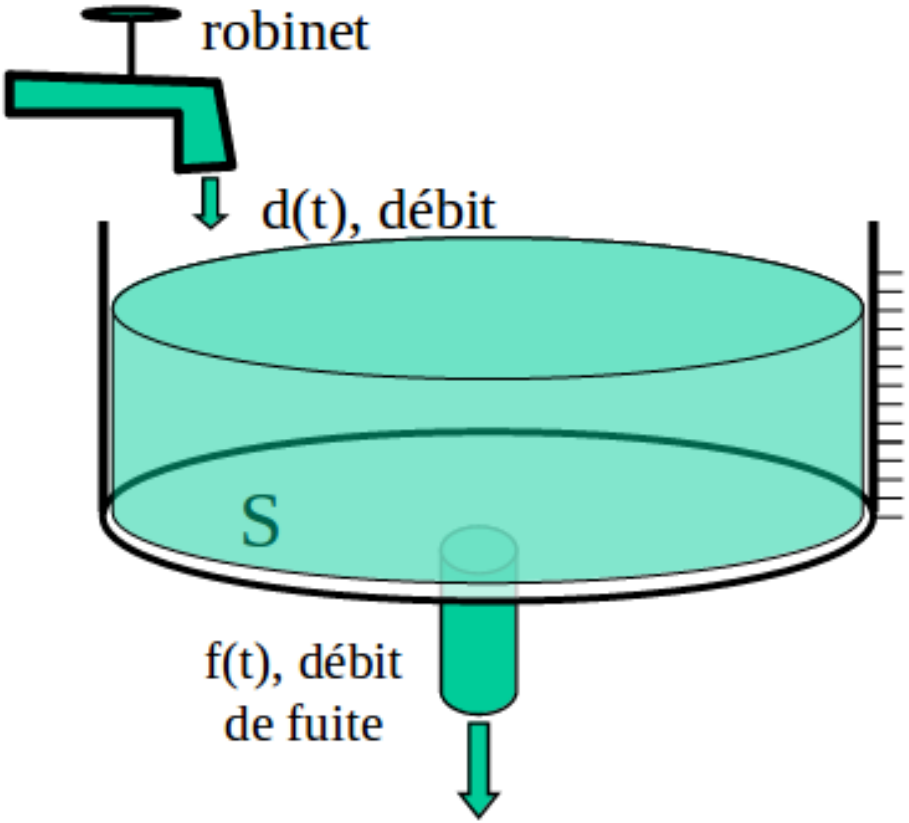
- **Processus physique** : entrées, sorties, équations, temps continu
- **Système asservi** : capteurs, actionneurs, loi de contre réaction
- **Discrétisation** : échantillonnage, blocage, temps discret
- **Loi de commande** : boucle ouverte, retour d'état, commande optimale
- **Performance** : rapidité, précision, stabilité, consommation

Commande par ordinateur
Exemple d'un processus physique



- Processus physique : remplissage d'une cuve avec fuite
- Entrées sorties : d(t) entrée, f(t) perturbation, h(t) sortie
- Capteur : la règle
- Actionneur : le robinet
- Équation : la variation du volume V=S*h dans la cuve est la différence d-f

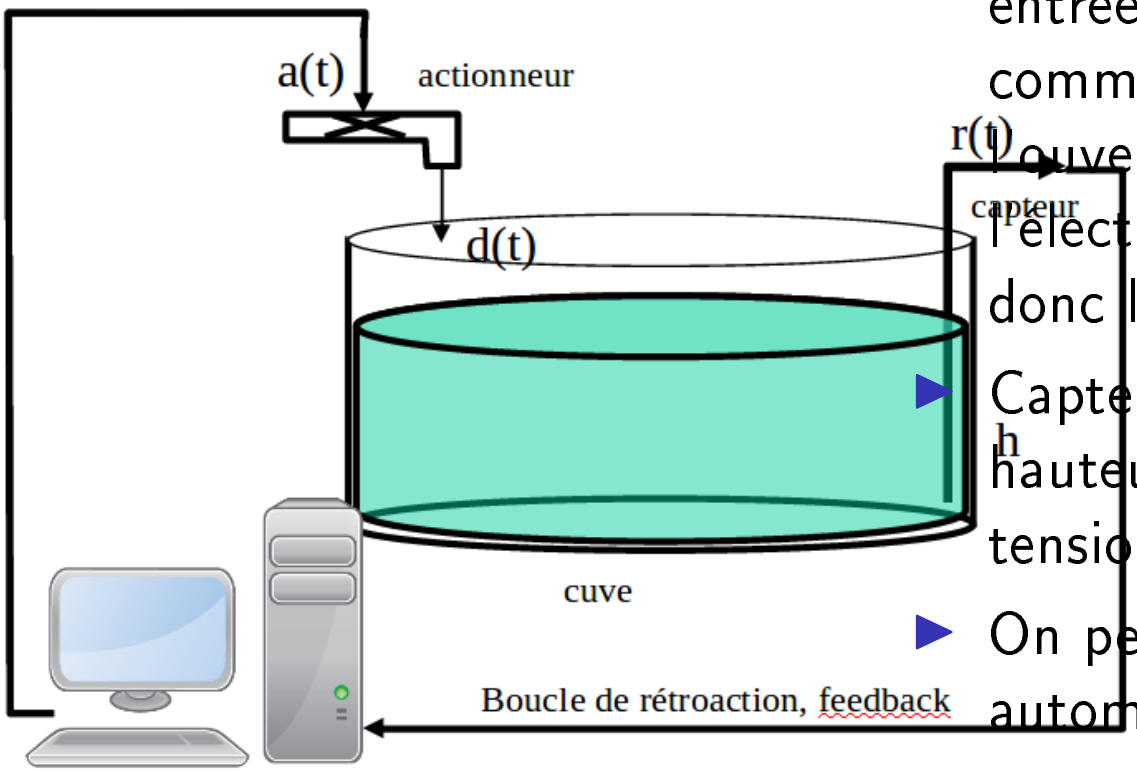
Commande par ordinateur
Exemple d'un processus physique



- Processus physique : remplissage d'une cuve avec fuite
- Entrées sorties : d(t) entrée, f(t) perturbation, h(t) sortie
- Capteur : la règle
- Actionneur : le robinet
- Équation : la variation du volume V=S*h dans la cuve est la différence d-f

Commande par ordinateur

Système asservi : Exemple d'un processus physique automatisé



- Actionneur : électrovanne
- Commande a(t) : entrée électrique qui commande l'ouverture de l'électrovanne et donc le débit d(t)
- Capteur : convertit hauteur h(t) en tension r(t)
- On peut maintenant automatiser le processus On fait pour simplifier :

Commande par ordinateur
Système asservi : Exemple d'un processus physique automatisé

Soit h_c, la hauteur de liquide souhaitée, ou consigne, ou encore référence :

- Loi de commande par tout ou rien (non linéaire) :
  $h(t) < h_c \Rightarrow d(t) = d_{max}$
  $h(t) \geq h_c \Rightarrow d(t) = 0$
- Loi de commande linéaire :
  $d(t) = Kx(h_c X h(t))$
  $d = Kx(h_c - h)$
  - d(t) est la commande
  - K : est le gain de contre réaction
  - $h_c - h$ est l'erreur d'asservissement
  - $h(t)$ est le retour

Principaux rôles de la commande par ordinateur

- Se substituer au correcteur analogique en apportant plus de souplesse dans le réglage de la loi de commande
- Exécuter le programme moniteur : gestion des tâches
- Fournir un journal de bord sur l'état du processus à commander : Supervision
- Optimisation de fonctionnement

Commande par ordinateur
Système asservi : Exemple d'un processus physique automatisé

Lorsqu'on décide de contrôler le processus physique par insertion *des calculateurs numériques* (**ordinateur**), la nécessité d'échantillonnage dans le contrôle automatisé des processus industriels (nature analogique) apparaît clairement
La communication entre le calculateur numérique et le processus à commander de nature analogique passe à travers deux convertisseurs : le convertisseur numérique analogique et le convertisseur analogique numérique.

Commande par ordinateur
Système asservi : Exemple d'un processus physique automatisé

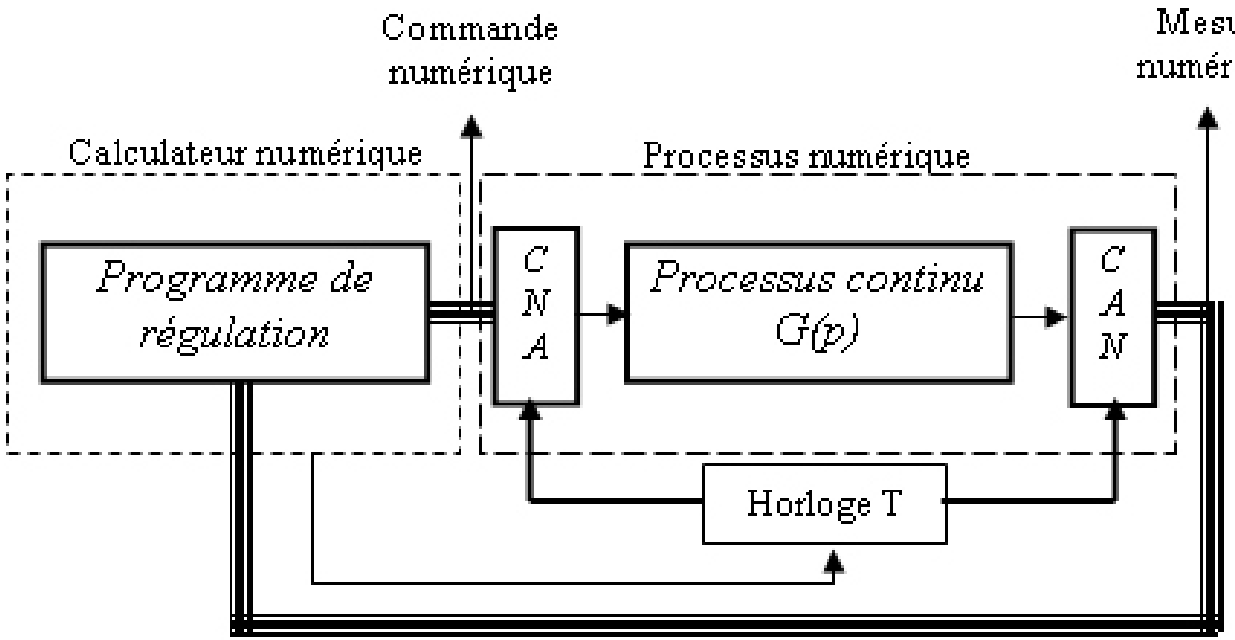


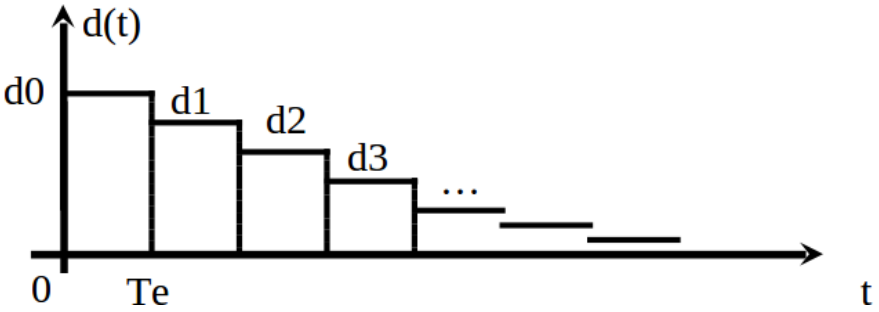
Figure – Schéma de commande en boucle fermée par calculateur numérique

- CAN= Convertisseur analogique numérique
- CNA= Convertisseur numérique analogique

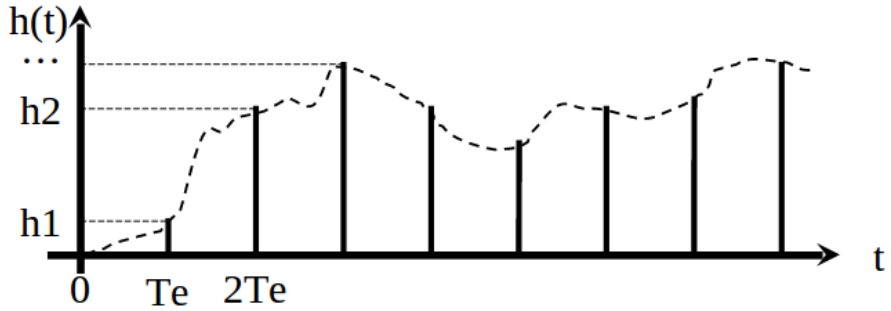
Échantillonnage des systèmes linéaires

Commande par ordinateur implique :

- Rythmée par une période d'échantillonnage T_e,
- blocage d'ordre zéro (BoZ) de l'entrée de commande d,



- échantillonnage de la sortie h(t)



Échantillonnage des systèmes linéaires
Choix de la période d'échantillonnage

Le pas d'échantillonnage T_e est un paramètre nouveau et fondamental dans la conception d'une commande par calculateur numérique. A des instants réguliers appelés instants d'échantillonnage, il y'a prélèvement d'une mesure et émission d'une commande. Ces deux opérations sont supposées synchrones. Dans tous les cas la période doit respectée le théorème de Shannon :

$$T_e < \frac{T_{max}}{2}$$
$$T_{max} = 2\pi/\omega_{max}$$

ω_{max}la plus haute pulsation contenue dans le signal à échantillonner.

Échantillonnage des systèmes linéaires
Choix de la période d'échantillonnage

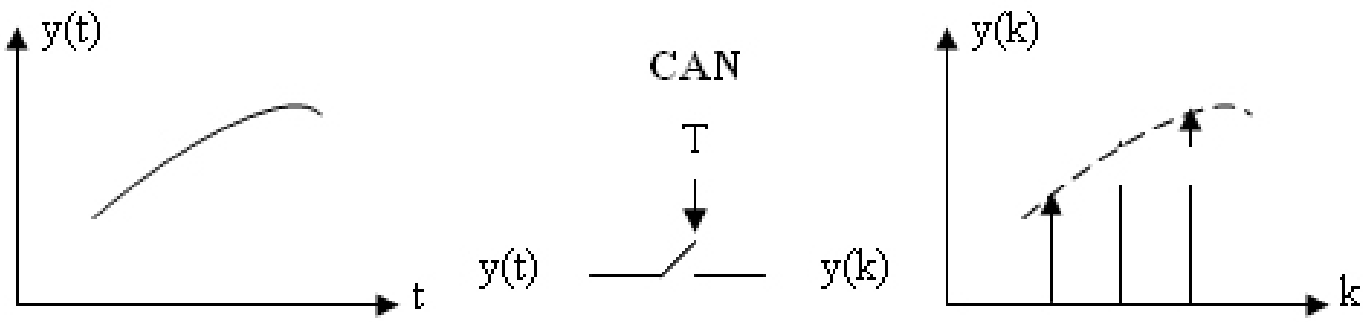
Quelques recommandations pour la période d'échantillonnage :

Type de variable physique	Période d'échantillonnage recommandée
Organe électrique	$T_e = 0.0001s$à 1s
Débit hydraulique	$T_e = 1s$
Niveau, pression	$T_e = 1s$ à 5s
Température	$T_e = 20s$

Échantillonnage des systèmes linéaires
Conversion analogique numérique et numérique analogique

Modélisation de CAN

D'un point de vue modélisation, l'ensemble capteur convertisseur analogique-numérique peut être assimilé à une prise d'échantillons de la sortie y(t) à période fixe T. Si l'on fait l'hypothèse que l'échantillonnage est instantané (le temps de codage est négligeable) et qu'il n'y a pas d'erreur de quantification, on peut représenter l'opération de conversion analogique-numérique par le schéma suivant :

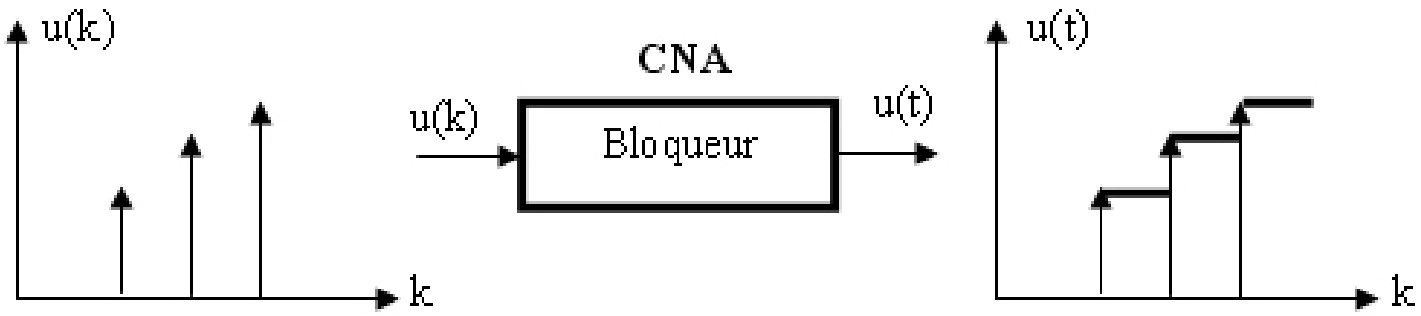


Échantillonnage des systèmes linéaires
Conversion analogique numérique et numérique analogique

Modélisation de CNA

Le calculateur numérique élaborant la commande à appliquer au procédé travaille de manière séquentielle et génère des valeurs numériques **u(k)** avec la même période **T** que celle qui a été choisie pour l'échantillonnage de la mesure.

L'opération de *conversion numérique-analogique* selon la plus courante consiste à produire un signal de commande continu par morceaux (en **escalier**) à partir des valeurs **u(k)** comme le montre la figure suivante :

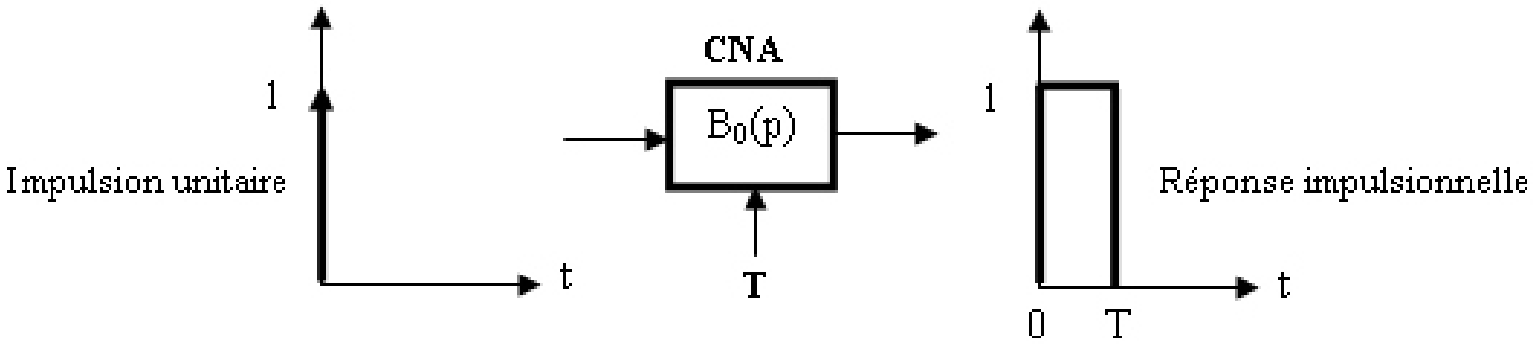


Échantillonnage des systèmes linéaires

Conversion analogique numérique et numérique analogique

Le bloqueur n'est pas un composant physique dans la chaîne de commande. Le blocage se fait par programme en maintenant la valeur de la commande u(k) à l'entrée du CNA pendant toute la période d'échantillonnage.

Ainsi, le calculateur numérique "voit" non pas le système continu à commander de fonction de transfert G(p) par exemple, mais un système constitué par la cascade bloqueur et le système à commander proprement dit.



La commande par ordinateur
Conclusion

Pour élaborer une loi de commande numérique :

- Choisir un pas d'échantillonnage ;
- Déterminer la transmittance G(z) du processus à commander ;
- Calculer la loi de commande par l'une des méthodes de synthèse des correcteurs numériques ;
- Programmer la loi de commande.