Systèmes temps réels (I)

Claude Guéganno

8 octobre 2001

Table des matières

Première partie

Des procédures aux processus

Chapitre 1

Le parallélisme

1.1 Les architectures parallèles

Le parallélisme n'est pas une notion récente en informatique. Il en existe différentes formes dans les ordinateurs. Par exemple, une impression de fichier qui s'opére sans bloquer le système, des accès aux disques de données gérés par des contrôleurs, pendant que le processeur travaille ... Ces formes restent transparentes pour l'utilisateur et visent à mieux tirer partie de la machine.

Au delà de ce parallélisme «matériel», est apparut un parallélisme «logiciel» : la programmation concurrente, dont le but est d'exécuter plusieurs tâches simultanément ou quasi-simultanément.

Parallélisme = coopération, au sein d'une même application de tâches s'exécutant en parallèle.

On divise les architectures parallèles en trois catégories :

- Les machines S.I.S.D. (*Single Instruction, Single Data*), ou un processeur unique exécute à un instant donné une seule instruction sur une donnée unique. Ce sont les architectures classiques. Il ne peut s'agir dans ce cas de vrai parallélisme, mais de parallélisme simulé. La mise en œuvre du parallélisme sur ce type de machines fera l'objet de l'étude des systèmes d'exploitation multi-tâches.
- Les machines S.I.M.D. (*Single Instruction, Multiple Data*), où les instructions portent sur des ensembles de données. (vecteurs ou matrices). La même instruction est distribuée sur plusieurs processeurs identiques qui traitent chacun une donnée différente. Ces machines sont particulièrement adaptées aux calculs scientifiques : traitements d'images, météorologie, simulations, ...
- Les machines M.I.M.D. (*Multiple Instruction, Multiple Data*), où plusieurs processeurs traitent simultanément des parties de programmes différents agissants eux mêmes sur des données différentes. Dans cette catégorie, on peut séparer :
  - les machines à **couplage fort**, lorsque les processeurs se partagent

la mémoire. C'est le cas des architectures multiprocesseurs classiques où toutes les ressources (mémoires, E/S) sont accessibles à tous les processeurs et sont allouées dynamiquement. Il peut y avoir un processeur maître qui coordonne l'ensemble.

- les machines à **couplage lâche**, où les processeurs sont dotés de mémoire locale. Ils coopèrent alors par un réseau qui les relie:
  - de type *bus* [1] : machines «raisonnablement» parallèles,
  - de type matrice ou hypercube [2] pour les machines «massivement» parallèles.
  - réseau point à point reliant des unités centrales distinctes : on parle alors d'architecture réparties. Pour ce dernier type d'architectures, le langage `java` propose une solution pour l'écriture d'applications réparties avec ses `R.M.I.` [3].

Dans ce cours nous nous limiterons à l'étude du parallélisme sur machine S.I.S.D.

1.2 Procédures

Les procédures, ou fonctions, permettent de regrouper, de nommer et de paramétrer un ensemble d'instructions opérant sur des données communes. On appelle **contexte** d'une procédure, l'ensemble des objets accessibles par elle (variables locales et variables globales) , ainsi que l'ensemble des informations qui caractérisent son état (valeur prises par les variables, instruction en cours, ...).

Lors de l'appel d'une procédure, sont réalisés :

- la création de son contexte (les variables locales prennent physiquement leur place sur la pile)
- le passage des paramètres (en général sur la pile, parfois dans des registres du processeur)
- l'activation de la procédure. Le compteur programme pointe sur sa première instruction.

Lors du retour de la procédure sont réalisés :

- le passage du résultat (en général dans un registre du processeur)
- la destruction du contexte de la procédure quittée (le pointeur de pile reprend sa position initiale en rendant disponible la zone occupée par les variables locales de la procédure)
- le retour du contrôle à l'appelant (le compteur programme ne pointe plus sur une instruction de la procédure)

1. Un seul ensemble de lignes relie l'ensemble des processeurs
2. Un processeur est relié à plusieurs autres
3. `Remote Method Invocation`

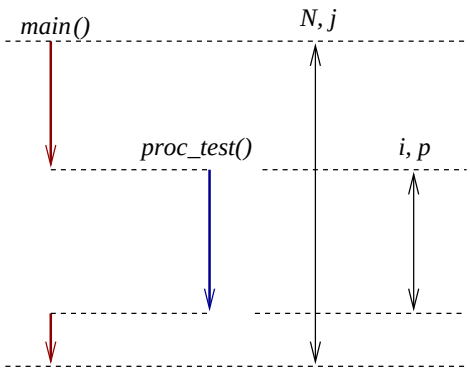
Exemple

```
#include <stdio.h>

int N;

int proc_test(int p) {
  int i;
  for (i=0; i<N+p; i++) printf("K");
  printf("\n");
  return(i);
}

void main() {
  int j;
  N = 12;
  j = proc_test(2);
  printf("Retour = %d\n", j);
}
```



Exécution :

```
% a.out
KKKKKKKKKKKKKK
Retour = 14
%
```

† les variables p et i n'existent que pendant l'exécution de la procédure `proc_test`.

† l'exécution de la procédure est matérialisée par l'affichage des lettres `K`

1.3 Coroutines

Dans ce mode de structuration, on considère que le système est constitué de plusieurs composants de même niveau et de même durée de vie qui coopèrent. Chaque composant peut décider lui même de s'endormir ou de lancer un aute composant à partir du point ou il s'était précédemment interrompu.

Il s'agit d'une généralisation du système de procédures dans le sens où:

→ le nombre d'interlocuteurs n'est pas limité à 2,

→ les contextes sont sauvegardés quand un coprogramme passe le contrôle à un autre,

→ le contrôle n'est pas systématiquement rendu à l'appelant.

C'est une structure symétrique où les opérations d'appel et de retour sont similaires.

Exemple. Le programme `Modula2` suivant met en œuvre quatre coroutines. Les trois premiers: `proc1`, `proc2` et `proc3` jouent des rôles similaires.

Ils bouclent en affichant à chaque fois leur identité ainsi qu'un compteur. Le processus principal est l'ordonnaceur. C'est lui qui détermine laquelle des co-routines `proc1`, `proc2` ou `proc3` aura le processeur.

```
MODULE ex1;

FROM InOut IMPORT WriteString,
                  WriteLn,
                  WriteCard;
FROM SYSTEM IMPORT ADDRESS,
                   NEWPROCESS,
                   TRANSFER;
FROM Storage IMPORT ALLOCATE;

VAR mem           :POINTER TO CHAR;
    p0, p1,p2, p3 :ADDRESS;

(* ----------------------------- *)
PROCEDURE proc1;
VAR i: CARDINAL;
BEGIN
  i := 1;
  LOOP
    WriteString("[p1:");
    WriteCard(i,2);
    WriteString("] ");
    INC(i);
    TRANSFER(p1, p0);
  END;
END proc1;

(* ----------------------------- *)
PROCEDURE proc2;
VAR i: CARDINAL;
BEGIN
  i := 1;
  LOOP
    WriteString("[p2:");
    WriteCard(i,2);
    WriteString("] ");
    INC(i);
    TRANSFER(p2, p0);
  END;
END proc2;

(* ------------------- *)
PROCEDURE proc3;
VAR i: CARDINAL;
BEGIN
  i := 1;
  LOOP
    WriteString("[p3:");
    WriteCard(i,2);
    WriteString("] ");
    INC(i);
    TRANSFER(p3, p0);
  END;
END proc3;

(* ------------------- *)
(* processus principal *)
VAR
  k:CARDINAL;
BEGIN
   WriteString ("Début...");
   WriteLn;
   ALLOCATE(mem, 512 );
   NEWPROCESS(proc1,mem,512,p1);
   ALLOCATE(mem,512);
   NEWPROCESS(proc2,mem,512,p2);
   ALLOCATE(mem,512);
   NEWPROCESS(proc3,mem,512,p3);
   FOR k:=1 TO 10 DO
     TRANSFER(p0, p1);
     TRANSFER(p0, p2);
     TRANSFER(p0, p1);
     TRANSFER(p0, p2);
     TRANSFER(p0, p1);
     TRANSFER(p0, p3);
   END;
   WriteLn;
   WriteString ("Fin...");
   WriteLn;
END ex1.
```

Trace d'exécution

```
> ex1
Début...
[p1: 1] [p2: 1] [p1: 2] [p2: 2] [p1: 3] [p3: 1] [p1: 4] [p2: 3] [p1: 5]
[p2: 4] [p1: 6] [p3: 2] [p1: 7] [p2: 5] [p1: 8] [p2: 6] [p1: 9] [p3: 3]
[p1:10] [p2: 7] [p1:11] [p2: 8] [p1:12] [p3: 4] [p1:13] [p2: 9] [p1:14]
[p2:10] [p1:15] [p3: 5] [p1:16] [p2:11] [p1:17] [p2:12] [p1:18] [p3: 6]
[p1:19] [p2:13] [p1:20] [p2:14] [p1:21] [p3: 7] [p1:22] [p2:15] [p1:23]
[p2:16] [p1:24] [p3: 8] [p1:25] [p2:17] [p1:26] [p2:18] [p1:27] [p3: 9]
```

```
[p1:28] [p2:19] [p1:29] [p2:20] [p1:30] [p3:10]
Fin...
```

Cet affichage est bien entendu cohérent avec la répartition de l'ordonnanceur. D'autre part, on vérifie bien que les co-routines conservent la valeur courante de leur variable locale.

1.4 Processus

Le concept de processus a été introduit à l'origine pour formaliser la notion d'activité parallèle dans les systèmes d'exploitation (travaux des utilisateurs ou des périphériques, ...). Le comportement global du système était décrit comme la coopération de ces divers processus synchronisés. Peu à peu, le concept s'est intégré aux langages d'écriture de systèmes.

1.4.1 Mise en œuvre des processus

Les premières primitives de gestion des processus furent `join` et `fork`. Un processus père p créée par `fork` un processus fils q. Le fils se termine par une primitive `exit` qui provoque sa disparition. Par la primitive `join(q)`, le père se bloque tant que q n'a pas exécuté `exit`. Ces primitives présentent l'avantage de la simplicité, mais donnent aux programmes un manque de lisibilité et de structuration.

L'exemple suivant montre la mise en œuvre des processus sous `UNIX`. La primitive `join(q)` est remplacée par la primitive `wait` qui synchronise un processus père sur la mort de l'un de ses fils (sans savoir lequel des fils).

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

pid_t q1, q2, q;
int status;

int main() {
  printf("Père : début\n");
  q1 = fork();
  if (q1 == 0) {
    printf("Fils 1 : début\n");   sleep(2);
    printf("Fils 1 : fin\n");
    exit(0);
  }
  else {
    printf("Père : 1\n");
      q2 = fork();
      if (q2 == 0) {
        printf("Fils 2 : début\n"); sleep(3);
        printf("Fils 2 : fin\n");
        exit(0);
```

```
      }
      else {
        printf("Père : 2\n");
        q = wait(&status);
        printf("Père : fin du fils %d\n",q==q1? 1 : 2);
        q = wait(&status);
        printf("Père : fin du fils %d\n",q==q1? 1 : 2);
        printf("Père : fin du programme\n");
      }
  }
}
```

Trace d'exécution

```
> a.out
Père : début
Père : 1
Fils 1 : début
Père : 2
Fils 2 : début
Fils 1 : fin
Père : fin du fils 1
Fils 2 : fin
Père : fin du fils 2
Père : fin du programme
> _
```

1.4.2 Langages concurrents

Les déclarations de processus et l'activation par une instruction spécifique se sont imposés dans beaucoup de langages. `Ada` , par exemple permet de déclarer des type processus (`task`) et des objets de ces types. Leur activation est implicite lors de l'élaboration des déclarations d'objets, ou explicite si l'on a créé un «type accès» pour pointer vers des tâches. Les tâches sont alors lancées par utilisation de l'allocateur `new`

Il ne faut pas confondre «programmation concurrente» et architecture parallèle». Ce sont des concepts différents. Le parallélisme, au niveau matériel, sous entend que des opérations seront effectuées simultanément (rigoureusement). Des séquences d'instructions séquentielles seront dites concurrentes si elles peuvent être exécutées en parallèle, mais sans pour autant qu'elles le soient. On peut concevoir une application en utilisant les spécificités d'un langage concurrent sans que le programme s'exécute sur un architecture parallèle[4].

La notion de **processus** est fondamentale en programmation concurrente.

4. On peut également exécuter un programme non concurrent sur une machine parallèle.

Processus = exécution séquentielle de code; un processus possède son propre *thread of control.*
thread of control = succession des contextes d'exécution atteints lors de l'exécution des différentes lignes de code du processus.
contexte d'exécution = toutes les valeurs contenues dans les registres du processeur et dans les variables du programme à un instant donné de son exécution.

Tâches concurrentes avec `Ada`

Le langage `Ada` propose des solutions adaptées aux problèmes de la programmation concurrente. L'aspect séquentiel de `Ada` est similaire à `Modula2` .

Un programme `Ada` est constitué par la structure globale suivante :

```
procedure <nom>
   <declarations>
begin
   <instructions>
end nom;
```

Exemple. le programme suivant affiche à l'écran le message

`Ada est un langage concurrent`

```
with Text_Io; use Text_Io; -- importe Put_Line

procedure Hello is
begin
   Put_Line("Ada est un langage concurrent");
end Hello;
```

Remarques :

- la séquence `--` indique un commentaire qui va jusqu'à la fin de la ligne;
- `Ada` ne fait pas la distinction entre les lettre majuscules ou minuscules.

Les processus apparaissent en `Ada` sous le nom `task` . Un processus `Ada` est déclaré en deux parties :

1. une spécification
2. un corps (`body`).

Une tâche qui n'est synchronisée avec aucune autre (donc, complètement indépendante) est simplement spécifiée par :

`task` <*nom*>;

Le corps de la tâche est composé de déclarations optionnelles et d'une séquence d'instructions entre `begin` et `end` . Le corps est l'implémentation de la tâche.

Exemple. Ce premier exemple montre la création de trois processus constituant un système **statique**, c'est à dire que le nombre de processus est déterminé au moment de l'initialisation de l'application.

```
with Text_Io; use Text_Io;

procedure Multi is

   task P;  -- spécification de P
   task body P is  -- corps de P
      begin
         Put_Line("p");
      end P;

   task Q;
   task body Q is
      begin
         Put_Line("q");
      end Q;

begin -- corps de la procédure
   Put_Line("r");
end Multi;
```

Le corps de la procédure est considéré comme le processus «parent» des processus P et Q. Les processus P, Q et le processus parent s'exécutent en concurrence. Il n'y a pas d'ordre prédéterminé pour les affichages de chacun de ces processus. Il peut se faire selon l'un des enchaînements suivants :

p	p	q	q	r	r
q	r	p	r	p	q
r	q	r	p	q	p

1.5 Les relations entre processus

L'ensemble des processus doivent **coopérer** à la réalisation de l'application. Cette coopération recouvre deux aspects souvent mêlés :

→ les coordinations ou **synchronisations** des processus entre eux,

→ les **communications** de valeurs entre processus.

1.5.1 Exclusion mutuelle et synchronisation

Exclusion mutuelle Dans de nombreuses applications, les processus doivent partager des ressources qui ne doivent être utilisées que par un seul processus simultanément (impression, voie de communication, ...). On parle alors d'**exclusion mutuelle** et de **ressource critique**. Une ressource critique correspond à une séquence d'instructions non partageable. C'est à dire qu'une seule tâche peut être rendue à ce point d'exécution à un instant donné.

Exemple. Les lignes suivantes permettent de réaliser une conversion analogique $\rightarrow$ numérique sur une carte industrielle présentant plusieurs voies. Il est clair qu'une fois une voie sélectionnée, il faut faire complètement l'acquisition (jusqu'à obtenir le résultat) pour garantir la cohérence du programme.

```
Selection_Voie(V);
Attend_Fin_Conv;
Lecture(R);
```

Synchronisation: un processus doit attendre que un ou plusieurs autres processus aient établi un état du système adéquat, pour qu'il puisse se continuer.

Par exemple, un processus p doit attendre qu'une donnée ait été produite par un processus q avant de s'en servir. D'une maniére plus générale, il existe dans le processus q une instruction I_q qui doit être exécutée avant une instruction I_p figurant dans le processus p.

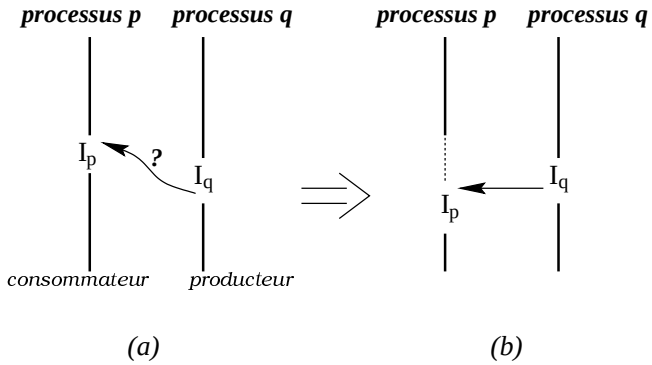


FIG. 1.1 – *Synchronisation : le processus q produit une donnée qui doit être consommée par le processus p. Situation (a): on ne sait pas si I_q sera exécutée avant I_p. Situation (b): la synchronisation est réalisée.*

Solutions. Parmi les solutions qui ont été apportées pour gérer ces deux problèmes de coopération, citons :

1. les sémaphores, qui sont des outils «systèmes». Ils sont utilisés sur les systèmes ou noyaux multi-tâche, lorsque la programmation de l'application utilise un langage impératif ou objet ne présentant pas d'outils de synchronisation.
2. les «rendez-vous» de `Ada` , mécanisme de haut niveau prenant en compte tous les problèmes classiques de synchronisation.

1.5.2 Les sémaphores

Définition du sémaphore

Un sémaphore S est constitué d'une variable entière $e(S)$ et d'une file d'attente $f(S)$. À sa création, la file d'attente est vide et $e(s)$ est initialisé à une valeur entière positive ou nulle, $e_0(S)$. Deux opérations indivisibles [5] et exclusives [6] permettent d'agir sur ces sémaphores : $P(S)$ et $V(S)$.

$P(S)$

début
e(S) ← e(S) - 1;
si (e(S) <0) ***alors***
début
état(*q*) [a] ← *bloqué*;
mettre *q* dans la file f(S)
fin
fin

[a] *q* étant le processus qui a appelé $P(S)$

$V(S)$

début
e(S) ← e(S) + 1;
si (e(S) ≤ 0) ***alors***
début
sortir un processus *q* de la file f(S)
état(*q*) ← *actif*;
fin
fin

À ces deux méthodes d'accès, on peut ajouter la phase de création du sémaphore qui fixera la valeur initiale $e_0(S)$ et qui créera la file d'attente pour les processus.

Utilisation du sémaphore pour gérer une exclusion mutuelle

L'utilisation d'un sémaphore passe par le respect de règles élémentaires :

- À chaque ressource critique, on associe un sémaphore spécifique S.
- La valeur initiale $e_0(S)$ est égale au nombre de processus pouvant utiliser simultanément la ressource (pour une exclusion mutuelle, $e_0(S) = 1$).
- Le début du code code de la région critique ($\Leftrightarrow$ le code non partageable) doit être précédé d'un appel à $P(S)$, ce qui équivaut, pour le processus à se mettre dans une file d'attente, en attendant que la ressource soit disponible.

5. Les instructions qui les constituent ne sont jamais interrompues.
6. Il n'y en a pas d'autre.

– La fin de la région critique doit être suivie d'un appel $V(S)$, qui annonce que la ressource est libérée.

Ainsi, les lignes de programmes à protéger de l'exemple précédent seront entourées par les deux requètes P et V :

```
...
P(S);
Selection_Voie(V);
Attend_Fin_Conv;
Lecture(R);
V(S);
...
```

La figure 1.2 représente la chronologie des actions dans le cas de la compétition entre deux processus p et q pour l'accès à une ressource critique pouvant être utilisée par un seul processus simultanément..

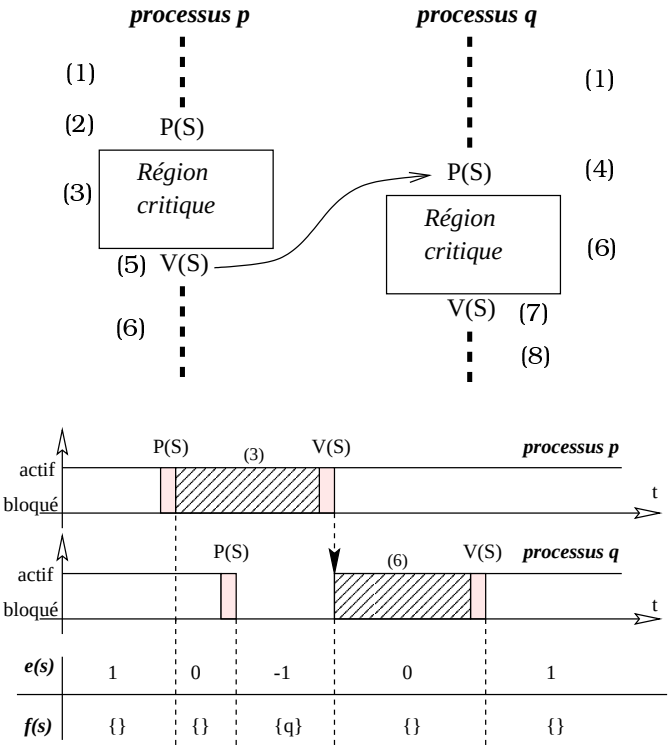


FIG. 1.2 – *Protection d'une région critique par un sémaphore.*

La valeur initiale $e_0(S) = 1$ affectée au sémaphore correspond au nombre de «places disponibles» dans la ressource critique. Lorsque la valeur du sémaphore est négative, sa valeur absolue donne le nombre de processus en attente dans la file $f(S)$.

Utilisation du sémaphore pour gérer une synchronisation

À chaque fois fois qu'un processus q produit une donnée qui doit être consommée par un processus p, le problème de la synchronisation se pose. Dans l'exemple général suivant, le processus q contient une instruction «`Produire`» qui doit être exécutée avant l'instruction «`Consommer`» du processus p.

Le sémaphore étant initialisé à $e_0(S) = 0$, le tableau suivant représente l'état du système dans le cas le moins favorable où le processus p cherche à consommer la donnée avant qu'elle ne soit produite.

processus p	processus q	$e(S)$	$f(S)$
...	...	0	{ }
P(S);	...	-1	{ p }
	Produire;	-1	{ p }
	V(S);	0	{ }
Consommer;	...	0	{ }
...	...	0	{ }

La figure 1.3 *(a)* représentent cette même situation sous forme de chronogramme. Le deuxième cas *(b)* illustre la situation la plus favorable (la donnée est prête).

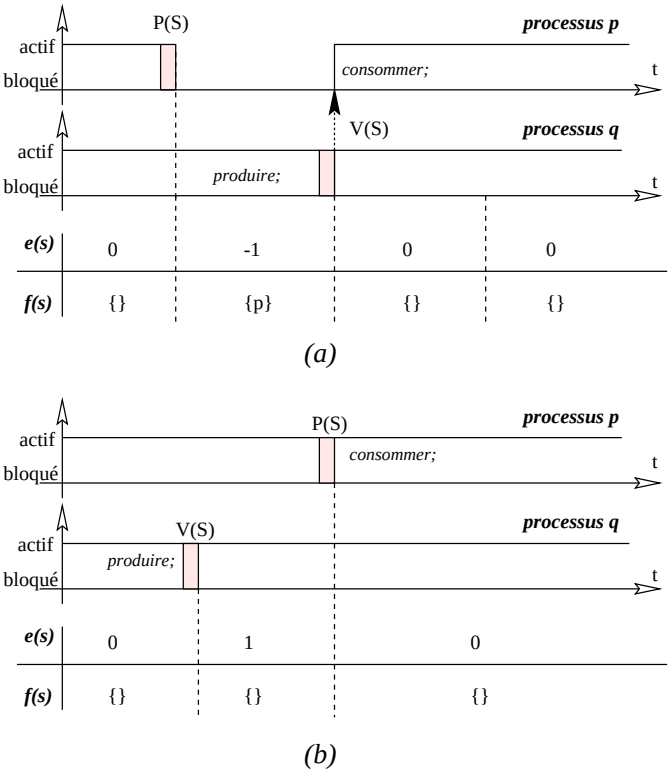


FIG. 1.3 – *Synchronisation dans le cas défavorable (a) et dans le cas favorable (b).*

Deuxième partie

Systèmes temps-réel

Chapitre 1

Systèmes temps-réel

Contrairement à l'informatique de gestion, les systèmes informatiques temps réel travaillent au rythme des événements qui surgissent à chaque instant dans le monde physique auquel ils sont connectés. Le temps et la nature de leur réactions à des sollicitations aléatoires qui, parfois, peuvent arriver en rafales sont par principe garantis.

Les mots clés du temps réel – préemptivité et déterminisme – résument bien les contraintes de cette informatique technique, scientifique, industrielle, militaire ...

> *Un système est temps réel lorsqu'une «activité de calcul doit répondre à des stimulis générés par le monde extérieur en un temps fini et spécifié».*
>
> YOUNG

1.1 Origine

L'informatique temps réel est apparue peu à peu dans l'histoire de l'informatique dont voici quelques brefs rappels :

1950 Les systèmes exécutent des travaux (*jobs*) séquentiellement. Un seul programme est présent en mémoire et est exécuté du début à la fin.

1955 C'est le début des premiers systèmes d'exploitation chargés des entrées-sorties. La programmation se fait en langage symbolique (assembleur). C'est aussi le début du `fortran`.

1960 Apparition des *batch-processing*. Les systèmes d'exploitation s'occupent en plus de l'enchaînement des travaux, selon la stratégie du «premier entré, premier servi».

1964 Les ordinateurs sont dotés d'unités d'échanges ayant la charge de gérer les entrées-sorties. L'unité centrale ne se consacre plus qu'à l'exécution des travaux. Les unités d'échanges communiquent avec l'unité centrale par des signaux. Les systèmes d'exploitation utilisent des mémoires-

tampon afin de réduire les temps d'attente au niveau des entrées-sorties .

1965 On essaie d'exploiter les temps d'inactivité de l'unité centrale et des périphériques. Les systèmes d'exploitation se chargent d'allouer les ressources aux programmes demandeurs. On aborde la multi programmation gérée par files d'attente.

1967 Apparition des systèmes multipostes : l'occupation de l'unité centrale par chaque programme est gérée par une horloge émettant des interruptions périodiques. La multiprogrammation se fait par partage de temps ou *time sharing*. Les programmes utilisateurs oscillent entre mémoire centrale et mémoire secondaire (technique de *swapping*).

1968 Les systèmes d'exploitation différencient les travaux de forte priorité de ceux de faible priorité. Les modes conversationnels (question réponse) et transactionels (modification de bases de données) sont de plus en plus courant. Les temps de réponse deviennent de plus en plus critiques. Les canaux d'entrées-sorties se multiplient. La notion de temps réel se concrétise.

1970 C'est la première apparition du terme «temps réel » tel que nous l'entendons aujourd'hui. L'arrivée massive des microprocessurs a rendu possible la généralisation du temps réel dans deux domaines essentiels : le pilotage de processus industriels et la gestion des entreprises.

En réalité, ce n'est que vers le début des années 1980 que l'informatique temps réel a commencé à se structurer, au travers de normes ou à défaut, de modélisation. C'est ainsi que le projet SCEPTRE[1] fut publié en 1984. Il proposait alors des spécifications à respecter pour l'écriture des exécutifs temps réel dans le but de rendre ceux-ci portables.

Une avancée déterminante pour la normalisation des noyaux temps réel a été la prise en compte du temps réel par le comité POSIX[2] de l'IEEE en 1987 (POSIX 1003.4 ou POSIX.4). La norme POSIX.4b va même jusqu'à la prise en compte du temps réel embarqué.

1.2 Le multitâche

1.2.1 Objectifs du multitâche

Un programme n'occupe jamais 100% du temps de l'unité centrale. En particulier, lorsque le programme est en attente d'une entrée-sortie sur un périphérique, le processeur serait inutilisé pendant le temps d'attente s'il ne devait gérer que ce programme. L'une des idées du multitâche est de récupérer tout les temps d'inactivité d'un programme au profit d'un autre programme.

Il s'agit alors de mettre en œuvre les règles de possession et le partage des ressources. Une contrainte importante, est la garantie de l'intégrité de

1. Standardisation du Cœur des Exécutifs des Produits Temps Réel Européens
2. Portable Operating System

chaque programme. Un programme donné ne doit pas être «nuisible» pour les autres.

Le multitâche fait coexister en mémoire centrale plusieurs programmes, sachant que ces derniers seront exécutés alternativement dans le temps par une seule unité centrale. La répartition temporelle se fait en fonction des ressources d'entrées-sorties requises par les programmes en compétition. La notion de priorité intervient également sur le partage du temps.

1.2.2 Les processus

Un **programme** est une entité composée de une ou plusieurs séquences d'instructions agissant sur un ensemble de données. Un programme est **statique**. C'est un ensemble séquentiel d'informations occupant une partie de la mémoire.

Un **processus** est une entité dynamique. Un processus représente l'exécution de un ou plusieurs programmes. Il présente des caractéristiques évoluant dans le temps.

On appelle souvent **tâche** un processus cyclique. C'est souvent le cas dans les applications industrielles.

Un processus peut être

→ créé,

→ exécuté,

→ détruit.

Un processus comporte en général trois zones (figure 1.1):

1. une zone **programme** qui contient les instructions à exécuter et parfois les constantes de l'application,
2. une zone de **données** accessible en lecture et en écriture : elle contient les variables globales de l'application.
3. une zone de **pile** permettant de ranger les données temporaires de l'application (paramètres pour les sous-programmes, adresse de retour de sous-programme, variables locales ...).

L'accès aux différentes zones se fait par rapport aux contenus des registres du processeurs qui ont été initialisés au lancement du processus. Dans la figure 1.1, les registres représentés sont ceux du M68000. L'instruction en cours du programme est marquée par le compteur programme du microprocesseur `PC`[3]. La pile est marquée par le registre `USP`[4] appelé aussi `A7`. La zone de donnée est marquée par un registre imposé par le système d'exploitation. Ici, `A6` marque la zone de donnée. (Ceci correspond à ce qui ce passe lors de l'exécution d'un programme sous OS9).

3. Program Counter
4. User Stack pointer

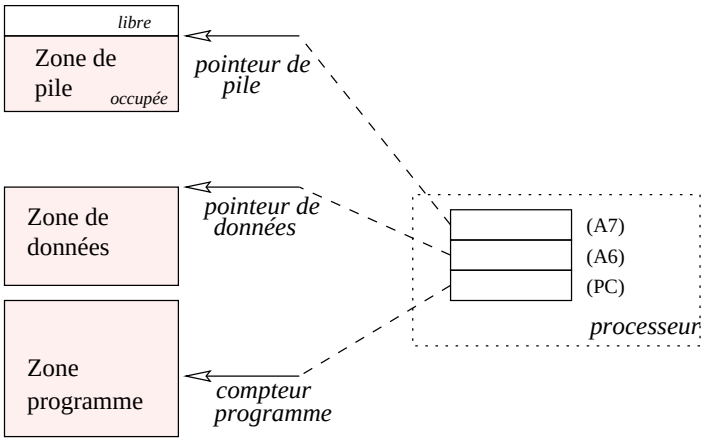


FIG. 1.1 – *Occupation de la mémoire par un processus.*

Les différentes informations contenues dans les registres du processeur caractérisent le processus à un moment de son exécution effective ou il est propriétaire du processeur. On les appelle **contexte d'exécution** du processus.

Si on parvient à stopper le processus en cours en conservant en mémoire le contenu des registres du processeur et en protégeant ses trois zones de mémoire, alors, on est capable de le relancer en rechargeant les registres du processeur avec ces mêmes valeurs. Nous verrons plus loin le mécanisme qui permet de sauvegarder les registres du processeurs sans perdre le contexte d'exécution du processus interrompu. La sauvegarde des contenus des registres d'un processus lorsqu'il est momentanément interrompu font partie du **contexte sauvegardé** de ce processus .

Plus généralement, dans le **contexte** d'un processus , on trouvera :

- l'état sauvegardé de chacun des registres du processeur, y compris le registre d'état;
- le nom du processus ;
- un identificateur (donné au moment de la création du processus);
- une priorité permettant de quantifier le degré de priorité;
- une variable d'état permettant au système de mémoriser l'état courant du processus («en attente», «en cours d'exécution» ...);

Toutes ces informations seront contenues dans une structure de données appelée **descripteur de processus** .

1.2.3 Gestion des descripteurs de processus

Le système multitâche a pour objet de gérer l'ensemble des descripteurs de processus . En général, ils sont stockés dans des files. Il y a autant de files

que d'état possibles pour les processus . Par exemple, il y aura une file pour les processus en cours, une autre file pour les processus en attente . . .

Faire changer d'état à un processus consiste alors à le supprimer dans une file pour le placer dans une autre. Supprimer un processus , c'est récupérer la mémoire qu'il occupait et l'éliminer de la file ou il figure.

Dans la pratique, de nombreux exécutifs utilisent le mécanisme du double chaînage consistant à relier un à un et dans les deux sens tous les descripteurs de processus de façon à pouvoir intervenir rapidement sur un processus donné.

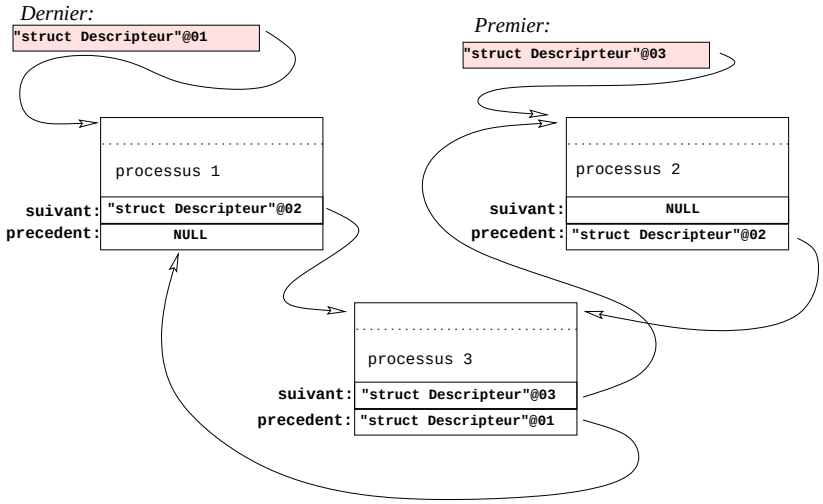


FIG. 1.2 – *Chaînage des descripteurs de processus*

Chapitre 2

Les tâches, et mise en œuvre avec VxWorks

Les systèmes temps réel modernes sont basés sur les deux concepts complémentaires du multitâche et de la communication inter-tâches. Un environnement multitâche permet de construire une application comme un ensemble de taches indépendantes, chacune possédant son propre *thread* d'exécution.

2.1 États d'une tâches et transitions entre états

2.1.1 Contexte sauvegardé

Le déroulement des processus temps réel est dépendant des événements externes au calculateur. Ces événements n'étant pas sous le contrôle du système, plusieurs tâches peuvent se trouver en concurrence pour l'octroi du processeur. Ce conflit est réglé en attribuant à chacune d'entre elles un niveau de priorité. La tâche exécutée est choisi par un algorithme de choix: **l'ordonnanceur**.

L'ordonnanceur (*scheduler*) constitue le cœur du noyau temps réel . Il est invoqué à chaque commutation de tâches ainsi qu'à chaque appel à une fonction système (sémaphore, signal, file d'attente, ...)

Le principe de commutation de tâche conduit à sauver en *RAM* l'état de chaque tâche : on parle de **contexte sauvegardé**. On y trouve :

- le *thread* d'exécution de la tâche (c'est simplement le compteur programme pointant sur l'adresse ou «est rendu» le programme);
- les registres du *CPU* et, éventuellement ceux de l'unité de calcul en virgule flottante;
- un pile mémorisant les variables dynamiques et les appels à des fonctions;
- les canaux d'E/S standards associés;
- un délai d'attente (compteur);
- un compteur de tranche de temps (*timeslice*)
- les structures de contrôles pour le noyau;

- un gestionnaire de signaux (*signal handler*);
- données pour le *debugging*.

2.1.2 États d'une tâche

Le noyau met à jour l'état de chaque tâche en cours dans l'application. Une tâche change d'un état à un autre suivant le résultat de chaque appel fait à une fonction du noyau par une tâche quelconque de l'application.

Lorsqu'une tâche est créée elle est dans l'état *suspendu*. Pour passer à l'état *prêt*, elle doit être activée. La phase d'activation est très rapide. Ceci permet de «pré-créer» plusieurs tâches ($\Rightarrow$ préparation des contextes), puis de les activer en un temps très court. Une autre possibilité est le lancement des tâches (*spawning*), qui permet d'enchaîner la création et l'activation d'une tâche. Les états possibles des tâches sont donnés dans le tableau 2.1.

`Prête`	$READY$	État d'une tâche qui n'attend aucune autre ressource que le CPU.
`En attente`	$PEND$	État d'une tâche bloquée suite à l'indisponibilité d'une ressource (matérielle, logicielle, information, ...)
`Endormie`	$DELAY$	État d'une tâche endormie pour une durée déterminée.
`Suspendue`	$SUSPEND$	État d'une tâche inéligible. Elle n'est jamais exécutée. Cet état est principalement utilisé pour la mise au point (*debugging*). La suspension d'un tâche n'empèche pas le changement d'état logique, seulement l'exécution. Ainsi, une tâche `en attente` et `suspendue` peut être débloquée. De même, une tâche `endormie` et `suspendue` peut être réveillée. Dans les deux cas elle reste `suspendue`.
`Endormie-suspendue`	$DELAY + S$	État d'une tâche à la fois `endormie` et `suspendue`.
`En attente-suspendue`	$PEND + S$	État d'une tâche à la fois `en attente` et `suspendue`.
`En attente-avec «time-out»`	$PEND + T$	État d'une tâche `en attente`, mais avec un *time out*
`En attente-suspendue-avec «time-out»`	$PEND + S + T$	État d'une tâche `en attente` avec un *time out*, et en plus `suspendue`. À l'issue du *time out* elle reste `suspendue`.

TAB. 2.1 – *États d'une tâche (*VXWORKS *)*

Le diagramme de la figure 2.1 représente les différents états d'une tâche

et les transitions possibles d'un état à l'autre.

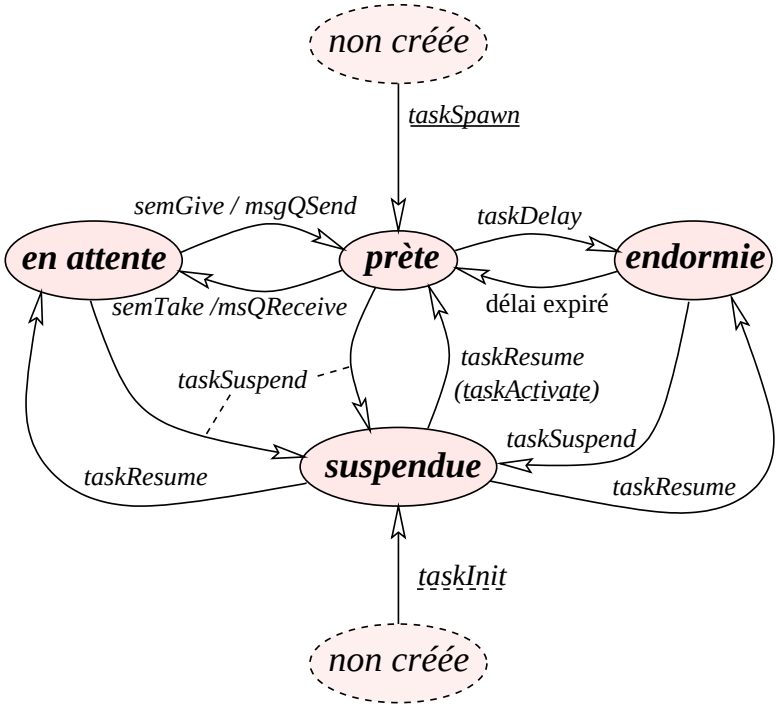


FIG. 2.1 – *Transitions possibles entre les états.*

2.1.3 Mise en œuvre

Considérons l'exemple suivant dans lequel une tâche `controle` arrête et réactive alternativement deux tâches `tache1` et `tache2` conformément au diagramme de la figure 2.2 :

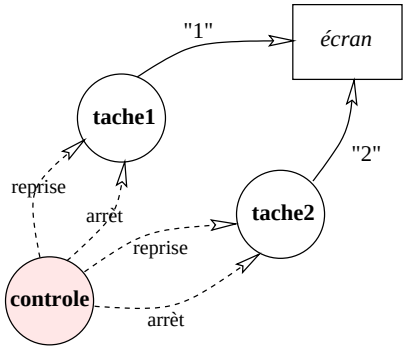


FIG. 2.2 – *Créations, suspensions et reprises.*

Les tâches `tache1` et `tache2` se contentent d'afficher respectivement des '1' et des '2' sur l'écran pour marquer leur activité.

```
#include "vxworks.h"

/* Identificateurs des tâches :*/
int id_tache1, id_tache2, id_cont;

tache1() {  while (1) printf("1"); }

tache2() {  while (1) printf("2"); }

controle() {
  while (1) {
    taskSuspend(id_tache1);  taskResume(id_tache2);   taskDelay(100);
    taskSuspend(id_tache2);  taskResume(id_tache1);   taskDelay(100);
  }
}

void usrAppInit (void) {
  /* Création des tâches */
  id_tache1 = taskSpawn("Tache1",90,0,4000,tache1,0,0,0,0,0,0,0,0,0,0);
  if (id_tache1 == ERROR){
    printf("Erreur de création de 'T_1'\n\r");
    return;
  }
  id_tache2 = taskSpawn("Tache2",90,0,4000,tache2,0,0,0,0,0,0,0,0,0,0);
  if (id_tache2 == ERROR){
    printf("Erreur de création de 'T_2'\n\r");
    return;
  }
  id_cont = taskSpawn("Controle",90,0,4000,controle,0,0,0,0,0,0,0,0,0,0);
  if (id_cont == ERROR){
    printf("Erreur de création de 'controle'\n\r");
    return;
  }
}
```

Remarques :

† Les tâches sont créées par la fonction `taskSpawn`. Son prototype est

```
int taskSpawn(
      char *nom,          Nom de la tâche (pour le debugging).
      int priorité,       Priorité (0 ≤ priorité ≤ 255).
      int options,        En général = 0.
      int taillePile,     Taille de pile nécessaire en octets.
      FUNCPTR tache,      C'est le nom de la fonction principale
                          associée à la tâche à créer.
      int arg1,           Premier argument passé à la fonction
                          tache.
      ...
      int arg10)          10ème argument passé à la fonction tache.
```

† La fonction `taskDelay` permet à la tâche appelante d'être endormie

pendant le délai entier spécifié. L'unité est la période de l'horloge temps réel (appelée *tick*). Son prototype est :

```
int taskDelay(int ticks)
```

† La fonction `taskSuspend` permet de suspendre la tâche dont le numéro identificateur est donné en argument. Son prototype est :

```
int taskSuspend(int t_ID)
```

† La fonction `taskResume` permet de restituer à la tâche dont le numéro identificateur est donné en argument son état normal. Son prototype est :

```
int taskResume(int t_ID)
```

† Les fonction `taskSpawn`, `taskDelay`, renvoient l'une des valeurs symboliques `OK` ou `ERROR`.

Utilisation des paramètres `arg1` ...`arg10` de `taskSpawn`

Ils sont utilisés pour passer des arguments à la fonction principale de chaque tâche créée. Ils peuvent avoir deux rôles :

- Lorsqu'une tâche est détruite, on peut la relancer avec des valeurs de paramètres différents.
- On peut créer plusieurs tâches à partir d'un code unique pouvant être paramétré.

Notre exemple peut illustrer ce deuxième cas. En effet, le code des fonctions `tache1` et `tache2` pourrait se résumer à une seule fonction paramétrée :

```
tache(int n) {
  while (1) {
    printf("%d", n);
  }
}
```

Pour obtenir le même fonctionnement de l'application, la création des tâches se ferait alors de la manière suivante :

```
id_tache1 = taskSpawn("Tache1",90,0,4000,tache,1,0,0,0,0,0,0,0,0,0);
id_tache2 = taskSpawn("Tache2",90,0,4000,tache,2,0,0,0,0,0,0,0,0,0);
```

2.2 Ordonnancement des tâches

La mise en œuvre du multitâche nécessite un algorithme de choix destiné à attribuer le processeur à l'une des tâche prêtes. Deux possibilités existent pour allouer le processeur :

- tenir compte de la priorité des tâches,
- donner à chaque tâche le même temps CPU .

2.2.1 Partage du temps en *round-robin* simple

Dans un système en temps partagé, l'ordonnancement des tâches est **préemptif** : chaque processus est interrompu après un temps d'exécution

prédéfini dans le système. Une horloge temps réel génère une interruption toutes les x millisecondes. Le noyau prend alors le processeur et l'attribue à la tâche suivante. Le partage du temps s'effectue sur une unique file d'at-

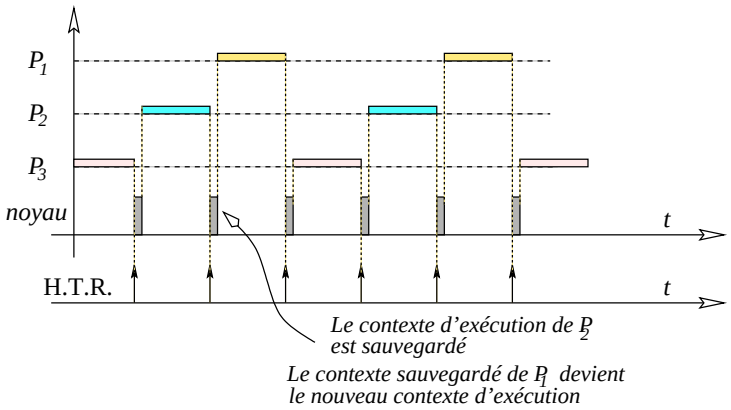


FIG. 2.3 – *Partage du temps par* round-robin .

tente de processus éligibles. À chaque processus est alloué un **quantum** de temps δt, encore appelé tranche de temps ou *time-slice*. En général, le quantum de temps vaut le moment entre deux impulsions de l'horloge temps réel ($\delta t = T_{HTR}$).

La période de l'horloge temps-réel est l'unité de temps du système. On l'appelle un *tick*. Le choix de l'unité de temps est primordial pour une bonne optimisation des temps d'attente et d'occupation du processeur. Un temps trop court conduit à des temps de service trop importants. Un temps trop long nuit à l'aspect multitâche . Il faut trouver un compromis.

Le *round-robin* avec VXWORKS **:** ce n'est pas le partage de temps par défaut du système. Cependant, il est possible de le sélectionner avec la fonction `kernelTimeSlice(int TIMESLICE)`. La valeur positive `TIMESLICE` donne le temps accordé à chaque tâche (en nombre de périodes de l'horloge temps réel). La valeur `TIMESLICE=0` annule le *round-robin* et fait repasser le système dans le mode de temps partagé avec priorités.

2.2.2 Approche temps réel du partage du temps

Une des caractéristiques du temps réel est d'assurer un **déterminisme** entre deux utilisations successives d'un processeur par un ensemble de processus . L'exécution d'un processus est associée à un temps maximum qu'il a pour le faire.

La prise en compte de ce type de contraintes conduit à associer une **priorité** au processus. La file d'attente des processus est triée par ordre de priorité décroissante, et le processus de priorité la plus forte est élu. Un processus conservera donc le processeur tant qu'aucun autre processus de la file d'attente n'aura une priorité supérieure.

Le choix de la priorité des tâches est déterminant :

- † les tâches les plus prioritaires sont associées aux traitements des défauts du système (pannes, arrêts d'urgence, alarmes ...)
- † viennent ensuite les tâches associées aux interruptions matérielles (activées suite à un signal provenant d'un programme d'interruption);
- † enfin, on trouve les tâches de fond et les tâches utilisateur.

Dans tous les cas, ce qui importe, c'est la priorité relative des processus .

Le noyau VxWorks gère 256 niveaux de priorités allant de 0 (priorité la plus forte) à 255 (priorité la plus faible). Á chaque *tick*, les niveaux de priorités sont décrémentés. La tâche de priorité la plus forte ($\Leftrightarrow$ de niveau le plus bas) est élue. Au *tick* suivant elle reprend son niveau d'origine.

Troisième partie

Communication entre tâches avec VxWorks

Chapitre 1

Les données partagées

C'est la manière la plus simple de communiquer des informations entre les tâches. Une application VxWorks s'exécute dans un espace de mémoire linéaire et continu. Plus simplement, il y a un seul programme principal (`main`) à partir duquel les fonctions sont exécutées, certaines (les tâches) en concurrence.

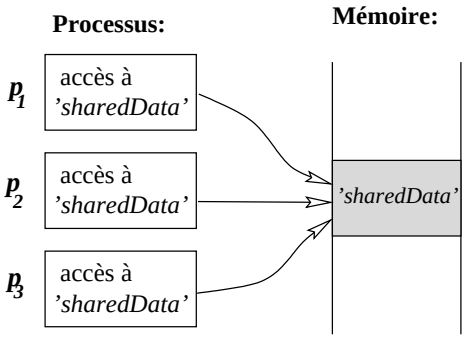


FIG. 1.1 – *Structure de données partagées.*

Les données partagées sont tout simplement des variables globales (simples, structurées ou en tableau). Les différentes tâches et fonctions y accèdent directement.

1.1 Protection des données partagées

Lorsqu'une donnée est composée de plus d'un mot machine, il est nécessaire d'en protéger l'accès.

1.1.1 Interdiction des interruptions

La solution la plus radicale pour protéger les informations des accès concurrents est d'interdire toutes les interruptions. Ainsi, la commutation

de tâches est stoppée, et les tâches matérielles sont interdites. Ceci est possible avec VxWorks par la fonction `intLock`. .

Exemple

```
fonctionAccesAuxDonnees() {
    int S = intLock();
     .
     .  /* accès aux données communes */
     .
    intUnlock(S);
}
```

Remarques :

- Pour autoriser à nouveau les interruptions (et donc le multi-tâche), il faut appeler la fonction `intUnlock`..
- La fonction `unLock` renvoie le masque courant d'interruptions. Il doit être restitué par la fonction `intUnlock`.

1.1.2 Arrêt du multi-tâche

Il existe une solution moins contraignante pour le système qui permet aux interruptions matérielles de s'exécuter. Ceci n'est possible, bien entendu, que si aucune tâche matérielle n'a besoins d'accéder aux données partagées. On utilise les fonctions `taskLock` et `taskUnlock`.

Exemple

```
fonctionAccesAuxDonnees() {
    taskLock();
     .
     .  /* accès aux données communes */
     .
    taskUnlock();
}
```

1.1.3 Inconvénient des arrêts préemptifs

La protection des données par blocage du multi-tâche pose le problème crucial du temps de réponse pour les autres processus. En effet des tâches de priorité supérieure peuvent être bloquées par une tâche de moindre importance.

Si cette exclusion est simple, elle doit rester exceptionnelle et réservée pour des accès très courts aux données. On préférera en général l'utilisation de sémaphores d'exclusion mutuelle.

Chapitre 2

Les sémaphores

Le sémaphores sont utilisés dans deux cas de figure :

- l'**exclusion mutuelle** : pour garantir un partage cohérent des ressources critiques (une seule tâche à la fois !),
- la **synchronisation** : l'exécution d'une tâche est synchronisée sur un événement extérieur (exécution d'une instruction donnée dans une autre tâche).

VxWorks propose des sémaphore spécialisés en fonction des problèmes précédents.

- les **sémaphores binaires** : réservés aux problèmes de synchronisation,
- les **sémaphores d'exclusion mutuelle** : pouvant prendre en compte les priorités ...
- les **sémaphores à compteur** : pour les synchronisations particulières ou il est possible de donner (V(s)) un sémaphore plusieurs fois.

2.1 Les fonctions associées

Appel	Description	B	M	C
`semBCreate()`	Crée un sémaphore binaire	×		
`semMCreate()`	Crée un sémaphore d'exclusion mutuelle		×	
`semCCreate()`	Crée un sémaphore à compteur		×	
`semDelete()`	Détruit un sémaphore et libère la mémoire associée	×	×	×
`semTake()`	Prend un sémaphore (P(s))	×	×	×
`semGive()`	Rend un sémaphore (V(s))	×	×	×
`semFlush()`	Débloque toutes les tâches en attente d'un sémaphore	×	×	×

TAB. 2.1 – *Fonction associées aux sémaphores*

Un sémaphore de VxWorks est une variable de type `SEM_ID` (défini dans le fichier *semLib.h*). Pour que plusieurs tâches puissent y accéder, il convient

de le déclarer comme une variable globale.

2.1.1 Création d'un sémaphore

Sémaphore d'exclusion mutuelle

Le prototype est :

```
SEM_ID semMCreate(int option)
```

avec `option` pouvant être :

`SEM_Q_PRIORITY :` les tâches en attente du sémaphore sont mises dans une file d'attente. L'élection de la tâche qui prend le sémaphore se fait en fonction des priorités.

`SEM_Q_FIFO :` la tâche élue est la plus ancienne qui a été candidate : la file est simplement gérée en FIFO.

La valeur retournée est l'identificateur du sémaphore (type `SEM_ID`). En cas d'erreur la valeur `NULL` est renvoyée.

Exemple

```
#include "semLib.h"

SEM_ID semaM;
 .
 .
int usrAppInit () {
  semaM =  semMCreate(SEM_Q_FIFO);
  if (semaM==NULL) {
    printf("Création du sémaphore impossible\n");
    return 0;
  }
  .
  .
}
```

Sémaphore de synchronisation

Le prototype est :

```
SEM_ID semBCreate(int option, SEM_B_STATE)
```

avec `option` comme précédemment, et `SEM_B_STATE` pouvant être :

`SEM_FULL :` le sémaphore est initialisé à l'état «arrivé», (la première tâche qui se mettra en attente ne sera donc pas bloquée).

`SEM_EMPTY :` le sémaphore est initialisé à l'état «non arrivé».

La valeur retournée est l'identificateur du sémaphore (type `SEM_ID`). En cas d'erreur la valeur `NULL` est renvoyée.

Exemple

```
#include "semLib.h"

SEM_ID semaB;
 .
 .
int usrAppInit () {
  semaB =  semBCreate(SEM_Q_FIFO, SEM_EMPTY);
  if (semaB==NULL) {
    printf("Création du sémaphore impossible\n");
    return 0;
  }
  .
  .
}
```

Sémaphore à compteur

Le prototype est :

```
SEM_ID semCCreate(int option, int valeurInitiale)
```

avec `option` comme précédemment, et `valeurInitiale` le compte initial du sémaphore. À chaque appel à `semGive`, le compteur est incrémenté. À chaque appel à `semTake`, le compteur est décrémenté; s'il est ≥ 0 la tâche n'est pas bloquée. Dans le cas contraire la tâche est bloquée en attendant que le compteur soit à nouveau incrémenté par des appels à `semGive`.

La valeur retournée est l'identificateur du sémaphore (type `SEM_ID`). En cas d'erreur la valeur `NULL` est renvoyée.

2.1.2 Mettre un sémaphore dans l'état «arrivé»

Pour donner un sémaphore – c'est à dire le mettre dans l'état «arrivé» – il faut appeler la fonction de prototype :

```
STATUS semGive(SEM_ID s)
```

Le sémaphore identifié par `s` est mis dans l'état «arrivé». La fonction renvoie `OK` si tout s'est bien passé ou `ERROR` si l'identificateur ne correspondait pas à un sémaphore.

2.1.3 Attendre qu'un sémaphore soit dans l'état «arrivé»

Pour prendre un sémaphore il faut appeler la fonction de prototype :

```
STATUS semTake(SEM_ID s, int timeOut)
```

Un *time-out* doit être spécifié. La valeur `WAIT_FOREVER` oblige la tâche à attendre que le sémaphore soit dans l'état «arrivé». Les valeurs positives de `timeOut` donne le temps maximum (en *ticks*) d'attente du sémaphore.

La valeur retournée est `OK` si le sémaphore a été pris dans les conditions normales. En cas de *time-out* ou d'erreur de numéro identificateur, c'est `ERROR` qui est retourné.

Exemple : protection d'une région critique. Le code à protéger fait l'objet de la fonction `fonctionCritique()`. Le sémaphore `semaM` a été créé comme dans l'exemple du paragraphe 2.1.1.

```
.
semTake(semaM);  /* attend que la ressource soit disponible */
fonctionCritique();
semGive(semaM);  /* libère la ressource */
.
```

2.2 Application

2.2.1 Le sémaphore d'exclusion mutuelle

Question 1 - Mise en évidence d'une région critique.

Créer un nouveau projet dans le répertoire Entrez le programme suivant dans le fichier `usrAppInit.c` du projet :

```
#include <stdio.h>
#include <stdlib.h>
#include "vxworks.h"
#include "sysLib.h"
#include "semLib.h"
#include "string.h"

void affiche(char *s){ logMsg(s, 0,0,0,0,0,0); }

/* Simulation d'une conversion: */
int fonctionCritique(int n) {
  static int V=0;
  char b[128];
  sprintf(b,"Selection voie %d\n",n); affiche(b);
  taskDelay(2);
  affiche("Lancement conversion\n");
  taskDelay(2);
  sprintf(b,"Lecture voie %d: %x\n",n, V); affiche(b);
  taskDelay(2);
  V++;
  return V;
}

/* Tâche: */
STATUS Conv(int voie){
  int G;
  char b[32];
```

```
  taskDelay(10);
  while(1) {
    G = fonctionCritique(voie);
    sprintf(b,"Lu : %x\n", G); affiche(b);
  }
}

/* Lanceur: */
void usrAppInit (void) {
  char *Tache[] = {"TRT", "TD", "TP", "TH"}
  int id[3],i;
  for (i=0; i<3; i++) {
    id[i] = taskSpawn(Tache[i],90,0,4000,Conv,i,0,0,0,0,0,0,0,0,0);
    printf("Lancement de la tache %s (%d)\n",Tache[i],id[i]);
    if (id[i] == ERROR){
      printf("Erreur de creation de '%s'\n",Tache[i]);
      return 0;
    }
  }
}
```

Compilez, téléchargez puis exécutez ce programme. Que constate t-on? La région critique (représentée par les affichages successifs de la fonction `fonctionCritique`) est-elle respectée?

Question 2 - Mise en œuvre d'un sémaphore d'exclusion mutuelle

Identifiez le code qui doit faire l'objet d'une protection par sémaphore d'exclusion mutuelle. Réalisez la protection en respectant les principes suivants :

- le sémaphore est déclaré en variable globale (type `SEM_ID`),
- il est créé dans le lanceur **avant** la création des tâches,
- le code protégé est le plus court possible.

2.2.2 Le sémaphore binaire

Il sont essentiellement utilisés pour la synchronisation des tâches. On décide de donner à chacune des trois tâches `TRT`, `TD` et `TP` une période de lancement. Pour cela, dans le code de la fonction `Conv`, on supprime le `taskDelay`, et chacune de ces trois tâche se synchronise sur un sémaphore binaire qui lui est spécifique (figure 2.1).

Les périodes des tâches seront :

- tâche `TRT` : 1,5 s
- tâche `TD` : $\frac{2}{3}$ s
- tâche `TP` : 1,0 s

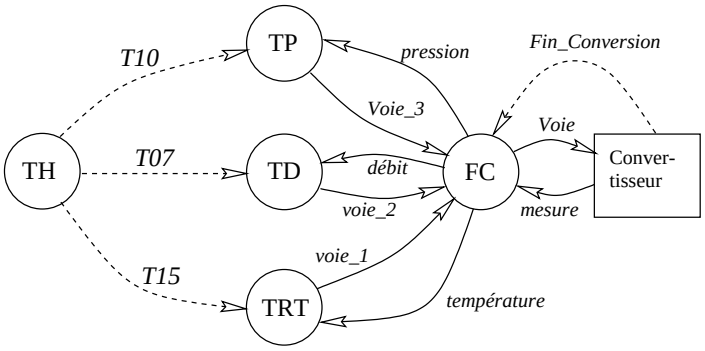


FIG. 2.1 – *Synchronisation de tâches.*

Question 3

Réalisez successivement les modifications suivantes :

1. déclarez en variable globale un tableau de trois identificateurs de sémaphores binaires;
2. dans la fonction «lanceur» `usrAppInit`, créez les trois sémaphores (avec `semBCreate`);
3. modifiez le code des tâches (fonction `Conv`) pour prendre en compte la synchronisation (supprimez le `taskDelay` dans cette fonction);
4. écrivez le code de la tâche `"TH"` dans une fonction

   ```
   STATUS horloge();
   ```

   On calculera le temps de sommeil de cette tâche le plus grand possible.
5. Créez la tâche `"TH"` dans le lanceur.

Chapitre 3

Les signaux

3.1 Généralités

3.1.1 Principes

Les signaux permettent aux tâches de communiquer de manière asynchrone. Une tâche recevant un signal voit son exécution modifiée : dès qu'elle reprend le processeur, elle abandonne son *thread* d'exécution pour exécuter une routine particulière. N'importe quelle tâche logicielle ou matérielle peut envoyer un signal à une autre tâche.

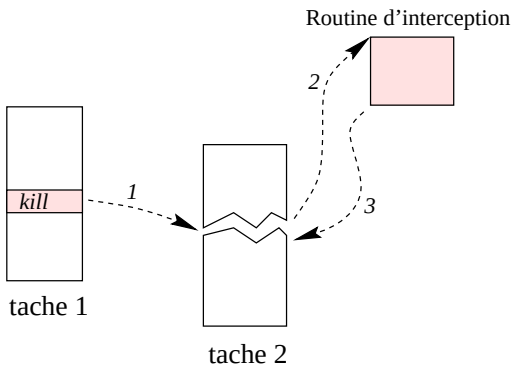


FIG. 3.1 – *Principe des signaux.*

Un signal est l'équivalent logiciel d'une interruption. Une tâche envoie un signal à une autre tâche pour indiquer qu'un événement s'est produit. Par principe, la tâche réceptrice n'est pas en attente du signal[1]. Par contre, elle doit au préalable avoir «installé» une routine d'interception (*signal handler*) qui s'exécute de manière asynchrone avec le code principal de la tâche réceptrice (et dans le même contexte). Après exécution de la routine d'interception, la tâche réceptrice continue son exécution normale.

1. Dans ce cas l'utilisation d'un sémaphore binaire paraît mieux indiquée.

3.1.2 Les signaux

Ils sont aux nombre de 31. Chacun représentant un événement particulier. Certains d'entre eux sont envoyés lorqu'une erreur système se produit (`SIGBUS, ...`). D'autres sont envoyés à l'initiative des tâches utilisateur (`SIGUSR1, ...`).

Symbole	**Valeur**	**Rôle**
SIGHUP	1	*hangup*
SIGINT	2	interruption
SIGQUIT	3	quitter
SIGILL	4	instruction illégale
SIGTRAP	5	mode trace
SIGABRT	6	*abort*
SIGEMT	7	instruction *EMT*
SIGFPE	8	exception virgule flottante
SIGKILL	9	*kill* (système $\rightarrow$ ne peut être ni intercepté ni ignoré)
SIGBUS	10	erreur bus
SIGSEGV	11	erreur de segmentation
SIGFMT	12	erreur de format de pile
SIGPIPE	13	écriture dans un tube sans lecteur
SIGALRM	14	alarme horloge
SIGTERM	15	fin de tâche provoquée logiciellement
SIGSTOP	17	signal *stop* ne provenant pas de la console *tty*
SIGTSTP	18	signal *stop* provenant de la console *tty*
SIGCONT	19	pour continuer un processus stoppé
SIGCHLD	20	pour la tâche parent, d'un fils qui se termine
SIGTTIN	21	
SIGTTOU	22	
	$23 \cdots 29$	signaux «temps-réel»
SIGUSR1	30	signal utilisateur 1
SIGUSR2	31	signal utilisateur 2

TAB. 3.1 – *Les signaux* VXWORKS

3.1.3 Précautions

Les signaux **ne sont pas recommandés** pour réaliser une communication entre tâches. Un signal :

- † Perturbe le déroulement normal des tâches. Il est préférable de créeer plusieurs tâches plutôt que de paralléliser l'activité d'une tâche par des routines d'interception.
- † Peut **provoquer des problèmes de réentrance** entre la routine d'interception et le code principal de la tâche. Il faut être vigilant en ce qui concerne les données partagées.

† Peut conduire à un **problème d'interblocage** (*deadlock*) lorsque la routine d'interception et le code principal utilisent un sémaphore d'exclusion mutuelle.

† Peut être utilisé pour mettre fin à la tâche.

3.1.4 Traitement des exceptions

Une exception est en général provoquée par une instruction qui met en défaut le processeur (erreur bus, division par zéro, ...). En général, lorsqu'une exception est générée, un message d'erreur est affiché («`segmentation fault`», ...), puis la tâche responsable est arrêtée.

Avec VxWorks , une exception peuvent être interceptée, à condition qu'elle corresponde à un signal pour lequel une routine d'interception a été enregistrée.

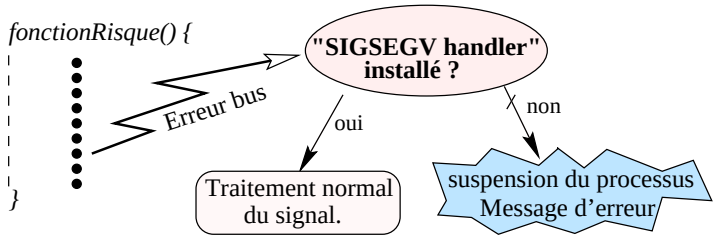


FIG. 3.2 – *Traitement des exceptions: certains signaux correspondent à des exceptions (problème «hardware»). L'erreur bus correspond sur une architecture 68k au signal* `SIGSEGV`*. Traiter ce signal permet à la tâche de poursuivre son traitement même si elle provoque une «erreur bus».*

Remarque : la correspondance entre signaux et exceptions dépend de l'architecture.

3.1.5 Envoyer un signal

L'envoi d'un signal se fait avec la fonction `kill`. Son prototype est

```
int kill(int tid, int signo);
```

`tid` est le N^o identificateur de la tâche réceptrice du signal. `signo` est le N^o du signal à émettre. Cette fontion n'est pas blocante.

Elle renvoie `OK` ou `ERROR` lorsque si la tâche `tid` n'existe pas ou si le signal `signo` n'existe pas.

Exemple:

```
#include "sigLib.h"

...
int taskId = taskSpawn(...);
```

```
...
kill(taskId, SIGUSR1);
```

3.1.6 Intercepter un signal

Pour intercepter un signal, une tâche doit installer une routine d'interception. Dans le cas de plusieurs signaux à intercepter, il faut plusieurs routines.

Lorsqu'une tâche reçoit un signal alors qu'elle ne dispose pas de routine d'interception pour lui, le signal est ignoré.

Lorsqu'une tâche endormie reçoit un signal pour lequel elle dispose d'une routine d'interception, elle est mise dans la liste des tâches actives, juste le temps d'exécuter la routine d'interception.

Pour enregistrer une routine d'interception, on utilise la fonction `signal` ou la fonction `sigaction`.

Avec la fonction `signal`

Son prototype est :

```
void (*signal(int signo, void (*routine)()))()
```

`signo` est le numéro du signal à traiter. `routine` est l'adresse du code à exécuter lors d'une réception de signal.

Elle renvoie l'adresse du *handler* précédent.

Exemple :

```
/* Handler du signal 'SIGUSR1' ----- */
void routineIT(int signal) {
  printf("----------->>  signal 'SIGUSR1' capturé ! \n");
  ...
}

/* Tache de reception de signaux -------------------- */
void recoitSignaux(void) {
  signal(SIGUSR1, routineIT);
  ...
}
```

Avec la fonction `sigaction`

Son prototype est :

```
int sigaction(int signo, const struct sigaction *pAct, struct
                    sigaction *oldAct);
```

`signo` : numéro du signal (voir le tableau 3.1),

`pAct` : adresse d'une structure `sigaction`. Cette structure décrit le comportement de la tâche lorsqu'elle reçoit le signal.

`oldAct` : adresse d'une structure de même type, qui est initialisée avec l'ancien contexte de traitement du signal par la tâche. La valeur `NULL` permet de désactiver cette possibilité.

La structure `struct sigaction` est définie par :

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

avec :

`sa_handler` : est l'adresse de la fonction à exécuter;

`sa_mask` : est un masque spécifiant les signaux devant être bloqués durant l'exécution du *handler*;

`sa_flags` : spécifie le comportement du *handler*;

Exemple :

```
/* Handler du signal 'SIGUSR1' ----- */
void routineIT(int signal) {
  printf("----------->>  signal 'SIGUSR1' capturé ! \n");
  ...
}

/* Tache de reception de signaux --------------------- */
void recoitSignaux(void) {
  struct sigaction ActionUSR1;

  /* Initialisation de la structure 'sigaction' */
  ActionUSR1.sa_handler = routineIT; /* adresse du code  */
  sigemptyset(&ActionUSR1.sa_mask);  /* pas d'autre signal */
  ActionUSR1.sa_flags = 0;            /* pas d'option */

  /* Installation de la routine d'interception */
  if(sigaction(SIGUSR1, &ActionUSR1, NULL) == -1)
    printf("Erreur d'installation du 'handler'\n");


  ...
}
```

3.2 Application

Le programme suivant met en œuvre deux tâches :

- `recoitSignaux` : qui installe une routine d'interception pour le signal `SIGINT`;
- `genereSignaux` : qui envoie régulièrement des signaux à la tâche `recoitSignaux`.

```
#include "vxWorks.h"
#include "sigLib.h"
#include "taskLib.h"
#include "stdio.h"
```

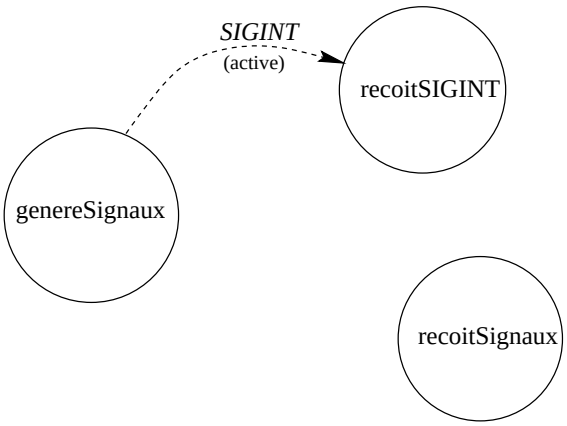


FIG. 3.3 – *Les deux tâches s'expriment par trois fonctions (il faut tenir compte de la routine d'interception)*

```
/* 'Handler' du signal 'SIGINT' ---------------------- */
/* (routine d'interception) */
void recoitSIGINT(int signal) {
  printf("----------->>  signal 'SIGINT' capture \n");
}


/* Tâche de réception de signaux --------------------- */
void recoitSignaux(void) {
  int i, j;

  /* Installation de la routine d'interception */
  signal(SIGINT, recoitSIGINT);

  /* Activité normale */
  for (i=0; i < 100; i++){
    for (j=0; j < 1000000; j++); /* pour simuler une activité ... */
    printf("Activité  de 'recoitSignaux'\n");
  }

  printf("Fin de 'recoitSignaux\n");
}


/* Tâche d'émission de signaux ----------------------- */
void genereSignaux() {
  int i, j, taskId;
  STATUS tacheOK;

  /* Création de la tâche de réception */
  if((taskId = taskSpawn("Signaux",100,0x100,20000,
                         (FUNCPTR)recoitSignaux,
                         0,0,0,0,0,0,0,0,0,0)) == ERROR)
```

```
    printf("Erreur de création de \n");


  /* Attente (pour que le 'handler' de signaux)
     soit effectivement installé) */
  taskDelay(30);

  /* Envoi de signaux ... */
  tacheOK = OK;
  for (i=0; i<100 && tacheOK==OK; i++) {
    if ((tacheOK = taskIdVerify(taskId)) == OK) {
      printf("signal 'SIGINT' genere\n");
      kill(taskId, SIGINT); /* genere le signal */
    }
  }
  printf("Fin de genereSignaux\n");
}

/* ------------------------------------------------- */
void usrAppInit (void){
  genereSignaux();
}
```

Question 1

Testez le programme précédent.

Question 2

En vous inspirant de cette mise en œuvre, créez une application qui met en action 3 tâches :

- `Process` : est une tâche qui affiche périodiquement un caractère simulant une activité. Elle installe au préalable trois routines d'interception correspondant aux signaux `SIGINT`, `SIGUSR1` et `SIGUSR2`. Le caractère affiché dépendra du dernier signal reçu:
  - `'1'` après `SIGUSR1`,
  - `'2'` après `SIGUSR2`,
  - `'T'` après `SIGINT`,
  - `'-'` tant qu'aucun signal n'est reçu.
- `Controle` : interroge le clavier, et envoie un signal à `Process` en fonction du caractère lu :
  - si le caractère lu est un nombre, elle envoie `SIGUSR1`
  - si le caractère lu est une lettre, elle envoie `SIGUSR2`
  - sinon, elle envoie `SIGINT`.
- `usrAppInit` : c'est la tâche d'initialisation. Elle lance les deux autres tâches puis disparaît.

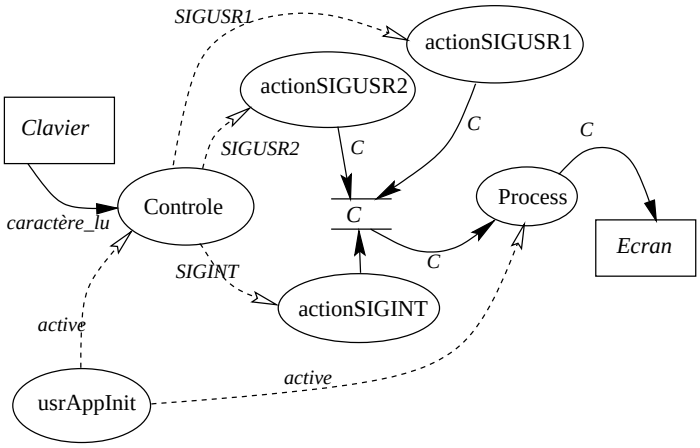


FIG. 3.4 – *Mise en œuvre*

Chapitre 4

Les files de messages

4.1 Définition

Une file de communication est un mécanisme de haut niveau permettant à des tâches concurrentes d'échanger des informations. Les messages en quantité variable peuvent être chacun d'une longueur quelconque.

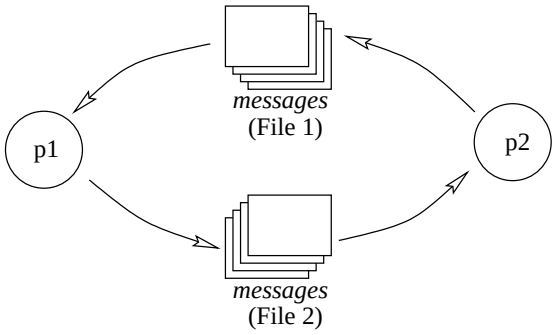


FIG. 4.1 – *Communication «full-duplex» avec deux files de messages.*

La circulation des messages se fait selon une stratégie ***FIFO*** (*first in, first out*). La **synchronisation** et l'**exclusion mutuelle** sont réalisées.

4.2 Création d'une file de messages

L'initialisation d'une file de messages se fait avec la fonction `msgQCreate` [1]. Son prototype est :

```
#include "msgQLib.h"
MSG_Q_ID msgQCreate( int maxMsg, int maxLongMsg, int opt)
```

`maxMsg` : nombre maximal de messages pouvant être stockés dans la file de message;

1. pour *message queue create*

`maxLongMsg` : nombre maximal d'octets pour un message;

`opt` : la file peut être créée avec les options suivantes :

- `MSG_Q_FIFO` : les tâches sélectionnées pour lire les messages sont prises dans l'ordre ou elles ont fait la requète de lecture,
- `MSG_Q_PRIORITY` : la sélection des tâches pour la lecture se fait en fonction de leur priorité.

L'allocation de mémoire se fait à la création de la file en fonction du nombre maximal de messages et de la taille maximale d'un message. La fonction renvoie `NULL` en cas d'erreur.

Exemple

```
#include "msgQLib.h"

MSG_Q_ID idFile;
 .
 .
   idFile = msgQCreate(256, 16, MSG_Q_FIFO);
   if (idFile == NULL) return 0;
 .
```

4.3 Envoi d'un message

N'importe quelle tâche immédiate (ISR) ou différée peut envoyer un message avec la fonction `msgQSend`. Si aucune tâche n'est en attente de message, le message envoyé est stocké en queue de file, sinon, le message est délivré à la première tâche en attente. Le prototype de cette fonction est :

```
STATUS msgQSend( MSG_Q_ID idFile,
                 char *buffer,
                 UINT N,
                 int time_out,
                 int priorite);
```

`idFile` : identificateur de la file destinatrice;

`buffer` : adresse des caractères à émettre;

`N` : nombre de caractères à émettre;

`time_out` : spécifie le nombre de *ticks* à attendre qu'il y ait assez d'espace disponible dans la file pour écrire le message; deux valeurs particulières sont prévues :

- `NO_WAIT` : pas de *time-out*, le retour s'effectue immédiatement même en cas d'échec;
- `WAIT_FOREVER` : la tâche émettrice reste bloquée jusqu'à ce qu'une place soit disponible dans la file;

`priorite` : priorité du message émis :

- `MSG_PRI_NORMAL` : priorité normale, le message est ajouté à la fin de la file;
- `MSG_PRI_URGENT` : priorité urgente, le message est ajouté en tête de liste.

Cette fonction renvoie `OK` ou `ERROR`.

Exemple

```
char *b = "1234567890";
struct _data {
  float X;
  int   K;
  char  c;
} Data = { 3.14, 200, 'P'};
 .
 msgQSend(ifFile1, b, strlen(b)+1, WAIT_FOREVER, MSG_PRI_NORMAL);
 msgQSend(idFile2, (char*)(&Data), sizeof(struct _data),
                          WAIT_FOREVER, MSG_PRI_NORMAL);
 .
```

Utilisation par un programme d'interruption. La file de messages est un moyen de communication idéal entre les interruptions matérielles et le reste de l'application (figure 4.3). Dans le cas où un message est délivré par une IT, l'envoi doit se faire avec le paramètre `NO_WAIT`.

4.4 Réception d'un message

La réception se fait grâce à la fonction `msgQReceive` dont le prototype est :

```
STATUS msgQReceive( MSG_Q_ID idFile,
                    char *buffer,
                    UINT N,
                    int time_out)
```

Les paramètres sont les mêmes que pour la fonction `msgQSend`.

La valeur renvoyée est le nombre d'octets lus, ou `ERROR` en cas d'erreur.

Attention : cette fonction ne doit pas être appelée dans un programme d'interruption.

Exemple

```
char [16];
struct _data D;
 .
 msgQReceive(ifFile1, b, 16, WAIT_FOREVER);
```

```
msgQReceive(idFile2,(char*)(&D),sizeof(struct _data),WAIT_FOREVER);
.
```

4.5 Destruction d'une file de messages

C'est l'objet de la fonction `msgQDelete` dont le prototype est :

```
STATUS msgQDelete(MSG_Q_ID idFile)
```

Toutes les tâches en attente de lecture ou d'écriture sont alors débloquées avec un code de retour `ERROR`.

4.6 Applications

4.6.1 Modèle client-serveur

La tâche *serveur* propose des services aux clients (partage de matériel, ...). Les clients font des requètes à la tâche *serveur* (figure 4.2).

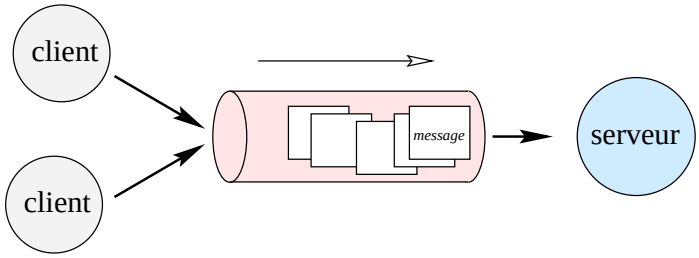


FIG. 4.2 – *Modèle «client-serveur»*

4.6.2 Acquisition de données en temps-réel

Il est fréquent d'avoir à réaliser l'acquisition de données sous interruption, et de devoir différer le traitement afin de respecter une durée courte pour le programme d'IT.

Il convient dans ce cas de déléguer le traitement des données à une tâche de priorité inférieure, pouvant être interrompue pendant son traitement par l'acquisition d'une nouvelle donnée.

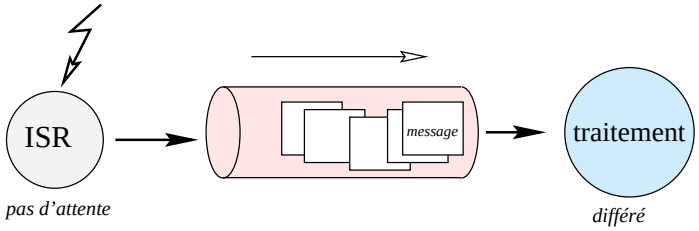


FIG. 4.3 – *Lien entre tâche immédiate et tâche différée.*

Quatrième partie

Caractéristiques industrielles

Chapitre 1

Les entrées / sorties industrielles

1.1 Lectures et écritures absolues en mémoire

Dans bien des applications industrielles, il est nécessaire de faire des E/S en écrivant directement dans les registes des composants, voire dans la mémoire à des adresses précises[1]. Avec VxWorks , l'accès impératif en utilisant un pointeur constant, provoque une exception. La solution prévue pour ces cas de figure est l'utilisation de *driver* de mémoire (*memDrv*). Le pilote *memDrv* permet au système d'accéder directement à la mémoire. Il n'y a aucun système de fichier d'installé[2]. La mémoire est directement accessible. L'adresse du premier octet et le nombre d'octets doivent être précisés à la création du descripteur.

1.2 Méthode

L'accès à la mémoire ne peut se faire qu'après avoir exécuté avec succès trois opérations préliminaires :

1. Lancement du pilote *memDrv* :

```
if (memDrv() == ERROR) {
  printf("Erreur d'exécution du pilote 'memDrv'\n");
  return 0;
}
```

2. Création d'un descripteur pour la zone de mémoire visée :

```
if (memDevCreate("/vmodIO", 0xffffa800, 0x200) == ERROR) {
  printf("Erreur de creation de  '/vmodIO'\n");
  return 0;
}
```

1. C'est le cas par exemple pour une carte d'imagerie industrielle, ou de la mémoire partagée entre plusieurs processeurs ...

2. Voir pour cela le pilote *ramDrv*.

La fonction `memDevCreate` prend en paramètre le nom qui est donné au descripteur, l'adresse de base de la zone visée, la taille (en octets) de cette zone.

3. Ouverture du descripteur, (comme pour un fichier) :

```
VmodIo = open("/vmodIO",O_RDWR, 0);
if(VmodIo ==  ERROR) {
  printf("Erreur d'ouverture de la VmodIo\n\r");
  return 0;
}
```

Dans cet exemple, le module est ouvert en lecture et en écriture (`O_RDWR`). C'est le cas d'utilisation le plus fréquent. Le paramètre de mode (ici mis à 0) n'est, en principe, pas exploité dans le cas des pilotes de mémoire.

1.3 Accès à la mémoire

L'écriture et la lecture des informations se fait de manière similaire à ce que l'on fait pour les accès aux fichiers sélectifs. Pour écrire un caractère c à la $n^{\text{ème}}$ adresse du bloc de mémoire initialisé, il faudra :

1. Positionner un pointeur sur cet octet. Par exemple, si le N^o identificateur du descripteur est `VmodIo` :

```
ioctl(VmodIo, FIOSEEK, n);
```

La fonction `ioctl` avec le paramètre `FIOSEEK` place le pointeur courant à la $n^{\text{ème}}$ place du bloc de mémoire considéré comme un fichier.

2. Il ne reste plus qu'à écrire l'octet ;-)

```
write(VmodIo, &c, 1);
```

1.4 Exemple complet

Cet exemple met en œuvre le module `VMOD-TTL` de la carte `VMOD-IO`.

Le module est logé sur le connecteur N^{o}0 de la carte. Son adresse de base est donc `0xFFFFA800`. Un module occupe au plus `0x200` octets. Les adresses relatives des registres d'entrées/sorties sont données dans le tableau 1.1

Adresse	Fonction	Accès
0	Registre de données du port C (`CIO Z8536`)	*r/w*
2	Registre de données du port B (`CIO Z8536`)	*r/w*
4	Registre de données du port A (`CIO Z8536`)	*r/w*
6	Registre de contrôle (`CIO Z8536`)	*r/w*

TAB. 1.1 – *Registres du module `VMOD-TTL`.*

Les E/S de `VMOD-TTL` sont accessibles sur un connecteur standard (DB25) (figure 1.1).

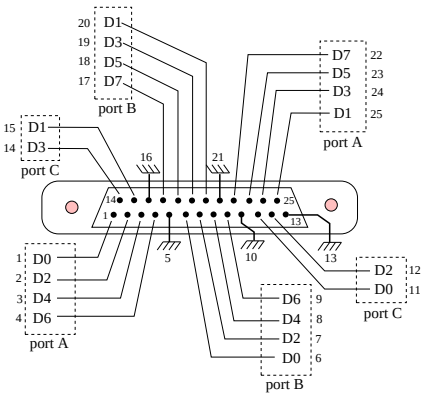


FIG. 1.1 – *Connecteur DB25 (vue de face)*

Afin de rendre la programmation plus conventionnelle (en tout cas en ce qui concerne les E/S), deux fonctions ont été écrites :

- `outpw(int n, unsigned short S)` : écrit un entier court à l'adresse n du module.
- `char inp(int n)` : renvoie le $n^{\text{ème}}$ octet lu sur le module.

L'application finale est un processus infini (`while (1) ...`) qui fait clignoter la sortie correspondant au *bit* b_0 du port A (programmé en sortie) du `z8635` présent sur le module.

```
#include <stdio.h>
#include <stdlib.h>
#include "vxworks.h"
#include "ioLib.h"
#include "selectLib.h"
#include "fcntl.h"

/* Sorties parallèles */
int VmodIo;

/* Identificateurs des tâches :*/
int id_led;

void outpw(int n, unsigned  short S) {
  ioctl(VmodIo, FIOSEEK, n);
  write(VmodIo, &S, 2);
}

char inp(int n){
  char c;
  ioctl(VmodIo, FIOSEEK, n);
  read(VmodIo, &c, 1);
  return c;
}

void initVmodIo() {
  outpw(6, 0);    outpw(6, 1);    outpw(6, 0);  /* Reset */
  outpw(6, 0x24); outpw(6, 0xff);  /* 0xff dans le port 0x24; A : sortie */
  outpw(6, 0x2b); outpw(6, 0xff);  /* 0xff dans le port 0x2b; B : entrée */
  outpw(6, 6);    outpw(6, 0xf);   /* 0xf dans le port 6; C : sortie */
  outpw(6, 1);    outpw(6, 0x94);  /* 0x94 dans le port 1; A,B,C : validés*/
}

tache_led() {
  initVmodIo();
  while (1) {
    printf("1");    taskDelay(40);    outpw(4, 1);
    printf("0");    taskDelay(20);    outpw(4, 0);
  }
}

void usrAppInit (void) {
  if (memDrv() == ERROR) {
    printf("Erreur d'exécution du driver 'memDrv'\n");
        return 0;
  }
  if (memDevCreate("/vmodIO", 0xffffa800, 0x200) == ERROR) {
    printf("Erreur de creation de  '/vmodIO'\n");
        return 0;
  }
  VmodIo = open("/vmodIO",O_RDWR, 0);
```

```
  if(VmodIo ==  ERROR) {
    printf("Erreur d'ouverture de la VmodIo\n\r");
        return 0;
  }
  id_led = taskSpawn("Led",90,0,4000,tache_led,0,0,0,0,0,0,0,0,0,0);
  printf("\n lancement de la tache %d",id_led);
        if (id_led == ERROR){
    printf("Erreur de création de 'Led'\n\r");
    return 0;
  }
  printf("Tâche  lancée\n\r");
  return 1;
}
```

1.5 Application

Question 1

Testez le programme précédent.

Question 2

Compléter le fichier en y ajoutant les fonctions

- `inpw` : écriture d'un entier court à une adresse donnée
- `outp` : lecture d'un octet à une adresse donnée.

Question 3

Écrire une fonction qui permet de programmer un port avec une valeur donnée. Le prototype de cette fonction est

```
void ecritPort(int num_port, unsigned char valeur);
```

Utiliser cette fonction pour améliorer la fonction `initVmodIo()`. (Ne pas modifier la ligne correspondant au *reset*).

Question 4

Écrire une fonction qui affiche en notation binaire l'état du port B et du port C (programmés en entrée).

Question 5

Compléter l'application avec une tâche qui lit et affiche deux fois par seconde l'état des entrées.

Chapitre 2

Les interruptions matérielles

Le traitement des interruptions matérielles est primordial pour l'informatique temps-réel. C'est ce qui la rend sensible aux événements extérieurs. Le traitement se fait à travers un mécanisme particulier. Le temps qui s'écoule entre l'événement physique qui provoque l'interruption et le traitement de la première instruction du programme associé est une caractéristique importante.

On utilisera les interruptions dans différents cas :

† production d'événements périodiques (horloge)

† traitement d'événements extérieurs «imprévisibles» (instant aléatoire)

† gestion de périphériques lents (coupleurs série, ...)

2.1 Le mécanisme des interruptions

2.1.1 Généralités

Une interruption est un signal électrique utilisé par un périphérique pour indiquer au processeur qu'un événement s'est produit : arrivée d'un caractère sur un port série, fin d'émission d'un caractère, évolution sur une entrée tout ou rien, fin de conversion analogique numérique, ...

Les sources d'interruptions pouvant être multiples, il est en général prévu de leur attribuer une priorité. Par exemple le processeur 68000 de MOTOROLA peut traiter jusqu'à 7 niveaux de priorité, dont un est non masquable.

On peut considérer quelque soit le type d'interruption, de périphérique et de processeur utilisé, trois étapes dans le processus d'interruption :

1. la fin du traitement en cours,
2. la prise en compte de l'interruption,
3. le traitement de l'interruption.

Une interruption est la conséquence directe de la validation d'une ligne d'interruption appelée en général `IRQ` (`Interrupt ReQuest`). Le processeur mémorise la condition d'interruption sur un état bas de cette ligne.

D'une manière générale, la plus petite partie insécable dans l'exécution d'un programme est l'instruction. Un cycle d'instruction est lui-même com-

posé de plusieurs cycles bus (lecture ou écriture) , eux-mêmes constitués de plusieurs cycles d'horloge. Les nombreuses instructions d'un processeur combinées aux différents modes d'adressage font qu'il n'est pas possible de connaître avec précision le temps qui sépare l'envoi de l'interruption par le périphérique du moment de sa prise en compte. Au mieux, on peut dire que le temps maximum garanti correspont à l'exécution de l'instruction la plus longue[1].

Interruptions et multi-tâche

L'asynchonisme inhérent aux interruptions impose de prendre un certain nombre de précautions :

- il faut éviter d'accéder à des régions critiques (risque de blocage),
- le code du programme d'interruption doit être le plus court possible,
- il est souhaitable de se limiter à des opération d'entrées/sorties simples, et/ou des signalisations d'événements.

La prise en compte de l'interruption

La prise en compte d'une interruption intervient à partir du moment où le processeur réalise des opérations propres à l'interruption reçue. En général, toutes les procédures de prise en compte des interruptions effectuent le même type d'opérations. Elles placent le contexte courant du processeur dans la pile (opération de sauvegarde), recherchent une adresse appelée «vecteur» (souvent associée à un «numéro de vecteur») spécifique à l'interruption, puis sautent à cette adresse.

Ces différentes étapes sont indépendante de l'exécutif temps-réel. C'est simplement un automatisme «câblé» du processeur, dépendant uniquement du matériel.

Le traitement de l'interruption

Le traitement de l'interruption débute à la première instruction du programme spécifique à l'interruption, couramment appelé programme ou **routine d'interruption** ou *ISR*[2]. L'application ou l'exécutif intervient uniquement à ce niveau.

Pour la plupart des processeurs courants, aucun traitement n'est à effectuer dans le programme d'interruption pour rechercher la source d'interruption. La procédure est souvent la suivante :

1. acquitter (ou réarmer) l'interruption,
2. lire (ou écrire) les données,
3. éventuellement réactiver ou signaler un événement à une tâche différée (par un sémaphore binaire).

1. La possibilité de masquer les interruptions complique cette notion
2. Interrupt Service Routine

2.1.2 Les vecteurs d'interruptions

En général, une source d'interruption (périphérique) est associée à un vecteur. Au moment de l'acquittement de l'interruption, le processeur obtient un numéro vecteur en provenance du périphérique. L'adresse du programme à exécuter est une fonction de ce numéro de vecteur.

> Le **vecteur** d'interruption est l'adresse du programme d'interruption. Le **numéro de vecteur** est un code sur 8 *bits* obtenu au moment de la reconnaissance d'interruption et utilisé par le processeur pour calculer le vecteur.

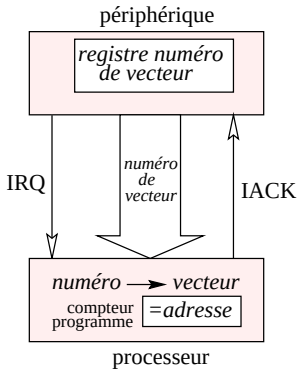


FIG. 2.1 – *Génération d'un numéro de vecteur.*

2.2 Mise en œuvre avec VXWORKS

VXWORKS propose plusieurs fonctions de gestion des interruptions dans les librairies `intLib` et `intArchLib`.

2.2.1 Détermination des vecteurs libres

Il est essentiel, lors d'une configuration matérielle de connaître les vecteurs disponibles. Certains sont en effet utilisés par le noyau. Lorsqu'un numéro de vecteur est libre, VXWORKS l'associe automatiquement à une routine d'interruption «par défaut» qui ne fait rien. Cette routine s'appelle `excIntStub`. Faire l'inventaire des vecteurs libres revient donc à voir quels numéros de vecteurs sont associés à cette routine. L'application entre l'ensemble des numéros de vecteurs et l'ensemble des vecteurs est surjective.

L'exemple suivant affiche la liste des numéros de vecteurs occupés.

```
/* vecteur.c - montre les numéros de vecteurs d'interruptions utilisés */

#include "vxWorks.h"
#include "intLib.h"
```

`intConnect`	associe une fonction `'C'` à une interruption
`intContext`	renvoie `TRUE` si le contexte courant est celui d'une interruption (utile dans une fonction pouvant être appelée à partir d'un programme d'interruption ou d'une tâche)
`intCount`	renvoie le nombre d'interruptions imbriquées
`intLevelSet`	fixe le niveau d'interruptions autorisées
`intLock`	interdit les interruptions de niveau spécifié par un appel préliminaire à `intLockLevelSet`.
`intUnlock`	autorise à nouveau les interruptions interdites par `intLock`. Elle prend en général comme argument la valeur renvoyée par l'appel préliminaire à `intLock`.
`intVectBaseSet`	précise l'adresse de la table des vecteurs
`intVectBaseGet`	renvoie l'adresse de la table des vecteurs
`intVectSet`	associe une adresse de routine à un vecteur d'interruption
`intVectGet`	renvoie l'adresse d'une fonction associée à un vecteur

TAB. 2.1 – *Fonctions de gestion des interruptions.*

```
#include "stdio.h"
#include "iv.h"

extern excIntStub; /* programme d'interruption par défaut */

void afficheVecteurs(void) {
  int i;
  int vecBase = (int) intVecBaseGet();

  for (i = (vecBase + 64); i < (vecBase + 256); i++) {
    if ( *(int *)(UINT)(INUM_TO_IVEC(i)) != (int)&excIntStub) {
      printf ("Le numéro de vecteur %d est utilisé.\n", i);
    }
  }
}
```

2.2.2 Installation d'une routine d'interruption

VXWORKS propose la fontion `intConnect()` qui permet de connecter une fonction `C` à n'importe laquelle interruption.

```
STATUS intConnect(VOIDFUNCPTR *p_vecteur,
                  VOIDFUNCPTR routine,
                  int parametre)
```

avec :

– `p_vecteur` est l'adresse ou sera stocké le pointeur sur la routine d'interruption (c'est le vecteur lui-même).

– `routine` est l'adresse de la fonction `C` (représentée dans le programme par le nom de la fonction).

– `parametre` est un paramètre de type entier qui sera passé à la fonction au moment de l'interruption. Ceci permet d'utiliser la même fonction pour plusieurs interruptions différentes. Elles seront alors différenciées par le paramètre.

Il est possible (et souhaitable) de calculer l'adresse `p_vecteur` à partir du numéro de vecteur d'interruption, grâce à la «macro» `INUM_TO_IVEC`. Ainsi, l'adresse de la routine d'interruption associée au numéro de vecteur `V` est écrite à l'adresse de référence `INUM_TO_IVEC(V)`.

Exemple dans lequel on installe une routine d'interruption dont la source est une carte se trouvant sur le bus `VME`, configurée pour fournir le numéro de vecteur `0xAE` lors de la reconnaissance de l'interruption.

```
if (intConnect(INUM_TO_IVEC(0xAE), VmodIoISR, 0) == ERROR) {
  printf("Erreur d'installation de l'interruption\n\r");
  return;
}
sysIntEnable(3);    /* autorisation des IT VME de niveau 3 */
```

2.2.3 Test d'une interruption *bus* par simulation

La fontion `sysBusIntGen()` permet de provoquer une interruption *bus* matérielle. Ceci permet, de tester une routine d'interruption, c'est à dire de vérifier son installation et son comportement, indépendemment du matériel. Cette possibilité est intéressante pour identifier un problème lors du développement d'une application.

```
STATUS sysBusIntGen(int niveau,
                    int numVecteur)
```

avec :

– `niveau` le niveau d'interruption à provoquer;
– `numVecteur` le numéro de vecteur d'interruption à fournir lors de l'identification de l'interruption.

Ainsi, pour provoquer une interruption de niveau 3 et de numéro de vecteur `0xAE`, il suffit d'écrire

```
sysBusIntGen(3, 0xAE);
```

Cette ligne provoquera l'exécution de la routine `VmodIoISR` de l'exemple précédent. Dans le cas contraire, il faudra vérifier l'installation de cette routine en tant que programme d'interruption.

Remarque : la fonction

```
sysIntEnable(int niveau)
```

permet d'autoriser une interruption *bus* de niveau passé en paramètre; la fonction

```
sysIntDisable(int niveau)
```

a l'effet contraire : elle inhibe les interruptions *bus* du niveau donné en argument.

2.3 Exemple complet

Il s'agit de déclencher une interruption périodique à partir d'un *timer*. Le *timer* utilisé est le `CIO Z8536` de la carte `VMOD-TTL` .

2.3.1 Programmation de bas niveau du matériel

Le fichier suivant donne les fonctions d'accès aux registres de la carte ainsi que la fonction d'initialisation du *timer*.

```
#include "ioLib.h"
static int VmodIo;

/* écriture d'un mot S à une adresse n */
void outpw(int n, unsigned  short S) {
  ioctl(VmodIo, FIOSEEK, n);
  write(VmodIo, &S, 2);
}

/* Initialisation d'un registre du cio */
void cioReg(unsigned char num_registre, unsigned char valeur) {
  outpw(6, num_registre);  outpw(6, valeur);
}

/* Reset du cio */
void cioReset() { outpw(6,0); outpw(6,1); outpw(6,0); }

/* Programmation du timer 1 avec interruption en fin de comptage */
void initTimer(unsigned int consigne){
  cioReset();
  taskDelay(10);
  cioReg(0,0xAE);  /* valid it timer sans vecteur */
  cioReg(0x1C, 0); /* timer 1 utilisé */
  cioReg(0x10,(consigne >> 8) & 0xFF); /* consigne MSB1 */
  cioReg(0x11,consigne & 0xFF);  /* consigne LSB1 */
  cioReg(0xA,0xC0); /* autorise les interruptions timer */
  cioReg(1,0x40);   /* valide le timer 1 + timer indépendant dans  MCCR */
  cioReg(0xA,0x6);  /* lancement timer 1 */
}

/* Acquittement de l'IT du timer 1 */
void acquitItTimer() { cioReg(0xA,0x20); }

/* Réarmement du timer */
void reactiveTimer() { cioReg(0xA,0x6); }

/* Accès à la mémoire occupée par la carte VMOD IO */
int init_vmodIO() {
  if (memDrv() == ERROR)    return ERROR;
```

```
  if (memDevCreate("/vmodIO", 0xffffa800, 0x200) == ERROR)   return ERROR;
  if ((VmodIo = open("/vmodIO",O_RDWR, 0)) ==  ERROR) return ERROR;
  return OK;
}
```

Remarques :

- Le paramètre `consigne` de la fonction `initTimer` est un mot de 16 *bits*, représentant un entier non signé qui est le nombre que le compteur doit atteindre avant de déclencher l'interruption. Le comptage est cadencé par une horloge à 4 MHz.
- Les accès aux registres du `CIO Z8536` se font en deux fois : il faut spécifier le numéro du registre visé, puis lire ou écrire la valeur.
- Le comptage s'arrête au bout d'une fois. Il faut réarmer le *timer* pour obtenir une nouvelle interruption.
- Compte tenu de la conception de la carte `VMOD-TTL` et de sa configuration, le numéro de vecteur est fixé par des cavaliers à 0xAE.

2.3.2 Création d'une tâche immédiate

L'application suivante permet de déclencher une interruption périodique.

```
#include <stdio.h>
#include "intLib.h"
#include "iv.h"
#include "logLib.h"

#define NIVEAU 3
#define NUM_VECTEUR 0xAE
#define MASQUE 3
#define PREDIVISEUR_TIMER 40000  /* prédiviseur du timer */

void routineIt(int param) {
  static int compteur=0;
  acquitItTimer();
  compteur++;
  if (compteur%param == 0){
    logMsg(" <<IT Timer>>\n",0,0,0,0,0,0);
  }
  reactiveTimer();
}

int installeIt(void) {
  int result;
  result = intConnect(INUM_TO_IVEC(NUM_VECTEUR),
                      (VOIDFUNCPTR)routineIt,
                      100);
  if (result == ERROR) return ERROR;
  intLevelSet(MASQUE);
  sysIntEnable(NIVEAU);
  return OK;
```

```
}

void usrAppInit(void) {
  if (init_vmodIO()==ERROR) return;
  if (installeIt()==ERROR) return;

  initTimer(PREDIVISEUR_TIMER);

  while (1) {
    taskDelay(30); /* 0.5 s */
    printf("Travail tache principale\n");
  }
}
```

Avec une horloge à 4 MHz et un prédiviseur de 40000, la période entre deux interruptions est donc de $\frac{40000}{4\times10^6} = 0.01$ s.

Compte tenu de l'utilisation faite de la variable statique de la routine d'interruption, ainsi que de la valeur du paramètre reçu (100) , un message sera affiché toutes les secondes ($= 100 \times 0.01$ s).

On remarquera

- la procédure utilisée pour installer la routine d'interruption, puis pour autoriser les interruptions,
- les différentes étapes de la routine d'interruption :
  1. acquittement de l'interruption,
  2. action,
  3. réarmement du *timer*.

2.4 Applications

Question 1 Table des vecteurs

Écrire une fonction `int testNumVecteur(unsigned char numVecteur)` qui renvoie `ERROR` si le numéro de vecteur passé en argument est utilisé par le système, et `OK` sinon.

Question 2 `sysBusIntGen()`

Modifier l'exemple fourni pour que l'interruption soit simulée par une action sur le clavier. On pourra :

- créer une tâche `lireClavier`,
- utiliser la fonction système `sysBusIntGen)()`.

On rappelle que le numéro de vecteur est 0xAE et que le niveau d'interruption est 3.

Question 3 Tâche «horloge»

Reprendre l'application du paragraphe 2.2.2 sur les sémaphores binaires en remplaçant la tâche `TH` par une tâche matérielle.

Bibliographie

[1] P. Pillot A. Dorseuil. *Le temps réel en milieu industriel.* dunod, 1991.

[2] E. Schonberg C. Comar, F. Gasperoni. *The GNAT Project: A GNU-Ada9X Compiler.* Courant Institute of Mathematical Sciences - New York University.

[3] M. Boasson C.H. Smedema, P. Medema. *Les langages de programmation, Pascal, Modula, CHILL, Ada.*

[4] Dijkstra. An introduction to concurrent programming. *1968.*

[5] A.B. Fontaine. *modula-2.* masson, 1986.

[6] P Lignelet. *Structures de données avec ADA.*

[7] J. Longchamp. *Les langages de programmation, concepts essentiels, évolution et classification.* masson, 1989.

[8] J.P. Perez. *Systèmes temps réel.* dunod, juillet 1990.

[9] Real-time Systems'93. *Actes des conférences*, 1993.

[10] D. Tschirhart. *Commande en temps réel.* dunod, juin 1990.

[11] H. Emmelmann J. Vollmer. *GMD MODULA SYSTEM MOCKA.* Institut für Programm und Datenstrukturen - Universität Karsruhe, Avril 1994.

[12] WindRiver. *VxWorks, Programmer's guide*, 1999.

Index