Un environnement pseudo-parallèle pour Unix

Joel Berthelin, Bernard Cousin

▶ To cite this version:

Joel Berthelin, Bernard Cousin. Un environnement pseudo-parallèle pour Unix. 1989, pp.1-18. hal-01551471

HAL Id: hal-01551471
https://hal.science/hal-01551471

Submitted on 30 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Un environnement pseudo-parallèle pour le système Unix

Joël BERTHELIN Bernard COUSIN

Laboratoire MASI
Université Pierre et Marie Curie
4 place Jussieu
75252 PARIS cedex 05

Résumé

Le pseudo-parallélisme permet à plusieurs applications de s'exécuter en simultanéité apparente sur une même machine. Fondée essentiellement sur la notion de coroutine, ce concept existe dans de nombreux systèmes et langages. Nous proposons un environnement efficace et élémentaire permettant l'exploitation aisée de ce concept inexistant en langage C, afin de bénéficier d'une approche simplifiée du parallélisme. Cet environnement portable est totalement écrit en langage C, et n'utilise que des fonctions de la bibliothèque standard du système Unix. Les évaluations de performances effectuées ont permi de dégager une baisse importante de l'utilisation du processeur grâce à cet environnement.

Mots clés : pseudo-parallélisme, parallélisme, coroutine, système.

Abstact

The quasi-parallelism enables several applications in one processor to be simultaneously executed. Based on the coroutine notion, this concept is used in numerous systems and languages. In order to have a straightforward approach of parallelism, we propose an efficient and basic environment. It allows an easily use of the coroutine concept which is non-existent in the C language. This portable environment in its entirety is coded in C language, and only uses standard library functions of the Unix system. The evaluation shows the good performance of our quasi-parallel environment.

Keywords: quasi-parallelism, parallelism, coroutine, system.

1. Introduction

L'évolution des outils de programmation doit suivre celle de l'architecture (machines parallèles et systèmes répartis), et celle des applications de plus en plus interactives (réagissant à des événements externes et asynchrones). C'est pourquoi nous présentons un environnement propice à la **programmation pseudo-parallèle,** ce qui nous permettra d'introduire les principaux concepts liés au parallélisme.

Les efforts de conception des systèmes d'exploitation ont provoqué l'émergence de la notion de "**coroutine**", notamment pour la gestion en temps partagé. Cette notion, à la base de la gestion du pseudo-parallélisme, permet à plusieurs applications de s'exécuter en simultanéité apparente sur une même machine. Elle existe de manière évidente ou sous-jacente dans de nombreux langages (Simula[1] Modula-2[2], Scheme, Smalltalk[3], etc...) et applications (multi-fenêtrage, compilateur, système d'exploitation, etc...).

Le noyau du système Unix est lui aussi conçu autour de cette notion. Malheureusement, ce mécanisme de coroutine est inaccessible à l'utilisateur de ce système. C'est pourquoi nous proposons un **environnement simple** écrit en langage C permettant l'exploitation aisée de cette notion.

2. Les coroutines

2.1 Présentation

Préalablement à toute étude, nous allons définir à quel type de systèmes parallèles et d'applications parallèles la notion de coroutine est adaptée. Pour ce faire, on peut classifier dans une première approche, **les différents systèmes parallèles** avec ou sans mémoire commune.

Les systèmes parallèles à mémoire commune ont des processeurs qui partagent un espace commun de mémorisation. Les échanges peuvent alors s'effectuer à relativement grand débit (vitesse d'accès à la mémoire), ils sont de plus rémanents (conservation des informations par la mémoire), mais cela impose aux processeurs d'être très proches (généralement situés dans la même machine). Les machines multiprocesseurs en sont un bon exemple.

Dans les systèmes distribués cet espace commun n'existe pas. Les échanges s'effectuent alors par l'intermédiaire d'un réseau (ou bus), ils sont moins rapides (à technologie égale), ils sont fugitifs (ils passent sur le réseau). Généralement le mécanisme d'échange utilise des messages. L'architecture du système est plus souple, on rencontre des systèmes parallèles, soit situés dans une même machine, soit éloignés de plusieurs mètres (plusieurs processeurs reliés par un réseau local).

A ces deux classes, on peut habituellement associer respectivement, deux modes de synchronisations. Une synchronisation forte ("tightly coupled"), où les processus s'échangent de manière répétée de nombreuses informations. Une synchronisation faible ("loosely coupled"), où les processus des différents processeurs ne partagent que peu d'informations pour mener à bien leur exécution.

Il est clair que la première classe d'architecture répondra mieux aux besoins du premier mode de synchronisation. Néanmoins cette architecture, bien que plus performante, est souvent plus coûteuse, car elle demande le développement d'un matériel à même de régler les problèmes d'accès multiples à la mémoire partagée. C'est à elle que nous nous intéresserons dans cet article.

On peut distinguer différents **types de parallélisme** utilisant une mémoire commune :

Le premier type est basé sur un ordinateur possédant plusieurs processeurs capables d'assurer chacun une partie de l'exécution (ces processeurs sont en général identiques). Les synchronisations dans ce genre de machine doivent s'effectuer fréquemment et à un niveau très fin. Ce genre de machine possède donc des instructions spécialisées assurant les traitements parallèles et leurs synchronisations. Ce type de machine autorise un vrai parallélisme du traitement des exécutions qui se déroulent en parallèle sur plusieurs processeurs.

Le deuxième type est caractéristique des machines où l'on ne parallélise qu'une certaine catégorie de traitement (généralement les entrées/sorties), à l'aide de processeurs spécialisés. Seuls les traitements appartenant à ces catégories sont réalisés en parallèle. En fait, l'immense majorité des ordinateurs possède cette caractéristique.

Le troisième type est possible sur les systèmes ne possédant qu'un seul processeur de traitement. Il consiste à multiplexer dans le temps les différents processus à exécuter sur le seul processeur. Les ordinateurs autorisant une gestion multi-tâches possèdent un système d'exploitation permettant ce multiplexage de la puissance de traitement du processeur entre les différents processus (appelé gestion en temps partagé "time sharing"). Ce type de parallélisme n'est qualifié que de pseudo-parallèlisme. Car le parallélisme apparent entre les processus (il existe effectivement plusieurs processus en cours d'exécution) disparaît dès que l'on regarde au niveau des instructions (à tout moment, il n'y a qu'une instruction d'un des processus en exécution). Il n'y a qu'un seul processeur, qui ne supporte qu'une seule exécution séquentielle.

Nous développons notre environnement pour ce troisième type de parallélisme.

Notre environnement veut promouvoir l'utilisation du pseudo-parallélisme, qui s'avère dans de nombreuses applications propice et efficace. Il permet au programmeur de bénéficier d'**une approche simplifiée du parallélisme**. Notre objectif est d'offrir une alternative aux trois façons de programmer suivantes :

- Une programmation en langage séquentiel strict, utilisant un seul processus.

Dans cette approche, la conception de certaines applications devient vite difficile et leur maintenance inextricable. Surtout si ces applications gèrent plusieurs entités possédant un contexte propre mais évolutif, et interagissent à des instants variables suivant l'évolution de leur exécution. Les coroutines proposent une façon plus simple et plus naturelle d'appréhender ces applications.

- Une programmation en langage séquentiel utilisant une coopération de plusieurs processus.

Dans cette approche, il est coûteux de faire partager aux différents processus un même contexte car les mécanismes disponibles ("pipe", "socket", segment partageable, sémaphore, signaux, etc...) ont souvent une mise en oeuvre complexe, et sont parfois mal adaptés aux besoins particuliers. Enfin, la gestion des différents processus est assurée par le "scheduler" du système, ce qui naturellement est beaucoup moins souple et moins efficace que celle offerte par notre environnement qui laisse entière liberté au programmeur.

- Une programmation dans un langage intégrant les concepts de pseudo-parallélisme.

Ces langages sont souvent d'un niveau conceptuel plus élevé, et intègrent des notions supplémentaires qui risquent de dérouter le néophyte. Notre environnement n'introduit aucune contrainte supplémentaire par rapport au langage C (pas de nouveaux mots clefs, pas de contraintes d'utilisation), et n'offre qu'un seul et nouveau concept : les coroutines! On a besoin de pseudo-parallélisme directement en C, sans l'intermédiaire d'un compilateur disproportionné avec certaines applications, et qui risque de poser des problèmes de portabilité.

2.2 Définition

Une coroutine est une **unité élémentaire d'exécution séquentielle**, possédant en propre un contexte local (généralement constitué d'une pile), et partageant avec les autres coroutines un espace global (une mémoire commune). Une coroutine (l'appelante) appelle une autre coroutine

(l'appelée) et lui cède son droit d'exécution en se suspendant. Il y a donc toujours au plus une seule coroutine en activité, les autres étant toutes suspendues. L'enchaînement des exécutions des coroutines est exprimé de manière explicite : chaque coroutine possède un nom unique et connu de tous.

L'exécution d'une coroutine peut être perçue comme l'exécution d'un sous-programme, qui peut être interrompue pour exécuter une autre coroutine, puis reprise ultérieurement. Plus précisément, un sous-programme a deux mécanismes de base : l'appel (nommant explicitement le sous-programme appelé), et le retour (intervenant implicitement en fin de code de l'appelé). Alors que la coroutine ne possède qu'un mécanisme (communément appelé "resume"), servant à la fois d'appel, de reprise et de retour (en nommant explicitement à chaque fois la coroutine suivante qui est reprise là où elle s'était arrêtée).

La notion de coroutine a déjà prouvé sa grande généralité, et son adaptation à de nombreux problèmes. Elle seule semble propice à répondre à nos nombreuses contraintes :

- Implantation dans un système Unix pour permettre une grande diffusion. Cependant ce choix présente quelques difficultés car les processus Unix sont séquentiels. Les chapitres suivants montreront comment contourner ces problèmes.
- Ecriture en langage évolué pour assurer une plus grande portabilité, et pour permettre une mise en oeuvre aisée.
- Efficacité tant en occupation mémoire, et nombre de fonctions de base, qu'en temps d'exécution. Ce qui permet à notre environnement de ne pas perdre les avantages inhérents à la simplicité de la notion de coroutine.
- Notre environnement doit être stable et cohérent, car la mise au point d'applications parallèles est beaucoup plus difficile que pour les applications séquentielles. Notre environnement veut promouvoir l'utilisation de coroutine. Il doit donc être d'un emploi aisé pour faciliter l'abord de la programmation pseudo-parallèle aux novices.

Un programme écrit en langage C se caractérise par quatre espaces : le code (les fonctions), les données (les variables globales), la pile (les variables locales, et les paramètres), le tas (les variables dynamiques). Il suffit d'attribuer **à chaque coroutine une pile** distincte pour permettre une exécution pseudo-parallèle de plusieurs coroutines à l'intérieur d'un même programme. Cette attribution permet à toutes les coroutines de partager code, données globales, tas. Et chaque coroutine est assurée de posséder toutes les propriétés inhérentes au langage C (réentrance, récursivité, localité des variables, sous-programme, paramétrage).

Cependant le compilateur impose les contraintes suivantes sur notre implantation des coroutines:

- L'ensemble du code (toutes les fonctions) est accessible à toutes les coroutines. N'importe quelle fonction peut donc être utilisée par n'importe quelle coroutine. Puisque dans le langage C, il n'y a aucun moyen d'imbriquer les déclaration de fonctions (au contraire d'Algol ou de Pascal). Cependant pour chaque coroutine, on distingue une fonction particulière : la fonction d'appel. La fonction d'appel est la première fonction exécutée lors du démarrage de la coroutine.
- De manière similaire, il n'existe que deux espaces de données dans un programme écrit en C, les données globales à l'ensemble des sous-programmes, et les données locales à chaque sous-programme. Il en sera de même pour nos coroutines. Ainsi, il n'est pas possible à cause du compilateur, d'avoir des données propres à chaque coroutine et cependant globales à l'ensemble de ses sous-programmes.

Le choix d'un autre langage support aurait permis, nous en sommes certains, de lever sans problème l'ensemble de ces contraintes. Néanmoins le développement de notre environnement pseudo-parallèle sur un système Unix nous a conduit à conserver le langage C.

Notre environnement est totalement **écrit en langage C**, et n'utilise que des fonctions de la bibliothèque **standard** du système Unix. Il ne comporte que 3 fonctions, dont le code figure en annexe :

```
CO_CONTEXT *co_alloc(longueur)              /* Allocation d'un contexte de coroutine */
int      longueur;                          /* Taille du contexte en octets */

co_init(coroutine, fonction, paramètre)     /* Initialisation d'une coroutine */
CO_CONTEXT * coroutine;                     /* Contexte de la coroutine à initialiser */
int*    (*fonction) ()                      /* Fonction d'appel de la coroutine */
int *    paramètre;                         /* Paramètre de la fonction d'appel */

resume(coroutine, in, out)                  /* Exécution d'une coroutine */
CO_CONTEXT * coroutine;                     /* Contexte de la coroutine à reprendre */
int *   in;                                 /* Paramètre passé à la coroutine appelée */
int *   out;                                /* Paramètre reçu de la coroutine appelante */
```

La création d'une coroutine consiste à réserver un nouvel espace pour sa pile (co_alloc()), et à initialiser cette pile avec une structure compatible avec celle du compilateur (co_init()). La commutation entre deux coroutines revient à échanger le pointeur de pile (resume()).

La mise en place de l'ensemble de ces fonctions (et d'autres plus évoluées...) dans une librairie permet à chacun d'y accéder, d'en faciliter la mise en oeuvre, et d'en promouvoir l'utilisation [7]. Sur les ordinateurs où nous avons réalisé l'implantation, la librairie "/usr/lib/liblocal/cor.a" contient les principales fonctions, et le fichier "/usr/include/local/cor.h" contient les types et constantes. Un fichier d'aide en ligne "/usr/cat_man/p_man/man3/local/coroutine" complète notre environnement pour le pseudo-parallélisme.

3. Réalisation

3.1 Introduction

Comme nous venons de le dire, le mécanisme de base de commutation des coroutines est fourni par un appel à la fonction "resume()". Cette fonction permet de suspendre l'exécution de la coroutine appelante et d'enchaîner sur l'exécution de la coroutine appelée. Cette dernière est reprise à l'endroit du code où elle avait auparavant été suspendue par un précédent appel de la fonction "resume()".

On note l'unicité et la symétricité de ce mécanisme, au contraire de celui associé au sous-programme, qui est double (appel et retour) et de ce fait distingue appelant et appelé. De plus, le contexte associé à chaque occurrence d'un sous-programme ne dure que le temps de l'appel de ce sous-programme, tandis que celui associé aux coroutines est rémanent. Ainsi, les objets locaux aux sous-programmes sont créés à chaque appel, détruit au moment du retour d'appel, le code lui-même redémarre systématiquement au même point d'entrée à chaque appel du sous-programme. Par contre, durant la suspension d'une coroutine, les objets locaux appartenant à cette coroutine sont conservés, et l'exécution reprend là où on l'avait suspendue.

3.2 Le Contexte de coroutine (CONTEXT)

On associe à chaque coroutine l'ensemble de ses objets, ce que l'on nomme son contexte. Le contexte est modifié lors de l'exécution de la coroutine, et est sauvegardé lors de la suspension de cette dernière. Commuter des coroutines, revient à substituer au contexte de coroutine courante (de l'appelante) celui de la coroutine appelée. En fait cette commutation demande d'effectuer deux opérations :
(1) **la sauvegarde** du contexte de la coroutine appelante (pour sa reprise ultérieure),
(2) **la restauration** à partir de la sauvegarde du contexte de la coroutine appelée.

Dans notre implantation, on peut définir **trois types de contextes** associés aux différents objets manipulés par le programme (Figure 1):

- Le contexte associé à chaque sous-programme contient ses variables locales, ses paramètres, et

une sauvegarde du descriptif du sous-programme appelant. Ce descriptif est utilisé lors du retour pour restaurer le contexte du sous-programme appelant. Chaque appel d'un sous-programme crée un contexte qui lui est associé.

- Le contexte de chaque coroutine (CO_CONTEXT) contient son espace propre et son descriptif (CO_DESCR). L'espace propre à chacune des coroutines est déterminé par l'enchaînement des sous-programmes et est défini par leur contexte. Le descriptif de coroutine contient une sauvegarde du jeu de registres (registre compteur ordinal, registre pointeur de pile, registres généraux, etc...).

- Le contexte du programme comporte l'espace global du programme et un contexte pour chacune des coroutines du programme. L'espace global contient les variables globales, dynamiques, ou externes accessibles par l'ensemble des coroutines. On place habituellement dans le contexte de programme les informations relatives au processus supportant le programme (descriptif de processus, registre de gestion de la mémoire, registre d'état, etc...).

Il convient de rappeler que la plupart des implantations des langages évolués utilise une pile pour stocker les variables locales, les paramètres, et pour sauvegarder les contextes d'exécutions lors de l'appel de sous-programme. Car la pile propose une structure propice à la gestion dynamique inhérente aux appels imbriqués de sous-programme. Enfin les machines actuelles offrent des instructions spécifiques pour gérer efficacement l'empilement lors de l'appel de sous-programme, et le dépilement lors du retour.

Les moments où les appels et les retours de sous-programme s'effectuent, sont déterminés durant l'exécution. Donc pour pouvoir bénéficier de la notion de sous-programme à l'intérieur des coroutines, il est nécessaire que chacune des coroutines possède en propre une pile.

Plutôt que d'avoir d'un côté la pile contenant les contextes des sous-programmes mémorisant la dynamique de leurs appels, et de l'autre le descriptif de coroutine, il nous est possible de les regrouper à l'intérieur de la pile. Il suffit pour que le descriptif se trouve en haut de pile au moment de la commutation et de le définir comme une variable locale de la fonction de commutation. On s'aperçoit que **la totalité du contexte d'une coroutine est mémorisée dans sa pile.**

Une coroutine sera donc référencée par l'adresse de sa pile, plus exactement, par l'adresse du bas de pile fournie au moment de sa réservation. Il faut que le descriptif de la coroutine, quand elle est suspendue, soit référencé par un pointeur. Car lors de la commutation, on recharge les registres du processeur avec les valeurs sauvegardées dans le descriptif. Or la situation du haut de pile étant éminemment variable, il faut la localiser avec précision. Ce qui est fait en plaçant en bas de pile (qui est à une adresse fixe!) une variable contenant lors de chaque suspension l'adresse du haut de pile. La Figure 2 illustre ces propos. On trouve, à gauche la structure habituelle d'une pile d'un langage évolué, à droite celle modifiée d'une pile associée à une coroutine lorsqu'elle est suspendue.

L'implantation en C définit les types suivants :

- CO_CONTEXT est le type associé au contexte de coroutine.
  Les fonctions retournent ou ont des paramètres de ce type :
  CO_CONTEXT * co_alloc();
  CO_CONTEXT * co_init();
  resume(cor)
  CO_CONTEXT * cor;
- CO_DESCR est le type associé au descriptif de coroutine.
  La variable du bas de pile pointant sur le descriptif se déclare donc :
  CO_DESCR * pdescr;

3.3 La Commutation de coroutines (resume())

Lors de la commutation, l'échange de contextes s'effectue en échangeant les piles (c'est à dire l'espace de donnée associé aux coroutines), la valeur des registres généraux, et le code. En fait, les espaces relatifs à la pile et à l'instruction à exécuter sont eux-mêmes référencés par deux registres : le

registre pointeur de pile et le registre compteur ordinal. Ce qui revient à dire que commuter deux coroutines, c'est échanger la valeur de l'ensemble des registres entre la coroutine appelante et appelée.

Sauvegarder et restaurer la valeur de la plupart des registres est très simple. Cependant, tous les informaticiens (notamment ceux familiers avec le langage machine) savent que la manipulation du compteur ordinal est dangereuse. Car il progresse sans arrêt avec l'exécution du code, en indiquant la prochaine instruction à exécuter. Ainsi, lorsque l'on exécute l'instruction de sauvegarde du compteur ordinal la valeur sauvegardée pointe sur l'instruction suivante. De ce fait, cette instruction a toutes les chances d'être exécutée deux fois. La première fois dans la suite normale à l'instruction de sauvegarde, la deuxième fois lors de la restauration après une reprise de la coroutine.

Cette réexécution peut être évitée à l'aide d'une des trois techniques suivantes :
- On saute les instructions suivantes en augmentant la valeur du compteur ordinal avant sa sauvegarde. La réexécution redémarrera quelques instructions plus loin.
- La valeur d'une variable permet de distinguer entre la première exécution et la deuxième. Une mise en oeuvre de cette technique repose par exemple sur la présence d'un instruction d'incrémentation en tant qu'instruction suivante. Le test de la variable permettra d'orienter l'exécution à volonté.
- La dernière solution utilise une instruction spécialisée qui effectue l'échange en une seule instruction (sauvegarde et restauration). Il n'y a dons plus de problème de réexécution. La pile est implicitement utilisée pour stocker la sauvegarde. Cette instruction spécialisée existe dans tous les langages machines, et est celle utilisée lors des appels de sous-programme.

Malheureusement, aucune des solutions proposées n'est réalisable directement à l'aide d'un langage évolué. L'alternative consiste donc :
- soit à utiliser l'assembleur pour réaliser les actions voulues. L'implantation du Vax 11/780 sous "Unix 4.3 BSD" choisit cette alternative en utilisant la troisième technique [5](instruction spécialisée CALLS).
- soit à utiliser les fonctions prédéfinies de la bibliothèque standard quand elles permettent de manipuler les registres. Ainsi l'implantation sur le Bull SPS7 sous "Spix" utilise les fonctions "setjmp()" et "longjmp()"[6]. En fait, ces fonctions masquent un code utilisant l'assembleur. Cette implantation utilise la seconde technique : la fonction "setjmp()" retourne une valeur modifiable entre les deux exécutions. A la première exécution la valeur retournée est nulle, à la seconde la valeur est celle du dernier paramètre de la fonction "longjmp()".

Dans le cas de l'implantation sur SPS7 le code de la fonction de commutation devient :

```
resume(descr_appelée)
CO_DESCR * descr_appelée;                 /* Descriptif de la coroutine appelée */
{
CO_DESCR * descr_appelante;               /* Descriptif de la coroutine appelante */
int setjmp();                             /* La fonction setjmp retourne 0 ou la valeur du */
                                          /* dernier paramètre de la fonction longjmp */
    if (setjmp(descr_appelante) == 0 ) {  /* Sauvegarde de l'appelante */
        longjmp(descr_appelée, 1);        /* Restauration de l'appelée, le paramètre doit être ≠ 0 */
    } else {
    ... ;                                 /* Code exécuté lors du deuxième retour !!! */
    }
}
```

On peut remarquer que toutes les commutations se passent dans la même fonction (celle qui commute, ici, "resume()"), sur la même instruction (celle qui modifie le compteur ordinal). Ainsi toutes les coroutines se suspendent sur cette instruction, et reprennent sur ce même point. Ces similitudes facilitent grandement la commutation, on sait d'avance d'où on part et où on arrive. Lors de leur reprise, le comportement des différentes coroutines diverge en fonction de l'empilement des contextes de sous-programmes (représentant la dynamique d'exécution des différentes coroutines), et de la valeur de leurs variables locales.

3.4 La Préparation des coroutines (co_init())

Nous venons de résoudre les problèmes soulevés lors de la commutation de coroutine. Mais que se passe-t-il lorsqu'une coroutine vient d'être créée ? Elle ne possède pas encore de contexte. Elle n'existe pas, on ne peut donc pas la reprendre!

Il faut préparer le contexte nécessaire à son démarrage (sa première reprise), qui doit être compatible avec la fonction de commutation. Il faut que le premier "resume()" trouve une organisation interne de la pile compatible avec celle habituellement attendue par cette fonction. Cette organisation est déterminée par les contraintes d'implantation de la fonction de commutation telles qu'elles ont été définies au chapitre précédent.

(1) Il faut trouver en haut de pile, le descriptif de la coroutine. Ce descriptif est utilisé pour restaurer les registres lors de la reprise. La structure de ce descriptif est déterminée par la technique employée pour effectuer la commutation de coroutine.

(2) Il faut créer un contexte de sous-programme pour la première fonction de la coroutine (fonction d'appel), avec ses variables locales, ses paramètres, et son descriptif de sous-programme. Ce descriptif est censé correspondre au sous-programme appelant qui aurait dû appeler la coroutine créée. En fait, ce sous-programme appelant n'existe pas vraiment, il n'est utile que pour exécuter la coroutine créée dans un environnement local et compatible. La structure du contexte de sous-programme est imposée par celle employée par le compilateur du langage cible.

(3) Le bas de pile est complété par un pointeur référençant le descripteur de coroutine (c'est à dire pointant sur le haut de pile). Dans cette position, le pointeur peut être considéré comme une variable locale du sous-programme appelant (censé avoir appelé la coroutine).

C'est ainsi que la fonction "co_init()" prépare du contexte d'une coroutine nouvellement créée. Les deux premières structures dépendent évidemment des particularités du processeur (de son jeu de registres), et de celles du compilateur (de sa structure d'appel de sous-programme).

Par souci d'homogénéité, il convient de reprendre la coroutine créée non pas à la première instruction de sa première fonction (sa fonction d'appel), mais comme toutes les autres coroutines lors de leur reprise, en fin de la fonction de commutation. La deuxième étape (2) de la préparation est modifiée comme suit :

(2a) Création d'un contexte pour le sous-programme de commutation "resume()".
(2b) Création d'un contexte pour le sous-programme d'appel.

3.5 La Terminaison d'une coroutine

Les coroutines se passent entre elles explicitement le contrôle du processeur par l'intermédiaire de la fonction de commutation "resume()". Cependant que se passe-t-il lors de l'exécution de la dernière instruction associée à une coroutine?

On aboutit à un état incohérent, car on retourne vers un sous-programme qui n'existe pas. Il convient plus tôt de rendre la main à une autre coroutine. Cette action et le nom de cette coroutine doivent être connus par défaut. Une première possibilité est de reprendre la coroutine ayant créée la coroutine venant de se terminer. Malheureusement, on ne peut pas connaître avec exactitude l'histoire de cette coroutine créatrice. Elle peut aussi bien être elle-même terminée. Il faut trouver une coroutine dont on est sûr de l'existence.

En fait, il existe bien une coroutine un peu spécifique. C'est la première coroutine du programme, qui est créée implicitement lors du lancement du programme. Cette coroutine initiale est la mère (directe ou indirecte) de toutes les coroutines qui forment l'application pseudo-parallèle. Nous décidons que la durée de vie du programme lui soit liée. C'est à dire que le programme débute par cette coroutine initiale, et finit lorsque cette coroutine se termine.

Lors de la terminaison d'une coroutine, on retournera donc le contrôle à cette coroutine initiale, dont on sait qu'elle est présente par définition. Ceci est réalisé en déclarant une variable globale "co_principale" référençant en permanence le contexte de la coroutine initiale, et en insérant à la fin du code de chaque coroutine un appel de fonction : "resume(co_principale)". Ce terme de principale s'explique par le fait que cette coroutine a pour fonction d'appel la fonction "main()" du programme.

En fait pour faciliter la mise en oeuvre, nous créons une fonction spéciale permettant d'effectuer et de contrôler de manière élégante à la fois le démarrage et la terminaison d'une coroutine. Cette fonction spéciale comporte, en tant que première instruction, la fonction d'appel associée à la coroutine créée, et en tant que deuxième instruction, une reprise de la coroutine initiale. La reprise de la coroutine initiale se fait ainsi naturellement à la fin de l'exécution de la fonction d'appel.

```
CONTEXT * co_principale;                           /* la coroutine principale */

void co_control (fonction_d_appel, argument)  /* fonction de contrôle des coroutines */
int (* fonction_d_appel) ();
int * argument;
{
     fonction_d_appel (argument);              /* fonction d'appel associé à la coroutine créée */
     resume(co_principale);                    /* reprise de la coroutine initiale */
}
```

Maintenant, la première fonction exécutée d'une coroutine n'est plus sa fonction d'appel, mais pour toutes les coroutines, la fonction "co_control()". Ce qui revient néanmoins au même, cette dernière fonction appelant aussitôt la fonction d'appel. Le contexte de sous-programme créé lors de la deuxième étape de préparation d'une coroutine est donc celui de la fonction "co_control()". Ceci simplifie la réalisation, car cette fonction est unique, alors que chaque coroutine peut avoir une fonction d'appel différente.

3.6 Les paramètres

Nous venons de voir que chaque coroutine peut posséder une fonction d'appel différente. La fonction "co_control()" possède donc un paramètre qui précise le nom de la fonction d'appel. Cependant le nom de cette fonction d'appel provient du deuxième paramètre de la fonction de préparation "co_init()". Il convient de transmettre ce nom entre la fonction "co_init()" qui prépare, et la fonction "co_control()" qui lance effectivement la fonction d'appel.

Malheureusement, ces deux fonctions ne s'exécutent pas dans le même contexte, ou plus précisément, pas par rapport à la même pile. Et la pile sert pour le passage des paramètres lors de

l'appel d'un sous-programme. Il faut donc que la fonction "co_init()" recopie son deuxième paramètre en tant que premier paramètre dans le contexte associé à la fonction "co_control()" de la pile de la coroutine crée. Cette recopie nécessite de connaître avec précision la place du paramètre dans la nouvelle pile. Bien entendu, cette recopie est nécessaire aussi pour l'argument de la fonction d'appel.

Un problème et une solution similaire existent pour les paramètres "in" et "out" de la fonction de commutation "resume()". Le paramètre "in" de la coroutine appelante doit être transmis à la coroutine appelée en tant que paramètre "out". La fonction "resume()" recopie la valeur du deuxième paramètre de la coroutine appelante dans le troisième de la coroutine appelée.

Ce problème se retrouve une troisième fois, lors de la terminaison d'une coroutine. La coroutine initiale devrait récupérer la valeur de retour de la fonction d'appel de la coroutine qui se termine. Dans la fonction "co_control()", lors de la reprise de la coroutine principale la valeur de retour de la fonction d'appel est attribué au paramètre "in".

Cet instant transitoire, où l'on est entre deux piles, oblige à déclarer certaines des variables globales de l'environnement. Ces variables sont utilisées par les deux fonctions "co_init()" et resume()".

3.7 Les Améliorations

Nous avons défini au chapitre 3.3 traitant de la préparation, et précisé au chapitre 3.4 traitant de la terminaison, les actions nécessaires à la création d'une coroutine.

Nous en déduisons qu'il faut bien connaître la structure de la pile, celle du processeur, le fonctionnement de l'appel (et de retour) de sous-programme. L'ensemble de ces connaissances est difficile à acquérir. Nous allons tenter d'alléger la charge de travail du porteur de l'environnement, en constatant qu'on pourrait automatiquement créer un nouveau contexte de coroutine en jouant astucieusement avec le pointeur de pile.

Par exemple, si avant d'appeler un "resume()" on modifie le pointeur de pile, la sauvegarde du descriptif de coroutine est effectuée sur la nouvelle pile. De même, si avant l'appel d'un sous-programme on modifie le pointeur de plie, le contexte du nouveau sous-programme est empilé sur la nouvelle pile.

C'est ainsi que nous proposons de modifier la fonction "co_init()". Au lieu d'empiler "à la main" sur la nouvelle pile chaque élément pour recréer la structure compatible et nécessaire à la commutation, nous allons utiliser ce principe pour qu'automatiquement cette structure se recréée.

Après avoir sauvegardé dans des variables globales les variables locales nécessaires à l'initialisation, et sauvegardé un point de reprise pour l'appelante dans son descriptif ("setjmp()"), le code de la fonction co_init() se structure en trois étapes.
La première étape prépare le changement de pile. On y parvient en rechargeant le descriptif de coroutine dont le pointeur de pile a été modifié. L'exécution de la fonction "longjmp()" modifie la pile courante, retourne au point de reprise, et passe à l'étape suivante.
La deuxième étape se déroule sur la pile de la coroutine créée. On prend soin de rétablir le pointeur de pile vers la pile de l'appelante. Puis on appelle la fonction "co_control()", dont le contexte tout naturellement s'empile. La première instruction de cette fonction est un appel à la fonction "resume()". Le contexte de la fonction "resume()" s'empile tout aussi naturellement sur la pile de la coroutine créée. L'exécution de cette fonction provoque la reprise de la coroutine appelante, car le pointeur de pile a été rétabli. On retourne alors au point de reprise.
On se retrouve à la troisième étape sur la pile de la coroutine appelante à la fin de la fonction "co_init()".

Durant l'exécution de la fonction "co_init()", si la première et troisième étapes se déroulent sur la pile de l'appelante, la seconde étape se déroule sur la pile de l'appelée. Ainsi durant un bref instant, la coroutine créée a pris vie en exécutant un code issu de la coroutine créatrice. Elle s'est exécutée en mode attaché. Plus tard, lorsque le premier "resume" aura provoqué l'exécution de sa fonction

d'appel, elle sera exécutée de manière totalement indépendante, en mode détaché.

Notre environnement est peu restrictif. Certaines erreurs de fonctionnements peuvent donc apparaître. Nous allons proposer, ici, deux modifications à même de corriger ou de détecter deux de ces erreurs : le dépassement de pile d'une coroutine, et la reprise d'une coroutine terminée. Le code que nous présentons en annexe ne possède pas ces traitements, car ils nous ont semblé peu portables, ou encombrants. Néanmoins, nous mettons à la disposition des utilisateurs potentiels les renseignements suffisants pour mettre en oeuvre ces traitements.

Le dépassement de pile est provoqué par un trop grand nombre d'appels imbriqués de sous-programmes, avec trop de paramètres, ou trop de variables locales. Ce dépassement peut ne pas être détecté immédiatement, et ainsi provoquer une exécution erronée. Une solution à ce problème consiste à attribuer à chaque coroutine un segment . Les fonctions mises en oeuvre sont les primitives de la version V : shmctl(), shmget(), shmat(), shmdt(). Le segment contient le contexte de la coroutine (sa pile). Tout débordement entraîne une violation des limites du segment, ce qui est immédiatement détecté. La récupération du signal émis lors de cette détection permet d'incriminer la coroutine courante, et de remédier au problème.

L'appel d'une coroutine terminée est un risque probable, dès lors que l'on ne peut effectuer aucun contrôle lors de la compilation sur l'existence supposée d'une pile. Une solution simple mais encombrante consiste à conserver le contexte de coroutine après sa terminaison, et à placer dans la fonction "co_control()" un message d'erreur après la reprise de la coroutine principale "resume(co_principale)". Ainsi toute reprise ultérieure aura pour effet de provoquer l'exécution de ce traitement d'erreur.

3.8. Le Portage

Le portage de notre environnement , initialement conçu sur Bull SPS7 sous Spix (proche de la Version V d'AT&T) a été effectué sur Vax 11/780, puis sur Sun 3/50 sous 4.3 BSD.

Il s'est avéré qu'il est impossible d'utiliser les fonctions de la bibliothèque standard "setjmp()" et "longjmp()" du Vax. Il est interdit de modifier le contenu de la structure "jmp_buf" contenant le jeu de registres, car la fonction "longjmp()" effectue un contrôle lors du rechargement. Toute modification aboutit inévitablement à la fonction "longjmperror()" et au message : 'longjmp botch'.

Nous avons en fait réalisé deux implantations sur Vax. Pour la première implantation, nous avons choisi de réécrire en assembleur une fonction" longjmp()" à notre goût. C'est à dire rechargeant les registres sans contrôle. En fait, nous l'avons décalquée de la précédente version en enlevant les tests gênants.

Pour la deuxième implantation, nous avons décidé de réécrire la totalité de la fonction "resume" en assembleur et de ne pas utiliser les fonctions "setjmp()" et longjmp()". Nous avons choisi d'utiliser alors la deuxième technique décrite au paragraphe 4.2 pour effectuer la commutation de coroutine. Cette technique utilise l'instruction spécialisée d'appel de sous-programme "Calls" pour effectuer en une seule instruction la sauvegarde et le restauration du jeu de registres. Cette deuxième implantation a pour avantage d'être légèrement plus rapide que la précédente.

Le portage sur Sun n'a consisté qu'en une modification de la valeur de la constante référençant le pointeur de pile dans la structure de sauvegarde des registres. Il est passé de 12 à 14. L'intégralité du code a été conservé.

L'ensemble de ces portages semble confirmer la justesse de nos affirmations. Il est possible dans toute machine Unix d'avoir un environnement identique favorisant la programmation du pseudo-parallèlisme. Le portage de cet environnement nécessite généralement que quelques judicieuses adaptations. De nouveaux portages sur Gould et sur MacIntosh sont d'ors et déjà envisagés.

3.9. L'Evaluation des performances

Nos mesures de performance ont montré que le temps moyen d'exécution d'un "resume" est de 0.05 milisecondes. Alors que celui des mécanismes équivalents sous le système Unix étaient de 0.45 milisecondes pour le "write/read" d'un "pipe". De même le temps moyen d'exécution d'un initialisation d'une coroutine à l'aide de la fonction "co_init()" se monte à 0.08 milisecondes, alors que ce temps est de 0.42 milisecondes pour la création d'un nouveau processus avec la fonction "fork()", et de 25.6 milisecondes pour un changement de code avec la fonction "execl()".

Nous avons étudié les temps d'exécution d'une application de transfert de caractères utilisant, soit deux processus et un "pipe", soit deux coroutines. Le temps système est nettement prépondérant dans l'option avec deux processus. Les primitives de lecture et d'écriture du "pipe" ("read" et "write") s'exécutent en mode système. Alors que c'est l'inverse avec les deux coroutines, le temps d'exécution en mode système est pratiquement nul. Cependant le cumul de temps d'exécution ("CPU time") donne toujours un net avantage en faveur des coroutines : un gain d'environ 900%.

Néanmoins, nous avons constaté la possibilité d'un effet pervers de cette inversion d'attribution du temps, entre le mode système et le mode usager. Un processus ne consommant que du temps système risque d'obtenir une plus forte priorité qu'un processus consommant la même somme de temps mais en mode usager. Car la priorité d'exécution d'un processus peut n'être calculé par le "scheduler" qu'à partir du temps de consommation du CPU en mode usager.

Cependant les coroutines utilisant moins le processeur, si le bénéfice n'en revient pas directement à son utilisateur (peu d'amélioration du temps de réponse, du fait de l'accroissement de la priorité), l'ensemble des utilisateurs y retrouvent leur compte.

Nos mesures mettent en évidence les avantages des coroutines :

- Gestion interne (donc optimale) de l'enchaînement des coroutines, ce qui minimise le temps d'exécution. Alors que les processus sont contrôlés par le "scheduler" du noyau.
- Les coroutines partagent un espace commun (les variables globales), ce qui minimise le temps et l'espace nécessaire à la transmission des informations. Cette transmission pénalise les processus sous Unix.

4. Conclusion

Notre étude, par une présentation détaillée de l'ensemble des problèmes rencontrés, et une discussion des différentes solutions possibles, veut éveiller le lecteur aux problèmes du système, du pseudo-parallélisme, et permet de rappeler des caractéristiques propres à la gestion des langages évolués. Les mécanismes relatifs aux sous-programmes sont tout particulièrement abordés :

- mémorisation et accès aux paramètres,
- mémorisation et accès aux variables locales,
- conflit lors de la manipulation du compteur ordinal,
- sauvegarde et restauration du descriptif de sous-programme.

La connaissance de ces mécanismes nous est indispensable pour comprendre l'implantation proposée, et peut permettre au lecteur de mieux utiliser les langages de programmation.

Une évaluation des performances nous a permis de mettre en évidence le coût important de la création et de la gestion des processus au regard de celui des coroutines, et de rappeler l'intérêt du pseudo-parallélisme vis-à-vis du vrai parallélisme, si le temps de communication est comparable au temps de calcul demandé.

Cet environnement nous a permis de développer un important ensemble d'applications :

- Un simulateur de système d'exploitation (utilisé à des fins pédagogiques en illustration d'un cours de maîtrise d'informatique (plusieurs milliers de lignes),
- Un éditeur utilisant le multi-fenêtrage (à chaque fenêtre est associée une coroutine, qui possède ainsi son contexte propre),

- L'ébauche d'un compilateur (à chaque phase du compilateur est associée une coroutine),
- Un traducteur de Modula-2 en C (les coroutines sont utilisées pour implanter le module Process associé à l'exécution pseudo-parallèle en Modula-2).

Il est **performant**, car la création et la commutation entre coroutines ne demande l'exécution que de quelques instructions (évaluation du coût : environ 25 micro-secondes).

Il est **minimal**, il ne comporte que 3 fonctions de quelques dizaines de lignes.

Il est **élémentaire**, car il permet de réaliser parfaitement de nombreux mécanismes de synchronisation ou de communication (cf les nombreux exemples réalisés : producteur/consommateur, moniteur, sémaphore, simulation de pipe, etc...), et est capable de s'adapter à de nombreuses politiques de gestion ou d'enchaînement des coroutines.

Il est **portable**, nous avons déjà implanté notre environnement sur différentes machines (Bull SPS7, Vax 11/780 et Sun 3/50). Cet ensemble est en voie d'accroissement sur d'autre machines Unix (Gould, Apple, etc..).

Simple, portable, peu coûteux, adaptable, notre environnement respecte la philosophie "Unixienne", et il pousse malheureusement la ressemblance au langage C jusqu'à permettre tous les abus d'utilisations!

Bibliographie

[1] O.J.Dahl, B.Myhrhaug, K.Nygaard. *Some features of the Simula 67 language.* 2nd conference on the applications of simulation. New-York -1968.

[2] N.Wirth. *Programming in Modula-2.* Springer-verlag, 1982.

[3] A.Goldberg, D.Robson. *Smalltalk-80, language and its implementation.* Addison Wesley, 1983.

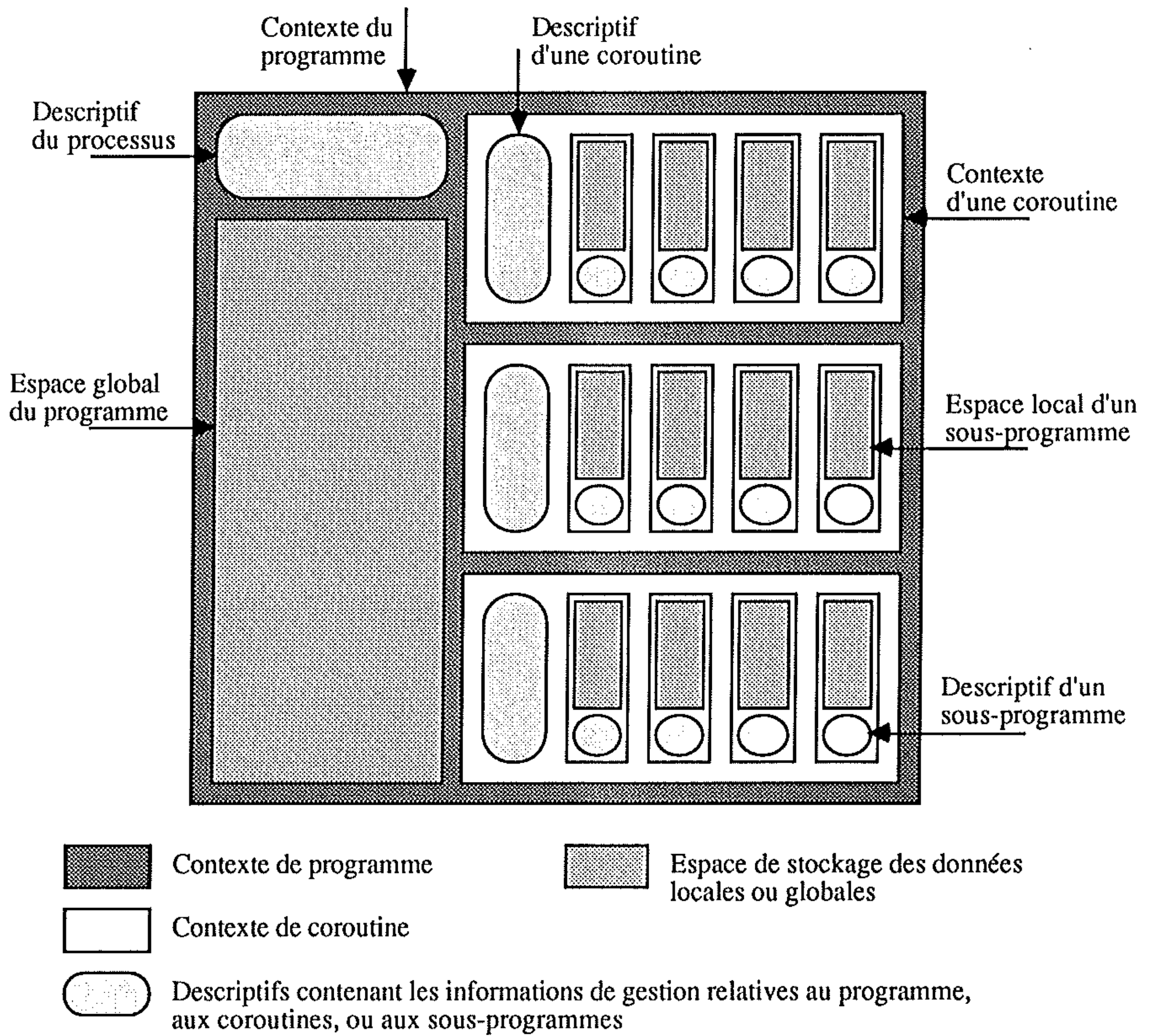
[4] B.Stroustrup. *The C++ programming langage.* Addison Wesley, 1987.

[5] *Vax 11, architecture reference manual.* Digital Equipment corporation, May 1982.

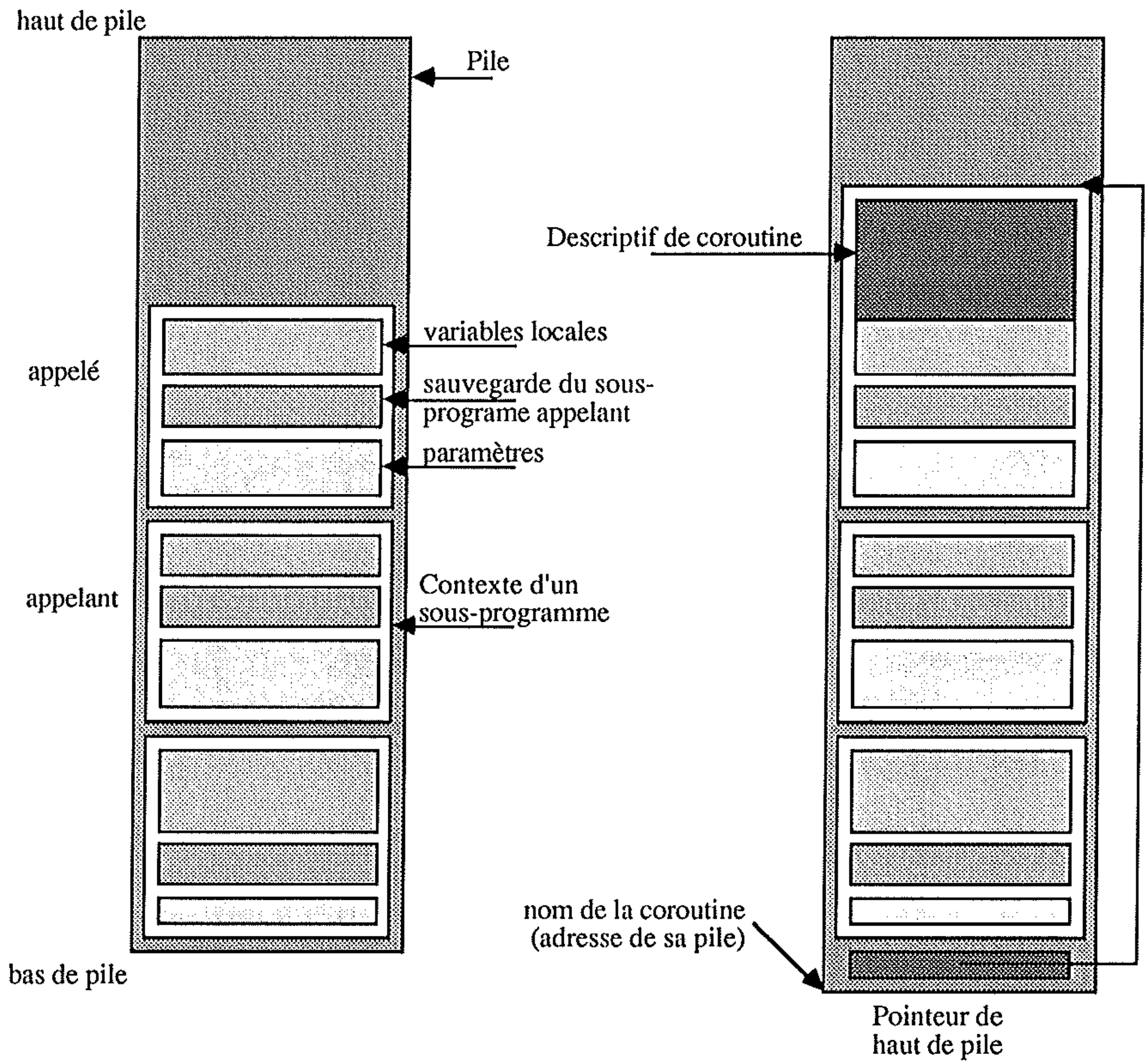
[6] *Spix system, programmer reference manual.* Bull, November 1986.

[7] J.Berthelin, B.Cousin. *Manuel d'utilisation de l'environnement pour le pseudo-parallèlisme (version 1.0).* Paris, Octobre 1988.

- Figure 1 - Contextes, espaces et descriptifs -



- Figure 2 - La structure d'une pile de coroutine -



```
/*************************************************************************************
 * (c)  J.Berthelin & B.Cousin  Sun Oct 23 15:23:25 MET 1988                          *
 *          Environnment des coroutines                                               *
 *___________________________________________________________________________________ *
 *                         CONSEILS POUR LE PORTAGE                                   *
 *      La mise en oeuvre des coroutines est dependante de la machine . Il faut donc :*
 *      1) Analyser le code genere par l'utilisation des fonctions                    *
 *              setjmp et longjmp pour mettre en evidence dans la                     *
 *              structure retournee, le champ contenant le pointeur de pile.          *
 *      2) Si votre machine a une fonction 'longjmperror' :                           *
 *              reecrire la fonction longjmp en supprimant le code qui maintient l'integrite de la pile. *
 *      3) Changer la definition de SP dans le fichier "cor.h" .                      *
 *************************************************************************************/
#include <local/cor.h>

CO_CONTEXT cor_principale;                                  /* Les variables globales utilisees */
static CO_CONTEXT *cor_appelante, *cor_courante = &cor_principale;
static int (*cor_fonction)(), * cor_parametre, save_SP;

/*______CO_ALLOC (Lenght);_____________________________________________
 * DESCRIPTION      Alloue un contexte de coroutine.
 *
 * ENTREE           Int lenght : taille de la pile demandee en octets.
 * RESULTAT         Pointeur de contexte de coroutine
 *                  ou NULL si  le contexte n'a pas pu etre alloue.
 *
 * REMARQUE Une verification de la taille de la pile demandee est effectuee :
 *                  Si la taille est negative ou nulle ou trop petite (lenght <= MINSTACK),
 *                  une pile minimum correcte est allouee (lenght = MINSTACK).
 *____________________________________________________________________________*/
CO_CONTEXT * co_alloc(lenght)
int lenght;
{
        CO_CONTEXT * pile ;

        lenght = lenght & ~(TAILLE_ELEM - 1);                       /* taille de l'allocation en nombre entier */
        if (lenght < MINSTACK) lenght = MINSTACK;                   /* verficatiion de taille de pile */
        if ((pile = (CO_CONTEXT *) malloc(lenght + sizeof(CO_DESCR))) < 0)    /* allocation du contexte */
                return((CO_CONTEXT *)NULL);
#ifdef DEBUG
        printf("%x = co_alloc(%x)\n", ((char *) pile) + lenght, lenght);
#endif
        return((CO_CONTEXT *)(((char *)pile) + lenght));
}

/*______CO_INIT(Cor_initialisee,Fonction,Parametre)____________________________
 * DESCRIPTION      Initialise le contexte d'une coroutine.
 *
 * ENTREE           CO_CONTEXT * cor_initialisee    : pointeur de contexte de la coroutine a initialiser.
 *                  int (*fonction)()               : adresse de la fonction d'appel de la coroutine.
 *                  int * parametre                 : pointeur sur un parametre initial de la coroutine.
 * RESULTAT         Pointeur de contexte de la coroutine initialisee.
 *                  ou NULL si l'intialisation de la coroutine n'a pu etre effectuee.
 *
 *                              |   Descriptif            |<-+
 *                              |-------------------------|  |
 *                              |       Pile              |  |
 *                              |-------------------------|  |
 *          (CO_CONTEXT *)  --> | pointeur sur le descriptif |---+
 *                              |-------------------------|
 *
 * REMARQUE Si parametre est NULL, on considere que la coroutine n'a pas de parametre.
 *                  On initialise le pointeur de descriptif.
______________________________________________________________________________*/
```

```
CO_CONTEXT * co_init(cor_initialisee,fonction,parametre)
CO_CONTEXT * cor_initialisee;
int (*fonction)();
int * parametre;
{
        CO_DESCR descriptif;
        extern void co_control();                          /* fonction du demarrage des coroutines */

        cor_fonction   = fonction;                         /* sauvegarde sous forme de variables globales */
        cor_parametre  = parametre;                        /* des variables utiles a l'execution du boot */
        cor_appelante  = cor_courante;
        cor_courante   = cor_initialisee;

        switch( setjmp(cor_appelante->pdescr = &descriptif) ) {  /* initialise le point de reprise pour les étapes suivantes*/
          case DEBUT_INIT : {                              /* Debut de l'initialisation (sur la pile de l'appelant) */
                save_SP = cor_appelante->pdescr->reg[SP];  /* sauvegarde le ptr de pile de l'appelante */
                cor_appelante->pdescr->reg[SP] = (int)(cor_initialisee - 2);/* chargement du ptr de pile de l'appelee */
                longjmp(cor_appelante->pdescr,SUITE_INIT); /* passage a l'etape suivante et changement de pile */
          }
          case SUITE_INIT : {                              /* Suite de l'initialisation (sur la pile de l'appelee)*/
                cor_appelante->pdescr->reg[SP] = save_SP;  /* chargement du ptr de pile de l'appelante */
                co_control(cor_fonction,cor_parametre);    /*  execute le demarrage de la coroutine */
          }
          case FIN_INIT : break ;                          /* Fin d'initialisation (retour sur la pile appelante) */
          default : return((CO_CONTEXT *)NULL);            /* Etape illegale */
        }
#ifdef DEBUG
        printf("%x = co_init(%x,%x,%x)\n",cor_initialisee,cor_initialisee,fonction,parametre);
#endif
        return(cor_initialisee);
}


/*________CO_CONTROL(Fonction,Parametre)______(Fonction Locale)________________
* DESCRIPTION      Initialise la pile de la coroutine en effectuant un debut d'execution.
*                  Initialise le descripteur de la coroutine en fixant un point de reprise.
*                  Execute la coroutine (*), puis reprend la coroutine principale.
*
* ENTREE           Int (*fonction)(): adresse de la fonction d'appel.
*                  Int * parametre     : pointeur sur un parametre initial de la coroutine.
* RESULTAT         Sans
*
* REMARQUE (*) L'execution de la coroutine ne sera faite que lors du prochain 'resume'
*_____________________________________________________________________________*/
static void co_control(fonction,parametre)                 /* initialise la pile */
int * (* fonction)();
int * parametre;
{
        int valeur_retournee;

#ifdef DEBUG
        printf("coroutine(fonction,%x)\n",parametre);
#endif
        resume(cor_appelante,FIN_INIT,&valeur_retournee);  /* initialise le decriptif de l'appelee*/
        if (parametre == (int *)NULL){                     /* passe un parametre si besoin */
                resume(&cor_principale, (*(fonction))(), &valeur_retournee); /* et execute la fonction d'appel */
        } else{                                            /* puis reprend la coroutine principale */
                resume(&cor_principale, (*(fonction))(parametre), &valeur_retournee); /* lors du retour de la fonction */
        }
}
```

```
/*______RESUME(Cor_appelee, In, Out)_________________________________________
 * DESCRIPTION        Reprend l'execution d'une coroutine en lui passant un parametre si besoin.
 *
 * ENTREE             CO_CONTEXT * cor_appelee : pointeur de contexte de la coroutine appelee.
 *                    int * in                 : pointeur sur le parametre passe a la coroutine appelee.
 *                    int * out                : pointeur sur la variable qui contiendra le parametre passe a la coroutine appelee.
 * RESULTAT           Sans, mais la variable 'out' permet de recuperer le parametre `in` entre coroutine.
 *
 * REMARQUE La pointeur 'out' permet de recuperer un parametre 'in' passe par la coroutine appelante.
______________________________________________________________________________*/
resume(cor_appelee, in, out)
CO_CONTEXT * cor_appelee;
int * in,* out;
{
	CO_DESCR descriptif;
#ifdef DEBUG
	printf("resume(%x, %x, %x)\n", (int)cor_appelee, (int)in, (int)out);
#endif

	cor_appelante = cor_courante;                          /* m.a.j la nouvelle coroutine courante */
	cor_courante = cor_appelee;
	descriptif.etat=ETAT_ZERO;                             /* initialise le nombre de passage le 'if' suivant : ZERO*/
	*out=(int)setjmp( cor_appelante->pdescr = &descriptif ); /* fixe un point de reprise pour l'appelante */
	if ( descriptif.etat == ETAT_ZERO)   {                 /* si premier passage ?           */
		descriptif.etat=ETAT_UN;                           /*  initialise le nombre de passage le 'if' : UN*/
		longjmp(cor_appelee->pdescr,in);                   /*  saute au point de reprise de la coroutine appelee */
	}
}
```

```
/*********************************************************************************************/
/**************************************** fichier cor.h **************************************/
/*********************************************************************************************/
#include <stdio.h>
#include <setjmp.h>

#undef DEBUG            1

#define TAILLE_ELEM     (sizeof(char *))           /* taille d'un element de la pile d'une coroutine */
#define MINSTACK        (int)(1000*TAILLE_ELEM)    /* taille minimale de la pile d'une coroutine */
#define DEBUT_INIT      0                          /* etape de la fonction 'co_init' */
#define SUITE_INIT      1                          /* etape de la fonction 'co_init' */
#define FIN_INIT        2                          /* etape de la fonction 'co_init' */
#define SP              12                         /* numero du registre pointeur de pile dans le descriptif */
#define ETAT_ZERO       0                          /* etat d'une coroutine : bloquée*/  suspendue
#define ETAT_UN         1                          /* etat d'une coroutine : libre*/    [illegible]

#define detacher(in, out) resume(&cor_principale,in,out)

typedef struct {
	jmp_buf     reg;
	int         etat;
} CO_DESCR;                                        /* definition du descritif de coroutine */
typedef struct {
	CO_DESCR * pdescr;
} CO_CONTEXT;                                      /* definition du contexte de coroutine */

CO_CONTEXT *    co_alloc();                        /* fonction d'allocation des coroutines */
CO_CONTEXT *    co_init();                         /* fonction d'initialisation des coroutines */
                resume();                          /* fonction de commutation des coroutines */

extern CO_CONTEXT cor_principale;                  /* nom de la coroutine principale */
```