

# OVIPAR : Une Plate-forme d'Initiation au Parallélisme sous UNIX

**Antonio ALMEIDA  
Pierre MOUKELI**

**Institut Africain d'Informatique  
(IAI) B.P. 2263 Libreville  
GABON**

**Tel. (241) 72 00 05**

**Fax. (241) 72 00 11**

**Résumé.** OVIPAR (Ordinateur Virtuel Parallèle) est un outil d'aide à l'apprentissage du parallélisme développé sous UNIX en langage C. Il s'adresse particulièrement aux étudiants et aux chercheurs désireux de découvrir ce domaine de l'informatique. OVIPAR se présente comme une bibliothèque de fonctions à intégrer dans un programme C classique. Ces fonctions mettent en oeuvre les instructions CSP. OVIPAR est en quelque sorte un modèle réduit de PVM, avec des ambitions très modestes.

**Mots-clés.** Outils de programmation, parallélisme.

**Summary.** Ovipar is a training platform for learning parallelism. It is based on Unix and C. It is well suited for students and researchers interested in this domain of computer science. Ovipar is designed as a library of functions to be inserted in usual C programs. These functions borrow the CSP model. Ovipar is a kind of reduced PVM, with modest ambitions.

**Key words.** Programming tools, parallelism.

## **Introduction.**

OVIPAR (Ordinateur Virtuel Parallèle) est une plate-forme de programmation destinée à l'initiation au parallélisme [Alm2] [Mou]. Cette plate-forme est réalisée en langage C, sous le système d'exploitation UNIX. OVIPAR a été mis en oeuvre suivant le modèle MIMD à mémoire distribuée. L'architecture est basée sur des processeurs à quatre liens inspirés des transputers de INMOS [INM].

OVIPAR offre à l'utilisateur la possibilité de créer un ordinateur virtuel multiprocesseur sur lequel il peut exécuter des programmes parallèles écrits en langage C suivant le modèle CSP. Les processus communiquent à travers des canaux

internes à chaque noeud ou reliant des noeuds entre eux. Les processus d'un même noeud disposent d'une zone de mémoire partagée.

OVIPAR a des ambitions plus modestes que PVM [Che] [Gei] et MPI [Wal]. D'abord il ne fonctionne que sur une seule machine UNIX. Il n'offre qu'une bibliothèque de fonctions mettant en oeuvre les principales instructions CSP, alors que PVM est un véritable environnement de programmation. L'installation et l'exploitation d'OVIPAR ne nécessitent pas de formation préalable. Il est conçu pour s'intégrer facilement aux environnements UNIX. L'apprentissage d'OVIPAR se fait en quelques minutes sur la base de son manuel d'utilisation [Alm1], pour quelqu'un qui connaît CSP. L'expérience d'utilisation à l'IAI depuis deux ans par les élèves ingénieurs montre qu'OVIPAR est un outil pédagogique simple à utiliser. Enfin, il est gratuit.

Le reste de ce document décrit dans la première partie, l'architecture d'OVIPAR. La deuxième partie donne quelques indications sur sa mise en oeuvre. La troisième et la quatrième parties présentent la bibliothèque des fonctions. Une annexe donne quelques exemples de programmes OVIPAR.

## 1. ARCHITECTURE D'OVIPAR.

### 1.1. MOTIVATION.

OVIPAR se définit comme un outil d'aide à l'apprentissage du parallélisme. Il s'adresse particulièrement aux étudiants et aux chercheurs désireux de découvrir ce domaine de l'informatique. L'Institut Africain d'Informatique, comme bien d'autres structures universitaires africaines ne disposait pas d'un ordinateur parallèle, ni d'un environnement de simulation. L'IAI ayant une vocation de formation professionnelle, l'enseignement du parallélisme pour être crédible ne pouvait pas être durablement envisagée dans un cadre strictement théorique. De plus nos futurs ingénieurs doivent commencer à se familiariser avec les ordinateurs parallèles, leur quotidien de demain. Il était donc indispensable de se doter d'une plate-forme de programmation parallèle. L'existence à l'IAI d'une technicité conséquente a donc permis de concevoir et réaliser OVIPAR.

### 1.2. MACHINE OVIPAR.

Une machine OVIPAR est conçue comme étant un ensemble de processeurs reliés entre eux par un réseau de communication. Chaque processeur dispose de quatre liens bidirectionnels lui permettant de communiquer avec autant de voisins. Chaque noeud offre une mémoire d'un kilo octet accessible à tous les processus qui lui sont associés.

La description de la topologie se fait lors de la création de la machine en spécifiant des connexions point à point entre les liens des différents noeuds. Chaque noeud peut avoir jusqu'à quatre voisins compte tenu du nombre de liens. Les liens sont nommés N (nord), S (sud), E (est) et W (ouest). Une connexion entre deux noeuds se fait en spécifiant les liens impliqués. Deux noeuds peuvent avoir plus d'une interconnexion. OVIPAR ne permet pas de spécifier des noeuds isolés.

Le nombre de noeuds d'OVIPAR et le nombre de processus par noeud ne sont limités que par les capacités de la machine UNIX utilisée (e.g. nombre de processus qu'un utilisateur peut créer).

### 1.3. PROCESSUS OVIPAR.

Un processus OVIPAR est un processus UNIX disposant d'une fenêtre de communication avec d'autres processus. Cette fenêtre est constituée de canaux de communication. Elle peut être constituée de liens, ou de canaux internes au noeud. La fenêtre d'un processus est toujours créée par le processus père avec des canaux ou des liens que ce dernier a hérité ou de canaux créés par le père lui-même pour ses fils.

Un processus peut créer autant de canaux qu'il veut. Ces canaux sont internes au noeud et ne peuvent être utilisés que par les fils du processus et ses descendants qui en ont hérités. A tout moment un canal ne peut être partagé que par deux processus. Comme le père se bloque en attendant la fin de ses fils, le père peut transmettre par héritage ses droits sur un canal à un fils à travers la fenêtre de celui-ci. Le père recouvre ses droits sur les canaux à la terminaison des fils.

Les communications sont synchrones et binaires. Si le nombre  $n_1$  d'octets envoyés par un processus est différent du nombre  $n_2$  qu'attend le destinataire, alors le nombre d'octets effectivement transférés sera égal au minimum de  $n_1$  et  $n_2$ . Les fonctions de communication retournent ce minimum.

Un processus peut créer plusieurs fils. Il spécifie pour chacun d'eux une fenêtre de communication. Le père se bloque après la création des fils, jusqu'à la terminaison du dernier fils. A tout moment un processus peut connaître l'identité de processeur sur lequel il s'exécute.

### 1.4. PROGRAMME OVIPAR.

Un programme OVIPAR est un programme C faisant référence aux fonctions de la bibliothèque OVIPAR. Ce programme doit comporter un appel à la fonction *ovipar* qui constitue le point d'entrée du programme parallèle. Cette fonction spécifie la topologie de la machine et doit comporter dans ces paramètres un pointeur sur une fonction de chargement de la machine. La fonction de chargement a pour rôle de spécifier la composante qui sera exécutée sur chaque noeud. Elle doit se terminer par un appel à la fonction *placedpar* qui réalise le placement des composantes sur les noeuds.

Une composante est le premier processus OVIPAR qui s'exécute sur un noeud. Un processus OVIPAR est spécifié comme une fonction C ayant le même format que la fonction *main*. Aucune restriction n'est imposée aux facilités habituelles de programmation C et UNIX pour la conception des processus. Un processus peut faire référence à toutes les fonctions d'OVIPAR exceptés *ovipar* et *placedpar*; toute référence à ces deux fonctions échoue en dehors du contexte décrit plus haut.

Le programme parallèle doit être compilé en faisant référence à la librairie d'OVIPAR. Comme tout environnement basé sur l'appel de fonctions, OVIPAR n'effectue aucun contrôle syntaxique et sémantique, et se limite à ceux qu'offre le langage C. C'est pourquoi le programmeur doit se conformer à la syntaxe du manuel d'utilisation d'OVIPAR.

### **1.5. UTILISATION D'OVIPAR.**

Le programme OVIPAR doit être exécuté comme toute commande UNIX. OVIPAR démarre quand la fonction *ovipar* est exécutée et s'arrête automatiquement dès que tous les processus qu'il supporte sont terminés.

Mettant en oeuvre des techniques de programmation avancées, le programme OVIPAR ne doit pas être arrêté de manière inopinée. Dans sa version actuelle OVIPAR n'offre pas de possibilités de terminaison de programmes autres que celles de C et UNIX. Cela peut sembler être une limitation sévère, surtout dans le contexte d'apprentissage dans lequel il est censé être exploité. En fait dans un tel contexte, des applications lourdes qui peuvent exiger de tels soins ne sont pas conçues. OVIPAR n'est pas un outil de développement.

## **2. ELÉMENTS DE MISE EN OEUVRE D'OVIPAR.**

OVIPAR a été mis en oeuvre en s'appuyant sur les mécanismes de programmation avancée du système UNIX version 5 (i.e. gestion des processus, mémoire partagée, sémaphores), et sur les techniques offertes par le langage C (fonctions à arguments variables, vecteurs de paramètres du style *argv* de *main*). Les choix de programmation ont été faits pour tirer au mieux partie de ces mécanismes [Alm2].

### **2.1. LES PROCESSEURS.**

Un processeur est mis en oeuvre sous la forme d'un processus dont les descendants sont les processus du noeud. Ce processus initialise une ensemble de variables qui seront héritées par duplication de l'espace des données (e.g. numéro de processeur). L'arrêt de la machine correspond à la fin de tous ces processus.

### **2.2. GESTION DES PROCESSUS.**

Les processus OVIPAR sont créés avec la fonction UNIX *fork*. Le processus ainsi créé exécute comme première action une fonction chargée d'initialiser la structure de données du processus. Pour créer plusieurs processus, il suffit de procéder par itération de ce principe.

Le processus père attend la fin des processus fils avec une itération sur la fonction *wait*, sachant que cette fonction retourne une valeur négative quand le processus n'a plus de fils actif. Chaque processus fils doit donc se terminer par la fonction *exit*.

L'appel à cette fonction est pris en charge par OVIPAR. La gestion d'un processus suit alors le schéma suivant :

```
if (fork() == 0) {
    init_process ();
    call_process_function (argc, argv);
    exit (0);
}
```

### 2.3. GESTION DES CANAUX ET LIENS.

Les canaux et les liens sont construits avec des segments de mémoire partagée. Le bout de chaque canal est marqué avec le numéro du processus qui en hérite à travers sa fenêtre. Ce marquage est réalisé par la fonction *init\_process* appelée juste après la fonction *fork* (voir §3.2). A chaque canal est associé un ensemble de sémaphores qui permettent d'assurer l'exclusion mutuelle. Les fonctions de communication exploitent ces informations pour garantir l'intégrité des canaux et liens. Pour reprendre ses droits sur les canaux et liens, le processus père restaure son propre marquage à la fin des fils.

### 2.4. MÉMOIRE PARTAGÉE.

Tout comme les canaux, la mémoire partagée sur chaque noeud est mise en oeuvre sur la base de segments de mémoire partagée. OVIPAR ne garantit pas l'accès en exclusion mutuelle dans ces segments. Leur taille est de un kilo octets chacun.

## 3. BIBLIOTHÈQUE DE FONCTIONS D'OVIPAR.

La bibliothèque d'OVIPAR comporte moins d'une dizaine de fonctions essentielles.

### 3.1. CRÉATION ET CONFIGURATION DE L'ORDINATEUR : OVIPAR.

Cette fonction permet de spécifier la topologie de la machine virtuelle. Elle prend en paramètre le nombre de noeuds, la spécification du réseau par des connexions point à point et un pointeur sur la fonction de chargement avec ses arguments. La topologie est décrite sous forme d'une chaîne de caractères composée d'une séquence de connexions. Chaque connexion se présente sous la forme: IC<sub>1</sub>JC<sub>2</sub>, tels que I et J sont des numéros de noeuds, et C<sub>1</sub> et C<sub>2</sub> sont des liens (i.e. N, S, E, W). La fonction de chargement dont le pointeur représente le troisième paramètre doit être écrite suivant le modèle de la fonction *main* du langage C. Le quatrième paramètre est le nombre de paramètres de la fonction de chargement. Enfin, le dernier paramètre est un tableau constitué de paramètres de la fonction de chargement. La fonction *ovipar* retourne 0 en cas de succès.

```
int ovipar (nodesnum, topo, fct, argc, argv)
    int    nodesnum;          Specify the number of nodes.
    char*  topo;              Describe the node interconnection topology in
```

<pre>int (fct*());</pre> <p>function is</p>	<p>term of couples of links to be connected. Entry point of the parallel program. This</p> <p>expected to load the nodes. It must be written in the format of C main function.</p> <p>Argument count of fct. Argument vector of fct.</p>
<pre>int argc ;</pre> <pre>char* argv ;</pre>	

### 3.2. CRÉATION D'UN DESCRIPTEUR DE PROCESSUS : INITPROC.

Cette fonction permet de créer un descripteur de processus, c'est à dire une structure de données OVIPAR contenant entre autres la fenêtre du processus et le pointeur de la fonction (point d'entrée) correspondant au code du processus. La fonction point d'entrée doit être écrite suivant le format de *main*. La spécification de la fenêtre du processus se fait en indiquant les liens et les canaux dont le processus hérite. La liste des liens est spécifiée sous la forme d'une somme booléenne CHN\_NORTH + CHN\_SOUTH + CHN\_EAST + CHN\_WEST selon les liens utilisables. Les canaux sont spécifiés à travers une liste. Les paramètres de la fonction point d'entrée sont spécifiés comme pour la fonction *main*. Il est recommandé d'utiliser un tableau pour regrouper tous les descripteurs.

<pre>void <b>initproc</b> (pdesc, fct, node, ll, chn, argc, argv)</pre> <pre>  csp_t pdesc ;</pre> <pre>  int (fct*());</pre> <p>be</p> <p><i>par</i></p> <pre>  char ll ;</pre> <pre>  chn_t chn ;</pre> <pre>  int argc ;</pre> <pre>  char* argv ;</pre>	<p>Empty process descriptor.</p> <p>Entry point function of the process. It is the first process to be executed on the node. It must</p> <p>written in the format of C main function. This function could also be a process model for the</p> <p>instruction.</p> <p>Bitmap specifying the node links to be used.</p> <p>Internal channel list. Only used with <i>par</i> instruction.</p> <p>Argument count of fct.</p> <p>Argument vector of fct.</p>
---	---

La fonction *initcomp* fait la même chose que la fonction *initproc* pour créer les composantes des noeuds. Elle admet un paramètre supplémentaire qui est le numéro du noeud sur lequel la composante est créée.

### 3.3. CHARGEMENT DE LA MACHINE : PLACEDPAR.

Cette fonction de style *main* permet de charger la machine OVIPAR. Elle prend en paramètre une table de descripteurs de processus créés avec la fonction *initproc*, crée les processus (appelés composantes) et les placent dans les différents noeuds de la

machine parallèle. Le programme ayant appelé cette fonction se bloque jusqu'à la fin des processus qui ont été créés.

```
int placedpar (sz, pdesctab)
    int    sz ;                Size of table pdesctab.
    csp_t  pdesctab[] ;      Table in which each item is a process descriptor.
                                These descriptors are created with the function
                                initproc.
```

La fonction *placedparl* est une version de *placedpar* qui prend en paramètre une liste de descripteurs de processus terminée par 0.

### 3.4. CRÉATION DE PROCESSUS SUR UN NOEUD: PAR.

Cette fonction prend en paramètre une table de descripteurs de processus créés avec la fonction *initproc*, et crée les processus sur le noeud à partir duquel elle a été appelée. Le processus ayant appelé cette fonction se bloque jusqu'à la fin des processus qui ont été créés.

```
int par (sz, pdesctab)
    int    sz ;                Size of table pdesctab.
    csp_t  pdesctab[] ;      Table in which each item is a process descriptor.
                                These descriptors are created with the function
                                initpdesc.
```

La fonction *parl* est une version de *par* qui prend en paramètre une liste de descripteurs de processus terminée par 0.

### 3.5. ÉMISSION D'UN MESSAGE: SEND.

Cette fonction permet au processus qui l'appelle d'envoyer un message à un autre processus. La fonction prend en paramètre le canal par lequel le message doit être envoyé, ainsi que l'adresse du message et sa taille. Le processus qui l'a appelé reste bloqué jusqu'à ce que le message soit transmis. La fonction retourne le minimum entre le nombre d'octets envoyés et le nombre d'octets attendus par le destinataire.

```
int send (chn, msg, msgsz)
    chn_t  chn ;                Channel descriptor.
    char*  msg ;                Message storage buffer.
    int    msgsz ;              Message size.
```

### 3.6. RÉCEPTION D'UN MESSAGE: RCEV.

Cette fonction permet au processus qui l'appelle de recevoir un message émis par un autre processus. La fonction prend en paramètre le canal par lequel le message sera

reçu, ainsi que l'adresse du buffer où il sera stocké, et le nombre d'octets attendus. Le processus qui l'a appelé reste bloqué jusqu'à ce que le message soit reçu. La fonction retourne le minimum entre le nombre d'octets envoyés et le nombre d'octets attendus.

int **recv** (chn, msg, msgsz)

chn_t	chn ;	Channel descriptor.
char*	msg ;	Storage buffer.
int	msgsz ;	Expected message size.

### 3.7. GESTION DE CANAUX INTERNES : CHANGET, CHANDEL.

La première de ces fonction permet de créer un canal sur un noeud, dont la taille donnée correspond au nombre maximal d'octets qui pourront transiter par ce canal. Si le nombre d'octets émis ou le nombre attendus dépassent cette taille, alors le nombre d'octets effectivement transmis sera égal à cette taille. La deuxième fonction permet de supprimer un canal.

chn\_t **changet** (sz) ;

int	sz ;	Maximum number of bytes which can transit by the channel.
-----	------	---

int **chandel** (chn) ;

chn_t	chn ;	Descriptor of the channel to be deleted.
-------	-------	--

### 3.8. CHOIX GARDÉ D'UNE COMMUNICATION : ALT.

La fonction *alt* permet d'attendre la présence d'une communication sur une liste de canaux. Si plusieurs communications sont présentes, la fonction choisit aléatoirement l'une d'elle et retourne le numéro dans la liste du canal associé. Sinon elle attend l'arrivée de la première communication et retourne le numéro du canal associé. Cette fonction ne reçoit pas le message. Le nombre de canaux passés en paramètre est variable. Le dernier paramètre doit être 0.

int **alt** (chn, ...)

chn_t	chn ;	The first of a list of channels ending by 0.
-------	-------	--

### 3.9. ACCÈS À LA MÉMOIRE PARTAGÉE D'UN NOEUD : VMEM.

Cette fonction permet à un processus de lire ou écrire une valeur dans la mémoire partagée du noeud dans lequel il s'exécute. L'accès se fait à partir d'un emplacement donné relativement au début de la mémoire partagée. Cette mémoire a une taille d'un kilo octets.

int **vmem** (buf, sz, os, cmd)

char*	buf ;	Buffer containing the message to be stored in the shared memory, or buffer in which the message extracted from the shared memory will be stored.
-------	-------	--



int	sz ;	Message size.
int	offset ;	Offset from the beginning of the shared memory.
int	cmd ;	0 to read and 1 to write.

## CONCLUSION.

L'outil qui vient d'être présenté est disponible à l'IAI et est en phase de diffusion. Cet outil n'est utilisable que dans un cadre strictement académique. Son principal avantage est le nombre de fonctions réduit qui en facilite l'apprentissage.

Nous avons dit qu'OVIPAR doit être arrêté par les procédures habituelles de terminaisons de programmes C. Une version améliorée permettrait de terminer proprement OVIPAR à la suite d'un incident. Il s'agira en particulier de terminer les processus pendants, et de libérer les sémaphores et les segments de mémoire partagée. La mise en oeuvre d'une telle procédure ne pose pas de problème particulier en exploitant par exemple un fichier de trace système.

Une autre amélioration possible est de rendre la machine OVIPAR permanente. Cela permettrait d'exécuter plusieurs programmes consécutifs sans arrêter OVIPAR. Là aussi, il n'y a pas de problème majeur, étant donné que les techniques de communication entre processus et avec le shell d'UNIX sont bien maîtrisées.

## Références.

- [Alm1] Almeida Antonio  
OVIPAR, Ordinateur Vituel Parallèle sous UNIX  
Manuel d'utilisation, 1995, IAI BP 2263 Libreville GABON.
- [Alm2] Almeida Antonio  
OVIPAR, Conception et Mise en Oeuvre d'un Ordinateur Vituel Parallèle sous UNIX  
Mémoire de Fin de Cycle Ingénieur, 1995, IAI BP 2263 Libreville GABON.
- [Che] Chergui Jalel, Escobar Juan, Girou Denis  
Programmation par Echange de Messages (PVM)  
Institut du Développement des Ressources en Informatique Scientifique, Fév. 95
- [Gei] Geist AI, et al.  
PVM 3 User's Guide and Reference Manual  
ORNL / TM-12187, Sept. 1994. Oak Ridge, Tennessee 37831,
- [INM] INMOS  
The T9000 Transputer, Product Overview Manual  
INMOS, First Edition 1991
- [Mou] Moukèli Pierre  
Parallel Computer Operating Systems

[Wal] Walker D.W.

The Design of a Standard Message Passing Interface for Distributed  
Memory Concurrent Computers  
Parallel Computing, Vol. 20, 1994, PP. 657-673

## ANNEXE: EXEMPLES DE PROGRAMMES OVIPAR.

Les programmes décrits dans cette section sont disponibles dans la bibliothèque des exemples OVIPAR et doivent être compilés avec la commande suivante (dans laquelle *demo.c* désigne le nom du programme) :

```
cc -o demo.c -lovipar
```

### A.1. PROGRAMMATION D'UN ANNEAU.

Cet exemple montre comment programmer un anneau. Il crée cinq processeurs reliés entre eux par les liens est ouest. Sur chaque processeur s'exécute un processus dont le rôle est de recevoir un jeton du voisin ouest, de le modifier, et de le transmettre au voisin est. Le premier processus sert de frontal.

```
void front (int argc, char* argv[], chn_t chnv[]) {          /* executed by the front node */
    int    token ;
            /* Read the token */
    printf (« Input an integer : ») ;
    scanf (« %d », &token) ;
            /* Send the token to the front node */
    send (chn_east, (char*)token, sizeof (int)) ;
            /* expect the token coming back */
    recv (chn_west, (char*)token, sizeof (int)) ;
    printf («New value of the token is : %d\n», token) ;
}

void transit (int argc, char* argv[], chn_t chnv[]) {      /* executed by the intermediate
nodes */
    int    token ;
            /* expect the token from west channel */
    recv (chn_west, (char*)token, sizeof (int)) ;
    token++ ;
            /* Send the token to the east channel */
    send (chn_east, (char*)token, sizeof (int)) ;
}

int tokenring () {          /* Create process descriptors. and specify node entries */
    csp_t  p[5] ;
    initcomp (&p[0], front, 1, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
    initcomp (&p[1], transit, 2, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
    initcomp (&p[2], transit, 3, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
    initcomp (&p[3], transit, 4, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
}
```

```

initproc (&p|4|, transit, 5, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
    return placedpar (5, p) ;
}

```

```

int main () {          /* Host entry point : computer creation and configuration */
                        /* E refers to EAST link and W refers to WEST link */
    ovipar (5, «1E2W 2E3W 3E4W 4E5W 5E1W», tokenring, 0 NULL)) ;
}

```

## A.2. PROGRAMMATION D'UN PRODUCTEUR CONSOMMATEUR.

L'objectif est de montrer comment deux processus peuvent accéder à la mémoire partagée de leur noeud, et comment utiliser la fonction *par*. Cet exemple crée deux processus auxquels sont associés les fonctions *père* et *arbitre*. Le père crée deux fils: *lire*, *écrire*, l'un récupère dans la mémoire partagée une valeur que l'autre a incrite. Les deux fils sont synchronisés par le processus *arbitre*.

```

void arbitre (int argc, char* argv[], chn_t chnv[]) {
    int    c ;
    recv (chn_north, &c, 1) ;
    send (chn_south, &c, 1) ;
}

void lire (int argc, char* argv[], chn_t chnv[]) {
    int    a = 0 ;
    int    c ;
    recv (chn_south, &c, 1) ;
    vpmem (&a, sizeof(int), 0, VP_READ) ;
    printf (« Valeur reçue: %d\n », a) ;
}

void écrire (int argc, char* argv[], chn_t chnv[]) {
    int    a ;
    printf (« Entrer une valeur entière: ») ;
    scanf (« %d », &a) ;
    vpmem (&a, sizeof(int), 0, VP_WRITE) ;
    send ((chn_north, «*», 1) ;
}

void père (int argc, char* argv[], chn_t chnv[]) {
    csp_t p[2] ;
    initproc (&p[0], écrire, CHN_NORTH, 0, NULL) ;
    initproc (&p[1], lire, CHN_SOUTH, 0, NULL) ;
    par (2, p) ;
}

void chargement () {
    csp_t p[2] ;
    initcomp (&p[0], arbitre, 1, CHN_NORTH + CHN_SOUTH, 0, NULL) ;
    initcomp (&p[1], père, 2, CHN_NORTH + CHN_SOUTH, 0, NULL) ;
    placedpar (2, p) ;
}

main () {
    if (ovipar(2, « 1N2N 1S2S, chargement, 0, NULL))
        printf (« Erreur ovipar n° %d\n », vp_errno) ;
}

```