Synchronisation d'Inter, le tutorial

Luc Weber

Observatoire de Genve
October 24, 2012

Contents

Chapter 1

Introduction

Inter est un interprteur de commandes. Comme tout interprteur de ce type il travaille en interactif. C'est dire qu'il attend des commandes de l'utilisateur en provenance du clavier. Pour signaler cette attente, il propose un prompt ("une invite" en franais). Quand le prompt n'est pas affich, on sait que l'interprteur travaille. Quand le prompt est prsent on sait que l'interprteur ne travaille pas (c'est tout simplement incroyable).

Inter travaille galement dans un mode nomm *serveur*. Dans ce mode, les commandes qui lui parviennent sont envoye par un autre programme (le *client*) et ne lui parviennent donc plus par le clavier, mais sont crites dans une zone de mmoire partage que l'on appellera ici *le bloc de communication*.

Ce mode de travail permet ainsi aux programmes de communiquer entre–eux. Ainsi un *serveur* peut recevoir des ordres de plusieurs *clients*. Le prompt n'existe plus et la notion de lancer une commande ou simplement savoir si le *serveur* travaille n'est pas aussi vidente que lors du travail en interactif.

Un *serveur* est considrer comme une *ressource* partageable mais ne pouvant excuter qu'une tche la fois. Le meilleure reprsentation d'une telle ressource est l'imprimante. Elle accepte des jobs de plusieurs expditeurs, et traite chacun de ces jobs les uns aprs les autres.

L'utilisation d'une ressource de type *serveur* implique le mise en place d'un mcanisme de rservation et de synchronisation bas sur les *smaphores*.

1.1 Qu'est-ce qu'un smaphores

Le smaphore est un objet informatique se prsentant (en schmatisant) sous la forme d'une variable globales accessible par tout les process. Cet objet a deux compteurs associs `NCNT` et `ZCNT`.

Un smaphore est associ un identificateur (de type entier).

Les proprits des smaphores sont les suivantes:

- Un smaphore supporte les oprations d'initialisations une valeur plus grande ou gale zro, ainsi que les oprations d'incrmentations et de dcrmentations.
- Si la valeur d'un smaphore est dcrmente alors que le smaphore vaut zro, le process excutant cette opration est mis en attente jusqu' moment o un autre process incrmente

le smaphore. En cas de mise en attente, le compteur NCNT est incrment. Il totalise le nombre de process en attente.

- Un process peut tre en attente sur la valeur zro d'un smaphore. Dans ce cas c'est le compteur ZCNT qui totalise le nombre de process en attente.
- Les process en attente sont ractivs dans leurs ordres d'arrive (FIFO)
- Un process en attente est ractiv:
  1. lorsque le smaphore a une valeur suprieure ou gale zro
  2. lorsque le process reoit un signal
  3. lorsque que le smaphore est dtruit.

1.2 Principe de la synchronisation

Les smaphores permettent de synchroniser l'accs au *serveur* en bloquant les *clients* dsirant l'utiliser. Il faut toutefois remarquer qu'un *client* peut accder un *serveur* sans utiliser ce mode de synchronisation. Cela peut tre utile pour communiquer avec un *serveur* excutant une tache en arrire plan (voir plus loin sous "communication asynchrone"), mais dans la plupart des cas les accs asynchrones gnreront des situations illgales difficile contrler ou identifier.

La synchronisation utilise 3 smaphores que l'on appelle `SEM0`, `SEM1` et `SEM2`. Ils ont les fonctions suivantes:

1.2.1 Fonctionnalits de SEM0

`SEM0` permet de grer l'accs au *serveur*. Il est initialis 1 par le *serveur*, indiquant par l que le *serveur* est libre. Chaque *client* voulant accder le *serveur* doit commencer par dcrmenter ce smaphore avant d'effectuer une quelconque opration sur le bloc de communication ou sur les autres smaphores. Si le *serveur* est occup, le *client* et mis en attente et `NCNT0` est incrment d'une unit.
Selon le mode de synchronisation, en fin de travail, c'est le *serveur* ou le *client* qui incrmente `SEM0` pour librer l'accs au *serveur* pour le *client* suivant.

1.2.2 Fonctionnalits de SEM1

`SEM1` bloque le *serveur* tant que le bloc de communication ne contient rien de valide. Il est initialis zro par le *serveur* qui se met tout de suite en attente par une dcrmentation (dans ce cas `NCNT1=1`). C'est le *client* qui incrmente ce smaphore lorsqu'il a obtenu l'accs au *serveur* et remplis le bloc de communication.
Le *serveur* se remet en attente automatiquement en dcrmentant `SEM1` en fin de travail.

1.2.3 Fonctionnalits de SEM2

`SEM2` est utilis comme un flag, il indique si le *serveur* excute une commande. Lorsqu'il vaut zro, le *serveur* ne travaille pas, lorsqu'il vaut 1, il travaille. C'est toujours le *client* qui le pose 1 avant d'ordonner l'excution d'une commande au *serveur* en incrmentant le `SEM1`. C'est le

serveur qui le pose zro la fin d'une excution. Si le *client* veut attendre la fin d'une excution, il se met en attente de valeur zro sur ce smaphore (dans ce cas `ZCNT2=1`).
Ce smaphore est utilis lors des oprations d'initialisation d'un *serveur* o un *serveur* peut savoir s'il a t tu durant l'excution d'une commande (`SEM2=1`) et ainsi le signaler au *client* qui peut tre toujours en attente.

1.3 Accs aux smaphores

Ces smaphores ne sont pas accds directement par *Inter* au niveau des procdures mais par une srie de fonctions. Par exemple, rserver un *serveur* se fait en dcrmentant `SEM0`. Dans une procdure on utilisera la fonction `shmwait()`.
Les smaphores d'*Inter* sont visualiss avec le moniteur d'tat des smaphores `ipcstat`.

1.4 Comportement des *clients* et *serveurs*

Voici la liste des proprits et comportements des *clients* et *serveurs* dans des situations extraordinaires:

- Un *serveur* a la proprit de pouvoir survivre la perte de ses smaphores (aprs un `ipckill` par exemple), il arrive les refabriquer.
- Si les smaphores sont tus deux fois de suite sans qu'aucune communication n'a eu lieu, le *serveur* se "suicide".
- Si un *serveur* est tu sans qu'il soit en communication, aucun problme.
- Si un *client* essaie de se connecter sur un *serveur* inexistant, le *client* est suspendu et sa commande sera excute lorsque le *client* sera lanc.
- Si un *serveur* est tu pendant une communication avec un *client* en attente. Le *client* reste en attente jusqu'au moment ou le *serveur* est relanc.
- Si un *serveur* est tu pendant une communication sans *client* en attente. Aucun problme.
- Un *client* la proprit de pouvoir se connecter un *serveur* n'importe quel moment. Il peut se dconnecter que s'il n'est pas en communication avec un serveur. Les *clients* peuvent ainsi tre lancs et tus un nombre de fois illimit
- Si un *client* en attente est tu, le smaphore ne sera pas libr. Il faut donc le librer avec un `shmfree()` sur un autre au *client* (voir aussi l'utilitaire `ipcstat`).

Chapter 2

Les bases

L'apprentissage de la confection des procdures va se faire pas pas dans ce chapitre. Commenons donc par le commencement.

2.1 Lancer Inter en mode *serveur*

Inter doit savoir qu'il ne travaille plus en interactif. Cela se fait au lancement avec les options de la ligne de commande. Dans le cas le plus simple o on ne dsire qu'un *Inter serveur*. On utilise l'identificateur de smaphore par dfaut (1000). La commande est:

```
SunOS 1> inter -server
```

Dans ce cas, l'*Inter* prend le bloc de donnes par dfaut (*application*.blk). On peut donner le nom d'un autre bloc sur la ligne de commande. Par exemple:

```
SunOS 1> inter -server -block standard.blk
```

Si en plus on dsire que l'*Inter* affiche son tat ainsi que l'cho des commandes, on tape:

```
SunOS 1> inter -server -block standard.blk -echo

 Inter: mode server, block = standard.blk

Configuration des matrices:
---------------------------

Maximum:   22 matrices sur  20 couches

 Pas de matrice alloues en mmoire dynamique
 ------------ Inter (mode serveur) en attente ------------
```

Si on lance `ipcstat` l'tat des smaphores est:

```
Sem #0 = 1   ncount=0   zcount=0
Sem #1 = 0   ncount=1   zcount=0
Sem #2 = 0   ncount=0   zcount=0
```

C'est l'tat normal d'un *Inter serveur* en attente de commande. A partir de cet instant il est prt.

2.2 Lancer Inter en mode *client*

Si on dsire connecter un *Inter client* sur un *Inter serveur* travaillant sur l'identificateur standard (1000) on tape:

```
SunOS 1> inter -client
```

Ce *client* travaille de manire normale et en plus s'accroche aux smaphores du *serveur*. Il peut donc lui envoyer des commandes.

2.3 Rserver un *serveur*

Comme on l'a vu prcdemment, tout accs synchrone un *serveur* doit tre prcd par une rservation du *serveur*. On le fait avec la commande `shmwait()`. Exemple:

```
INTER > shmstat=shmwait()
```

A ce moment, les smaphores passent dans l'tat suivant:

```
Sem #0 = 0   ncount=0   zcount=0
Sem #1 = 0   ncount=1   zcount=0
Sem #2 = 0   ncount=0   zcount=0
```

On voit ($Sem0 == 0$)qu'aucun autre *client* ne pourra rserver le *serveur* et que le *serveur* est toujours en attente de commande ($ncount1 == 1$).
On peut faire une attente de rservation durant un temps limit. Dans ce cas on prcise la dure en secondes comme argument de la fonction. Exemple d'attente de 30 secondes:

```
INTER > shmstat=shmwait(30)
```

Lorsqu'un timeout survient, le status (ici `shmstat`) vaut `114` (c'est dire `100` plus le code du signal `SIGALRM` (`14`).

2.4 Formulation de la commande

L'envoi de la commande se fait en deux temps. D'abord on l'crit dans le bloc de communication puis on signifie au *serveur* qu'il peut l'excuter.

Le bloc de communication est une structure qui contient plusieurs lments, (nous les dcrirons au fur et mesure de leur emploi dans ce document) dont une table compose de couples <cl;contenu>. Le nombre de couples est limit dans la librairie (actuellement 100 dans ipcdef.h pour libipc.a).
Ces couples permettent le passage de paramtres nomms entre un *client* et un *serveur*.

Un de ces paramtres est le mot cl `"COMMAND"`. C'est celui–ci que lit le *serveur* lorsqu'il dbute une excution de commande.

Remarque: si `COMMAND=""` ou si `COMMAND` n'est pas dfinit, le message `"Pas de keyword COMMAND valide"` apparat sur le *serveur* et aucune erreur n'est gnre.

L'criture de ce mot–cl se fait avec la fonction `shmput()`. Cette fonction est destructive, car elle vide le bloc avant d'y placer le mot–cl et son contenu. Par exemple:

```
INTER > shmstat=shmput("COMMAND", "show i")
```

On voit que la commande contient des arguments sur la ligne de commande (ici le `"i"`). Dans le cas du lancement d'une procdure, les neuf arguments standards se passent sur la ligne de commande, auxquels on peut ajouter des paramtres dans le bloc de communication.

Pour stocker ces paramtres, on utilise la fonction non–destructive `shmadd()`. Remarque: cette fonction remplace le contenu d'un mot–cl en cas de synonyme. Par exemple, si on veut donner la valeur de tout les `NX` et `NY` du *client* au *serveur*, on tape:

```
INTER > shmstat=shmput("COMMAND", "@dosomething")
INTER > do i=1,dim(nx)
INTER >   shmstat=shmadd(lcat("NX(",itoa(i),")"),nx(i))
INTER >   shmstat=shmadd(lcat("NY(",itoa(i),")"),ny(i))
INTER > enddo
```

Attention, la longueur des noms des paramtres est limite 12 et la longueur des contenus et limite 128.
Maintenant que la commande et ses paramtres facultatifs sont inscrit dans le bloc de communication, le *client* doit signifier au *serveur* de dmarrer la commande. Il peut le faire selon deux modes.

1. **Mode WAIT** dans ce mode, lorsque le *serveur* termine l'excution de la commande, le *client* a toujours la main. Le *serveur* lui est toujours rserv.
2. **Mode NOWAIT** dans ce mode, lorsque le *serveur* termine l'excution de la commande, il rend la main. Le *client*, s'il veut envoyer une nouvelle commande, devra nouveau rserver le *serveur*.

2.5 Excution de la commande en mode WAIT

Le *client* veut donc conserver son *serveur*. Il pourra, et cela est trs important, tester le status de l'excution de la commande. Une variable du bloc de communication nomm `ackno` indique si le *serveur* doit rendre la main ($ackno == 0$) ou s'il ne doit pas rendre la main ($ackno == 1$) En gardant la main, le *client* pourra envoyer de nouvelles commandes sans se faire perturber par d'autres *clients* en attente. Il faut toutefois tre conscient que si on garde la main, il faut absolument s'assurer que le *serveur* ait finis son travail avant d'crire ou lire dans le bloc. La fonction de lancement de l'excution de la commande sur le *serveur* en mode wait est `shmack()` et la fonction d'attente de la fin de cette excution est `shmwack()`. Par exemple:

```
INTER > shmstat=shmput("COMMAND", "@dosomething")
INTER > shmstat=shmack()
INTER > shmstat=shmwack()
```

Lors de l'excution, les smaphores sont dans l'tat suivant:

```
Sem #0 = 0   ncount=0   zcount=0
Sem #1 = 0   ncount=0   zcount=0
Sem #2 = 1   ncount=0   zcount=0       ackno=1
```

Ds la fin de l'excution, les smaphores sont dans l'tat suivant:

```
Sem #0 = 0   ncount=0   zcount=0
Sem #1 = 0   ncount=1   zcount=0
Sem #2 = 0   ncount=0   zcount=0       ackno=1
```

Si le temps de rponse doit tre limit par un timeout, on le donne en secondes dans l'appel `shmwack()`. Par exemple, une attente se 20 secondes s'crira:

```
INTER > shmstat=shmwack(20)
```

Le status retourn lors du `shmwack()` permet de dterminer s'il y a eu une erreur, de quel cot elle a eu lieu et le type d'erreur. Les valeurs de status possibles sont:

1	erreur Inter sur le serveur
2	serveur dconnect
3	SIGINT (CTRL–C) sur le serveur
101	SIGHUP sur le client
102	SIGINT (CTRL–C) sur le client
113	SIGPIPE sur le client
114	SIGALRM (timeout) sur le client

2.6 Le *client* rend la main

Dans le cas d'une communication en mode WAIT, c'est le *client* qui finalement doit rendre la main. Il le fait avec la commande `shmfree()`. Dans l'exemple suivant, le *client* envoie deux commande successive en gardant la main puis libre le *serveur*.

```
INTER > shmstat=shmput("COMMAND", "@dosomething")
INTER > shmstat=shmack()
INTER > shmstat=shmwack()
INTER > shmstat=shmput("COMMAND", "@dootherthings")
INTER > shmstat=shmack()
INTER > shmstat=shmwack()
INTER > shmstat=shmfree()
```

2.7 Excution de la commande en mode NOWAIT

Dans ce cas, c'est le *serveur* qui rendra la main. Le *client* n'aura pas le faire, mais il ne pourra pas connatre le status de la commande. C'est le prix payer! Ce mode est connu du *serveur* car la variable `ackno` du bloc de communication vaut `0`.
La fonction de lancement de l'excution de la commande sur le *serveur* en mode nowait est `shmcont()`. Par exemple:

```
INTER > shmstat=shmput("COMMAND", "@dosomething")
INTER > shmstat=shmcont()
```

Lors de l'excution, les smaphores sont dans l'tat suivant:

```
Sem #0 = 0   ncount=0   zcount=0
Sem #1 = 0   ncount=0   zcount=0
Sem #2 = 1   ncount=0   zcount=0       ackno=0
```

A la fin de l'excution, le *serveur* se met nouveau en attente de rservation.

```
Sem #0 = 1   ncount=0   zcount=0
Sem #1 = 0   ncount=1   zcount=0
Sem #2 = 0   ncount=0   zcount=0       ackno=0
```

2.8 Macro commandes

Il existe deux macro–commandes permettant de raliser les fonctions standards en un seul appel. Elles permettent d'envoyer une simple commande puis de librer le *serveur* selon les mode `wait` et `nowait`. Ces commandes sont dcrite ici pour le principe, car la tendance actuelle est plutt d'utiliser des procdures pour raliser ce type de commandes plus complexes. Ce sont `shmcmdw()` et `shmcmd()`. Il faut remarquer que ces commande n'accptent pas l'emploi des paramtres du bloc de communication ni les commandes successivent avec *serveur* rserv.
La syntaxe est:

```
shmcmdw(<commande>[, <timeout_wait>[, <timeout_wack>]])
shmcmd(<commande>[, <timeout_wait>])
```

Par exemple:

```
INTER > shmstat=shmcmdw("@dosomething",30)
INTER > shmstat=shmcmd("@dosomething")
```

Les status de retour sont les mme que pour la commande `shmwack()`.

2.9 Lecture des paramtres du bloc de communication

On a vu qu'un *client* pouvait crire des paramtres nomms dans le bloc de communication. En fait l'accs ces bloc est compltement libre. Un *serveur* peut galement crire dans ce bloc pour, par exemple, retourner des rsultats au *client*. La rcupration du contenu des paramtres se fait avec la fonction `shmget()`. Attention, on rcupre toujours des chanes de caractres. Par exemple, la rcupration des `NX` et `NY` donns dans un exemple prcdent se fait de la manire suivante:

```
INTER > do i=1,dim(nx)
INTER >   nx(i)=ator(shmget(lcat("NX(",itoa(i),")")))
INTER >   ny(i)=ator(shmget(lcat("NY(",itoa(i),")")))
INTER > enddo
```

Une erreur apparat si le paramtre n'existe pas.
Une fonction trs pratique permet de lire les paramtres du bloc s'il sont nomms comme les variable du bloc de donne. Cette commande est `fetch()`. Elle tente de lire tous les paramtres du bloc sauf le mot–cl `COMMAND`. Elle met un warning, mais pas d'erreur si un paramtre n'a pas son correspondant dans les variables *Inter*. Elle retourne toujours zro. L'exemple prcdent s'crit dans ce cas simplement:

```
INTER > shmstat=fetch()
```

Le bloc peut tre vid avec la commande `shminit()` et visualis avec la commande `shmshow()`. `ipcstat` permet galement de monitorer le contenu du bloc de communication.

Chapter 3

Travail avec plusieurs *serveurs*

Un *client* peut communiquer avec plusieurs *serveurs*. Dans ce cas, il est ncessaire de donner un identificateur numrique pour chaque *serveur*.

3.1 Spcifier un identificateur

Pour le *serveur*, on donnera simplement le numro de cl aprs l'option `-serveur`. Par exemple:

```
SunOS 1> inter -server 2000
```

Pour le *client* (partant du principe que dans la plupart des cas, si on spcifie un identificateur c'est que l'on se connecte sur plusieurs *serveurs*) on donnera un nom symbolique associ avec l'identificateur de *serveur*. Par exemple, si le *client* se connecte sur un *Inter*-CCD et un *Inter*-Spectro, on tapera:

```
SunOS 1> inter -Cccd 1000 -Cspectro 2000
```

Attention, le nom symbolique doit tre coll au `-C`.

3.2 Le *client* est un *serveur*

Si un *client* est le *serveur* d'un autre *client*, les mode dcrits prcdemment sont utiliss. Par exemple:

```
SunOS 1> inter -serveur 3000 -Cccd 1000 -Cspectro 2000
```

3.3 Slectionner un *serveur*

Lorsqu'un *client* veut s'adresser un de ses *serveurs*, il doit le slectionner au pralable. La slection s'effectue avec la fonction `select()` sur les noms symboliques donns sur la ligne de commande. Par exemple, on envoie une commande en mode wait sur les deux *serveurs* en parallle et on attend la fin des deux oprations:

```
INTER > shmstat=select("ccd")
INTER > shmstat=shmput("COMMAND", "@dosomething")
INTER > shmstat=shmack()
INTER > shmstat=select("spectro")
INTER > shmstat=shmput("COMMAND", "@dosomethingelse")
INTER > shmstat=shmack()
INTER > shmstat=shmwack()
INTER > shmstat=select("ccd")
INTER > shmstat=shmwack()
```

La fonction `showsel()` affiche la liste des *serveurs* et prcise le *serveur* courant slectionn

Chapter 4

Les autres fonctions

Dans les exemples cits ci–dessus, les commandes se droulent sans accrocs. Les cas problmatique peuvent se grer dans les procdures. Une srie de fonctions permettent de dterminer l'tat des *serveurs* et des *clients* et aident ainsi la ralisation d'un rtablissement automatique.

4.1 Qui suis–je

Les fonctions `shmcli()` (resp. `shmsrv()`) retourne `1.0` si *Inter* a t lanc en mode *client* (resp. *serveur*).

4.2 SRVWAIT()

Cette fonction retourne `1.0` si le *serveur* est rserv pour le *client* qui excute cette fonction. Par exemple:

```
INTER > sh srvwait()
   srvwait() =      0.
INTER > shmstat=shmwait()
INTER > sh srvwait()
   srvwait() =  1.0000000
```

`Srvwait()` peut tre trs utile pour librer un *serveur* alors que l'on est pas sur d'tre son *client*. Par exemple:

```
if srvwait() shmstat=shmfree()
```

4.3 SRVWORK()

Cette fonction retourne `0.0` si le *serveur* est sur le prompt, c'est dire $SEM2 == 0$ (indpendamment du fait qu'il soit ou non rserv). Si le *serveur* travaille ($SEM2 == 1$), elle retourne `1.0` si le *serveur* travaille pour le *client* qui excute cette fonction ou retourne `-1.0` si le *serveur* travaille pour un autre *client*. Par exemple:

```
INTER > shmstat=shmput("COMMAND","i=sleep(10)")
INTER > shmstat=shmack()
INTER > sh srvwork()
   srvwork() =  1.0000000
INTER > shmstat=shmwack()
INTER > sh srvwork()
   srvwork() =         0.
```

4.4 Communication bloquante

Si un *client* meurt alors qu'il a rserv un *serveur*, on obtient une situation blocante. Il faut librer le smaphores la place du dfunt. On le fait en posant $SEM0 = 0$ avec la fonction `shmzero()` ou avec `ipcstat`.

Lors d'un reset automatique, il faut prendre garde au fait que si d'autres *clients* sont en attente sur le *serveur* bloqu, le premier de la queue sera libr l'instant mme du reset du smaphore et pourra travailler avec le *serveur*. Cette situation ne doit pas arriver, si par exemple le *client* dfunt devait faire un action essentielle avant l'action du second *client*.

La fonction `shmncnt()` retourne le nombre de *client* en attente sur le *serveur*. Cela signifie que dans le cas o aucun *client* n'est en attente. Le reset peut s'oprer sans aucun risque. Si des *clients* sont en attente, il est souvent prfrable de laisser un contrle manuel l'utilisateur. Par exemple:

```
write "Attention rupture de squence"
if shmncnt().gt.0 then
    write "Des clients sont en attente"
    write "Il faut remettre le systme en tat manuellement"
else
    write "Il n'y a pas de client en attente"
    write "On  fait un reset du smaphore 0"
    shmstat=shmzero()
endif
erreur /set
```

4.5 Accs aux variables du bloc de communication

Les variables du bloc de communication donnent le status de la dernire communication. Ce sont `status`, `erreur` et `code d'erreur`.

- `status` est pos (par le *serveur*) `2` lorsque qu'une commande dmarre et passe `0` lorsqu'elle se termine. Ainsi, si aprs un `shmwait()` (resp. un `shmwack()`) `status` vaut `2`, cela signifie que le *serveur* a t tu durant la commande prcdente (resp. courante).
- `erreur` donne un code d'erreur qui donne le type d'erreur:

  1 erreur Inter sur le serveur
  3 SIGINT (CTRL–C) sur le serveur

  Il est important de remarquer que cette valeur n'est pas quivalente au status de la commande `shmwack()`. D'une part `shmwack()` retourne les erreur qui surviennent ct *client* et d'autre part le status prend la valeur de la variable `status` si `status` vaut `2`.
- `code d'erreur` nom symbolique des erreurs *Inter* (ex: "`int_errsyn`").

Lorsque l'on rserve un *serveur*, si on ne dsire pas connatre le status de la dernire communication, on peut effacer tout les status avec la fonction `clearsv()`. Par exemple:

```
shmstat=shmwait()
shmstat=clearsv()
```

Par contre, si on dsire connatre le status de la dernire communication les fonctions `shmgerr()`, `shmgsta()` et `shmgcod()` permettre de les rcuprer. Par exemple:

```
local code_err=""
shmstat=shmwait()
if shmgsta.ne.0 then
  ...
endif
if shmgerr().ne.0 then
   if shmgerr().eq.1 then
      code_err=shmgcod()
      ...
   else
      ...
endif
shmstat=clearsv()
```

Ces diffrents status sont valides chaque fois que le *client* a la main sur le bloc de communication.
Un autre exemple montre le traitement d'une erreur survenant sur le *serveur*:
Par exemple, une fonction `xxx` peut retourner deux codes d'erreur en cas de problmes `"no_space"` ou `"no_name"` . Ces situations sont grable:

```
shmstat=shmwait()
shmstat=shmput("COMMAND","@xxx")
shmstat=shmack()
shmstat=shmwack()
if (shmstat.ne.0) then
    if (shmstat.eq.1) then
        if leq(shmgcod(),"no_space") then
           ...
        endif
        if leq(shmgcod(),"no_name") then
           ...
        endif
    else
        ...
    endif
endif
```

4.6 Kill des smaphores

Si les smaphores sont tus (avec `ipckill` ou le bouton `kill` sur `ipcstat`), la fonction `connect()` permet un *client* de se reconnecter sur le *serveur* courant (une fois qu'il existe). La dtection de la perte des smaphores se fait par exemple au moment du `shmwait()` par une erreur *Inter*.

Chapter 5

Procdures de communication

Deux procdures de communication ont t cres pour lancer des commandes (et plus particulirerement des procdures) de manire transparente que ce soit en local ou sur un *serveur*. Ce sont les procdures nommes `@.prc` et `@@.prc`. Par exemple:

```
@dosomething i j k      ! en local
@@ @dosomething i j k   ! en remote (sans interprtation des arguments)
@@@ @dosomething i j k  ! en remote (avec interprtation des arguments)
```

Ces procdures de communication font plusieurs choses:

1. elles se chargent du traitement du status de retour
2. elles transmettent le `CTRL-C` au *serveur*
3. elles mettent jour les variables du bloc de donnes du *client* slectionne par le *serveur* (c'est une option, voir plus loin `pput.prc`)

Ces procdures ont t dveloppe pour un fonctionnement d'*Inters clients—serveurs* qui doivent suivre les rgles suivantes:

1. le *client* et le *serveur* possdent des variables de bloc de mmes noms
2. seules ces variables peuvent tre utilises comme arguments non–interprts ou peuvent tre mises jour sur le *client* par le *serveur*.
3. les procdures qui veulent mettre jour des variables sur le *client* doivent le faire selon un schma prcis (voir plus loin `pput.prc`).

5.1 La procdure `@.prc`

Cette fonction n'interprte pas les arguments. Les arguments sont placs dans le bloc de communication avec la fonction:

```
shmstat=shmput("COMMAND",rm?("{1} {2} {3} {4} {5} {6} {7} {8} {9}"))
```

Donc si on envoie:

```
INTER-CLIENT > i=734
INTER-CLIENT > gdrive="hcposta4"
INTER-CLIENT > @@ show i gdrive
```

on a sur le *serveur*:

```
------------ Inter (mode serveur) en attente ------------
Reu : >sh i gdrive<
  I(001) =  45.000000
  GDRIVE(001) = "mx11"
```

Ce sont bien le `I` et le `GDRIVE` du *serveur*.

5.2 La procdure `@@.prc`

Cette fonction interprte les arguments. Les arguments sont placs dans le bloc de communication avec la fonction (remarque: la commande "`write /quo`" met des guillemets autour des chanes de caractres):

```
write /quo /key=cmd  {2} {3} {4} {5} {6} {7} {8} {9}
shmstat=shmput("COMMAND",lcat("{1} ",cmd))
```

Donc si on envoie:

```
INTER-CLIENT > i=734
INTER-CLIENT > gdrive="hcposta4"
INTER-CLIENT > @@@ show i gdrive
```

on a sur le *serveur*:

```
------------ Inter (mode serveur) en attente ------------
Reu : >show  734.00000    "hcposta4" <
  734.00000 =  734.00000
  "hcposta4" = "hcposta4"
```

On a effectivement pass la valeur.

5.3 Restrictions d'utilisation

Tout d'abord, il faut prciser que ces procdures sont prvue surtout pour l'utilisation simplifie de la communication lorsque l'on travaille sur un *Inter—client* en interactif. Ce sont des amliorations de la fonction `shmcmdw()`. Elles ne permettent pas de passer au *serveur* des paramtres par le bloc de communication. Elles grent le status de manire standard, et ne possdent pas de *timeout*.

De plus, du au fonctionnement de l'interprtation, certains type d'arguments sont interdit dans certains cas et les chanes de caractres sont passer selon des syntaxes prcises. D'un point de vue gnrale, il est de loin prfrable (contrairement l'exemple ci–dessus) de lancer uniquement des procdures par ce moyen.

5.3.1 Restrictions pour `@.prc`

Les chanes de caractres doivent tre mise entre deux paires de guillemets. Exemple:

```
INTER-CLIENT > @@ gdrive=""mx11""
INTER-CLIENT > @@ @dosomething ""mon commentaire""
```

5.3.2 Restrictions pour `@@.prc`

Les qualificateur sont interdits, par contre les paramtres peuvent tre mis normalement entre guillemets. Exemple:

```
INTER-CLIENT > @@@ catch /fit
Ce qualificateur est inconnu : "/FIT"
INTER-CLIENT > @@@ write "mon commentaire"
```

5.4 Le retour des paramtres

Comme nous l'avons vu, un bloc de communication permet d'assigner des variables *Inter* avec la fonction `fetch()`. Ainsi une procdure qui dsire retourner des rsultat au *client* doit simplement remplir le bloc de communication (avec la fonction `shmadd()`). C'est pour cette raison qu'il beaucoup plus efficace d'avoir les noms de variable communes entre un *serveur* et un *client*.

La fonction `fetch()` est excute dans les procdures `@.prc` et `@@.prc` lorsque le procdure *remote* se termine et s'il n'y a pas d'erreur.

La procdure `pput.prc` permet le passage simplifi de paramtres vers le *client*, et seulement si la procdure travaille en mode *serveur*. En fin de procdure, il suffit de taper par exemple:

```
@pput te.m2z te.m2x te.m2y te.m2tx te.m2ty
```

Remarque: cette procdure n'accpte que 9 paramtres.

5.5 Passage de variables *client—serveur*

En fait c'est trs simple. Pour assigner une variable sur un *serveur* on tape par exemple (attention la double paires de guillemets):

```
INTER-CLIENT > @@ i=1234 gdrive=""hposta4""
```

Attention, il n'y a pas d'interprtation possible.

Pour le passage de variables (de mme nom naturellement) du *serveur* au *client*, on tape par exemple:

```
INTER-CLIENT > @@ @pput i gdrive
```

Attention, dans ce cas, comme "`@pput`" est le premier argument de la procdure `@.prc`, on ne peut rcuprer de cette manire que huit arguments par appel.

Chapter 6

Exemples

Ce chapitre dcrits des exemples de procdures types.

6.1 Interruption d'un *serveur*

Si un *client* dsire interrompre un *serveur*, il peut lui envoyer un signal ($CTRL - C == 2$) avec la fonction `signal()`. Par exemple, stopper deux *serveurs* (`spectro` et `ccd`) se fait de la manire suivante:

```
shmstat=select("spectro")
shmstat=signal(2)
shmstat=select("ccd")
shmstat=signal(2)
```

6.2 Interruption d'un *client*

Le *serveur* doit simplement lui signifier l'erreur. Par exemple:

```
if (test.eq.bad) erreur /set "test is bad"
```

Remarque: ce type de gestion va bientt tre amliore afin de permettre d'envoyer un code d'erreur plus explicite.

6.3 Gestion des *timeout*

On peut donner des *timeout* pour l'attente de rservation (`shmwait()`) et pour l'attente de fin de d'excution sur *serveur* (`shmwack()`). Dans le cas d'un *timeout* le status de la fonction vaut `114`. Pour le traitement du *timeout* du `shmwack()` il est prfrable de librer (sous condition) le *serveur* en testant s'il est vraiment toujours notre *serveur* (s'il a *crash* il ne l'est plus!). Par exemple:

```
shmstat=shmwait()
if shmstat.eq.114 then
   erreur /set "timeout"
endif
...
shmstat=shmwack()
if shmstat.eq.114 then
   if srvwait() shmstat=shmfree()
   erreur /set "timeout"
endif
```

6.4 Propagation du `CTRL--C`

Lorsque le *client* attend la fin d'une procdure *remote*, il peut dcider de propager le `CTRL--C` si celui–ci survient. La mthode est la suivante:

1. enregistrer un *handler* de signal avant l'attente
2. l'enlever aprs la fin de la procdure *remote*

Cela donne:

```
action ctrlc call sendsig
shmstat=shmack()
shmstat=shmwack()
action ctrlc nocall
```

La subroutine `sensig.prc` est accessible avec les procdures standards. Elle contient:

```
subroutine sendsig
  local stat=signal(2)
  return
endproc
```

6.5 Gestion du status de retour d'une fonction

Comme dcrit plus haut, dans un environnement multi—*clients* multi—*serveurs* il faut absolument grer la valeur du status pour, par exemple, qu'un *client* ne libre pas de manire draisonnable un *serveur* qui aurait gnrer une erreur.
La procdure `shm_test_status.prc` notamment utilise dans `@.prc` et `@@.prc` fait un traitement standard. Prire de la consulter pour plus de dtail. Son appel est le suivant:

```
shmstat=shmwack()
call shm_test_status shmstat
if shmstat.eq.0 shmstat=fetch()
```

6.6 Communication asynchrone

Ce mode de fonctionnement est normalement prohiber, car l'accs concurrent d'une zone commune deux process doit justement se faire de manire synchrone. On toutefois travailler dans ce mode en respectant quelques rgles. Par exemple:

1. L'criture se fait d'un ct et la lecture de l'autre
2. Il n'y a qu'un *client* la fois qui doit crire

On peut par exemple contrler un *serveur* en crivant un mot cl dans le bloc de communication. Celui–ci peut tre lu dans une boucle infinie...