

UNIX COMMUNICATION INTERNE ENTRE PROCESSUS

1. GÉNÉRALITÉS

- Communications internes:
entre processus sur une même machine:
 - exec()
 - fichiers
 - moyens de communications Unix
 - signaux
 - tubes
 - IPCs:
 - file de messages
 - mémoire partagée
 - sémaphores

- Communications externes:
 - d'autres moyens pour la communication entre processus tournant sur des machines différentes
 - les sockets

2. INTERFACE PROGRAMME C

- Commande Unix sous Shell et main()

% prog a b c

main(**argc**, **argv**, **envp**)

- Paramètres d'appel

– à l'exécution de main(argc, argv, envp) on récupère

- **argc**: le nombre de paramètres du 2ème paramètre
- **argv**: adresse d'un tableau d'adresses de chaînes de caractères

sous Unix la première chaîne est le nom du programme (“prog”)

- **envp**: adresse d'un tableau d'adresses de chaînes de caractères donnant les valeurs des variables Shell

- **Exemple d'appel:**

- % prog par1 par2 par3 par4

avec variables d'environnement: TERM = vt100

HOME=/usr/moi

- La commande est composée de 5 paramètres
- main(**argc**, **argv**, **envp**)
- après exécution
 - argc= 5
 - argv= adresse tableau contenant “prog” “par1” “par2” “par3” “par4”
 - envp= adresse tableau contenant “TERM=vt100”
”HOME=/usr/moi”

- **Interface**

```
main(argc, argv, envp)
```

```
int argc;
```

```
char **argv, **envp;
```

argc: nombre d'éléments du tableau d'adresses pointé par argv

argv: pointe sur le tableau qui contient les adresses des paramètres d'appel

envp: pointe sur le tableau qui contient les adresses des variables d'environnement

- **Exemple de programme C**

- affichage des paramètres d'appel du programme lui-même

```
main(argc, argv, envp)
```

```
int argc;
```

```
char **argv, **envp;
```

```
{ int k;
```

```
for(k=0;k<argc;++k)
```

```
{ printf("paramètres %d: %\n",k+1,argv[k]);
```

```
    argv[k]; }
```

```
for(k=0;;++k)
```

```
{ { if(envp[k][0])
```

```
    printf("environnement %d: %\n",k+1,envp[k]);
```

```
else break;}} exit(0);}
```

- La variable d'environnement Unix: **environ**
 - la variable 'environ' sous Unix est un pointeur sur un tableau de pointeurs vers les chaînes de caractères des variables d'environnement

exemple:

```
extern char **environ;
```

```
main()
```

```
{ int k;
```

```
for(k=0;;++k)
```

```
{ if(environ[k][0])
```

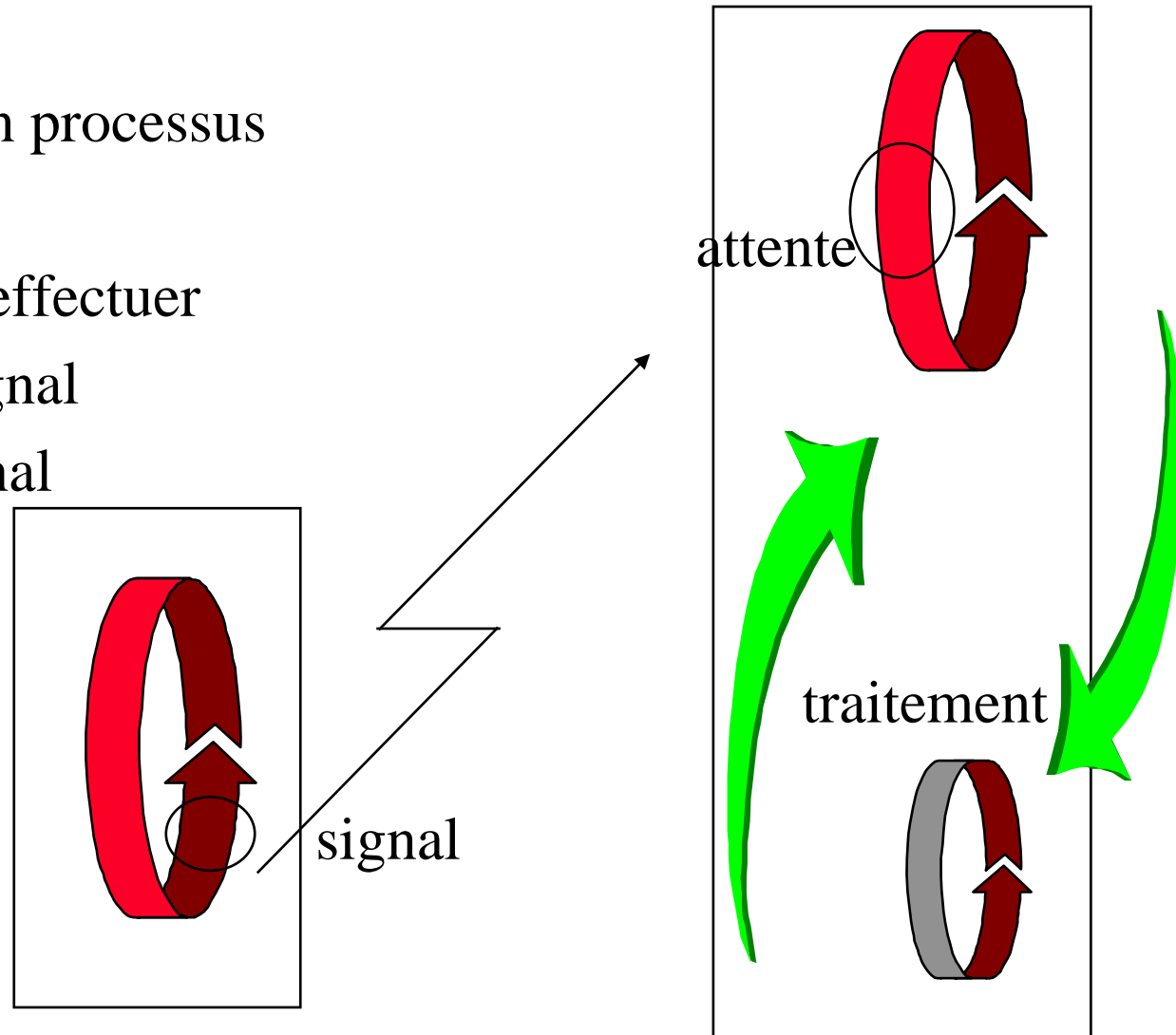
```
    printf("environnement %d: %\n",k+1,environ[k]);
```

```
    else break;} exit(0);}
```


3. LES SIGNAUX

- Plusieurs processus se partagent:
 - la mémoire
 - le processeur
- Interruption logicielle envoyée à un processus
 - signal ‘pendant’ si non pris en compte
 - signal ‘délivré’ si pris en compte
- Identification par un entier
- Traitement à effectuer
 - traitement par défaut
 - handler: fonction sans retour de valeur
- Réactiver le signal après utilisation

- Qu'est-ce qu'un signal?
 - interruption d'un processus
 - fonctions utiles
 - traitement à effectuer
 - attente du signal
 - envoi du signal



- Quelques utilisations d'un signal
 - cas 1: demande d'**E/S occupée**
 - processus endormi jusqu'à E/S libérée
 - Unix envoie un **signal** à tous les processus prêts
 - cas 2: **déconnexion** d'un terminal
 - tous les processus en cours reçoivent un **signal SIGHUP** et s'arrêtent
 - cas 3: fin d'un processus fils par **exit()**
 - un **signal SIGHLD** est envoyé au père qui est en attente **wait()**
 - cas 4: entre **processus utilisateurs**
 - un **signal SIGUSR** est envoyé par un processus à un autre processus

- `fork()` et `exec()`
 - Après `fork()`
 - fils hérite des traitements ou handler
 - Après `exec()`
 - traitements perdus
 - signaux ignorés restent ignorés
 - les autres reprennent leur traitement par défaut

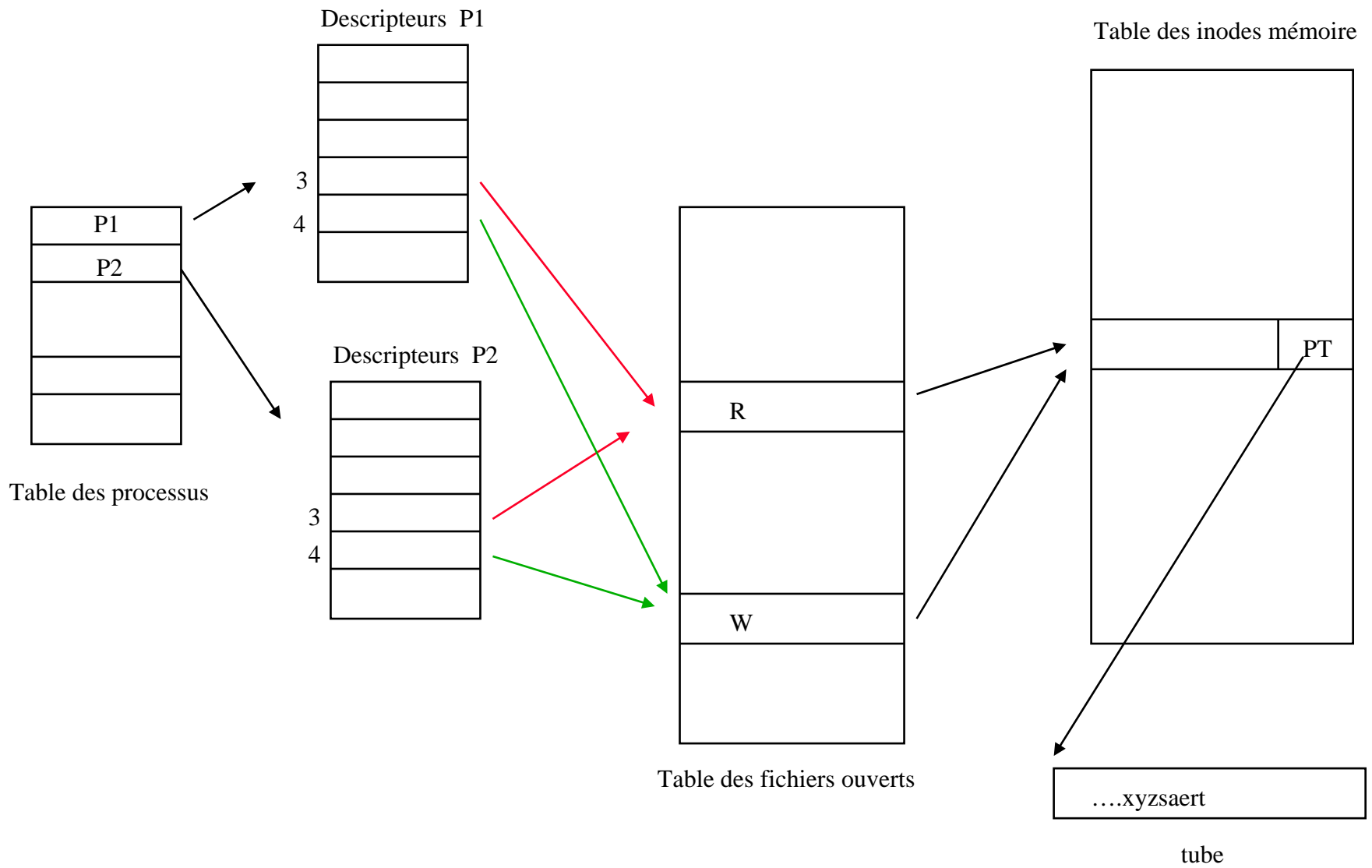
- Les signaux
 - chaque système possède un fichier de référence: `signal.h`
 - certains signaux génèrent un core dump
 - un signal -> un traitement par défaut `SIG_DFL`

Détail dans un cours prochain....

4. LES TUBES

- **LES TUBES OU PIPE**

- Types:
 - tube anonyme
 - tube nommé
- Moyen de communication entre deux processus s'exécutant sur une même machine
- Fichiers particuliers (SGF)
- Gérés par le noyau
- File de données en mémoire (FIFO)
- Lectures destructrices



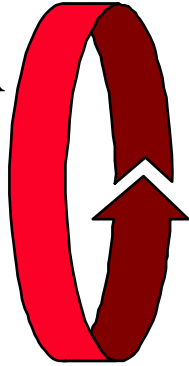
- **TUBE ANONYME**

- Structure sans nom
- Communication entre deux processus
- Deux descripteurs: lecture et écriture
- Deux pointeurs automatiques: lecture et écriture
 - pointeur de lecture sur le 1er caractère non lu
 - pointeur d'écriture sur le 1er emplacement vide
- Processus de même filiation

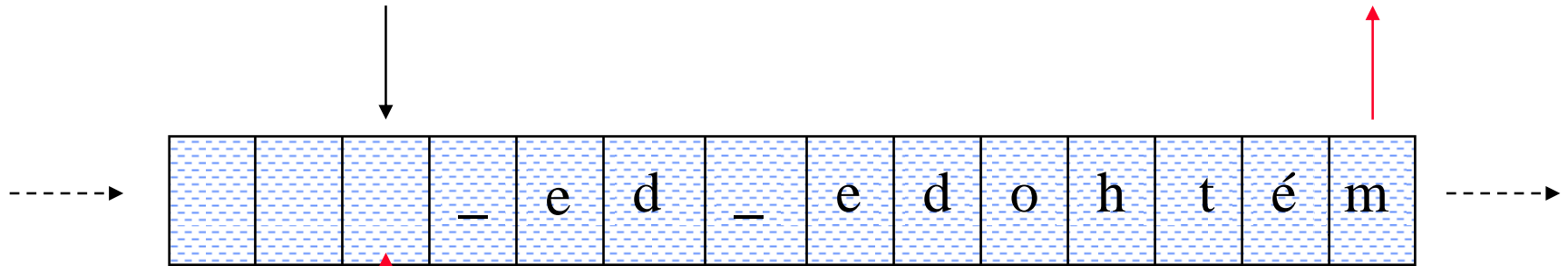
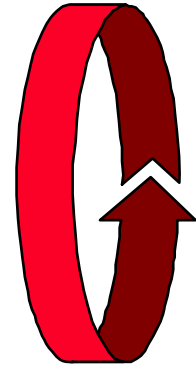
– Principe:

- `pipe()`: création du tube par le père
- `fork()`: création du processus fils
- héritage de l'ouverture du tube (fichier)
- `exec()`: passage des descripteurs en paramètres

Le processus A
dépose



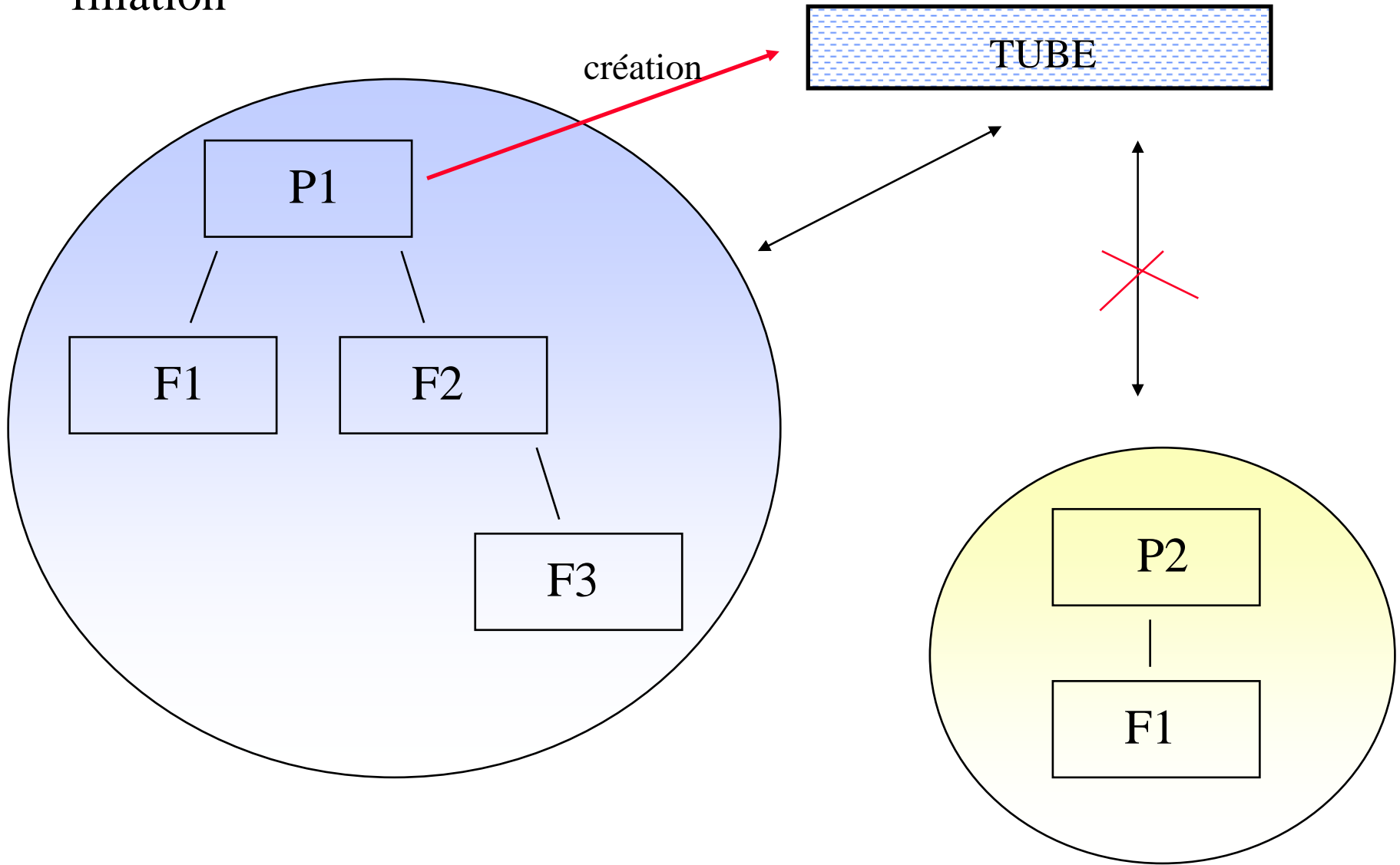
Le processus B
lit



Le processus B
dépose

Le processus A
lit

filiation

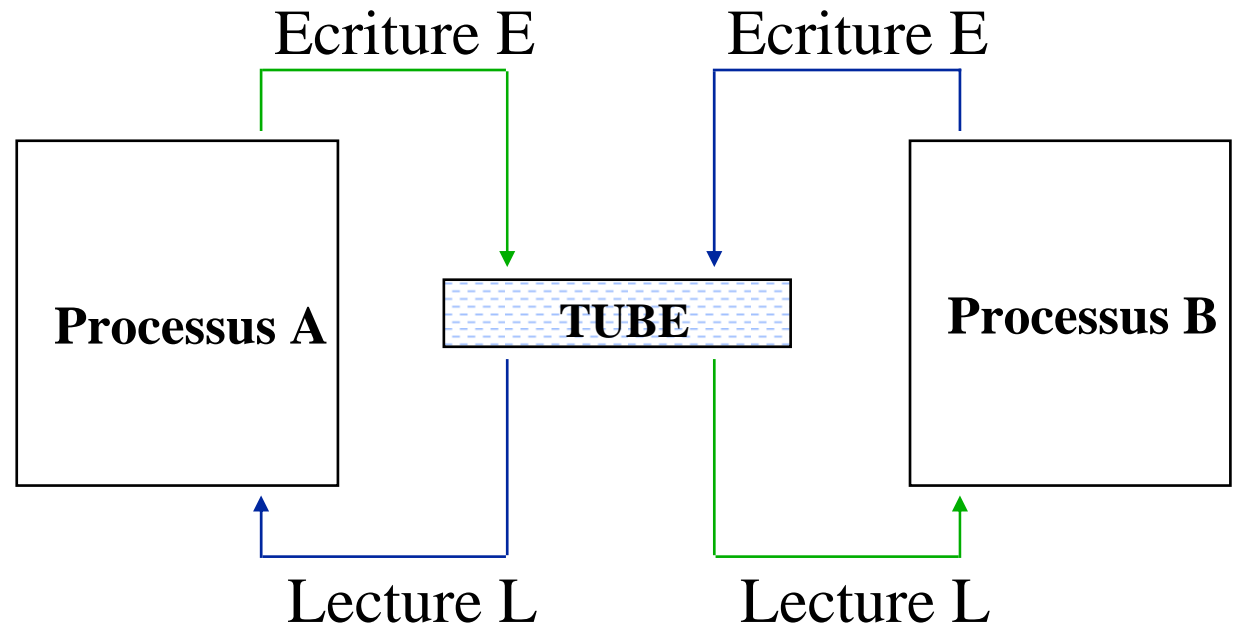


– CARACTÉRISTIQUES

- L'ouverture du tube se fait par `pipe()` et non `open()`
- `lseek()` interdit
- taille limitée / taille du tampon alloué
- descripteurs de lecture fermés et tentative d'écriture
 - signal SIGPIPE
 - arrêt du processus
- lectures multiples:
 - le 1er qui lit récupère les données

- tube vide et
 - lecture
 - code erreur = 0
 - processus bloqué jusqu'à dépôt de données
- tube non vide et
 - nombre de données à lire > données existantes
 - code erreur = 0
 - processus bloqué jusqu'à dépôt de données
- tube plein et
 - écriture
 - code erreur = 0
 - processus bloqué jusqu'à lecture de données

Synchronisation

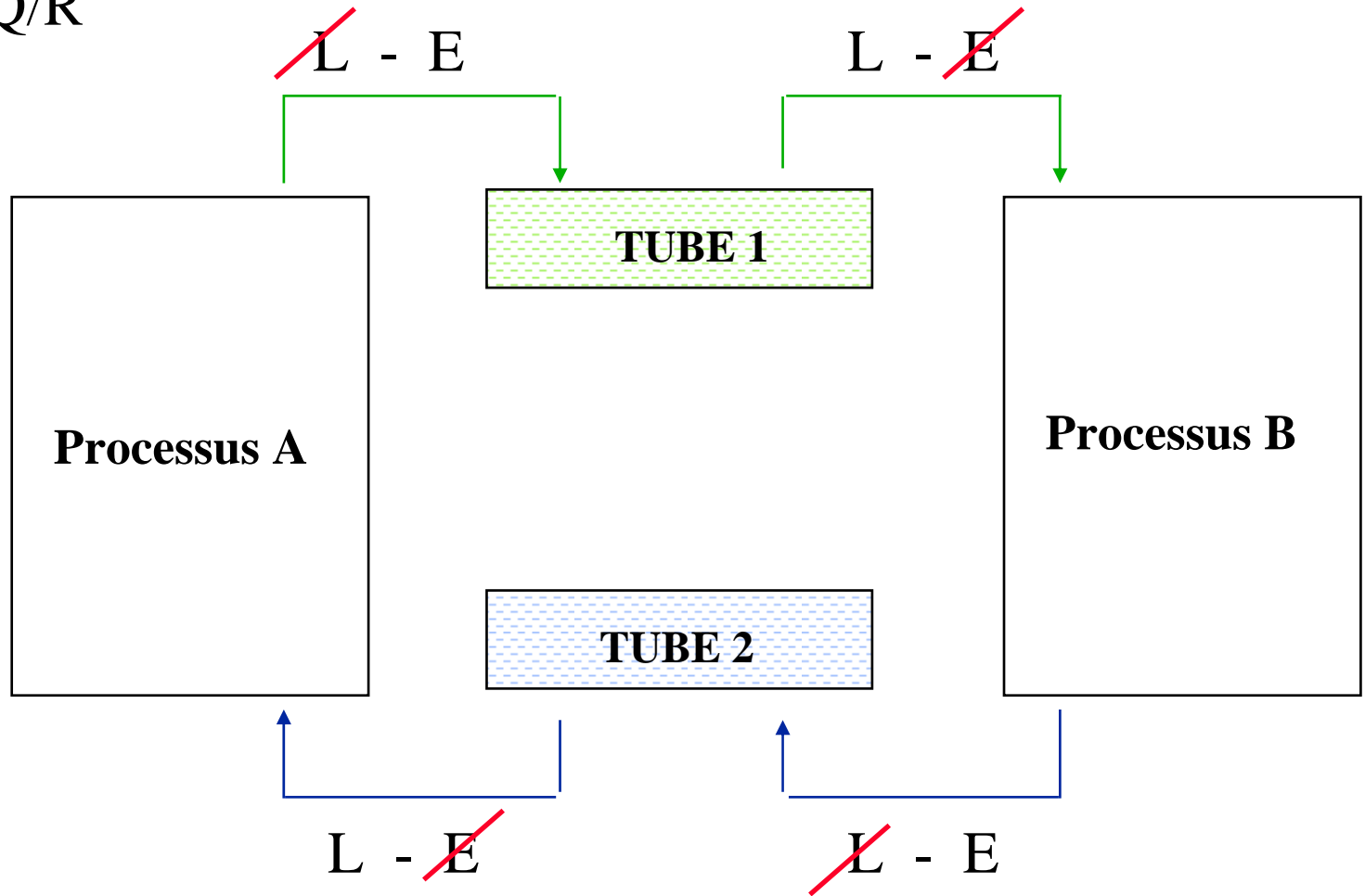


- Soit: PA transmet à PB ou PB transmet à PA

SI

- PA dépose et PA lit => PB bloqué
- PA et PB déposent et PB lit => risque que PB lise sa propre donnée

Modèle Q/R



Fermeture des descripteurs inutiles

– FONCTION TUBE

pipe (tab)

- crée un tube
- retourne les n° des deux descripteurs dans un tableau 'tab'
 - tab[0]: n° du descripteur de lecture: dl
 - tab[1]: n° du descripteur d'écriture: de
- remplit la fonction d'ouverture d'un fichier classique

– FONCTIONS SGF

read (dl, buf, nb)

- dl: n° descripteur lecture
- buf : zone de réception des octets
- nb : nombre d'octets à lire

write (de, buf, nb)

- de: n° du descripteur écriture
- buf: zone d'émission des octets
- nb: nombre d'octets à écrire

close (dl) et close (de)

- fermeture des descripteurs
- fermeture des descripteurs automatique si processus terminé
- suppression du tube si fermeture de tous les descripteurs

– EXAMPLE TUBE ANONYME


```
#include <stdio.h>

int pip[2];                                /* descripteur de pipe */
char buf [6];

{ main()
  pipe(pip);                               /* creation pipe */
  switch (fork())
    { case -1: perror("pipe"); exit(1);
      case 0:  fils();
      default: pere();}
  pere(){ write (pip[1], "hello", 5); exit(0);}          /* écriture pipe */
  fils() { read (pip[0], buf, 5); exit(0);}              /* lecture pipe */
}
```

Utilisation de dup() pour rediriger la sortie standard

descripteurs du processus A



0	STDIN
1	STDOUT
2	STDERR
3	tube input
4	tube output
5	
6	

DUP()

1) on crée un tube:

- deux descripteurs: 3 et 4 qui pointent sur la table des fichiers: ici tube

2) on ferme le descripteur 1

- l'entrée 1 est libre

3) on duplique le descripteur 4 avec retour = dup (4)

- le descripteur 4 est recopié dans le descripteur 1 (dup prend la première entrée libre)

- valeur de retour: le nouveau descripteur ici le 1

4) on ferme les descripteurs 3 et 4 qui ne servent plus

5) tout envoi vers le descripteur 1 concernera le tube

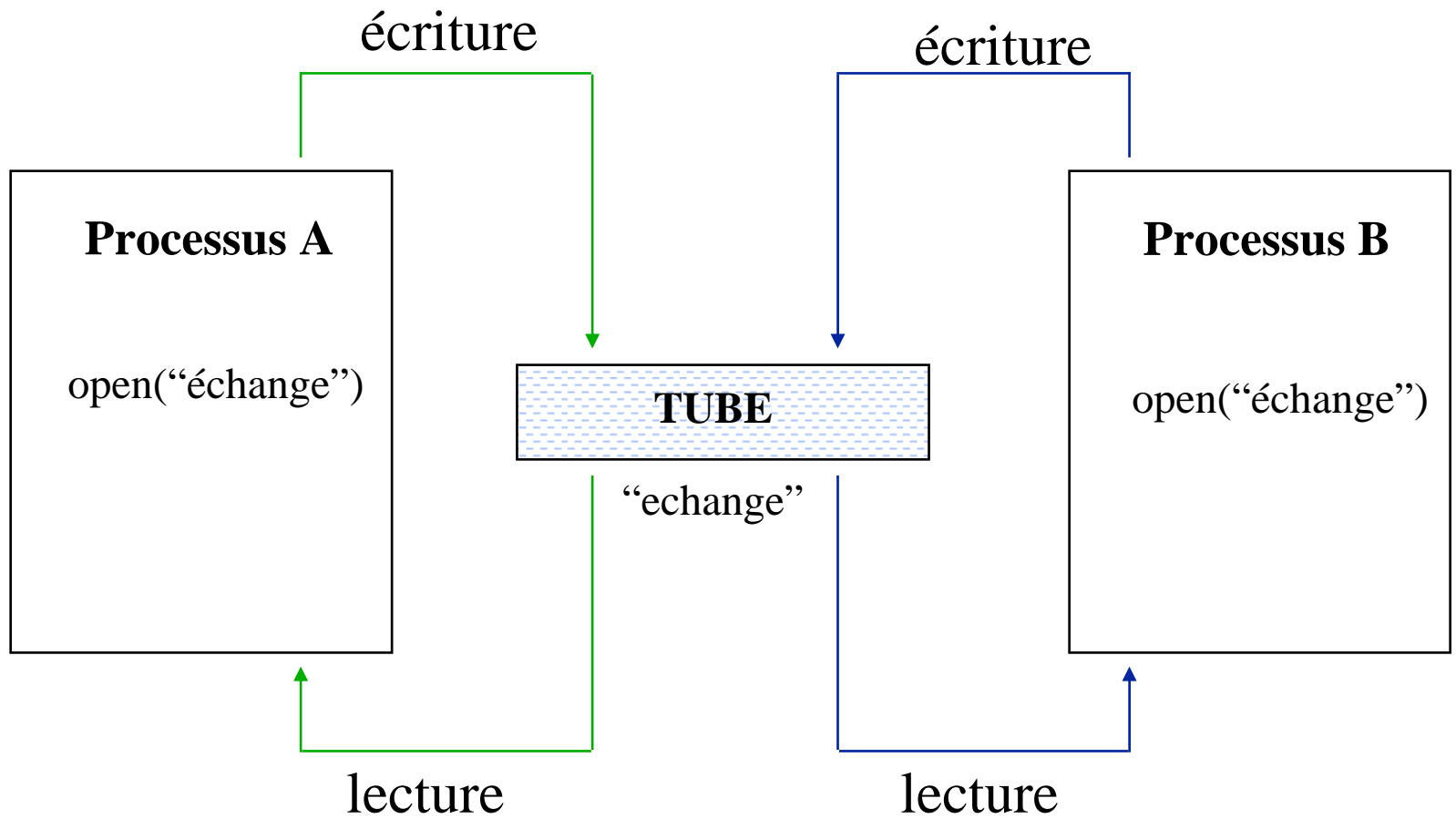
- **TUBE NOMMÉ**

- Caractéristiques communes aux tubes anonymes:

- communication entre processus s'exécutant sur une même machine
 - fichier particulier
 - file de messages en mémoire
 - pointeurs gérés automatiquement: `lseek()` inutilisable

– Différences:

- fichier portant un nom
- filiation non nécessaire
- création par la fonction SGF `mknod()`
- ouverture par `open()`
- un seul descripteur par ouverture de tube
- fichier persistant



– FONCTION CRÉATION SGF

mknod (nom_du_fichier, accès+S_IFIFO)

mkfifo (nom_fichier, accès)

- utilisation de la fonction de création d'un i-node particulier
- spécifier un nom de fichier
- donner les droits d'accès
- création d'un fichier fonctionnant en mode FIFO

– FONCTION OUVERTURE SGF

desc = open(nom_du_fichier, mode)

- ouverture en lecture si mode = O_RDONLY
- ouverture en écriture si mode = O_WRONLY
- ouverture en maj si mode = O_RDWR
- ouverture bloquante / non bloquante mode = O_NDELAY

mode	O_RDONLY	O_WRONLY
O_NDELAY	ouverture sans attente	ouverture avec retour code erreur si aucune ouverture en lecture
sinon	processus bloqué jusqu'à ouverture en écriture par un autre processus	processus bloqué jusqu'à ouverture en lecture par un autre processus

– FONCTIONS LECTURE / ÉCRITURE SGF

read (desc, buf, nb) lecture dans le tube

- si O_NDELAY à l'ouverture
 - retour code erreur si lecture dans un tube vide
- sinon
 - processus bloqué si tube vide, attente tube suffisamment de données à lire

write (desc, buf, nb) écriture dans le tube

- si O_NDELAY à l'ouverture
 - retour code erreur si tube plein
- sinon
 - processus bloqué si tube plein, attente tube suffisamment vide pour écrire

– FONCTIONS FERMETURE / DESTRUCTION SGF

close (desc) fermeture du fichier

- fermeture du descripteur du fichier ouvert dans chacun des processus

unlink (nom_du_fichier) destruction du fichier

ou

rm nom_du_fichier commande shell de destruction du fichier

- EXEMPLE TUBE NOMMÉ

```
/* Processus ecrivain */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
main()
{mode_t mode;
int tub;
mode = S_IRUST | S_IWUSR;
mkfifo ("fictub",mode)           /* création fichier FIFO */
tub = open("fictub",O_WRONLY)    /* ouverture fichier */
write (tub,"0123456789",10);    /* écriture dans fichier */
close (tub);
exit(0);}
```

```

/* Processus lecteur */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
main()
{int tub;
char buf[11];

tub = open("fictub",O_RDONLY)      /* ouverture fichier */
read (tub,buf,10);                 /* lecture du fichier */
buf[11]=0;
printf("J'ai lu %s\n", buf);
close (tub);
exit(0); }

```

5. LES IPCs

- IPC: Inter Processus Communication (SystemV)
 - Externe au SGF
 - Identification et manipulation par une clé
 - Interface commun aux IPC:
 - `/usr/include/sys/ipc.h`
 - `/usr/include/sys/types.h`
 - Les IPCs:
 - Files de messages
 - Mémoire partagée
 - Sémaphores
- Commandes: `ipcs` et `ipcrm` pour voir ou supprimer un IPC

– **Fichier types.h**

- définitions des types /machine

– **Fichier ipc.h**

```
type def long  mtyp_t;    /* ipc type message */
```

```
struct ipc_perm{  
    uid_t      uid        /* identification du propriétaire */  
    gid_t      gid        /* identification du groupe */  
    uid_t      cuid       /* identification du créateur */  
    gid_t      ugid       /* identification du groupe à la création */  
    mode_t     mode       /* mode d'accès */  
    ushort_t   seq  
    key_t      key;      /* clé */ }  

```


mode:

00400	lecture par utilisateur
00200	écriture par utilisateur
00040	lecture par groupe
00020	écriture par groupe
00004	lecture par les autres
00002	écriture par les autres

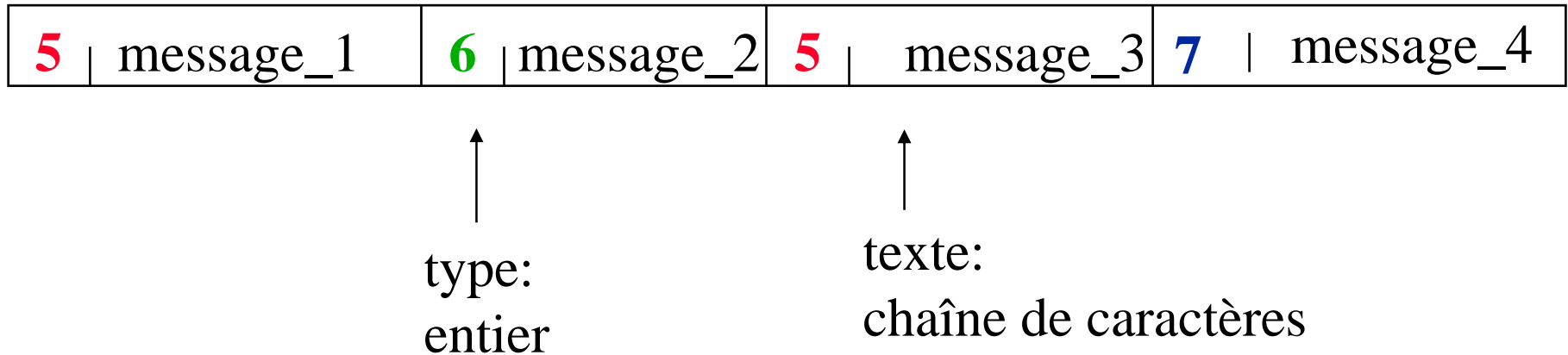
Variables symboliques:

IPC_ALLOC	0100000	/* indicateur objet alloué */
IPC_CREAT	0001000	/* création si clé n'existe pas */
IPC_EXCL	0002000	/* échec si clé existe */
IPC_NOWAIT	0004000	/* erreur en cas d'attente */

6. LES FILES DE MESSAGES

- FILE DE MESSAGES ou MESSAGE QUEUE ou MSQ
 - File de messages : mode FIFO particulier
 - **Identificateur** MSQ: entier fourni par le système à la création
 - msqid (similaire a un n° de descripteur)
 - Ensemble de **messages typés**
 - déposés par un ou plusieurs processus
 - relus par un ou plusieurs processus
 - aucune filiation exigée
 - lecture destructrice
 - structure associée: fichier msg.h

– STRUCTURE MSQ



Ici 3 types de messages: A, B et C

– MÉCANISME

- création MSQ avec une **clé**
 - récupération de l'identificateur msqid
 - autorisation d'accès par le créateur
- lecture ou écriture
 - récupération msqid en fournissant la clé
 - fournir **msqid**
 - si lecture
choisir le **type** du message à lire
 - si écriture
fournir le **type** du message
le **texte** du message

– **msgget()** **Création d'une MSQ**
Récupération de msqid

- **Interface:**

```
#include <sys/msg.h>  
int msgget (cle, msgflg);  
key_t cle;  
int msgflg;
```

- **Retour:**

si ok retour **msqid** de la MSQ
sinon -1 erreur

- **Description:**

Cas **création MSQ**

cle = IPC_PRIVATE création d'une MSQ sans cle
cle <> IPC_PRIVATE création d'une MSQ avec clé
 si msgflg = IPC_CREAT et si cle n'existe pas déjà
 . msq créée
 si msgflg = IPC_EXCL et si cle existe
 . retour erreur
 si msgflg <> IPC_EXCL et si cle existe
 . retour msqid

Cas récupération msqid

`msgget(cle, 0)` alors retour msqid

Droits d'accès indiqués dans `msgflg`

La MSQ créée, la structure associée est mise à jour

– msgctl()

Contrôle structure associée à MSQ

Destruction d'une MSQ

- **Interface:**

```
#include <sys/msg.h>
```

```
int msgctl (msqid, op, buf)
```

```
int msqid, int op;
```

```
struct msqid_ds *buf;
```

- **Retour:**

si ok 0

sinon -1 erreur

- **Description:**

Les opérations sur la structure de la MSQ sont:

si `op = IPC_STAT`

. lecture de la structure dans buf

si `op = IPC_SET`

. modif de la structure à partir de buf

si `op = IPC_RMID`

. destruction MSQ si vide et accès autorisé

– msgsnd() Ecriture dans MSQ

- **Interface:**

```
#include <sys/msg.h>
```

```
int msgsnd (msqid, msgp, msgsz, msgflg)
```

```
int msqid, int msgflg;
```

```
const void *msgp;
```

```
size_t msgsz;
```

- **Retour:**

```
si ok     0
```

```
sinon    -1 erreur
```

- **Description:**

Le message de la MSQ msqid est préparé dans la structure pointée par msgp est définie dans msg.h:

```
struct msgbuf {  
    mtyp_t    mtype          /* type du message */  
    char      mtext [ ] }    /* texte du message */  
  
mtext        texte de msgsz octets  
mtype        entier positif
```

`msgflg` en cas d'erreur

`si = IPC_NOWAIT`

- . message perdu et retour code erreur

`sinon`

- . processus bloqué jusqu'à place dans la MSQ ou
- . MSQ détruite ou
- . réception d'un signal

- **Interface:**

```
#include <sys/msg.h>
```

```
int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
```

```
int msqid, int msgflg;
```

```
const void *msgp;
```

```
size_t msgsz;
```

```
long msgtyp;
```

- **Retour:**

si ok longueur du message lu en octets

sinon -1 erreur

- **Description:**

Le message de la MSQ msqid est lu dans la structure pointée par msgp est définie dans msg.h:

```
struct msgbuf {  
    mtyp_t  mtype          /* type du message */  
    char    mtext [] }    /* texte du message */
```

`mtext` texte du message lu

`mtype` type associé au message lors de l'écriture

N.B. Le message lu est détruit

msgsz

longueur maxi du message à lire

si $\text{msgsz} < \text{long du message}$ et

si $\text{msgflg} = \text{MSG_NOERROR}$

. le message est tronqué à la long. msgsz

sinon

. le message n'est pas lu et code erreur

msgtyp

type du message selon

si $\text{msgtyp} = 0$ lecture du 1er message de la file

si $\text{msgtyp} > 0$ lecture du 1er message non lu
de type = msgtyp

si $\text{msgtyp} < 0$ lecture du premier message non
lu de type $< |\text{msgtyp}|$

`msgflg`

en cas d'erreur

si `msgflg = IPC_NOWAIT` et pas de message
de type `msgtyp` alors

- . retour code erreur

`sinon` processus bloqué jusqu'à :

- . arrivée d'un message de type `msgtyp` ou

- . MSQ détruite ou

- . réception d'un signal

- EXEMPLE MSQ

```
/* création d'une MSQ et envoi message*/
```

```
#include <sys/types.h>
```

```
#include <ipc.h>
```

```
#include <msgq.h>
```

```
#define CLE 17
```

```
struct msgbuf msgp;
```

```
char *msg="ceci est un message";
```

```
main()
```

```
{ int msqid; /* identificateur msq */
```

```
msqid = msgget((key_t)CLE,0750+IPC_CREAT); /* creation msq */
```

```
msgp.mtype=12; /* le type */
```

```
strcpy(msgp.mtext,msg); /* le message */
```

```
msgsnd(msqid, &msgp, strlen(msg), IPC_NOWAIT) /* envoi message */
```

```
exit(0); }
```

```

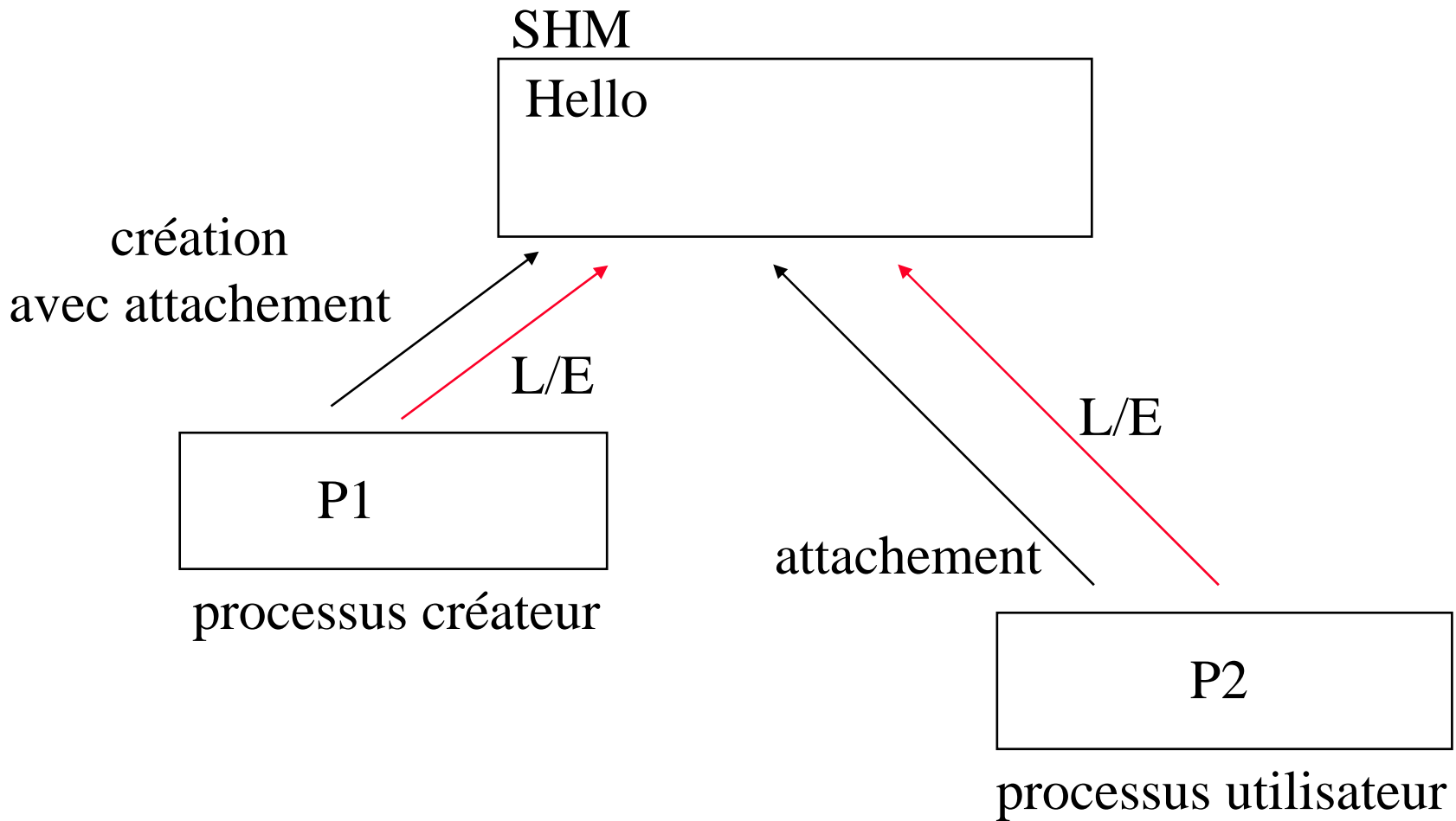
/* lecture message et destruction msq*/
#include <sys/types.h>
#include <ipc.h>
#include <msg.h>
#define CLE 17
struct msgbuf msgp;

main()
{ int msqid; int x;
msqid = msgget((key_t)CLE, 0); /* récup msqid */
x = msgrcv (msqid, &msgp, 19, (long)12, IPC_NOWAIT) /* lecture type 12*/
msgp.text[x] = 0;
printf ("message lu %s÷n",msgp.mtext);
msgctl(msqid(IPC_RMID, NULL);
exit(0); }

```

7. LA MÉMOIRE PARTAGÉE

- MÉMOIRE PARTAGÉE ou SHARED MEMORY ou SHM
 - Zone **mémoire commune** à plusieurs processus
 - **Identificateur SHM**: entier fourni par le système à la création
 - shmid
 - **Attachement** de cette zone par les processus utilisateurs
 - Données non typées
 - aucune filiation exigée
 - lecture non destructrice: zone mémoire
 - structure associée: fichier shm.h



– MÉCANISME:

- création SHM avec une **clé**
 - récupération de l'identificateur **shmid**
 - autorisation d'accès par le créateur
- attachement SHM par un processus
 - fournir **shmid**
 - récupération **pointeur** début zone SHM
- lecture ou écriture
 - accès mémoire
- détachement SHM par chaque processus
- libération SHM par le processus créateur

– **shmget()** **Création d'une SHM**
Récupération de shmid

- **Interface:**

```
#include <sys/shm.h>
```

```
int shmget (cle, sz, shmflg);
```

```
key_t cle;
```

```
size_t size;
```

```
int shmflg;
```

- **Retour:**

si ok retour **shmid** de la SHM

sinon -1 erreur

- **Description:**

Cas création SHM

création d'une SHM de taille `size+1`

`cle = IPC_PRIVATE` création d'une SHM sans cle

`cle <> IPC_PRIVATE` création d'une SHM avec clé

si `msgflg = IPC_CREAT` et si `cle` n'existe pas déjà
 . shm créée

si `msgflg = IPC_EXCL` et si `cle` existe
 . retour erreur

si `msgflg <> IPC_EXCL` et si `cle` existe
 . retour `shmid`

Cas récupération shmid

`shmget(cle, 0)` alors retour shmid

Droits d'accès indiqués dans msgflg

La SHM créée, la structure associée est mise à jour

– **shmctl()** **Contrôle structure associée à SHM**
Suppression d'une SHM

- **Interface:**

```
#include <sys/shm.h>
```

```
int shmctl (shmid, op, buf);
```

```
int shmid, int op;
```

```
struct shmid_ds *buf;
```

- **Retour:**

si ok 0

sinon -1 erreur

- **Description:**

Les opérations sur la structure de la SHM sont:

si `op = IPC_STAT`

. lecture de la structure dans buf

si `op = IPC_SET`

. modif de la structure à partir de buf

si `op = IPC_RMID`

. suppression SHM si n'est plus attachée à aucun processus

– shmat() Attachement SHM à un processus

- **Interface:**

```
#include <sys/shm.h>
```

```
char *shmat (shmid, shmadd, shmflg);
```

```
int shmid, int shmflg;
```

```
char *shmadd;
```

- **Retour:**

si ok retour adresse SHM

sinon -1 erreur

- **Description:**

SHM identifiée par `shmid`

La SHM est attachée au segment de données du processus
à l'adresse spécifiée par `shmadd`:

si `shmadd = 0`

. adresse attachement définie par le système

si `shmadd <> 0`

. adresse attachement = `shmadd`

Droits d'accès:

si `shmflg = SHM_RDONLY`

. lecture seule par le processus

– shmdt()

Détachement d'une SHM

- **Interface:**

```
#include <sys/shm.h>
```

```
int shmdt (shmadd);
```

```
char *shmadd;
```

- **Retour:**

si ok 0

sinon -1 erreur

- **Description**

La SHM dont l'adresse d'attachement est shmadd est détachée

- **Rappel**

SHM identifiée par `shmid`

La SHM est attachée au segment de données du processus
à l'adresse spécifiée par `shmadd`:

si `shmadd = 0`

. adresse attachement définie par le système

si `shmadd <> 0`

. adresse attachement = `shmadd`

EXEMPLE SHM

Le premier programme p1.c crée la shm de clé 217

et écrit un message « je suis le programme p1 »

Un autre programme p2.c relit le message déposé par p1.c et l'affiche.

Il détruit ensuite la shm.

```
/* programme p1.c */
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

```
#define CLE 217
```

```
char *shmat();
```

```
main()
{
    int      shmidx;          /* id de la shm */
    char     *mem             /* pointeur shm */

    /* création shm avec la clé CLE */
    if ((shmidx = shmget((key_t)CLE,1000,0750+IPC_CREAT) == -1)
        { perror (« shmget »);
          exit (1); }

    /* attachement */
    if ((mem = shmat(shmidx,NULL,0) == (char *)-1)
        { perror (« shmat »);
          exit(2); }
```



```
/* écriture sans shm */  
strcpy (mem, « je suis le programme p1 »);  
exit(0); }
```

```
/* programme p2.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define CLE 217
char *shmat();

main ()
{
/* récupération shmid */
if ((shmid = shmget ((key_t) CLE,0,0)) < 0)
    { perror (« shmget »);
      exit(1); }
}
```

```
/* attachement à la shm */  
if (mem = shmat(shmid,NULL,0)) == (char *) -1)  
    { perror (« shmat »);  
      exit(2); }
```

```
/* lecture de la shm */  
printf (« lu: %s\n », mem);
```

```
/* détachement du processus */  
if (shmdt(mem))  
    { perror (« shmdt »);  
      exit(3); }
```

```
/* destruction shm */  
shmctl (shmid, IPC_RMID, NULL) ;  
  
exit(0);  
}
```