

# **Synchronisation et Communication Inter-processus**

## Ce que cela comprend

- Les IPC :
  - Les sémaphores
  - Les messages
  - Les segments de mémoire partagée
- POSIX
  - Permet de faire des sémaphores  
(on abordera ce sujet dans le prochain cours)
- UNIX API
  - Les sémaphores
- Win32 API
  - Les messages, Sémaphores et segments de mémoire

## La problématique

- Jusqu'à présent on a:
  - Exécuté des processus dans des environnements protégés
  - Synchronisé ces processus à l'aide de signaux
- Mais on n'a pas protégé des ressources

## La problématique en détail

- Il ne suffit pas de **synchroniser l'exécution des processus**, mais aussi de **partager des données**, à la fois en mode **lecture** ou **écriture**.
- Parlons un peu des problèmes classiques posés par l'accès concurrentiel à des données
  - Si deux processus lisent le même ensemble de données, l'exécution est dite **CONSISTENTE**
  - Si un des processus est autorisé à modifier ces données, le second pourra renvoyer des résultats différents selon l'instant où il lira les données, avant ou après que le premier ait écrit ces données modifiées. (comme on a vu avec les signaux)

## Exemple

- deux processus "A" et "B" partagent un entier "i". Le processus A incrémente i de 1, **puis** le processus B affiche sa valeur.
- Dans un meta-langage nous pourrions l'exprimer ainsi :
  - **A { i->i+1 }**
  - **B { i->output }**

Shell\$

inter-processus

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 0

B

```
while(true)  
    printf(i);
```

On suppose que pendant  
un TS on exécute une boucle  
(ce qui est faux)

for(  
;

Shell\$

inter-processus

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 1

B

```
while(true)  
    printf(i);
```





Shell\$

inter-processus

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 2

B

```
while(true)  
    printf(i);
```



Shell\$  
2

inter-processus

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 2

B

```
while(true)  
    printf(i);
```



Shell\$  
2

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 3

B

```
while(true)  
    printf(i);
```



# B

```
Int i : 3
```

```
while(true)
    printf(i);
```



Shell\$  
233

A

```
for(i = 0; i<10000;i++)  
    ;
```

Int i : 3

B

```
while(true)  
    printf(i);
```



Shell\$  
233445810

inter-processus

233445810

!=

12345678

A

```
for(i = 0; i<10000;i++)  
    ;
```

```
while(true)  
    printf(i);
```



A A B A B B A B B A B A A B A B A A

```
Shell$  
233445810
```

**EN BREF**  
**CA MARCHE PAS**  
**!!!**  
**Tout dépend de**  
**l'ordonnanceur**

# Au fait, l'ordonnanceur...

auteur(s) © date - titre - version

Users/af/Documents/Cours/2012/module 2012/MTR/Cours/SCHEDULERV3.ppt



## Le risque d'INCONSISTENCE

- Il faut gérer de telles situations
- Essayez d'imaginer que ces données concernent votre compte bancaire, votre avion, et vous ne sous-estimerez plus jamais ce problème

## Petit rappel, une solution simple

- La fonction `waitpid(2)`, force un processus à attendre la réception d'un signal.
- Ceci permet en fait de régler une partie des conflits provoqués par la lecture et l'écriture des données :
  - dès que l'ensemble de données sur lequel un processus P1 doit travailler a été défini,
  - un processus P2 qui doit travailler sur le même ensemble ou sur un sous-ensemble doit attendre l'achèvement de P1 ou un signal avant de commencer lui-même à s'exécuter.
- **P2 va devoir attendre un temps indéterminé**

## La Solution , Les Sémaphores

## Définition : Sémaphores

- Feux rouge automobile
- Structure de données  $>$  ou  $=$  à 0
- Gestion des files d'attentes
- Contrairement aux apparences, de simples sémaphores sont très puissants et par conséquent entraînent une complexité grandissante.

## Des exemples pour bien comprendre

Semaphore

## Comment les écrit t'ont

- Plusieurs methodes sont possible
  - Les IPC assez difficile mais complet
  - La methode classique UNIX
  - La methode classique WINDOWS
  - La methode classique ...
  - La méthode POSIX
- Chaque OS à ses sémaphores
- et il y en a parfois de plusieurs types

## Avec les IPC

## Tout d'abord : Les clés de SysV

- Une clé IPC est un nombre qui identifie de manière unique une structure de contrôle IPC
- **key\_t ftok(const char \*pathname, int proj\_id);**
- L'unicité de la clé n'est pas assurée
- Il faut éviter les doublons
- Nous verrons bientôt à quoi cela peut servir!



## IPC

- La fonction qui crée un sémaphore se nomme **semget**
- **int semget(key\_t key, int nsems, int semflg);**
- **key** est une clé IPC,
- **nsems** le nombre de sémaphores que l'on désire créer
- et **semflg** est le contrôle d'accès représenté sur 12 bits,
  - les 3 premiers relatifs à la création
  - et les 9 autres aux accès en lecture et écriture de l'utilisateur ('user'), du groupe ('group') et des autres ('other')

# Notre premier Sémaphore

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(void)
{
    key_t key;
    int semid;
    key = ftok("/etc/fstab", getpid());
    /* create a semaphore set with only 1 semaphore: */
    semid = semget(key, 1, 0666 | IPC_CREAT);
    return 0;
}
```

## Avant d'aller plus loin

- Avant d'aller plus loin apprenons à gérer et à supprimer des sémaphores; ceci se réalise avec la primitive `semctl(2)`
- `int semctl(int semid, int semnum, int cmd, ...)`
- `union semun {`
  - `int val;` `/* value for SETVAL */`
  - `struct semid_ds *buf;` `/* buffer for IPC_STAT,IPC_SET*/`
  - `unsigned short *array;` `/* array for GETALL, SETALL */`
  - `/* Linux specific part: */`
  - `struct seminfo *__buf;` `/* buffer for IPC_INFO */``};`

## Initialiser / Détruire

**/\*create a semaphore set with only 1 semaphore\*/**

```
semid = semget(key, 1, 0666 | IPC_CREAT);
```

**/\* set value of semaphore number 0 to 1 \*/**

```
arg.val = 1;
```

```
semctl(semid, 0, SETVAL, arg);
```

...

**/\* set value of semaphore number 0 to 1 \*/**

```
arg.val = 1;
```

```
semctl(semid, 0, SETVAL, arg);
```

**/\* deallocate semaphore \*/**

```
semctl(semid, 0, IPC_RMID);
```

## Utilisation du sémaphore

- Le sémaphore peut maintenant être utilisé avec la fonction `semop(2)`
- **int semop(**  
    **int semid,**  
    **struct sembuf \*sops,**  
    **unsigned nsops**  
    **);**
- où **semid** identifie l'ensemble, **sops** est un tableau contenant les opérations à effectuer et **nsops** le nombre de ces opérations. Chaque opération est représentée par une structure **sembuf**
  - { `u_short sem_num;` `short sem_op;` `short sem_flg;` }

## exemple

- On définit 2 structures
  - struct sembuf **lock\_res** = {0, -1, 0};
  - struct sembuf **rel\_res** = {0, 1, 0};
- ```
/* Try to lock resource - sem #0 */  
if (semop(semid, &lock_res, 1) == -1) {  
    perror("semop:lock_res");  
}
```
- ```
/* Release resource */  
semop(semid, &rel_res, 1);
```

- **Prendre un sémaphore  $\Rightarrow \text{sem\_op} < 0$  :**
  - Si la valeur absolue du sémaphore est plus grande ou égale à celle de  $\text{sem\_op}$  l'opération est effectuée et  $\text{sem\_op}$  est ajoutée à la valeur du sémaphore (en fait elle est soustraite puisque  $\text{sem\_op}$  est négatif).
  - Si la valeur absolue de  $\text{sem\_op}$  est inférieure à la valeur du sémaphore le processus se met en sommeil jusqu'à ce que la quantité de ressources requise soit disponible.
- **Attente de la non dispo de sém.  $\Rightarrow \text{sem\_op} = 0$** 
  - Le processus est en sommeil jusqu'à ce que la valeur du sémaphore atteigne 0.
- **Rendre (Vendre) un sémaphore  $\Rightarrow \text{sem\_op} > 0$** 
  - La valeur de  $\text{sem\_op}$  est ajoutée à celle du sémaphore, libérant les ressources utilisées jusqu'alors.

## Pourquoi tant de complexité...

- Pour faire des chose compliqués
  - `semid = semget(key, 2, 0666 | IPC_CREAT);`
  - `struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};`
  - `struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};`
  - `arg.val = X`
  - `semctl(semid, 0, SETVAL, arg);`
  - `semctl(semid, 1, SETVAL, arg);`
  - `semop(semid, &push, 2) // 2 car 2 opérations`



# Création de Sem sous Windows (rappel)

```
HANDLE hSemaphore;  
LONG cMax = 10;
```

```
// Create a semaphore with initial and max. counts of 10.
```

```
hSemaphore = CreateSemaphore(  
    NULL, // default security attributes  
    cMax, // initial count  
    cMax, // maximum count  
    NULL); // unnamed semaphore
```

```
if (hSemaphore == NULL)  
{  
    printf("CreateSemaphore error: %d\n", GetLastError());  
}
```

# Sémaphore sous UNIX sans les IPC

## Soit en beaucoup plus simple (rappel)

- **La variable :** `sem_t bin_sem;`
- **La création :**  
`res = sem_init(&bin_sem, 0 /*nb procs*/, 1 /*val*/);`
- **Prendre :** `sem_wait(&bin_sem);`
- **Vendre :** `sem_post(&bin_sem);`
- **Tester :** `sem_getvalue(&bin_sem, &ret);`
- **Détruire :** `sem_destroy(&bin_sem);`
- **Compile :** `Gcc -lpthread toto.c`

En POSIX ...

Nous verrons ça avec les Threads

## Questions?

## Les Segments de mémoire partagé

# La problématique

- Jusqu'à présent on a:
  - Exécuté des processus dans des environnements protégés
  - Synchronisé ces processus à l'aide de signaux
  - Protégé des ressources à l'aide de sémaphores
- Mais on n'a pas partagé de données entre deux processus!

## Qu'est ce que...

- Les processus vont partager des pages physiques par l'intermédiaire de leur espace d'adressage
- Il n'y aura plus de copie d'information
- Cette mémoire partagée devient un espace critique
- Il faudra sans doute en protéger les accès avec des sémaphores par exemple...
- Un segment de mémoire est indépendant de tout processus.
- Il peut exister sans qu'aucun processus n'y accède.

## Avec Les IPC



## Comment on s'en sert

- Un processus rattachera le segment à son espace d'adressage,
- puis pourra manipuler cette mémoire de la même façon qu'il peut manipuler sa propre mémoire.
- `<sys/shm.h>`

## struct shmid\_ds

```
struct shmid_ds {  
    struct ipc_perm  shm_perm;    /* operation permission struct */  
    int              shm_segsz;    /* size of segment in bytes */  
    struct vas       *shm_vas;     /* virtual address space this entry */  
    pid_t            shm_lpid;     /* pid of last shmop */  
    pid_t            shm_cpid;     /* pid of creator */  
    unsigned short int shm_nattch; /* current # attached */  
    unsigned short int shm_cnattch; /* in memory # attached */  
    time_t           shm_atime;    /* last shmat time */  
    time_t           shm_dtime;    /* last shmdt time */  
    time_t           shm_ctime;    /* last change time */  
    char             shm_pad[24];  /* room for future expansion */  
};
```

## Comment on s'en sert

- **Le créer :**

```
int shmget(key_t cle, int taille, int options);
```

- **L'attacher :**

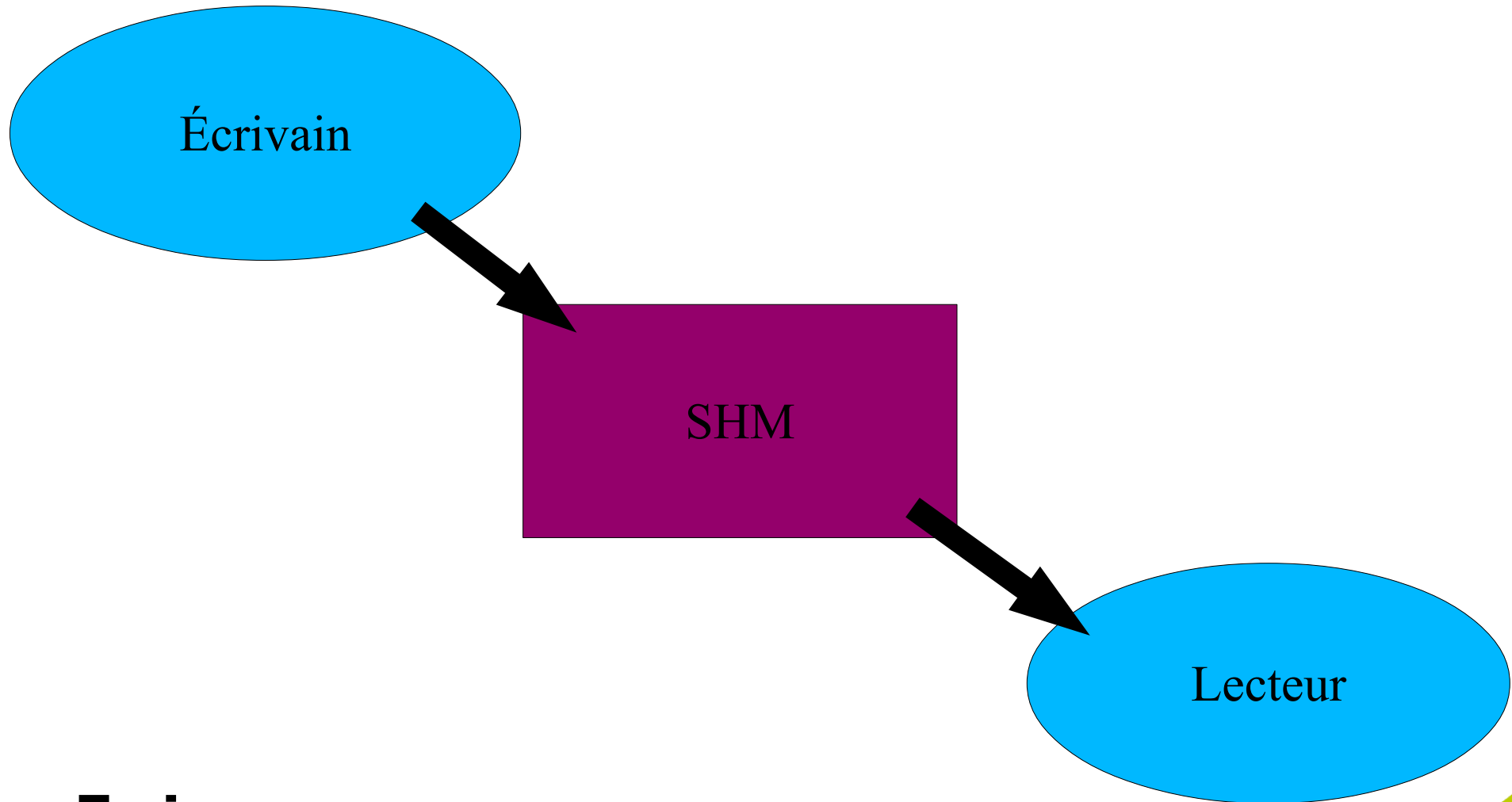
```
void *shmat(int dipc, const void *adr, int option);
```

- Demande d'attachement du segment **dipc** à l'adresse **adr** de l'espace d'adressage du processus.
- La valeur de retour est l'adresse où l'attachement a été effectivement réalisé, c'est-à-dire celle attribuée au premier octet du segment
- Le choix du paramètre **adr** est délicat. On utilisera de préférence **adr = NULL**, c'est-à-dire qu'on laisse le soin au système de sélectionner l'adresse.

## bis

- Le détacher :  
**int shmdt (const void \*adr);**
- Le contrôle :
  - **int shmctl(int dipc, int op, struct shmid\_ds \*pshmid);**
  - génériques à tous les IPC.
    - IPC\_STAT lecture de la structure shmid\_ds
    - IPC\_SET positionnement de la structure shmid\_ds
    - IPC\_RMID permet de détruire le segment (super-utilisateur, ou créateur du sémaphore)

## Un exemple un peu consistant



**En image...**

## Le lecteur

- **Initialise le SHM**

```
ftok("/etc/fstab", getpid());  
id=shmget(key, 4000, 0666 | IPC_CREAT);
```

- **S'y connecte**

```
ad=shmat(id,NULL,SHM_RDONLY);
```

- **Déroute le signal USR1**

```
signal(SIGUSR1,handler);
```

- **Attend le signal USR1**

```
pause(); (et pas wait)
```

- **Affiche le SHM**

```
printf("%s\n",ad);
```

## L'écrivain

- **Se connecte au SHM**  
`id=shmget(cle,4000,0)`
- **Attache adr au SHM**  
`ad=shmat(id,NULL,0)`
- **Ecrit dans le SHM**  
`strcpy(ad,"|msg=");`  
`strcat(ad,argv[3]);`
- **Prévient le lecteur par le signal USR1**  
`kill(pid,SIGUSR1)`

## Et sous Windows



## Named SHM chez Windows

- Le créer :

```
hMapObject = CreateFileMapping(  
    INVALID_HANDLE_VALUE, // use paging file  
    NULL,                  // no security attributes  
    PAGE_READWRITE,       // read/write access  
    0,                     // size: high 32-bits  
    sizeof(struct shared_struct), // size: low 32-bits  
    "MMF1");               // name of map object
```

- Le fermer  
 CloseHandle(hMapObject);

## SHM chez Windows (2)

- L'attaché à un pointeur :

```
shared_memory_loc = MapViewOfFile(  
    hMapObject,          // object to map view of  
    FILE_MAP_WRITE,      // read/write access  
    0,                   // high offset: map from  
    0,                   // low offset: beginning  
    0);                 // default: map entire file
```

- Pour le détacher et détruire  
    UnmapViewOfFile(shared\_memory\_loc)

## Les messages

# Les sémaphores et les SHM ne font pas tout

- Les sémaphores nous permettent de gérer l'accès concurrentiel à des ressources partagées, afin de synchroniser deux ou plusieurs processus.
- Les SHM permettent de partager des données entre des processus
- Synchroniser des processus nécessite de chronométrer leur travail, non pas dans un référentiel absolu, **mais dans un référentiel relatif,**

## Un référentiel relatif

- L'usage de sémaphores pour cet objectif se révèle complexe et limité
  - Complexe parce que chaque processus doit gérer un sémaphore pour chaque processus distinct avec lequel il doit se synchroniser
  - Limité parce que cette solution ne permet pas d'échanger des paramètres entre les processus
  - Possibilité de blocage permanent d'un processus si un autre processus libère la ressource puis la verrouille à nouveau avant qu'un autre ait pu se l'approprier
- pas de solution à des problèmes de synchronisation de processus un peu complexes

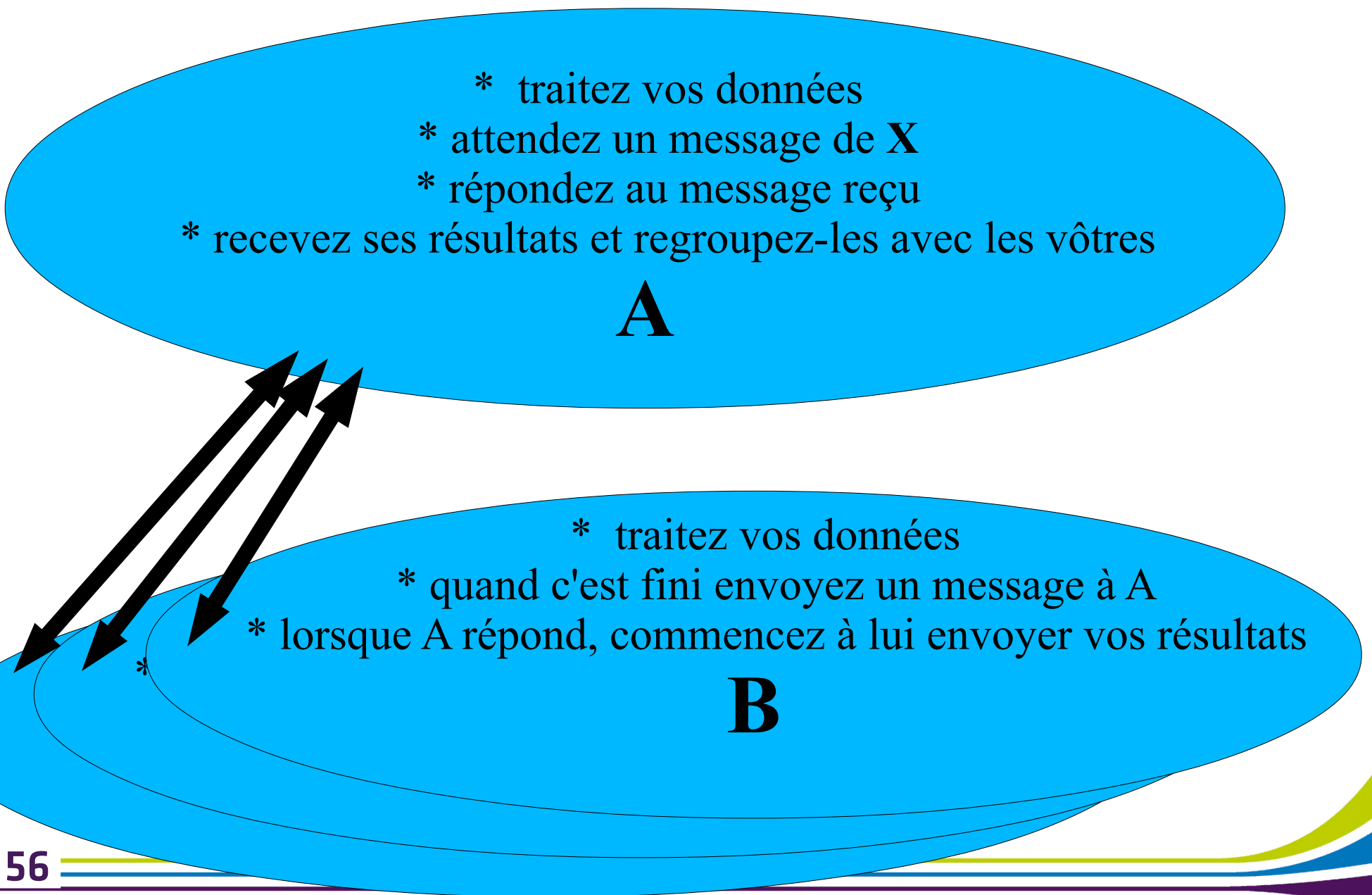
## Une solution élégante à ce problème

- L'utilisation de files d'attente de messages

# La Théorie des Files d'attente de Messages

- Tout processus actif peut créer une ou plusieurs structures dénommées files
  - Chacune de ces structures peut conserver un ou plusieurs messages de différents types
  - Et pouvant provenir de différentes sources
  - Et contenir des informations de nature quelconque
- Chaque processus peut envoyer un message à toute file disponible sous réserve de connaître son identificateur
- Le processus accède à la file séquentiellement

## A quoi ca peut servir

- 
- \* traitez vos données
  - \* attendez un message de **X**
  - \* répondez au message reçu
  - \* recevez ses résultats et regroupez-les avec les vôtres

**A**

- \* traitez vos données
- \* quand c'est fini envoyez un message à A
- \* lorsque A répond, commencez à lui envoyer vos résultats

**B**



## Comment ça s'écrit

linux/msg.h

/\* tampon de message pour les appels msgsnd et msgrcv \*/

```
struct msgbuf {  
    long mtype;      /* type du message */  
    char mtext[1];   /* texte du message */  
};  
  
ou  
  
struct message {  
    long mtype;      /* type du message */  
    long sender;     /* id émetteur */  
    long receiver;   /* id destinataire */  
    struct info data; /* contenu du message */  
    ...  
};
```

- Pour créer : **int msgget(key\_t key, int msgflg)**
- Pour envoyer :  
**int msgsnd(  
    int msqid, struct msgbuf \*msgp,  
    int msgsz, int msgflg)**
  - msqid est l'identificateur de la file,
  - msgp est un pointeur vers le message à envoyer,
  - msgsz est la taille du message  
(sizeof(struct message) - sizeof(long);)
  - msgflg un drapeau relatif aux situations d'attente.

- Pour recevoir :  
**int msgrcv(  
    msqid, \*msgp, int msgsz, mtype, msgflg)**
- Pour supprimer : **msgctl(qid, IPC\_RMID, 0)**

## Un exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

/* Redéfinir la structure msgbuf */
typedef struct mymsgbuf
{
    long mtype;
    int int_num;
    float float_num;
    char ch;
} mess_t;
msgkey = ftok(".", 'm');
/* Crée la file */
qid = msgget(msgkey, IPC_CREAT | 0660);
```

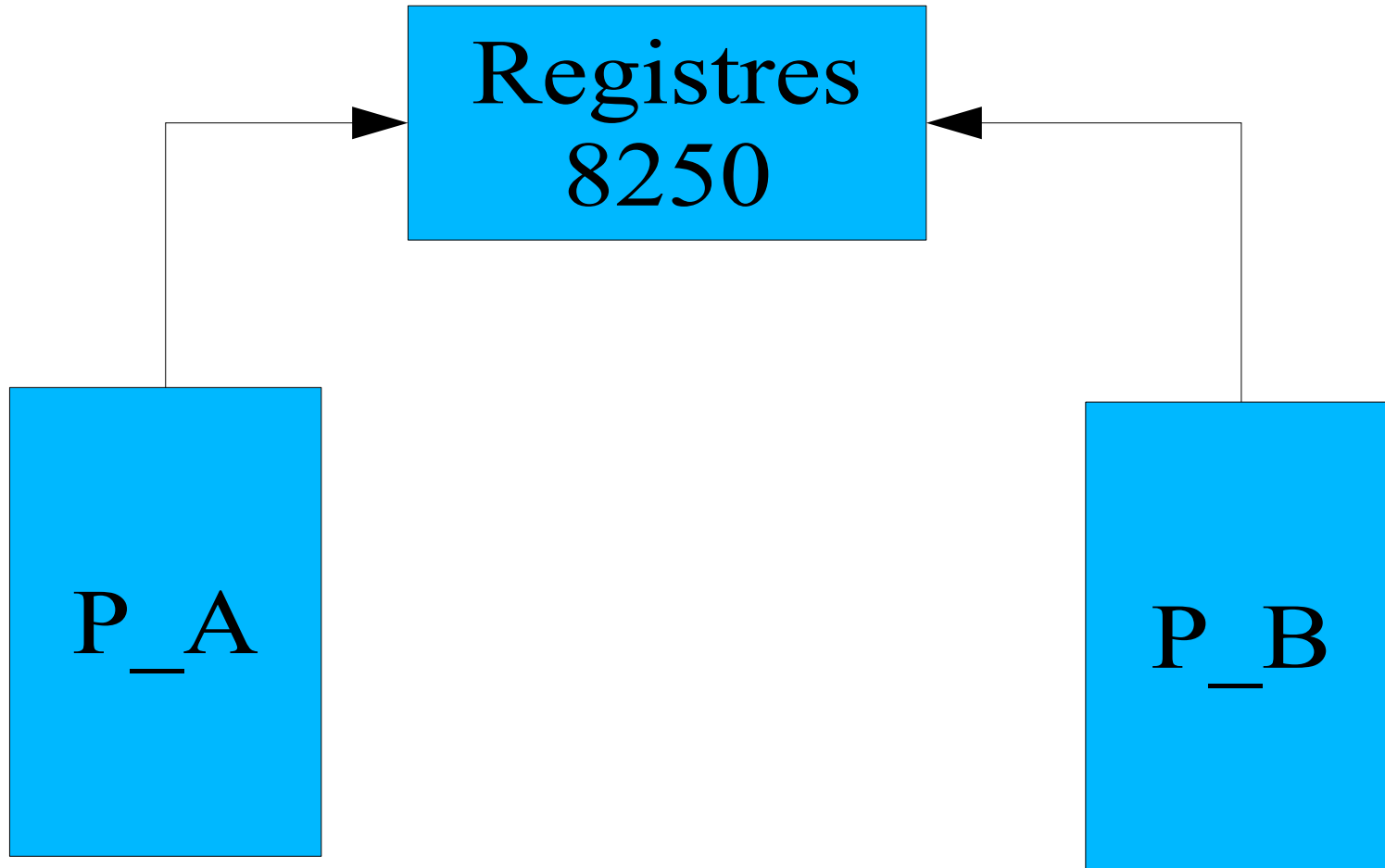
## suite

```
/* Construit un message */  
sent.mtype = 1;  
sent.int_num = rand();  
sent.float_num = (float) (rand()) / 3;  
sent.carattere = 'f';  
  
/* Envoie ce message */  
msgsnd(qid, &sent, length, 0);  
  
/* Recoit le message */  
msgrcv(qid, &received, length, sent.mtype, 0);  
  
/* Detruit la file */  
msgctl(qid, IPC_RMID, 0);
```

## Questions

C'est tout sur La synchronisation et sur la communication inter-processus

## Exercice



Proposer un algorithme pour l'accès au port série  
par le processus A ou le processus B