Cours de S.E. – Coopération, exclusion mutuelle, synchronisation des processus

A. B. Dragut

Univ. Aix-Marseille, IUT Aix en Provence

Plan

- Coopération, modèles
  - définition, motivation
  - modèles
  - outils
- Communication – les pipes et les sockets
  - concept
  - utilisation
  - fonctionnement
  - exemple
  - pipes nommés
  - sockets – brève présentation
- Synchronisation et exclusion mutuelle – les émaphores
  - concept
  - section critique
  - utilisation
- Autres IPC
  - mémoire partagée
  - verrous

Définition, motivation

- un processus indépendant n'est pas affecté par d'autres processus
- des processus coopérants peuvent s'affecter mutuellement (idéalement d'une manière contrôlée)
- motivation :
  - partage des informations
  - temps de calcul réduit
  - modularité, plus de facilité (processus plus facile à gérer)

Exemple

- le shell et les commandes lancées depuis le shell
- les commandes lancées une après l'autre avec des “pipe”s dans le shell
- serveurs et clients

Modèle – producteur-consommateur

- schéma
  - processus qui « produit » des données
  - données stockées dans un tampon
  - processus qui « consomme » les données
- contraintes :
  - le consommateur – attendre qu'il y ait des données à consommer
  - si tampon borné – producteur – verifier qu'il y ait de l'espace libre dans le tampon
  - $\Longrightarrow$ synchronisation, exclusion mutuelle

Modèle – accès concurrent ressource partagée

- schéma
  - ressource commune (e.g. périphérique, fichier, etc.)
  - processus qui doivent y accéder « chacun pour soi »
- contraintes :
  - un seul processus à la fois
  - $\Longrightarrow$ synchronisation, exclusion mutuelle

Outils (IPC) – InterProcess Communication

- pipes – « tuyaux » pour l'envoi de données
- sockets – généralisation – point de communication
- sémaphores, verrous – pour la synchronisation
- mémoire partagée – transfert plus rapide, plus facile à structurer

Pipes

- la plus ancienne forme d'IPC Unix
- disponible partout
- canal half-duplex
- utilisable seulement par processus à ancêtre commun (ayant créé le pipe)

Remarque

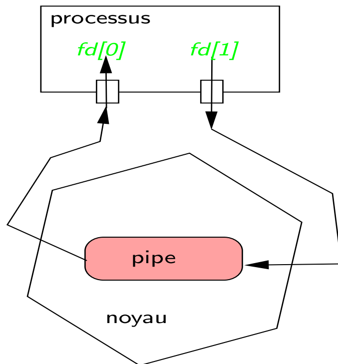
- *A* → *canal half-duplex* ← *B signifie A peut parler à B* **et** *B peut aussi parler à A* **mais pas** *en même temps — comme la poste pneumatique (cylindre avec documents, dans tube pressurisé)*
- *A* → *canal full-duplex* ← *B signifie A peut parler à B et viceversa* **en même temps** — *comme le téléphone*

Création

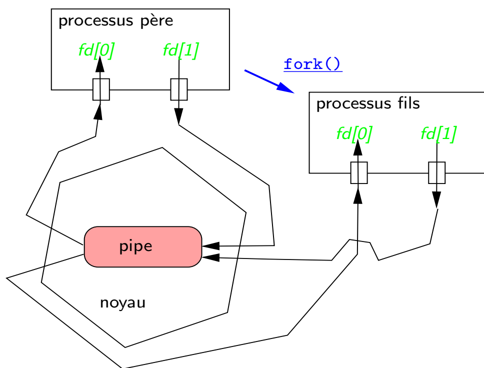
```
#include <unistd.h>
int pipe(int descrFic[2]);
```

- après création, la fonction pipe() rend deux descripteurs de fichier
  - *descrFic[0]* pour lire
  - *descrFic[1]* pour écrire
- si succès, rend zéro
- si erreur, rend -1 et configure errno (trop de descr. de fic, etc.)
- écriture atomique – au plus PIPE_BUF octets

Pipe – structure dans le noyau

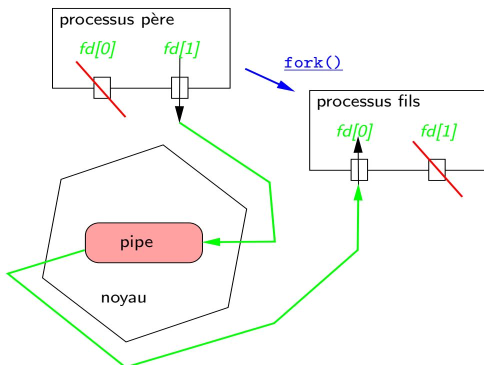


Pipe – père et fils



- le fils hérite les descripteurs ouverts dans le père
- ⇒ accès au pipe – communication

Pipe – père parle au fils

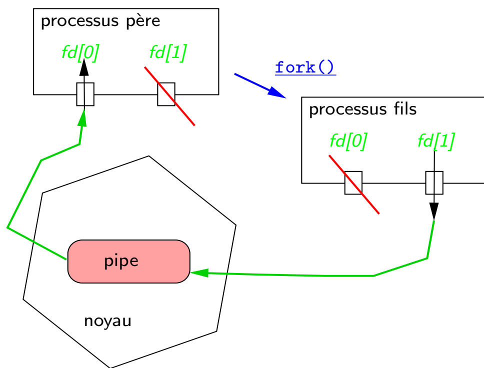


- le père ferme *fd[0]* et écrit sur *fd[1]*
- le fils ferme *fd[1]* et lit depuis *fd[0]*

Exemple

```
#include <sys/types.h>
#include <unistd.h>
// ...
int main() {
 const int Nbuff=1000;char buff[Nbuff];int pF,dFic[2];
 Pipe(dFic);
 if((pF = Fork()) > 0) { // pere
   Close(dFic[0]); Write(dFic[1],"Salut",5);
 }
 else if(pF == 0) { // fils
   Close(dFic[1]); int n = Read(dFic[0],buff,Nbuff-1);
   buff[n] = '\0'; cout << buff;
 }
 return 0;
}
```

Pipe – fils parle au père

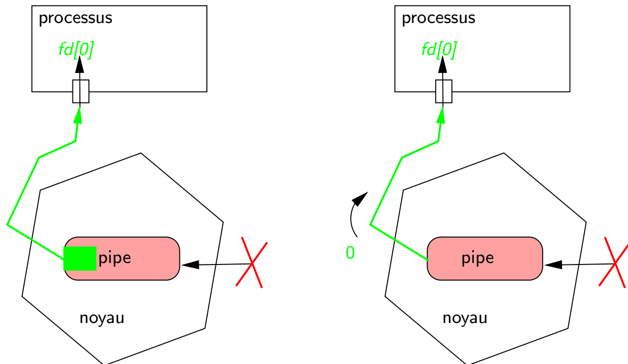


- le fils ferme *fd[0]* et écrit sur *fd[1]*
- le père ferme *fd[1]* et lit depuis *fd[0]*

Exemple

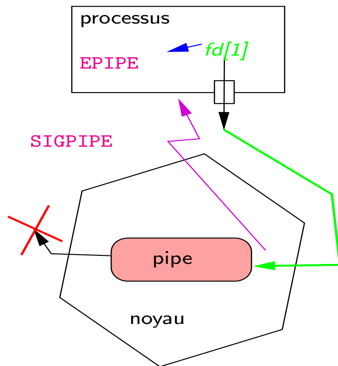
```
#include <sys/types.h>
#include <unistd.h>
// ...
int main() {
 const int Nbuff=1000;char buff[Nbuff];int pF,dFic[2];
 Pipe(dFic));
 if((pF = Fork()) > 0) { // pere
   Close(dFic[1]); int n = Read(dFic[0],buff,Nbuff-1);
   buff[n] = '\0'; cout << buff;
 }
 else if(pF == 0) { // fils
   Close(dFic[0]); Write(dFic[1],"Salut",5);
 }

 return 0;
}
```

Accès sans partenaires – lecture



- si lecture depuis pipe sans rédacteurs, alors, après avoir tout lu, `read()` renvoie zéro

Accès sans partenaires – écriture



- si écriture sur pipe sans lecteurs, alors, SIGPIPE, et write() revient en erreur, errno positionnée à EPIPE.

Les pipes : boîte à outils Unix/shell

- Appliquer successivement plusieurs traitements :
  - chercher une occurence (par exemple, parmi des occurences trouvées)
  - compter des occurences,
  - remplacer des occurences
  - faire des statistiques, trier, etc.
- Chaque étape peut
  1. gérer ses propres fichiers (entrée, sortie)
  2. utiliser l'entrée standard et/ou la sortie standard
- Ex : afficher les lignes distinctes commençant par 'A' :

```
cat monFichier.txt | egrep '^A' | sort | uniq
```

Mini-schéma de connecteur/lanceur de programmes h|f

- `pipe(p)`
- `fork()`, et puis dans le fils
  - `dup2(p[0],0)` (donc l'entrée standard est fermée et devient le pipe).
  - `exec()` un programme *f*
- et dans le père,
  - `dup2(p[1],1)` (donc la sortie standard est fermée et devient le pipe).
  - `exec()` un autre programme *h*
- **PROTOCOLE :** Il faut lire de la même manière dont on écrit ! !
- **FIN DE LA COMMUNICATION :**
  - après un nombre d'échanges,
  - quand on rencontre un charactère special

Remarque

- *Dans l'exemple h|f :*
  - *f lit depuis son entrée standard ce que produit h, mais sans « le savoir ».*
  - *h écrira sur sa sortie standard, sans « savoir » qu'on lit cette sortie standard.*
- *Les programmes (f ou h) peuvent apprendre quel type de fichier est physiquement relié au descripteurs – terminal, pipe, fichier sur disque, etc., et changer leur comportement (* `less` *le fait).*
  - *Ex : La bibliothèque standard I/O fait du full buffer pour un pipe, et non pas du line buffer*
- *Plus d'une connexion – f et g lisent simultanément ce que h produit ? – les pipes nommés.*

FIFO – pipes nommés

- similaires aux fichiers

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *chemin, mode_t mode);
// le 'mode' --- droits d'acces
```

- ensuite on l'ouvre avec open(), etc.
- tout le transfert — tampons du noyau
- le nom dans le système de fichiers — uniquement un point de rendez-vous
- souvent : un lecteur, plusieurs rédacteurs (serveur et ses clients)

Ouverture et fermeture

Ouverture

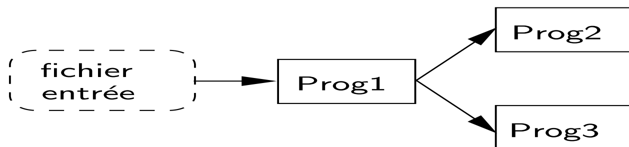
- sans O_NONBLOCK,
  - un open() O_RDONLY bloque jusqu'à ce qu'un autre proc. l'ouvre pour l'écriture
  - un open() O_WRONLY bloque jusqu'à ce qu'un autre proc. l'ouvre pour l'lecture
- avec O_NONBLOCK,
  - un open() O_RDONLY revient tout de suite
  - un open() pour O_WRONLY revient en erreur, errno configurée à ENXIO.
- si open() avec O_RDWR,
  - POSIX ne spécifie rien
  - Linux — succès avec ou sans O_NONBLOCK

Fermeture

- similairement aux pipes anonymes —
- écriture sans lecteurs – SIGPIPE
- lecture sans rédacteurs – read() renvoie zéro

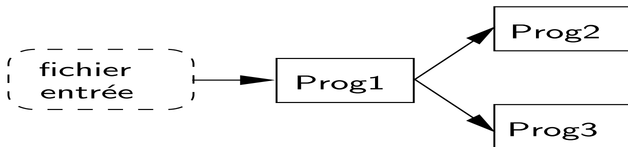
Utilisation des pipes nommés

- dans le shell, pour connexions non-linéaires, sans fichiers intermédiaires



- on dispose de la commande *tee <nomFic>* qui copie son entrée standard
  - sur sa sortie et
  - dans le fichier
- idée —
  - pipe nommé à la place du fichier
  - un des programmes – autre bout du pipe nommé

Exemple concret

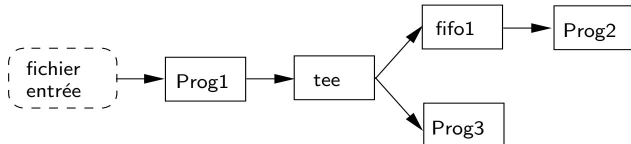


- par exemple, le fichier d'entrée contient des dates de transactions et un bref descriptif de chaque transaction (type, destination, etc.)

  20051012 trans T 52.3 a b
- on veut des statistiques sur ces transactions
  - quels sont les jours avec transactions de type T (on peut en avoir plusieurs par jour) '
  - combien de transactions de type T il y en a au total
- *Prog1* extrait ces transactions (peut-être un simple *grep ' T '*)
- *Prog2* énumère les jours (par exemple, *awk '{print $1;}'|sort|uniq*)
- *Prog3* compte les lignes (par exemple *wc -l*)
- la sortie de *Prog1* doit arriver dans l'entrée de chacun des *Prog2* et *Prog3*.

Connexions non-linéaires

```
mkfifo fifo1
Prog2 < fifo1 & Prog1 < fichierEntree | tee fifo1 | Prog3
```




```
awk '{print $1;}' < fifo1 |sort|uniq & grep ' T ' < fichierEntree | tee fifo1 | wc
```

et pour ce contenu de `fichierEntree`, par exemple

```
20051012 trans T 52.3 a b
20051012 trans B 42.2 bb cc
20051012 trans T 14.2 ee ff
20050920 trans T 10.2 cc ddd
```

on a

```
20050920
20051012
3
```

Client-serveur avec des pipes nommés

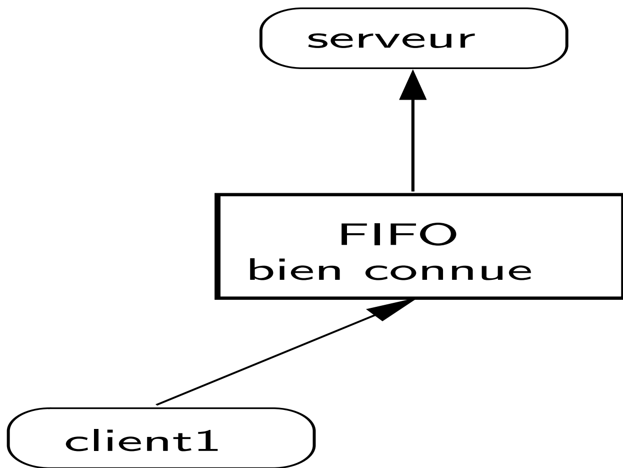
- serveur –
  - fournit des services sur demande des clients
  - attend les demandes sur un pipe nommé bien connu par les clients
  - atomicité requêtes – longueur au plus `PIPE_BUF`
  - réponse sur FIFO dédié pour chaque client – par exemple `/tmp/<NomServeur>.<PIDclient>` (le client donnant son PID au serveur)
  - serveur – robuste (par ex. client peut disparaître avant de lire la réponse – `SIGPIPE` envoyé au serveur – lequel doit le dérouter)

Client-serveur

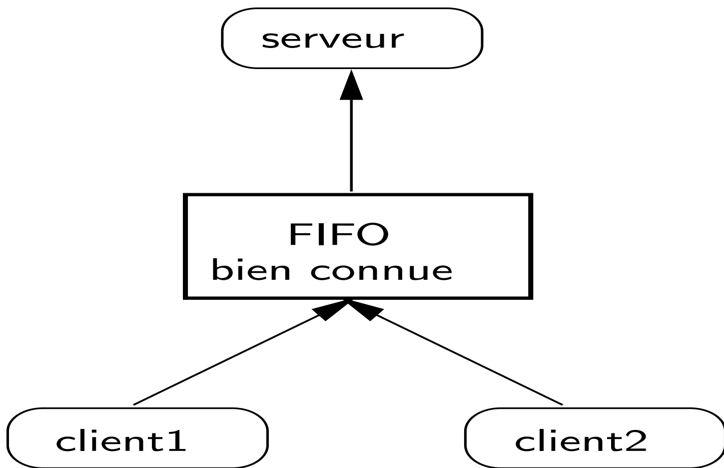
serveur

FIFO
bien connue

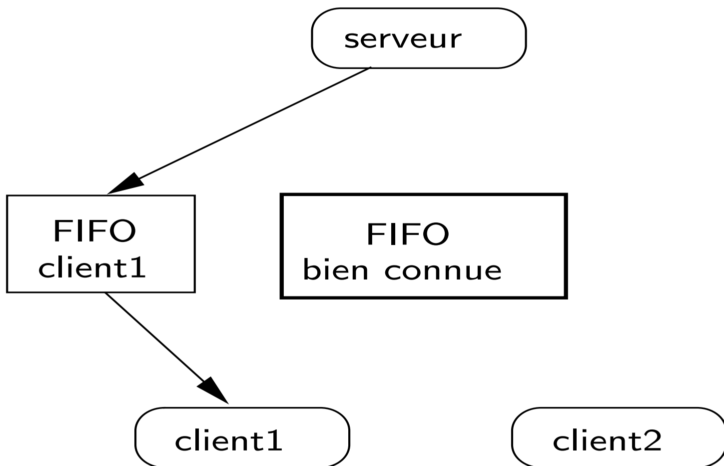
Client-serveur



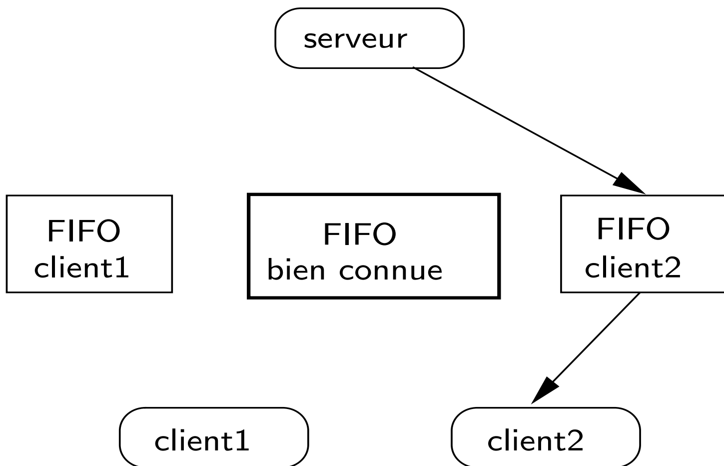
Client-serveur



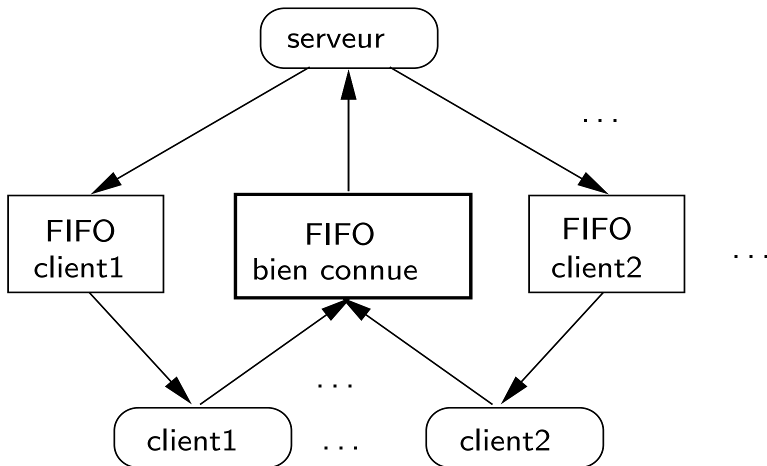
Client-serveur



Client-serveur



Client-serveur



Sockets

- ”prise de courant” — branchement — point de communication
- paradigme réseau, mais également domaine Unix (ou local)
- fonction socketpair() – généralisation de pipe() — full duplex

```
int socketpair(AF_UNIX, SOCK_STREAM, 0, descrF[2]);
```

- utilisation similaire

Sémaphores

- **contrôle de l'accès à des ressources partagées**
  - objets ayant une valeur entière
  - deux opérations atomiques
    - $P()$ — réservation — si sémaphore (ressource) dispo (i.e. valeur positive), décrémentation

$$P(s) : [\textbf{tantque}(s == 0)\{\textbf{attendre}\}; s \leftarrow s - 1]$$

    - $V()$ — libération

$$V(s) : [s \leftarrow s + 1]$$

  - l'attente peut se faire passivement : processus endormi, et réveillé quand s n'est plus 0.
- **gérer l'alternance de l'accès à des ressources partagées**
  - plusieurs sémaphores

Section critique

- plusieurs processus accèdent à la même ressource — besoin d'arbitrage
- section critique — portion du code où est effectué l'accès
- il faut qu'à un moment donné, un seul processus se trouve dedans
- solution : un sémaphore binaire (deux valeurs) :
  - $P(b)$
  - section critique
  - $V(b)$
- tous les processus compétiteurs doivent respecter la discipline (i.e. utiliser le sémaphore) — objet « global », géré par le noyau
- si la ressource existe en nombre limité n — valeur de départ du sémaphore sera n

Remarque

Attention, si utilisation de plusieurs sémaphores, danger étreinte fatale (deadlock) – chaque processus attend une ressource bloquée par un autre — ordre des opérations – très important.

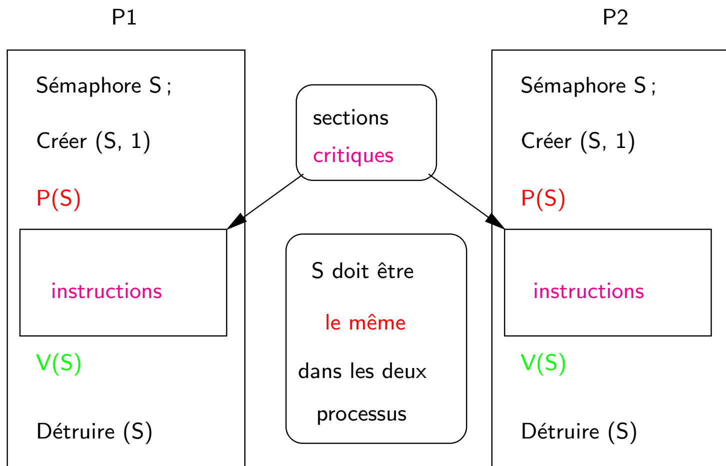
Schéma utilisation sémaphores

- opérations : $P(S)$ et $V(S)$
- le noyau – une liste de processus en attente
- $P(S)$ – si compteur nul, alors
  - liste $\leftarrow$ processus
  - processus endormi
- sinon décrémentation compteur
- $V(S)$ – incrémentation compteur
- et si liste non-vide, alors
  - reveil processus de la liste (si conditions remplies)

Schéma utilisation sémaphores et ressources

- opérations : $P(S)$ et $V(S)$
- le noyau – une liste de processus en attente
- $P(S)$ – si compteur nul, alors
  - liste $\leftarrow$ processus
  - processus endormi **en ATTENTE de ressources**
- sinon décrémentation compteur pour **PRÉLEVER une ressource**
- $V(S)$ – incrémentation compteur pour **RENDRE une ressource**
- et si liste non-vide, alors
  - reveil processus de la liste (si conditions remplies)

Sémaphores



Création sémaphores System V avec clé – séquences OK

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

1 Si (sémaphore de clé *Key* n'existe pas) alors
2 demander au S.E. de créer le sémaphore S
3 demander au S.E. de l'initialiser
sinon
4 récupérer le sémaphore S auprès du S.E.
fin si
5 P(S)
6 // instructions critiques
V(S)

P	P'
1	
2	
3	
5	
	1'
	4'
	5'

OK

P	P'
1	
2	
3	
	1'
	4'
	5'
5	

OK

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

doit être atomique

1 Si (sémaphore de clé *Key* n'existe pas) alors
2 demander au S.E. de créer le sémaphore S
3 demander au S.E. de l'initialiser
sinon
4 récupérer le sémaphore S auprès du S.E.
fin si
5 P(S)
6 // instructions critiques
V(S)

P	P'
1	
.	1'
.	.
.	.

Faux

P	P'
1	
2	
	1'
	4'
	5'
3	
5	

Faux

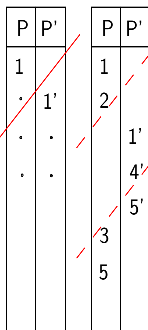
Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin

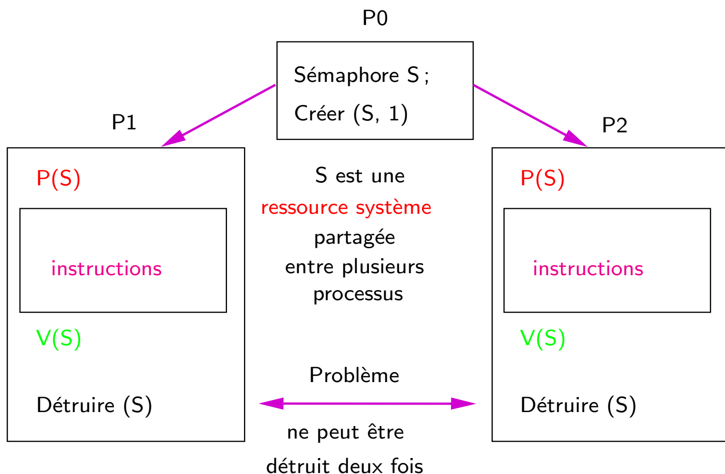
qui crée le sémaphore de clé *Key*

Linux/Unix : PAS ATOMIQUE

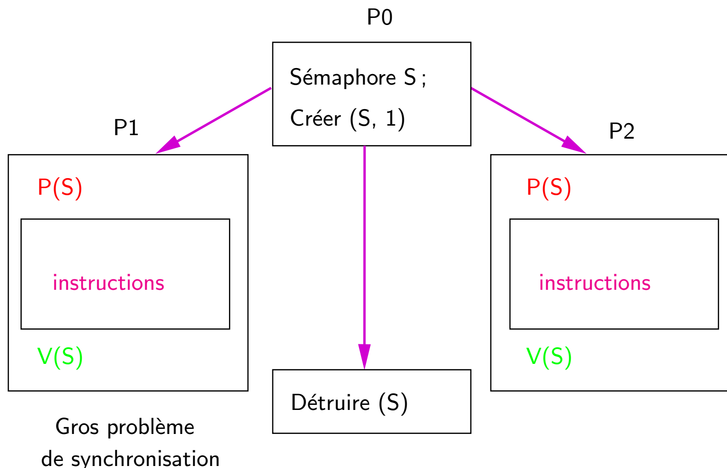
1 Si (sémaphore de clé *Key* n'existe pas) alors
2 demander au S.E. de créer le sémaphore S
3 demander au S.E. de l'initialiser
sinon
4 récupérer le sémaphore S auprès du S.E.
fin si
5 P(S)
6 // instructions critiques
V(S)



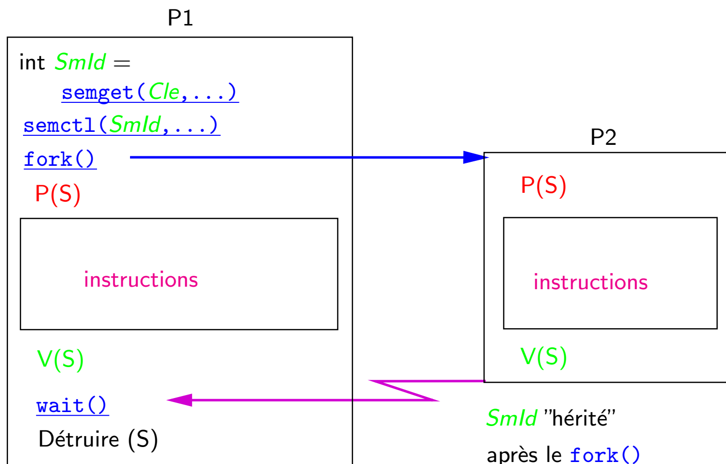
Sémaphores



Sémaphores



Sémaphores System V



Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin

qui crée le sémaphore de clé *Key*

Linux/Unix : ATOMIQUE seulement ceci

1 Si (sémaphore de clé *Key* n'existe pas) alors

2 demander au S.E. de créer le sémaphore S

3 demander au S.E. de l'initialiser

sinon

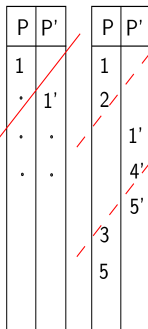
4 récupérer le sémaphore S auprès du S.E.

fin si

5 P(S)

6 // instructions critiques

V(S)



Sémaphores style System V

- identification — clé IPC (∼ noms des fichiers)
- ensembles de sémaphores obtenus simultanément – `semget()` — prend une clé IPC et rend un identificateur (∼ descripteur de fichiers) — utilisé pour les opérations
- initialisation, manipulation — `semctl()`
- le noyau maintient des infos — les UID du créateur et du possesseur (∼ fichiers)
- $P()$ et $V()$ — `semop()` — conditions à remplir simultanément sur tous les membres
- persistance — même après la fin des processus — besoin de les supprimer explicitement — uniquement par le créateur ou possesseur ou superutilisateur — `semctl()`

Obtention d'un identificateur de sémaphore. Création

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

int semget(key_t cleIPC, int nbreSems, int semFlg);
```

- rend l'**identificateur de la famille** de **nbreSems** sémaphores
  - création : **cleIPC** : *IPC_PRIVATE*, **semFlg** :*IPC_CREAT*|*IPC_EXCL*|0700
  - pour l'utilisation d'un déjà existent : **cleIPC** : un entier
- les valeurs des sémaphores créés sont indéterminées. (POSIX.1-2001)

Structures initialisés par **semget()**

- initialise une structure de <sys/sem.h> **struct semid_ds** {
  - struct ipc_perm sem_perm.mode// sur les 9 bits de poids faibles :semflg
  - unsigned shortsem_nsems// no. de sémaphores : nsems
  - time_t sem_otime // 0
  - time_t sem_ctime // l'heure actuelle
  - }; // semid_ds
- initialise une structure de <sys/ipc.h> **struct ipc_perm**{
  - key_t __key; // cleIPC de semget(2) */
  - uid_t sem_perm.cuid // l'UID effectif du proc. appelant
  - uid_t **sem_perm.uid** // ...
  - gid_t sem_perm.cgid //le GID effectif du proc. appelant
  - gid_t **sem_perm.gid** // ...
  - unsigned short **sem_perm.mode** // les 9 bits de poids faibles de semflg
  - unsigned short __seq;
  - }; // ipc_perm

Initialisation d'un sémaphore de la famille

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

int  semctl(int semIdent, int semNumero, int cmd, ...);
```

- **semctl** applique une commande au sémaphore **semNumero** de l'ensemble d'identificateur **semIdent** rendu par semget()
- **commande** : si **cmd**== **IPC_RMID** pas de quatrième argument
- "..." quatrième argument qui depend de la valeur de **cmd**
- **union semun** {
  - int val ; //cmd==SETVAL
  - struct ::semid_ds *buf ;//cmd==IPC_STAT ou IPC_SET
  - unsigned short int ∗array ;//cmd==GETALL ou SETALL
  - struct ::seminfo ∗__buf ; //cmd==IPC_INFO <sys/sem.h> Linux
  - } ; // semun

Opération sur un sémaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semIdent, struct sembuf *sOpers,
     unsigned nbreSOpers);
```

- **sOpers** un tableau de **nbreOpers struct sembuf**
- pour une opération à effectuer sur un seul sémaphore :
- une **struct sembuf** : {
  - unsigned short sem_num //numero du semaphore''
  - short sem_op //opération : nombre a rajouter ou soustraire, ou zero''
  - short sem_flg //IPC_NOWAIT, SEM_UNDO
  - }; // sembuf
- implementation : V() : sem_op positive : valeur semaph += sem_op
- implementation P() : sem_op negative et pas de IPC_NOWAIT :
  - si valeur semaph. >-sem_op => valeur semaph. += sem_op
  - sinon => attente
  - fin attente : soit valeur semaph. >= -sem_op, soit suppr.-> err. ERMID

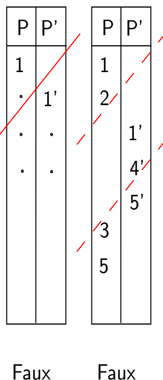
Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin

qui crée le sémaphore de clé *Key*

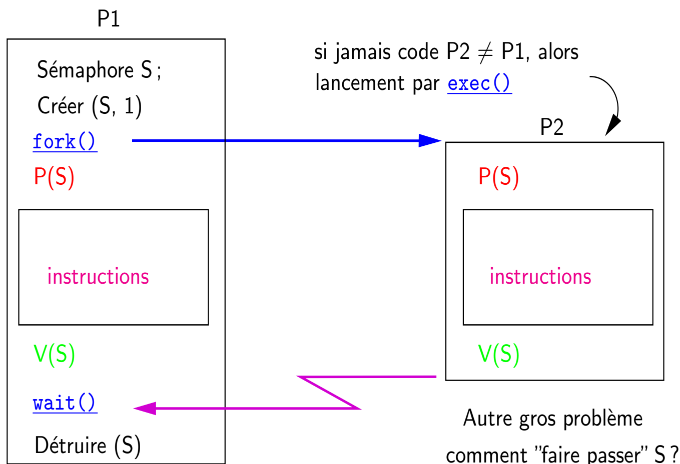
Linux/Unix : ATOMIQUE seulement ceci

```
1 } SmId=semget(Key,1,
2 }          IPC_EXCL|IPC_CREAT|0700) ;
    if(SmId ! = −1) {
3     semctl(SmId,SETVAL,NbRes) ;
    } else {
4     SmId=semget(Key) ;
    }
  sembuf P={0,-1,0},V={0,1,0} ;
5 semop(SmId,&P,1) ;
6 // instructions critiques
  semop(SmId,&V,1) ;
```

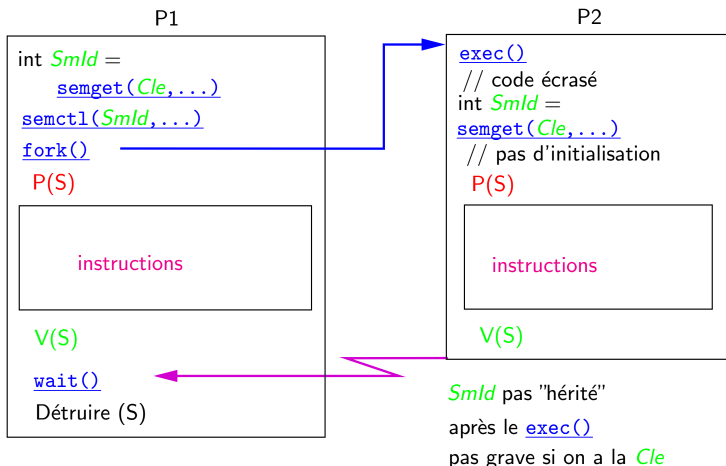


Faux Faux

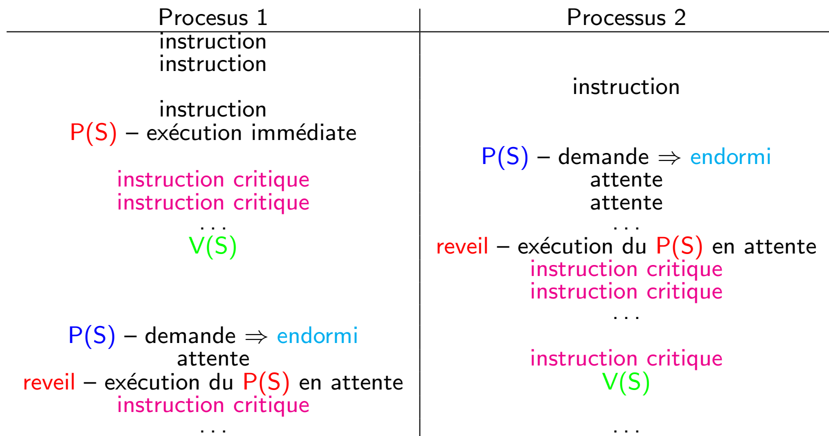
Sémaphores



Sémaphores System V



Exemple séquence utilisation sémaphores



Écriture programmes utilisation sémaphores

- instruction critique – utilisation d'une ressource commune R1 – tampon d'imprimante, segment mémoire partagée, etc.

Programme 1	Programme 2
...	...
P(S1)	P(S1)
utilisation R1	utilisation R1
V(S1)	V(S1)
...	...

Utilisation de deux ressources R1 et R2

- besoin de deux sémaphores S1 et S2

Programme 1	Programme 2
...	...
P(S1)	P(S1)
utilisation R1	utilisation R1
V(S1)	V(S1)
P(S2)	P(S2)
utilisation R2	utilisation R2
V(S2)	V(S2)
...	...

Remarque

Rappel, tous ces schémas doivent fonctionner quel que soit l'ordre d'exécution entrelacée des pas atomiques de tous les programmes sémaphorisés.

Utilisation de deux ressources R1 et R2

- besoin de deux sémaphores S1 et S2

Programme 1		Programme 2
...	ça	...
P(S1)		P(S2)
utilisation R1		utilisation R2
P(S2)	← peut bloquer →	P(S1)
V(S1)	chacun attend que	V(S2)
utilisation R2	son partenaire	utilisation R1
V(S2)	libère l'autre	V(S1)
...	sémaphore	...

Remarque

Il faut donc prêter beaucoup d'attention aux interblocages potentiels

Exemple utilisation un seul sémaphore

- lecture simultanée depuis fichier, le père affiche les '1', et le fils les '0'
- contenu du fichier
  00011110
  11101010
- sortie sans utiliser les sémaphores, chaqun écrit quand il veut :
  10101010
  1
  0101...
- ressource partagée : sortie standard, par père et fils, ligne par ligne — chaque ligne soit le père, soit le fils – avec sémaphores :
  1111
  0000
  11111
  000
  (chaque ligne du fichier ⇒ deux lignes en sortie, si elle a des 1 et des 0).

Schéma

- création d'un seul sémaphore, valeur 1
- duplication, et dans les deux (père et fils)
- ouverture fichier, et pour chaque ligne lue
  - attente sémaphore 1, mise à 0
  - pour chaque caractère de la ligne,
    - si père et caractère vaut 1 ou bien fils et caractère vaut 0, afficher
  - mettre sémaphore à 1
- dans le père, attente fils, destruction sémaphore avec `shmctl()` et `IPC_RMID`

Remarque

Ceci nous donne un mutex, pour assurer l'exclusion mutuelle lors du partage de la ressource.

Schéma du programme – sans tests d'erreur

```
... //include necssrs sys/{types,wait,ipc,...}.h,etc.
int feu = semget(IPC_PRIVATE,1,0700);
semctl(feu,0,SETVAL,1);
sembuf rouge = {0,-1,0},vert = {0,1,0};int p = fork();
// rouge -- decremente , vert -- incremente
ifstream inputStream ("./toto");
for(string line; getline(inputStream,line);) {
  semop(feu, &rouge, 1);  // attente pour 1, mise a 0
  // debut zone critique, exclusion mutuelle
  for(unsigned int i = 0; i < line.size(); ++i)
   if((p && line[i] == '1') || (!p && line[i] == '0')){
        cout  << line[i] << flush;
   }
  cout<<"\n";semop(feu, &vert, 1);//fin zone critique
}
inputStream . close();
if(p) { wait(0); semctl(feu, 0, IPC_RMID,0); }
```

Autres outils d'IPC :Mémoire partagée

- la plus rapide forme d'IPC
- création — `shmget()` — prend une clé IPC et rend un identificateur (∼ descripteur de fichiers) — utilisé pour les opérations
- attachement au processus — `shmat()` — prend un identificateur et rend l'adresse
- besoin de synchroniser l'accès — deux sémaphores
  - but : un processusécrira, un autre lira, à tours de rôles
  - mais dans le système — élection processus non-déterministe
  - nécessaire :
    1. exclusion mutuelle (une action à la fois)
    2. ordre – écriture d'abord, lecture ensuite, cycle
- à la fin — détachement du processus — `shmdt()`
- suppression avec `shmctl()` et `IPC_RMID`

Autres outils de synchronization :Verrous sur fichiers

- sur le fichier entier — `flock()` prend un descr de fichier et l'opération :
  - `LOCK_SH` – verrou partagé (lecture)
  - `LOCK_EX` – verrou exclusif (écriture)
  - `LOCK_UN` – enlèvement verrou
  - OU binaire avec `LOCK_NB` pour ne pas bloquer en attente
- sur une partie du fichier — `fcntl()` — région arbitraire en octets – `struct flock` , bloquant (`F_SETLKW`) ou pas bloquant (`F_SETLK`)

verrou existant	demande	
	lecture	écriture
pas de verrou	OK	OK
un ou plusieurs lect.	OK	refusée
un écrit.	refusée	refusée