

# *Système d'exploitation*

## *Partie 2: programmation système*

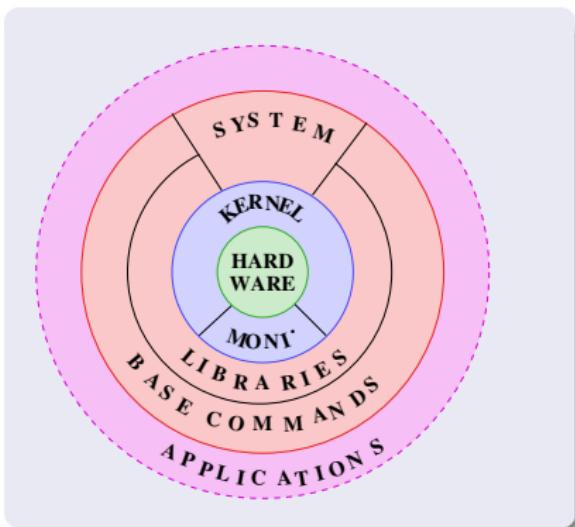
Le 30 août 2022, SVN-ID 425

30 août 2022

## 4 Appel système

- Organisation
- Format général d'un appel système

# Couches Principales



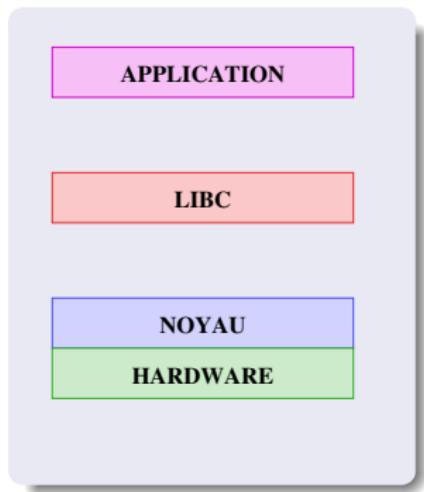
**matériel** CPU, RAM, contrôleurs et périphériques.

**moniteur** Petit programme en ROM, qui tourne au démarrage de la machine.

**noyau** Gère et donne accès au matériel

**système** Couche de standardisation applications

# Appel système et libc



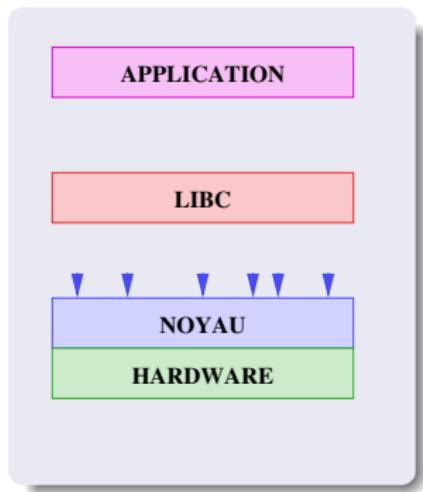
## Application

Utilisation de la libc  $\implies$  portabilité,  
Utilisation directe des appels système  
 $\implies$  difficile

libc ( $\downarrow$ ) Ensemble de services complets  
normalisés  $\implies$  portabilité

Appels système ( $\downarrow$ ) Services du noyau, peu  
nombreux  $\implies$  fonctions basiques

# Appel système et libc



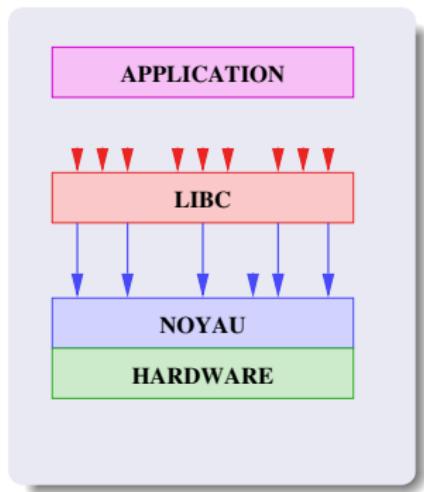
## Application

Utilisation de la libc  $\implies$  portabilité,  
Utilisation directe des appels système  
 $\implies$  difficile

libc ( $\downarrow$ ) Ensemble de services complets  
normalisés  $\implies$  portabilité

Appels système ( $\downarrow$ ) Services du noyau, peu  
nombreux  $\implies$  fonctions basiques

# Appel système et libc



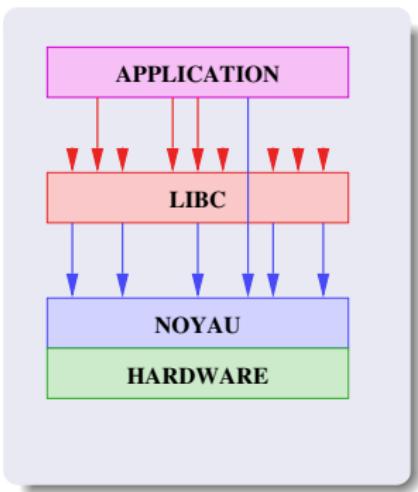
## Application

Utilisation de la libc  $\implies$  portabilité,  
Utilisation directe des appels système  
 $\implies$  difficile

libc ( $\downarrow$ ) Ensemble de services complets  
normalisés  $\implies$  portabilité

Appels système ( $\downarrow$ ) Services du noyau, peu  
nombreux  $\implies$  fonctions basiques

# Appel système et libc



## Application

Utilisation de la libc  $\implies$  portabilité,  
Utilisation directe des appels système  
 $\implies$  difficile

libc ( $\downarrow$ ) Ensemble de services complets  
normalisés  $\implies$  portabilité

Appels système ( $\downarrow$ ) Services du noyau, peu  
nombreux  $\implies$  fonctions basiques

# Appel système et libc

libc : module standalone service n'utilisant pas les appels système (ex : module strxxx)

libc : module interface

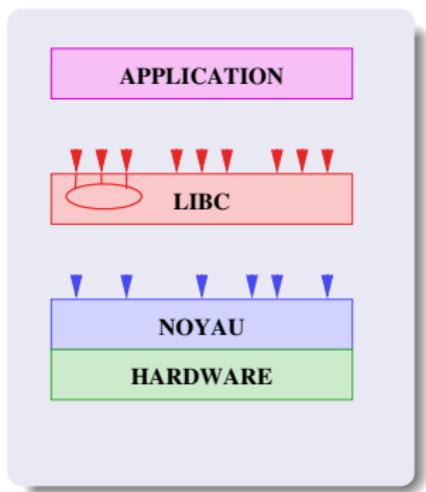
malloc, free, ...  $\Rightarrow$  brk utilisable  
fopen, fread, ...  $\Rightarrow$  performance acceptable

Libc : driver quasi-direct sur appels système

$\Rightarrow$  portabilité pour la plupart (ex : open, read)

$\Rightarrow$  souvent pas très pratique (ex : time)

$\Rightarrow$  risque d'utilisation non performante



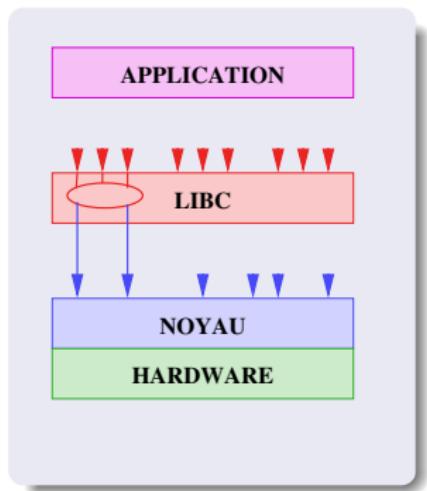
# Appel système et libc

libc : module standalone service n'utilisant pas les appels système (ex : module strxxx)

libc : module interface

malloc, free, ...  $\Rightarrow$  brk utilisable  
fopen, fread, ...  $\Rightarrow$  performance acceptable

Libc : driver quasi-direct sur appels système



$\Rightarrow$  portabilité pour la plupart (ex : open, read)

$\Rightarrow$  souvent pas très pratique (ex : time)

$\Rightarrow$  risque d'utilisation non performante

# Appel système et libc

libc : module standalone service n'utilisant pas les appels système (ex : module strxxx)

libc : module interface

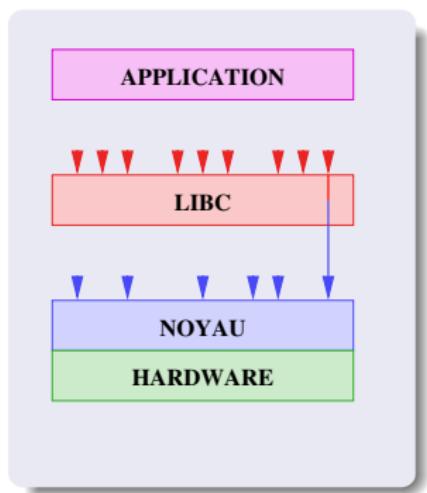
malloc, free, ...  $\Rightarrow$  brk utilisable  
fopen, fread, ...  $\Rightarrow$  performance acceptable

Libc : driver quasi-direct sur appels système

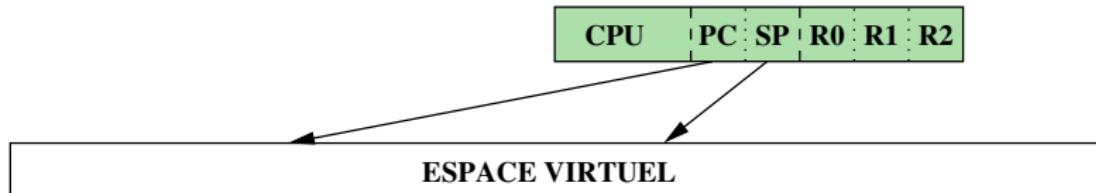
$\Rightarrow$  portabilité pour la plupart (ex : open, read)

$\Rightarrow$  souvent pas très pratique (ex : time)

$\Rightarrow$  risque d'utilisation non performante



# Espaces virtuels



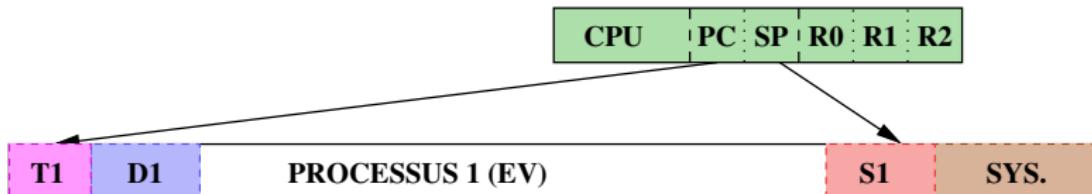
CPU Quelques registres

PC Program Counter, adresse de l'instruction à exécuter.

SP Stack Pointeur, adresse du haut de la pile d'exécution.

Espace virtuel La mémoire que voit le processeur.

# Espace virtuel d'un processus



T1 : segment **text** il contient les instructions. Le PC se balade dans ce segment.

D1 : segment **données** il contient les données globales du processus.

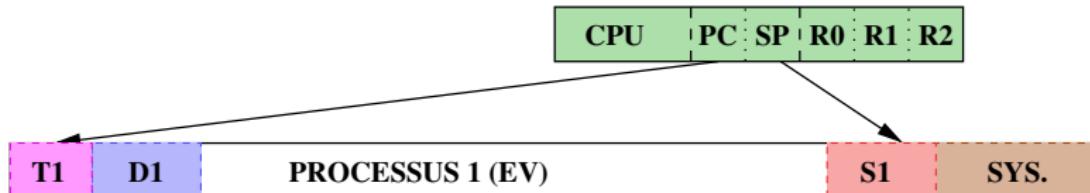
S1 : segment **pile** il contient la pile d'exécution : données locales aux fonctions, les paramètres d'appel des fonctions et les adresses de retour dans l'appelant.

**trous** un accès à une adresse dans un trou  $\implies$  exception  
"segmentation fault"

espace user/système

- Si le processeur est en mode système : il peut accéder à tout l'espace virtuel (sauf les trous).
- Si le processeur est en mode user : il ne peut pas accéder

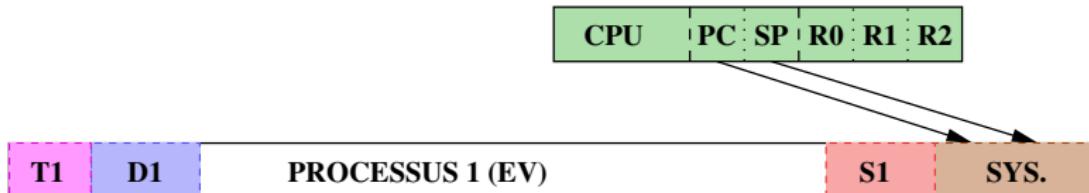
# Appel système



C'est le mécanisme qui permet à un processus en mode utilisateur :

- ① passer en mode système
- ② exécuter une fonction du noyau (en mode système)
- ③ revenir en mode utilisateur après l'exécution de la fonction.

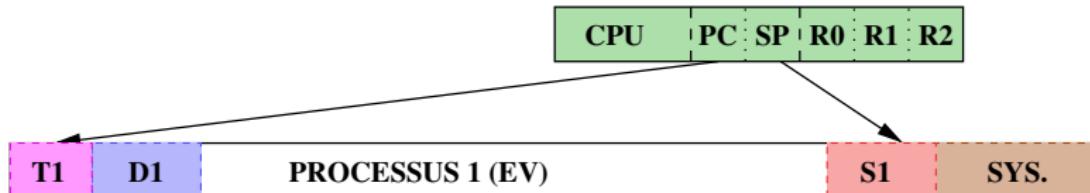
# Appel système



C'est le mécanisme qui permet à un processus en mode utilisateur :

- ① passer en mode système
- ② exécuter une fonction du noyau (en mode système)
- ③ revenir en mode utilisateur après l'exécution de la fonction.

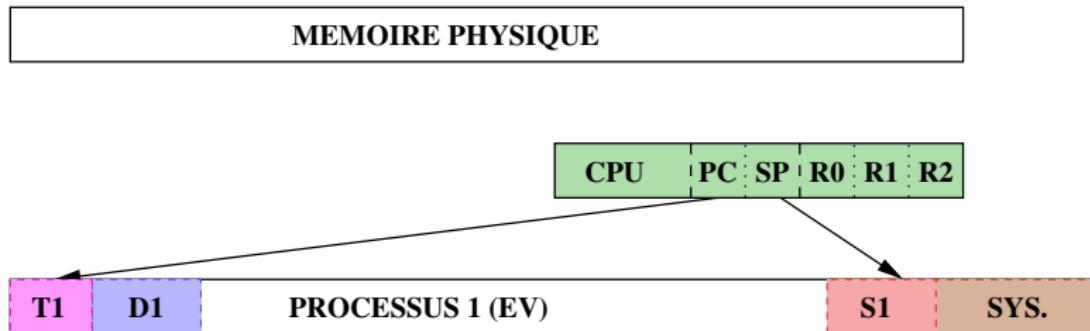
# Appel système



C'est le mécanisme qui permet à un processus en mode utilisateur :

- ① passer en mode système
- ② exécuter une fonction du noyau (en mode système)
- ③ revenir en mode utilisateur après l'exécution de la fonction.

# MMU : Memory Management Unit



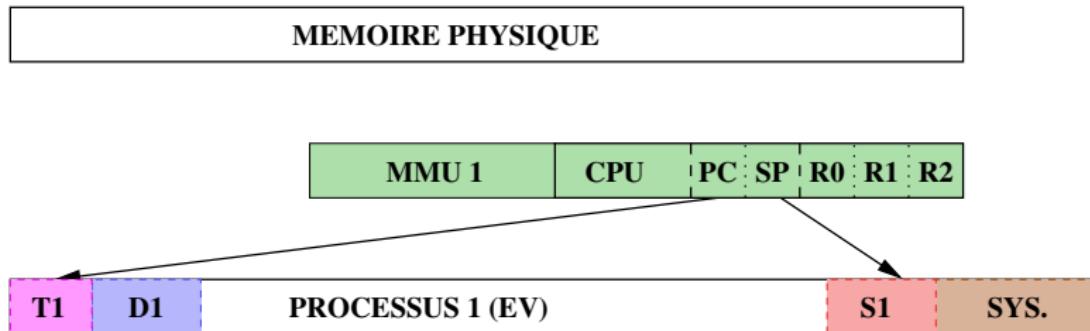
**MMU** Unité matérielle contenant une batterie de registre

**Fonction** En fonction des valeurs contenues dans les registres :

- Convertit les adresses virtuelles en adresses physiques.
- Assure les protections mémoire (trou, ...).

**MMU i** La MMU configurée pour le processus i.

# MMU : Memory Management Unit



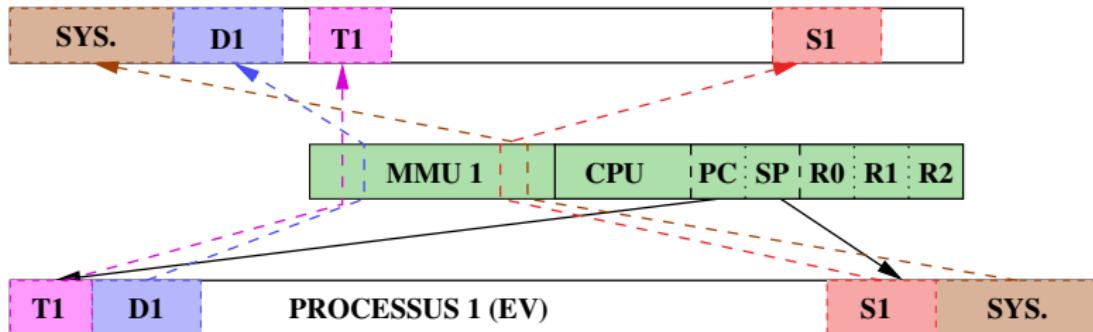
**MMU** Unité matérielle contenant une batterie de registre

**Fonction** En fonction des valeurs contenues dans les registres :

- Convertit les adresses virtuelles en adresses physiques.
- Assure les protections mémoire (trou, ...).

**MMU i** La MMU configurée pour le processus i.

# MMU : Memory Management Unit



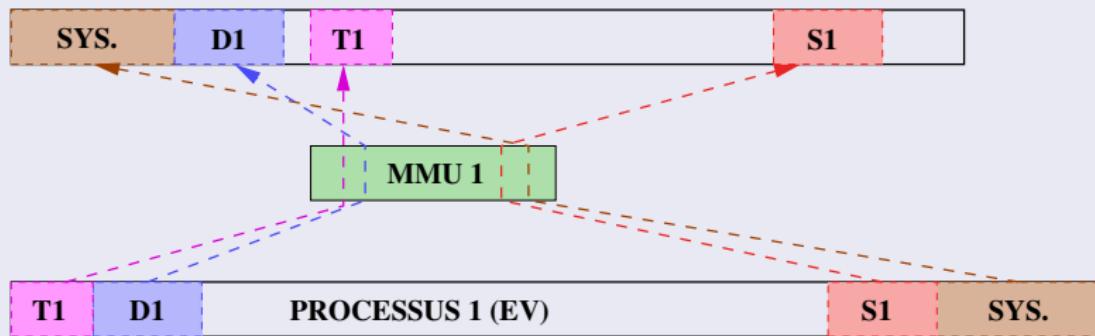
**MMU** Unité matérielle contenant une batterie de registre

**Fonction** En fonction des valeurs contenues dans les registres :

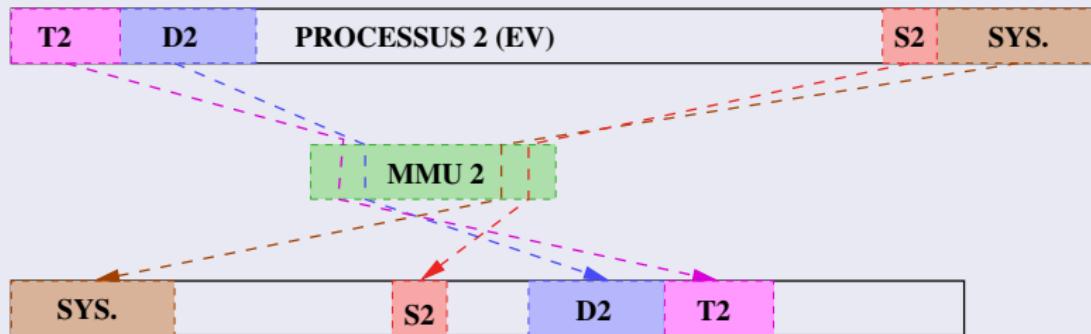
- Convertit les adresses virtuelles en adresses physiques.
- Assure les protections mémoire (trou, ...).

**MMU i** La MMU configurée pour le processus i.

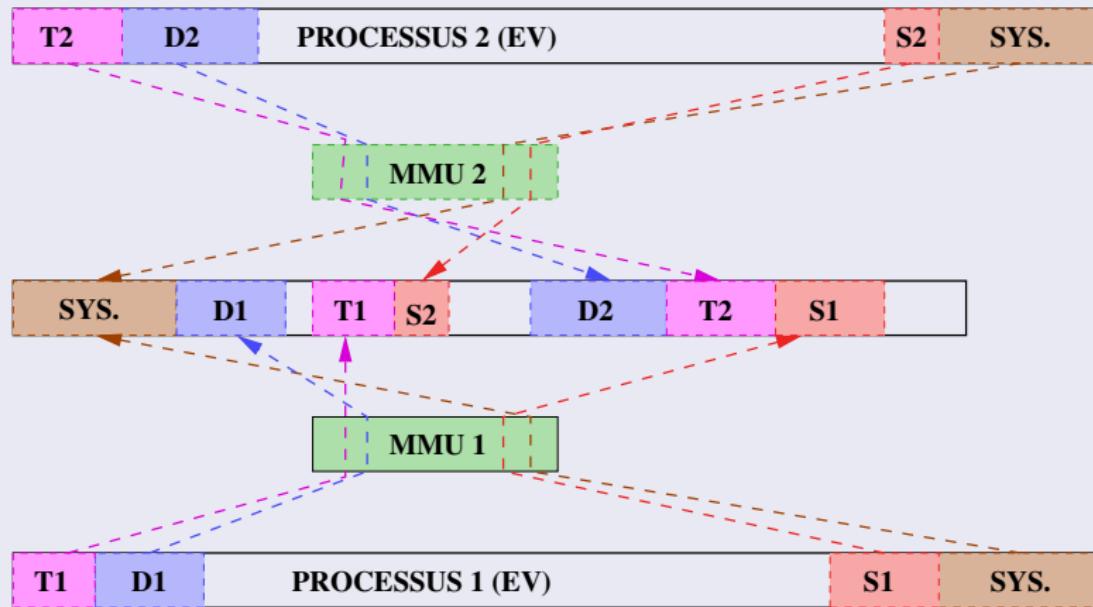
# Quelques configurations : Partage de l'espace système



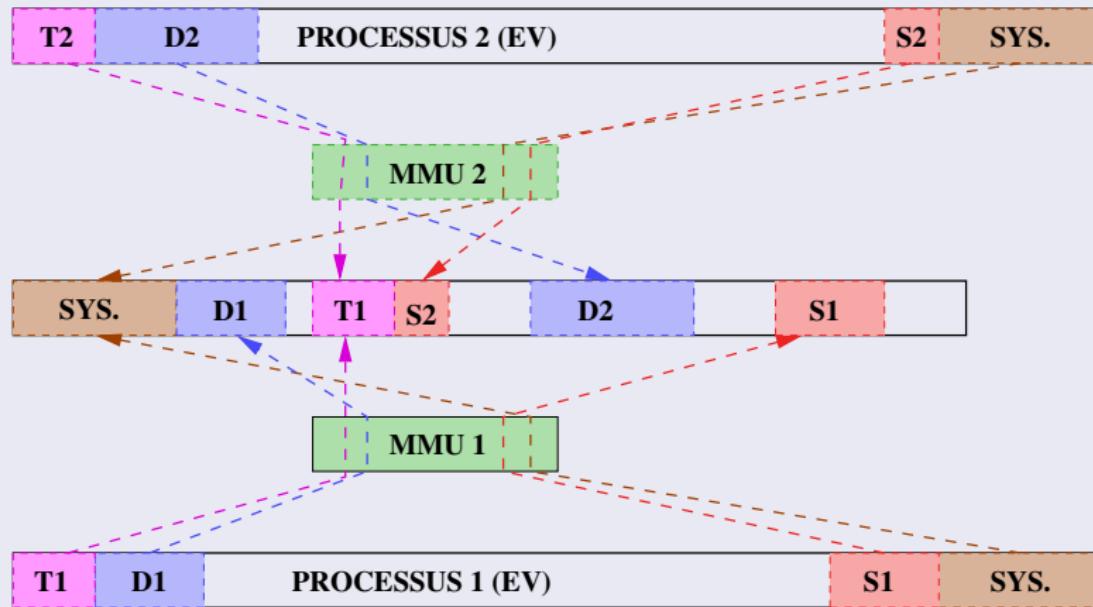
# Quelques configurations : Partage de l'espace système



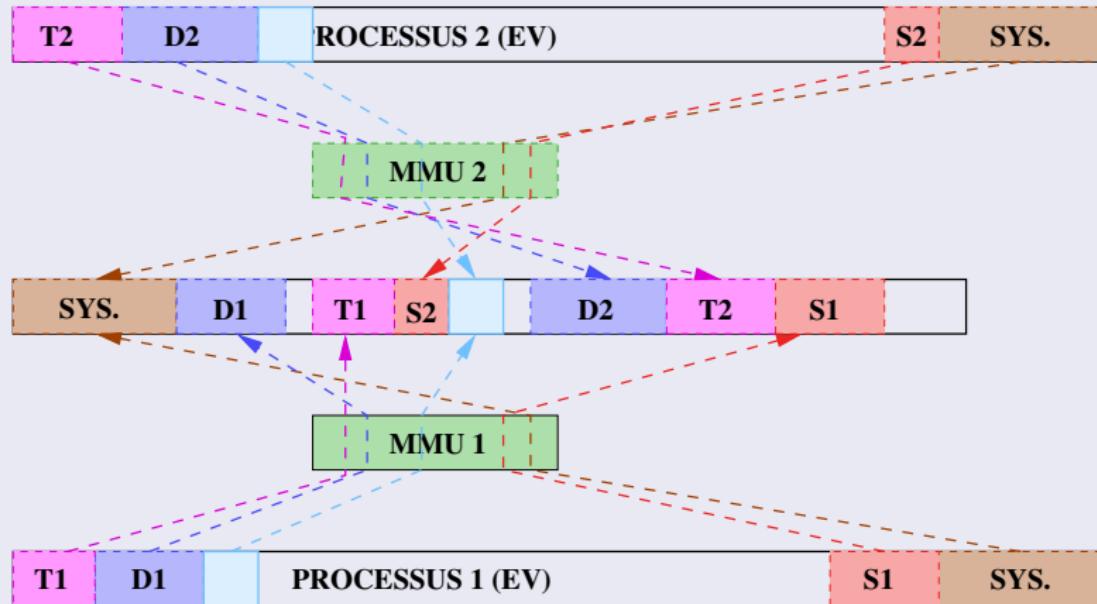
# Quelques configurations : Partage de l'espace système



# Quelques configurations : 2 processus du même exécutable

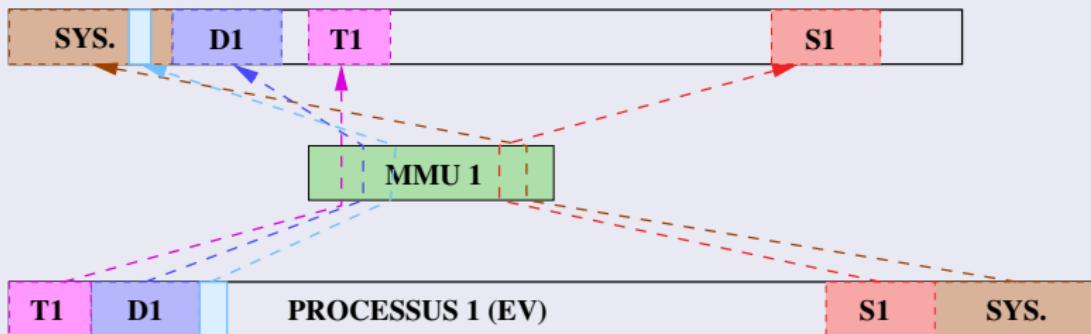


# Quelques configurations : Mémoire partagée



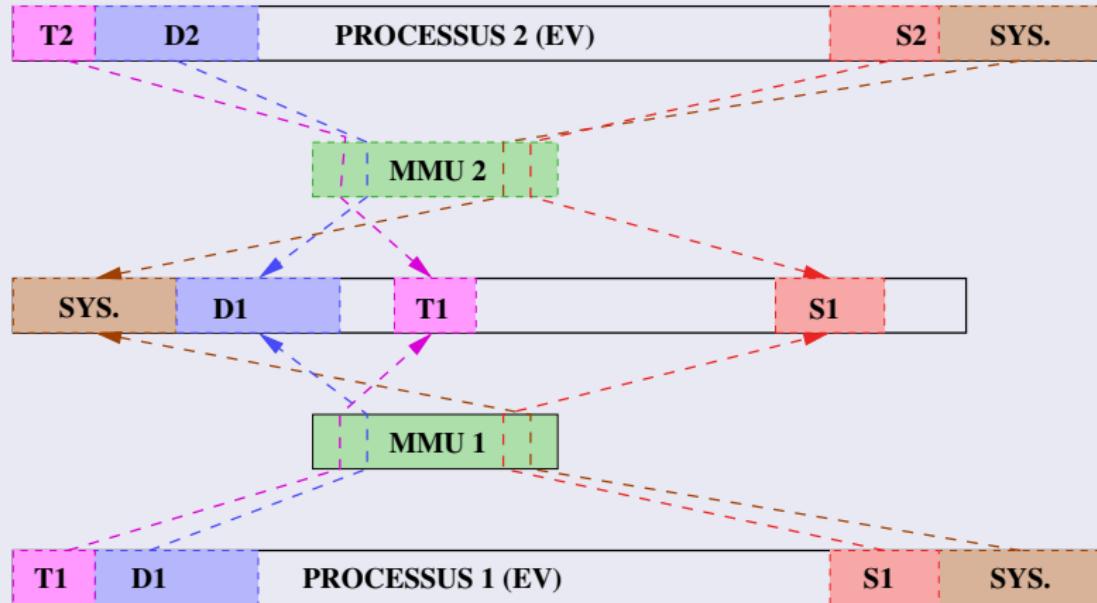
Le segment système et un bout de segment données sont partagés entre les deux processus. Les 2 processus peuvent s'échanger des

# Quelques configurations : Accès direct à un tampon système



Un tampon de donnée système est mappé dans l'espace utilisateur (segment donnée). Le processus en mode utilisateur peut accéder à cette partie de l'espace système.

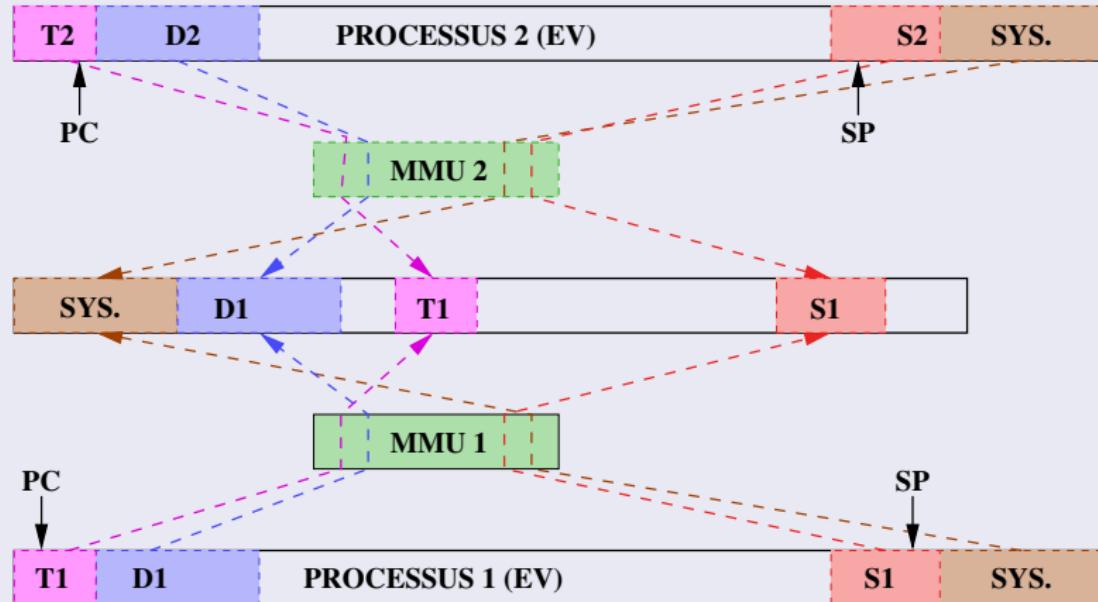
# Quelques configurations : Processus légers



MMU 1 = MMU 2

En quoi ces 2 processus diffèrent ils ?

# Quelques configurations : Processus légers



MMU 1 = MMU 2

En quoi ces 2 processus diffèrent ils ?

# Processus

Unité d'exécution	flux0
PID CWD UID GID0 ...	flux1
ARG0 ARG1 ARG2 ...	flux2
ENV0 ENV1 ENV2 ...	flux3
	...

Unité d'exécution  
Registres et MMU  
Espace virtuel  
Informations diverse  
Paramètres  
Flux

Où sont stockés ces éléments

# Processus

Unité d'exécution				
PC	SP	R0	R1	MMU
				flux0
				flux1
				flux2
				flux3
PID	CWD	UID	GID	0 ...
ARG0	ARG1	ARG2	...	...
ENV0	ENV1	ENV2	...	

Unité d'exécution  
Registres et MMU  
Espace virtuel  
Informations diverses  
Paramètres  
Flux

Où sont stockés ces éléments

# Processus



Unité d'exécution

Registres et MMU  
Espace virtuel

Informations diverse

Paramètres

Flux

Où sont stockés ces éléments

## 4 Appel système

- Organisation
- Format général d'un appel système

# Format général d'un appel système : Prototype

```
1 | extern int errno;  
2 |  
3 | int unAppelSysteme(  
4 |     Tp1 p1,  
5 |     Tp2 p2,  
6 |     ...  
7 | );
```

type retour toujours int

-1 : erreur

$\geq \emptyset$  : ok

$< \emptyset$  : impossible

arguments de 0 à 6

errno  $\geq \emptyset$ , code d'erreur en cas  
d'échec **seulement**

# Format général d'un appel système : Utilisation standard

```
1 #include <stdio.h>
2 #include <string.h> // pour strerror
3 #include <errno.h> // pour errno
4
5 int main( int argc , char*argv [] )
6 {
7     ...
8     if ( unAppelSystème ( ... ) == -1 ) {
9         fprintf(stderr ,
10             "%s : Fatal : unAppelSystème fails : %s\n" ,
11             argv[0] , strerror(errno));
12         return 1; // ou exit(1)
13     }
14     ...
15     return 0;
```

# Format général d'un appel système : Utilisation standard

```
1 #include <stdio.h>
2 #include <string.h> // pour strerror
3 #include <errno.h> // pour errno
4
5 int main( int argc , char*argv [] )
6 {
7     ...
8     if ( unAppelSystème ( ... ) == -1 ) {
9         fprintf(stderr ,
10             "%s : Fatal : unAppelSystème fails : %s\n" ,
11             argv[0] , strerror(errno));
12         return 1; // ou exit(1)
13     }
14     ...
15     return 0;
```

# Format général d'un appel système : Utilisation standard

```
1 #include <stdio.h>
2 #include <string.h> // pour strerror
3 #include <errno.h> // pour errno
4
5 int main( int argc , char*argv [] )
6 {
7     ...
8     if ( unAppelSystème ( ... ) == -1 ) {
9         fprintf( stderr ,
10             "%s : Fatal : unAppelSystème fails : %s\n" ,
11             argv[0] , strerror(errno) );
12         return 1; // ou exit(1)
13     }
14     ...
15     return 0;
```

## 5 Flux

- Algorithmes
- Les flux noyau
- Les flux libc
- Mapping
- Comparaison

# Algorithme : Lecture/écriture

```
1 fd = ouvrir f en lecture  
2 statut = lire(fd, n1, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
3 statut = lire(fd, n2, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

```
1 fd = ouvrir f en écriture  
2 statut = écrire(fd, n1, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
3 statut = écrire(fd, n2, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

**fd** les opérations sur fichier nécessitent un descripteur de flux  
**ouvrir** Crée un descripteur de flux, les modes sont RO, WO ou RW  
**fermer** Libère toutes les allocations associées au flux

# Algorithme : Lecture/écriture

```
1 fd = ouvrir f en lecture  
2 statut = lire(fd, n1, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
3 statut = lire(fd, n2, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

```
1 fd = ouvrir f en écriture  
2 statut = écrire(fd, n1, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
3 statut = écrire(fd, n2, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

**lire** transfère n octets du flux vers un tampon mémoire

**écrire** transfère n octets d'un tampon mémoire vers le flux

**statut** E O F seul lire peut le recevoir

# Algorithme : Lecture/écriture

```
1 fd = ouvrir f en lecture  
2 statut = lire(fd, n1, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
3 statut = lire(fd, n2, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

```
1 fd = ouvrir f en écriture  
2 statut = écrire(fd, n1, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
3 statut = écrire(fd, n2, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

**séquentiel** fd contient en autre un curseur de lecture dans le flux.

lire n octets avance le curseur de n octets dans le flux. (idem pour écrire)

# Algorithme : Lecture/écriture

```
1 fd = ouvrir f en lecture  
2 statut = lire(fd, n1, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
3 statut = lire(fd, n2, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

```
1 fd = ouvrir f en écriture  
2 statut = écrire(fd, n1, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
3 statut = écrire(fd, n2, tamp)  
si statut = ERR alors  
    aller à fin  
fin si  
... ...  
fin fermer fd
```

bloquant

**ouvrir** Non sur un fichier régulier local, possible sur d'autres flux

# Algorithme : Positionnement

Positionnement dans un fichier f.

- 1 fd = ouvrir f en lecture
- 2 statut = lire(fd, n<sub>1</sub>, tamp)  
si statut = ERR ou EOF alors  
    aller à fin  
fin si
- 4 statut = déplace(fd, m,  
POS)  
si statut = ERR alors  
    aller à fin  
fin si
- 5 statut = lire(fd, n<sub>2</sub>, tamp)  
si statut = ERR ou EOF alors  
    aller à fin

**ouvrir** Crée un descripteur de flux,  
les modes sont RO, WO ou  
RW.

**déplace** Déplace le curseur du flux  
de m octets par rapport à  
une position fixe (POS).  
⇒ POS = début ou fin ou  
curseur.

⇒ n'est possible que sur des  
flux le supportant (ex :  
fichier régulier).

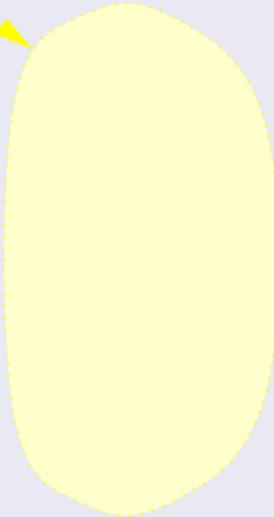
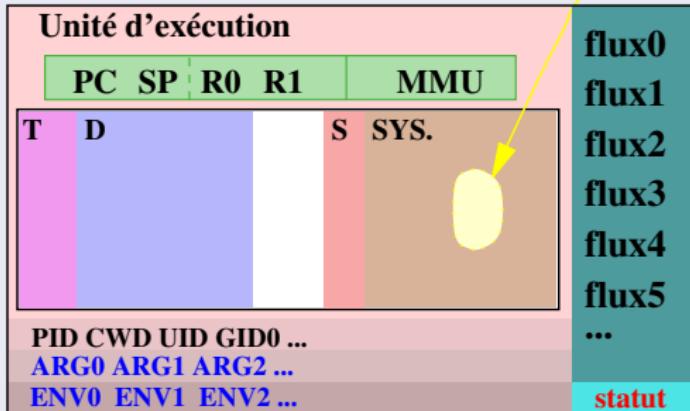
# Algorithme : Quizz

- ① Une lecture sur un flux avec un statut OK garantit que les données reçues sont bonnes.
- ② Une écriture sur un flux avec un statut OK garantit que les données sont arrivées à destination.
- ③ Une écriture sur un flux associé à un fichier régulier ne peut pas renvoyer un statut ERR.
- ④ Après une lecture sur un flux avec un statut EOF, une seconde lecture donne toujours EOF.
- ⑤ Sur un flux sans erreur possible, on obtiendra toujours un statut EOF après un certain nombre de lectures.
- ⑥ Il n'est pas utile de fermer les flux que l'on utilise plus puisque le noyau le fera à la terminaison du processus.

## 5 Flux

- Algorithmes
- **Les flux noyau**
- Les flux libc
- Mapping
- Comparaison

# Flux : Descripteurs de flux

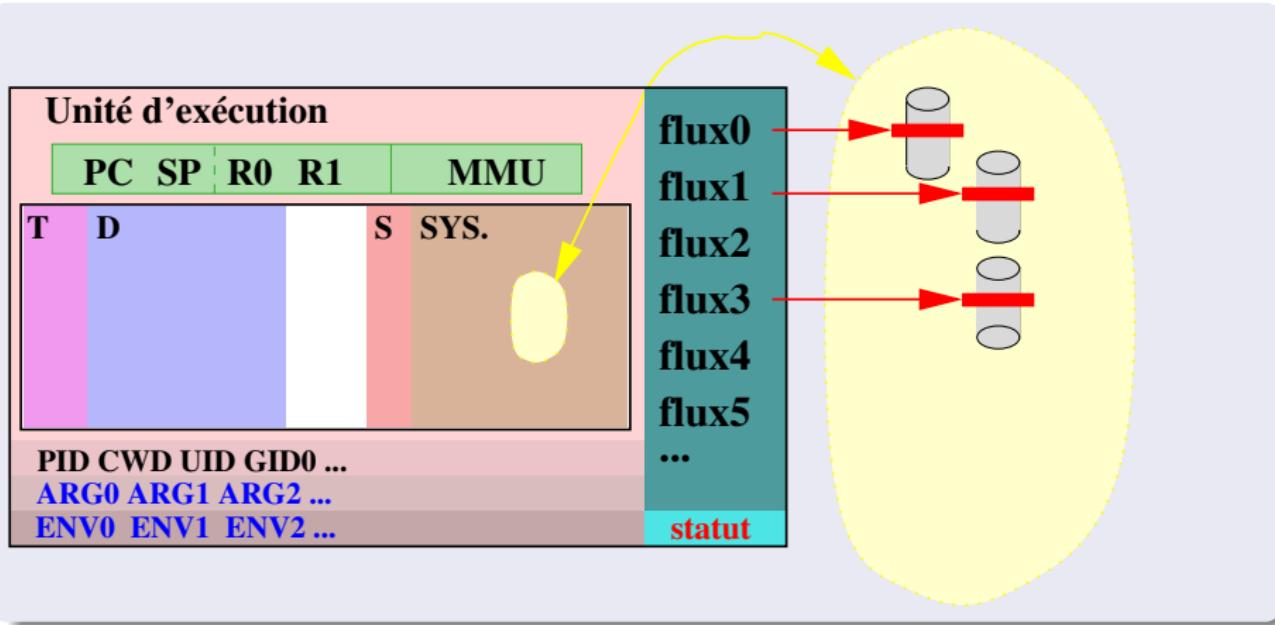


type Entier

correspondance Le  $i^{ième}$  flux du processus

exemple flux 0, 1, 3 définis, 2, 4, ... non définis

# Flux : Descripteurs de flux

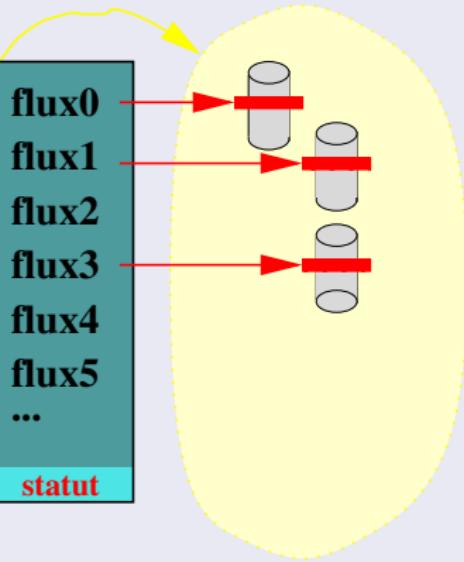
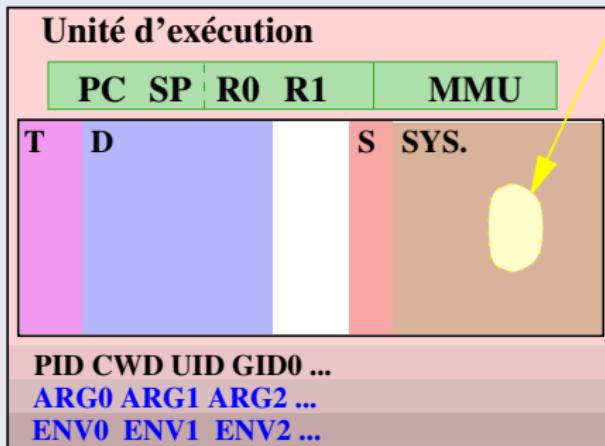


type Entier

correspondance Le  $i^{ième}$  flux du processus

exemple flux 0, 1, 3 définis, 2, 4, ... non définis

# Flux : Ouverture



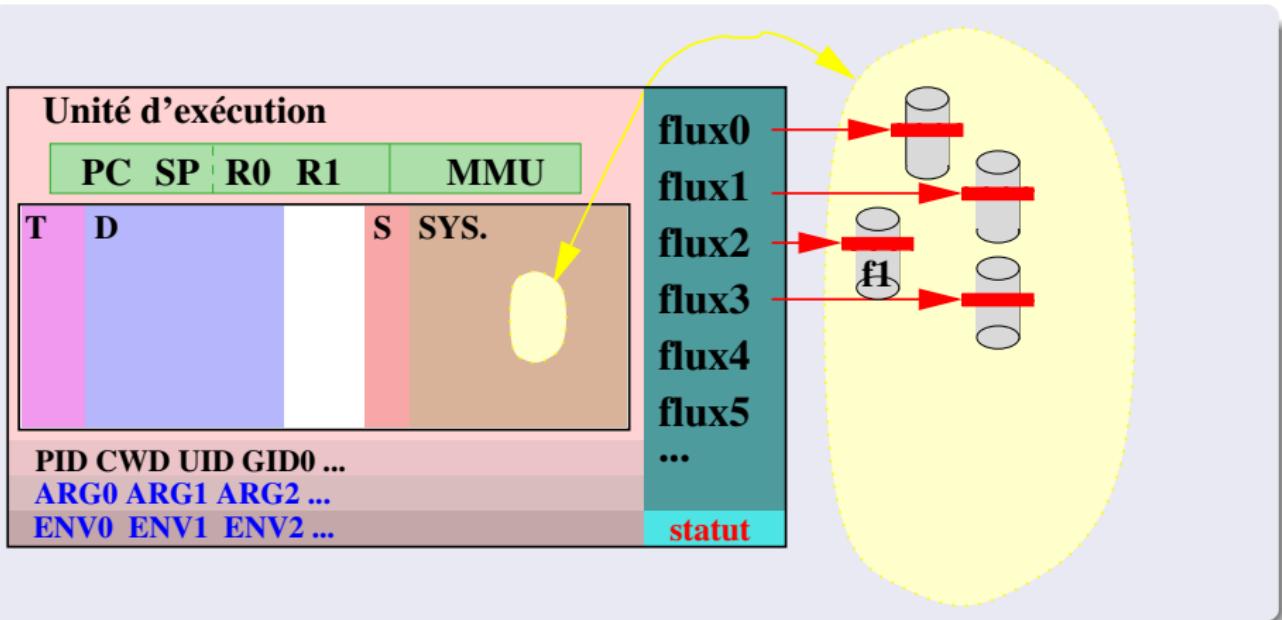
Synopsis sans création int open(const char\*f, int flags);

Synopsis avec création int open(const char\*f, int flags, mode\_t mode);

Fonction Associe un descripteur de flux au fichier f et le renvoie.

Retour Le descripteur de flux ou -1.

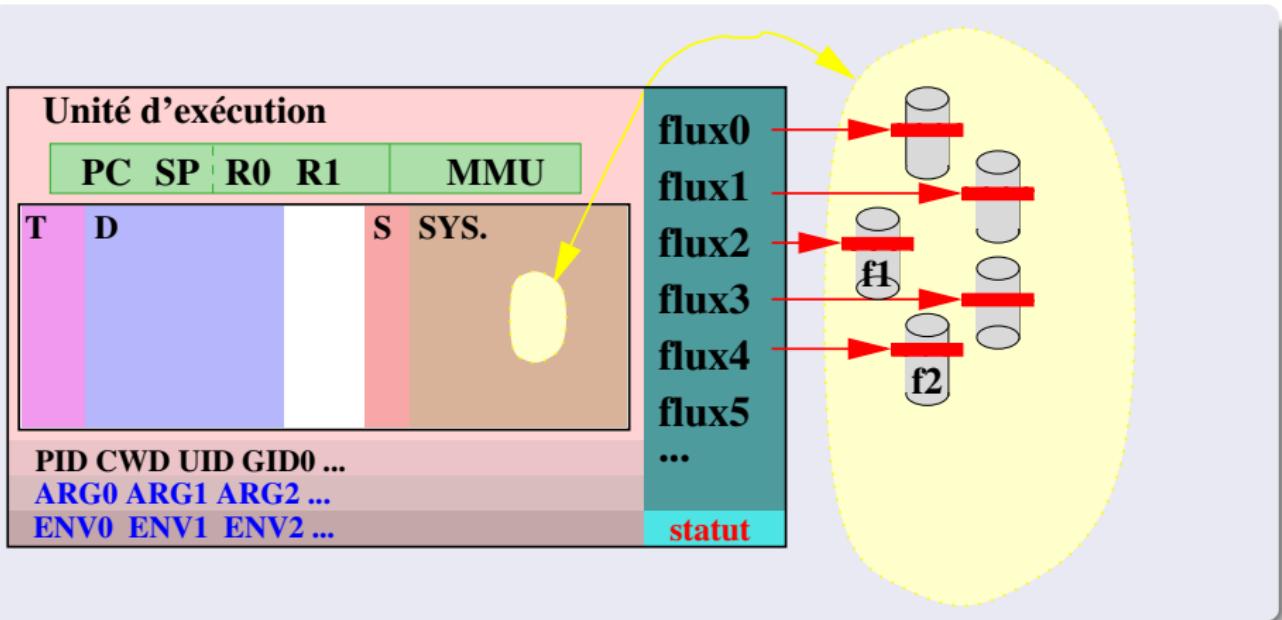
# Flux : Ouverture



Exemple 1 `open("f1", O_RDONLY)` recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  2.

Exemple 2 `open("f2", O_WRONLY)` recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  4.

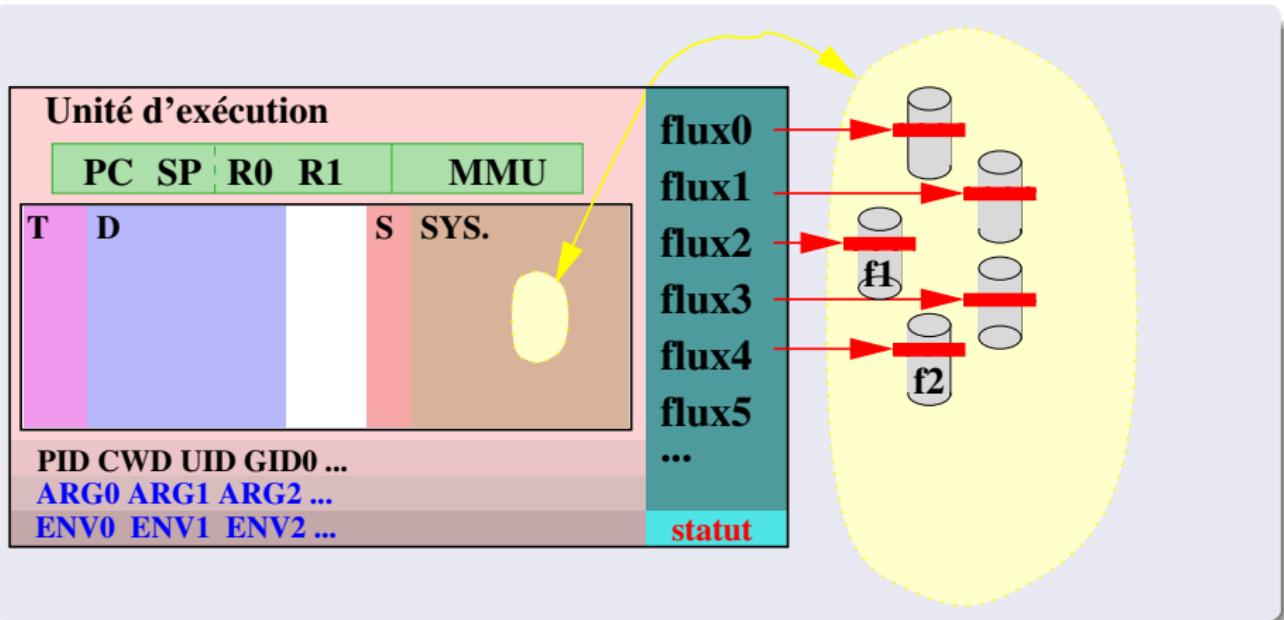
# Flux : Ouverture



Exemple 1 `open("f1", O_RDONLY)` recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  2.

Exemple 2 `open("f2", O_WRONLY)` recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  4.

# Flux : Ouverture

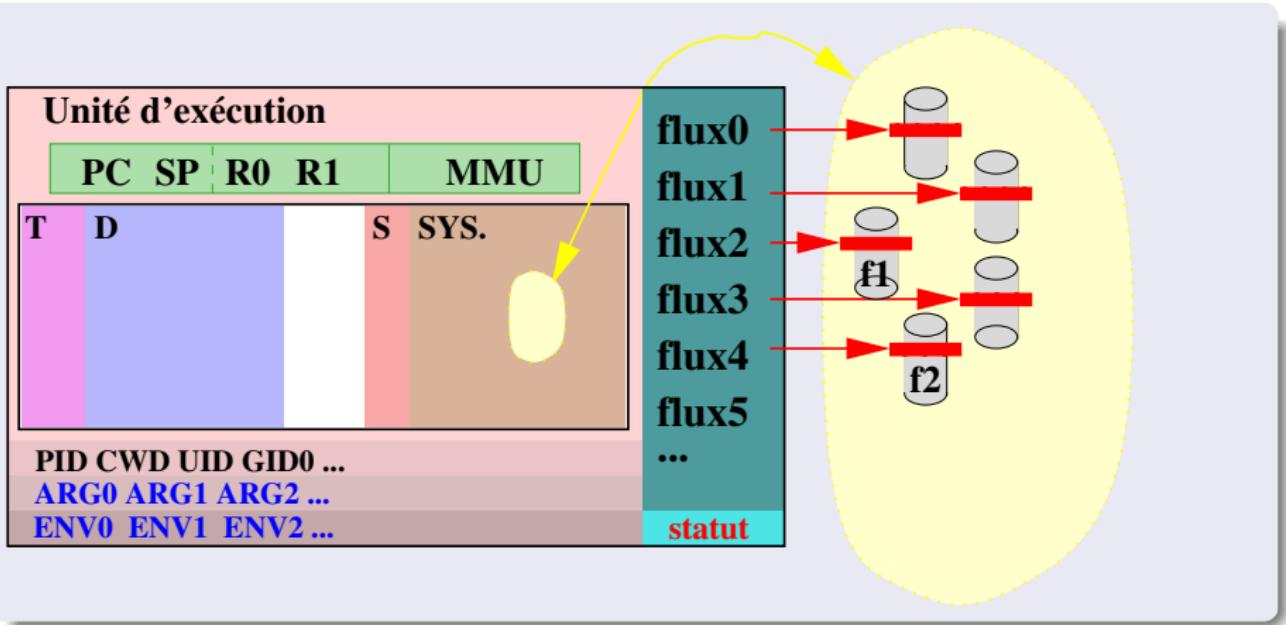


Synopsis avec création int open(const char\*f, int flags, mode\_t mode);

Flags mode O\_RDONLY, O\_WRONLY, O\_RDWR

Flags autres O\_TRUNC, O\_APPEND, O\_CREAT

# Flux : Fermeture

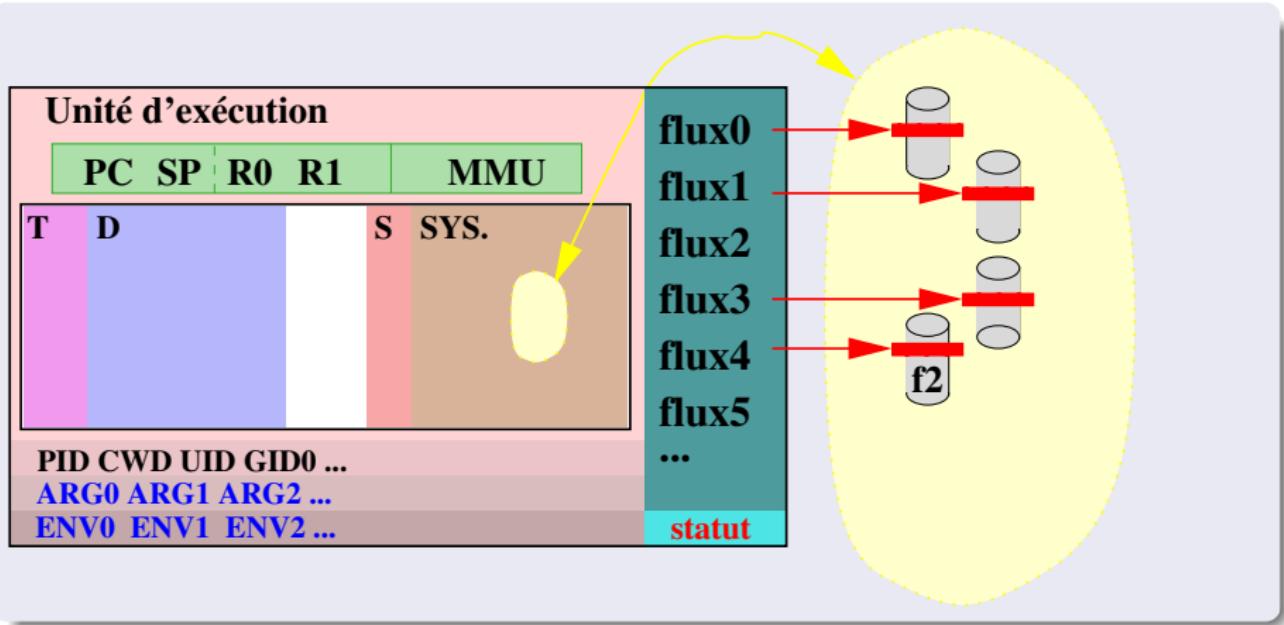


Synopsis `int close(int fd);`

Fonction Désalloue le descripteur de flux fd.

Exemple `close(2)` le descripteur 2 est libre

# Flux : Fermeture

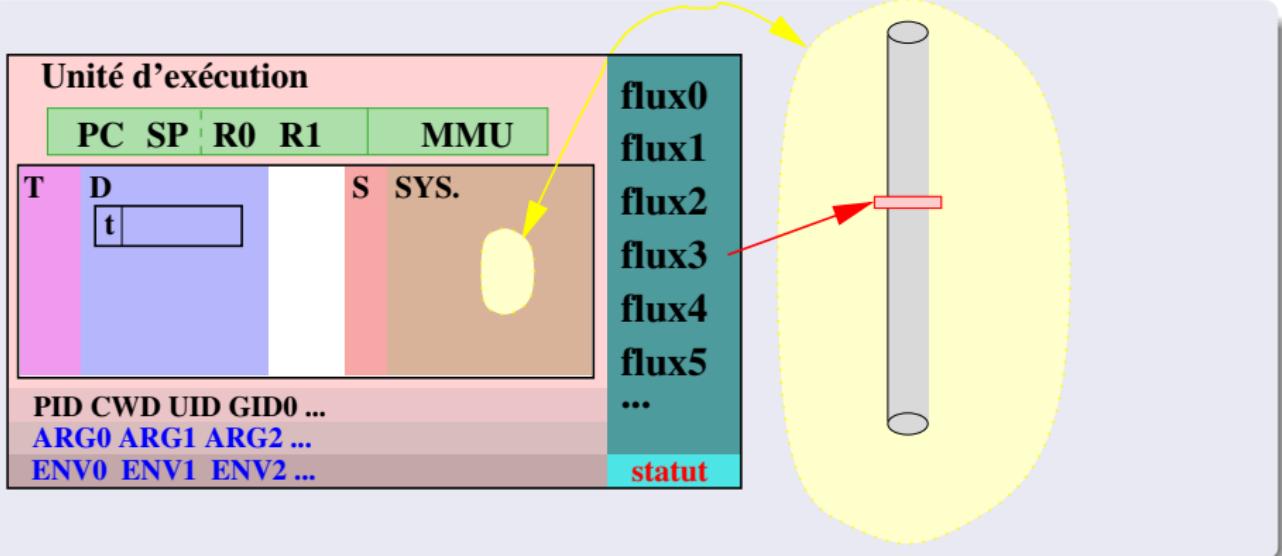


Synopsis `int close(int fd);`

Fonction Désalloue le descripteur de flux `fd`.

Exemple `close(2)` le descripteur 2 est libre

# Flux : Lecture



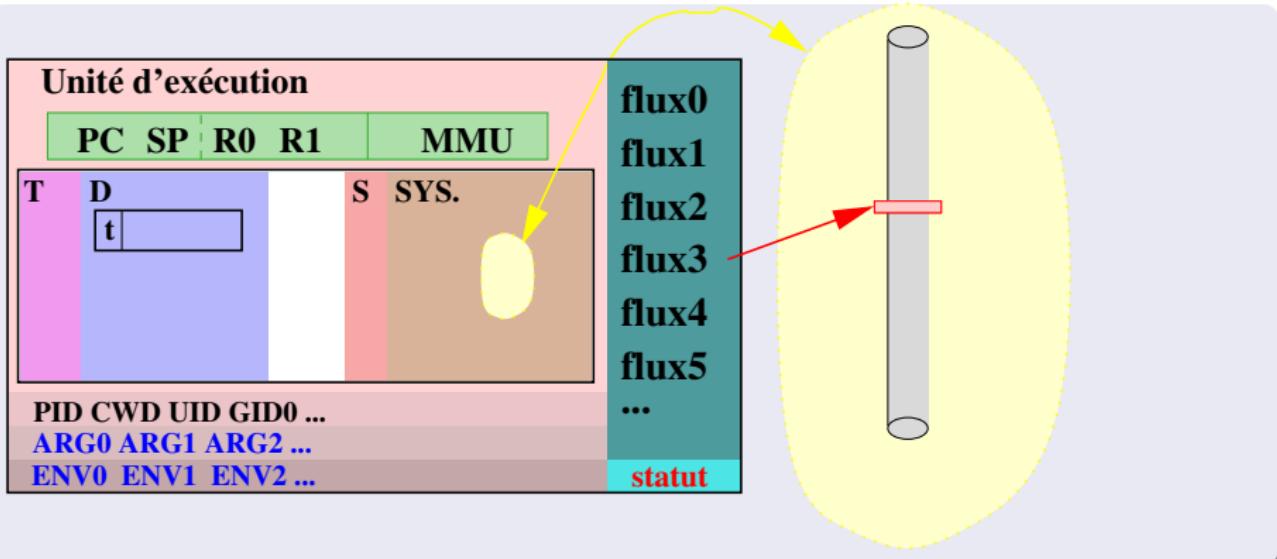
Synopsis `size_t read(int fd, void *buf, size_t count);`

Fonction Essaye de lire `count` octets du flux `fd` dans le tampon mémoire `buf` et incrémente le curseur de `count`.

Retour Le nombre d'octets lus.

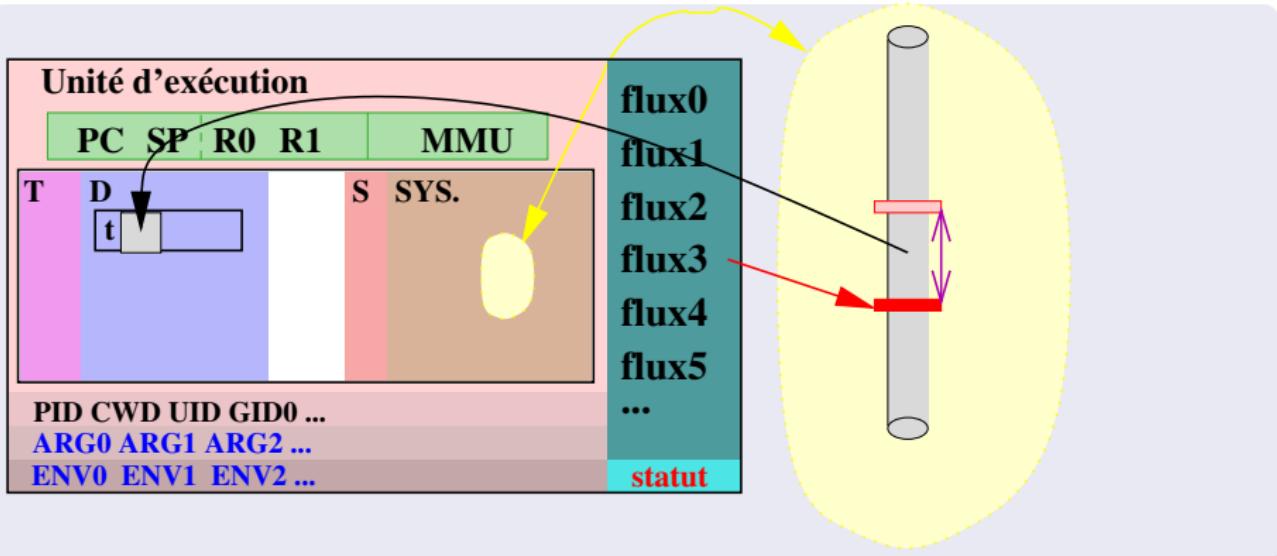
E.O.F Un retour de la valeur `Ø`.

# Flux : Lecture



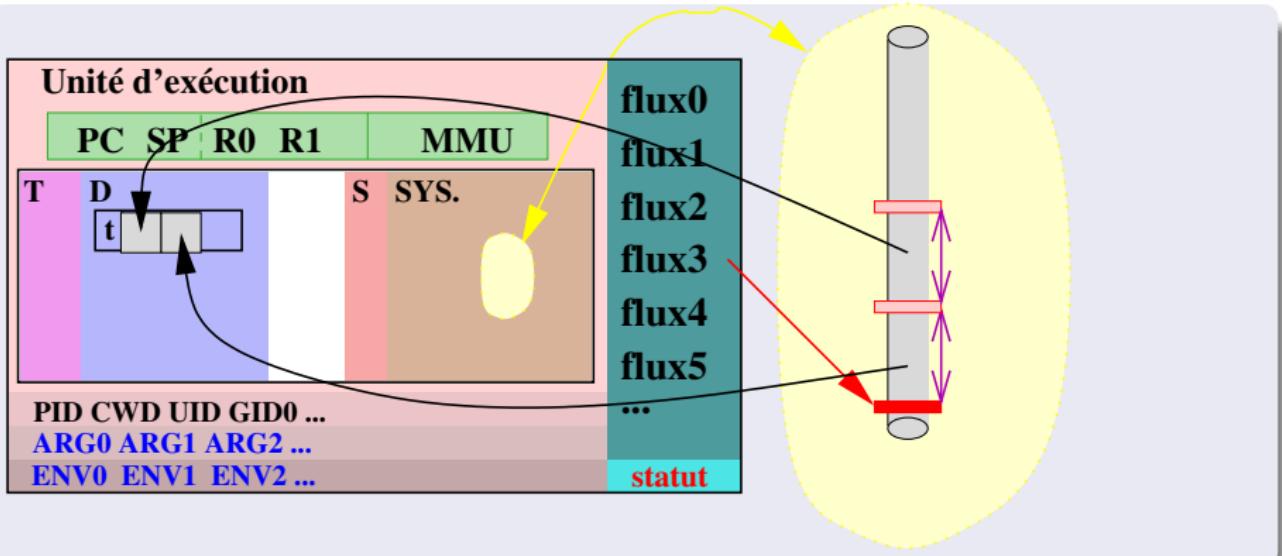
Exemple nbl=read(3,t,10); nbl=read(3,t+10,10);

# Flux : Lecture



Exemple nbl=read(3,t,10); nbl=read(3,t+10,10);

# Flux : Lecture



Exemple nbl=read(3,t,10); nbl=read(3,t+10,10);

# Flux : Écriture

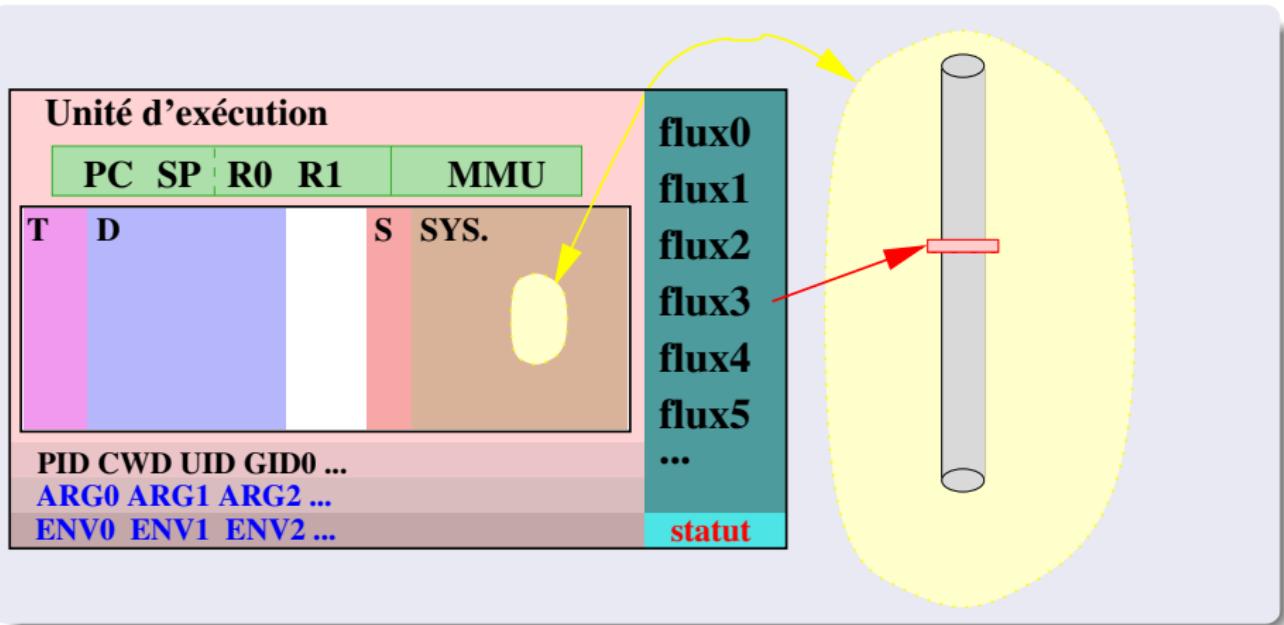
**Synopsis** `size_t write(int fd, void *buf, size_t count);`

**Fonction** Essaye de d'écrire `count` octets du flux tampon mémoire `buf` dans le flux `fd` et incrémenté le curseur de `count`.

**Retour** En cas de succès, le nombre d'octets écrits sinon -1 et `errno` est mis à jour.

**Exemple** `nbe=write(3,t,10);`

# Flux : Positionnement



Synopsis `int lseek(int fd, off_t offset, int whence);`

Fonction Positionne le curseur du flux fd à offset octets de la position whence.

Retour La nouvelle position de curseur en cas de succès, sinon -1 et















































































































































































































































































































































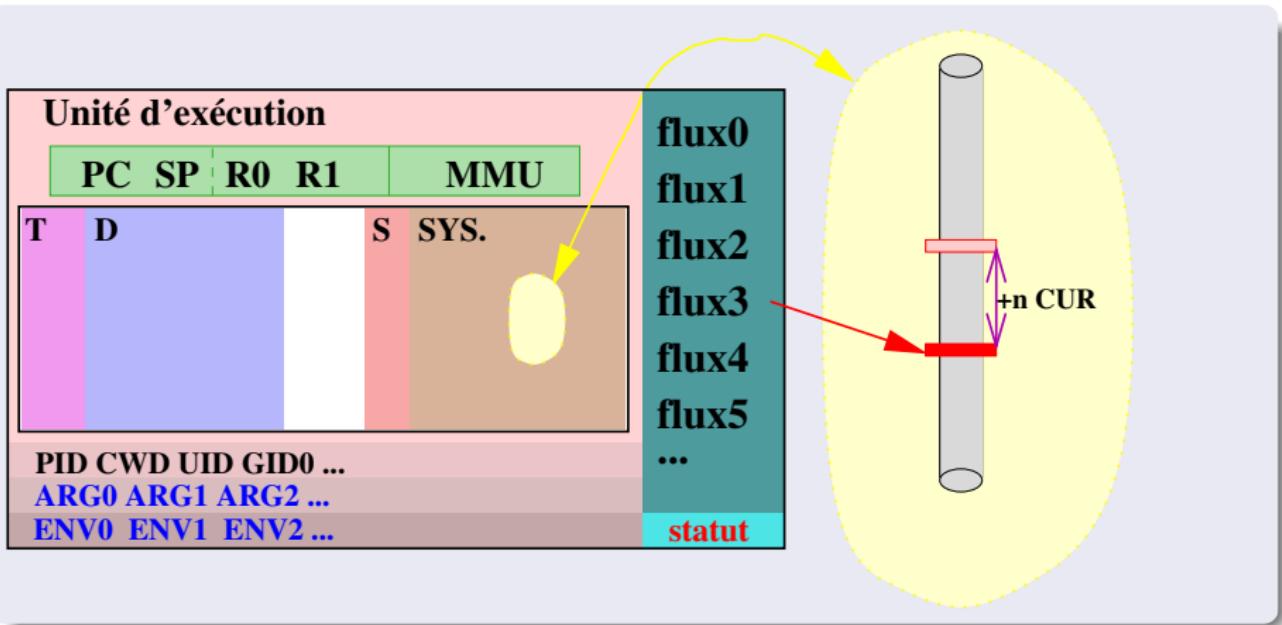








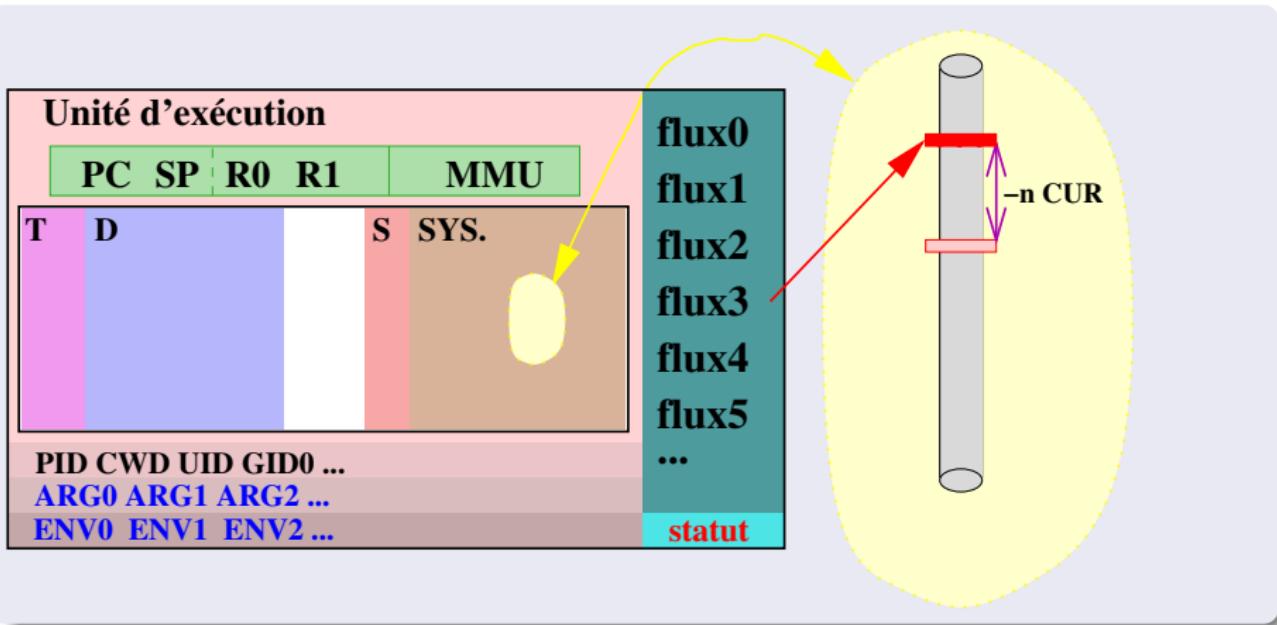
# Flux : Positionnement



Exemple "pos=lsseek(3,+10,SEEK\_CUR)" avance à partir de la position courante.

Exemple "pos=lsseek(3,-10,SEEK\_CUR)" recule à partir de la position courante.

# Flux : Positionnement



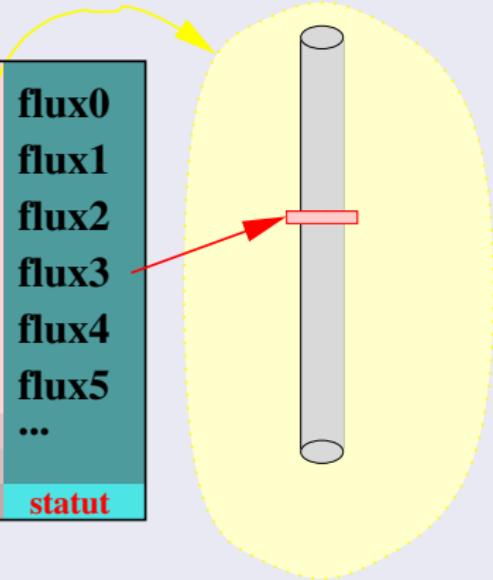
Exemple "pos=lsseek(3,+10,SEEK\_CUR)" avance à partir de la position courante.

Exemple "pos=lsseek(3,-10,SEEK\_CUR)" recule à partir de la position courante.

# Flux : Positionnement

## Unité d'exécution

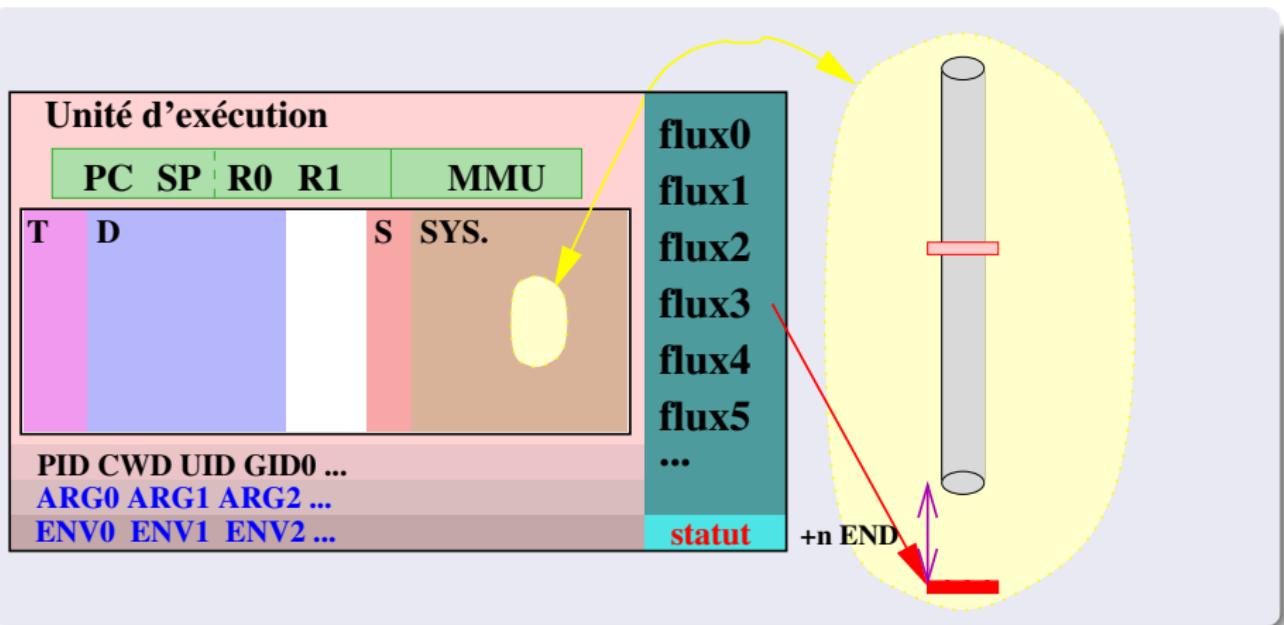
PC	SP	R0	R1	MMU
T	D		S	SYS.
PID CWD UID GID0 ...				
ARG0 ARG1 ARG2 ...				
ENV0 ENV1 ENV2 ...				statut



Exemple "pos=lsseek(3,+10,SEEK\_END)" avance à partir de la fin.

Exemple "pos=lsseek(3,-10,SEEK\_END)" recule à partir de la fin.

# Flux : Positionnement



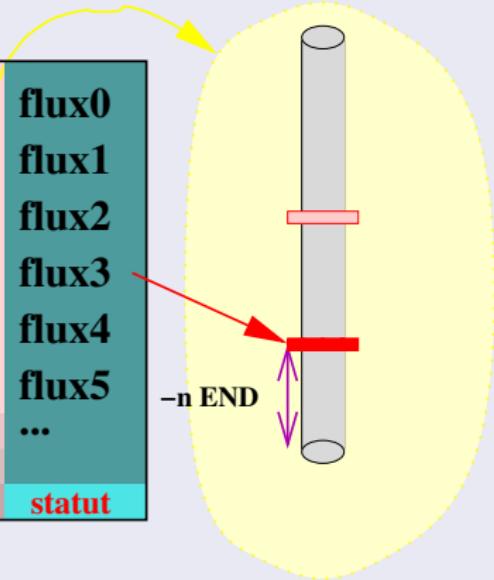
Exemple "pos=lsseek(3,+10,SEEK\_END)" avance à partir de la fin.

Exemple "pos=lsseek(3,-10,SEEK\_END)" recule à partir de la fin.

# Flux : Positionnement

## Unité d'exécution

PC	SP	R0	R1	MMU
T	D		S	SYS.
PID CWD UID GID0 ...				
ARG0 ARG1 ARG2 ...				
ENV0 ENV1 ENV2 ...				statut



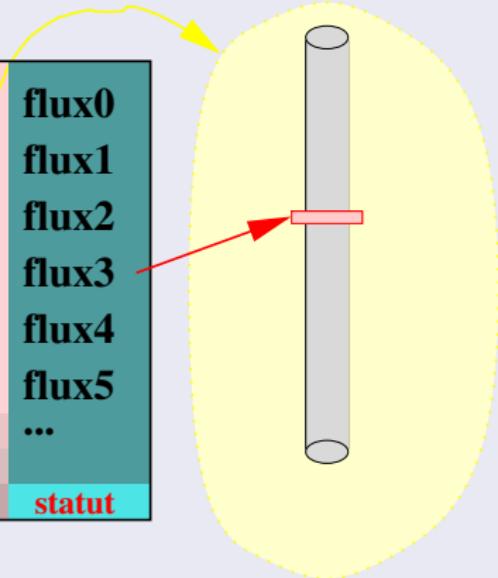
Exemple "pos=lsseek(3,+10,SEEK\_END)" avance à partir de la fin.

Exemple "pos=lsseek(3,-10,SEEK\_END)" recule à partir de la fin.

# Flux : Positionnement

## Unité d'exécution

PC	SP	R0	R1	MMU
T	D		S	SYS.
PID	CWD	UID	GID	0 ...
ARG0	ARG1	ARG2	...	
ENV0	ENV1	ENV2	...	statut



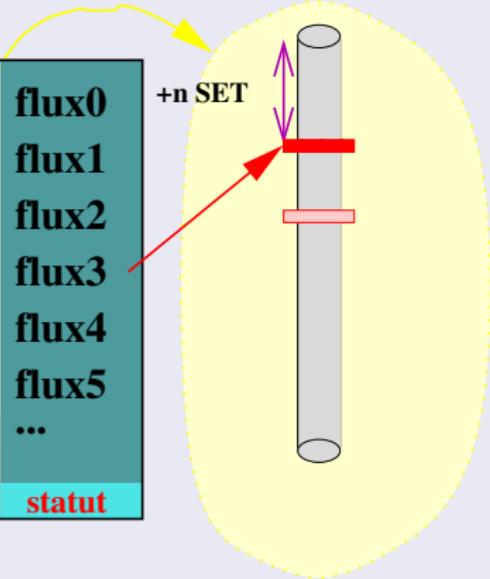
Exemple "pos=lsseek(3,+10,SEEK\_SET)" avance à partir du début.

Exemple "pos=lsseek(3,-10,SEEK\_SET)"  $\Rightarrow -1$

# Flux : Positionnement

## Unité d'exécution

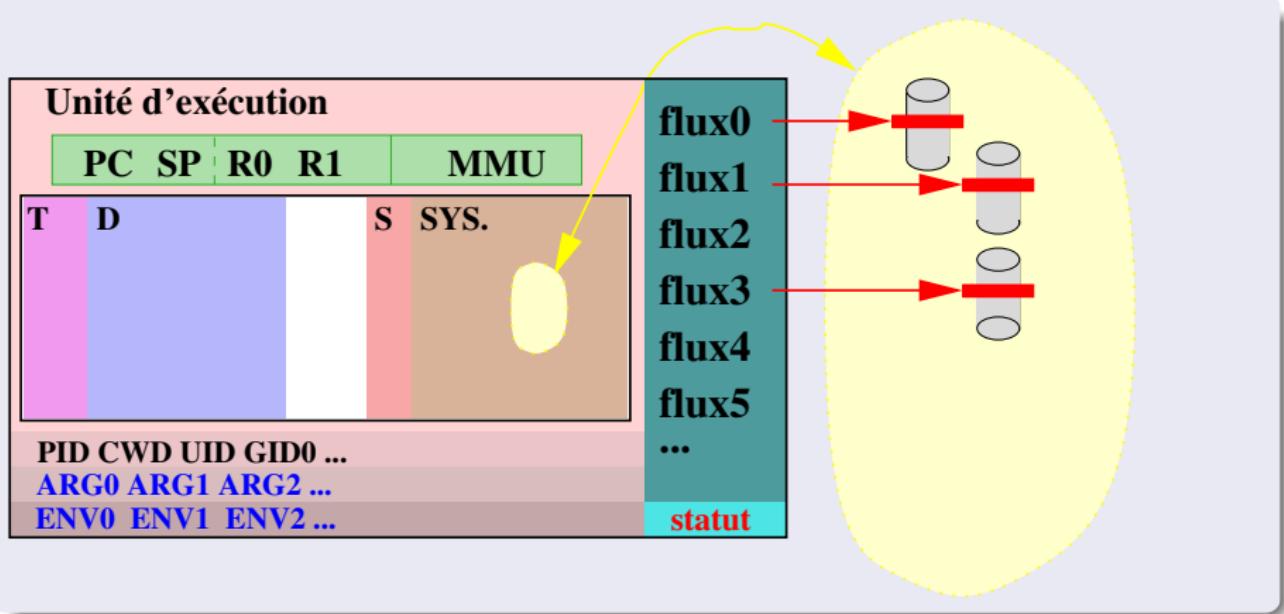
PC	SP	R0	R1	MMU
T	D		S	SYS.
PID CWD UID GID0 ...				
ARG0 ARG1 ARG2 ...				
ENV0 ENV1 ENV2 ...				statut



Exemple "pos=lsseek(3,+10,SEEK\_SET)" avance à partir du début.

Exemple "pos=lsseek(3,-10,SEEK\_SET)"  $\Rightarrow -1$

# Flux : Duplication de descripteurs



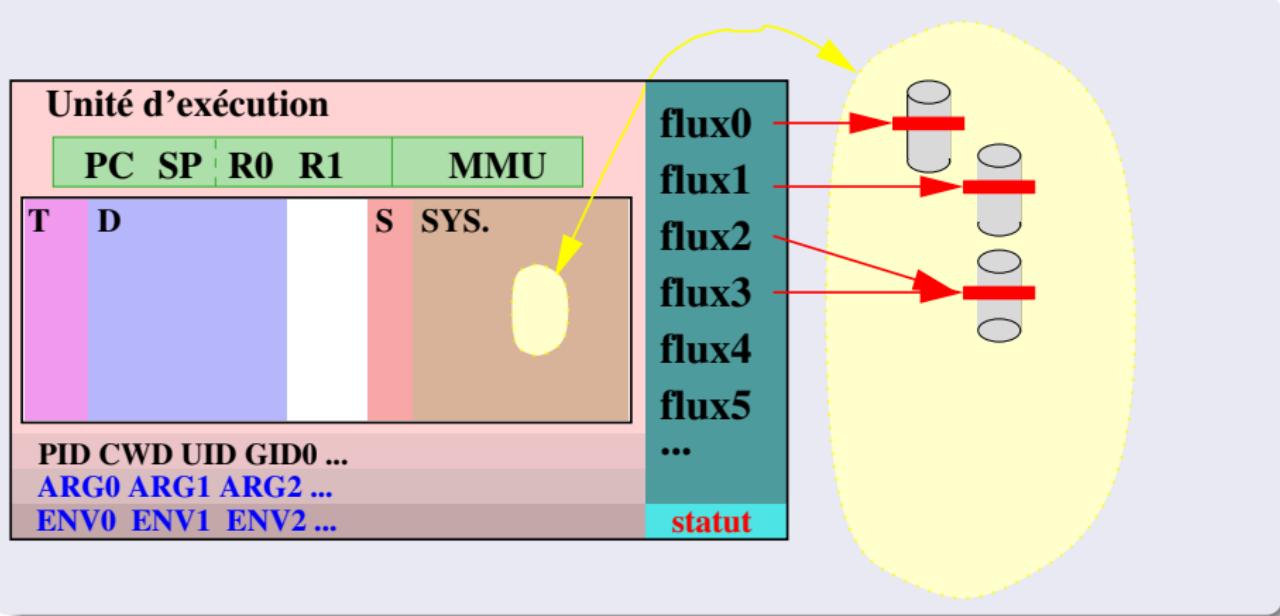
Synopsis `int dup(int fd);`

Fonction Duplique le descripteur de flux fd et le renvoie.

Exemple "`dup(3)`" Recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  2.

Les descripteurs 2 et 3 accèdent le même flux.

# Flux : Duplication de descripteurs



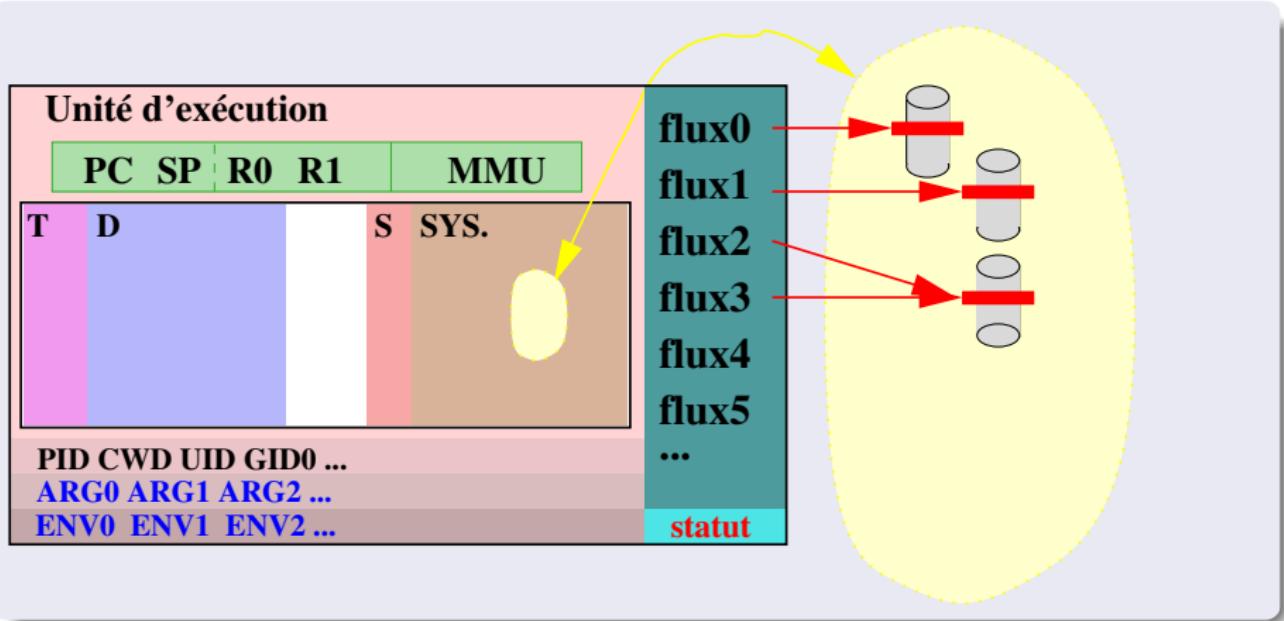
Synopsis `int dup(int fd);`

Fonction Duplique le descripteur de flux fd et le renvoie.

Exemple "`dup(3)`" Recherche le 1<sup>er</sup> fd libre  $\Rightarrow$  2.

Les descripteurs 2 et 3 accèdent le même flux.

# Flux : Duplication de descripteurs



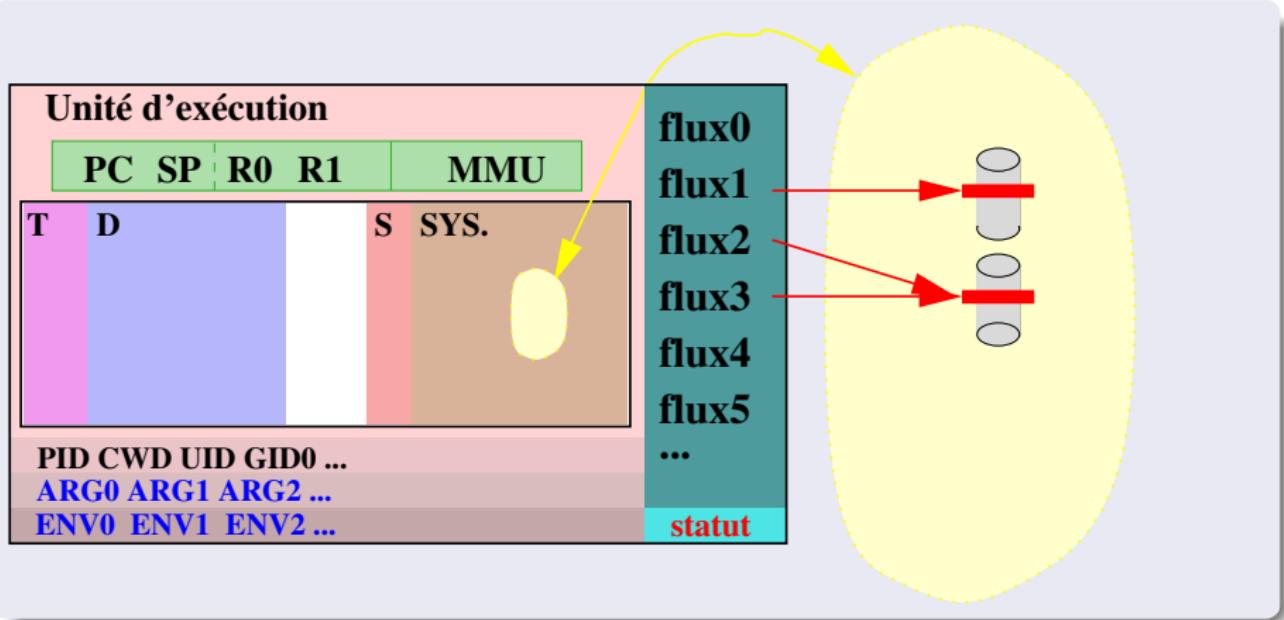
Synopsis `int dup2(int oldfd, int newfd)`

Fonction Fait que le descripteur de flux `newfd` accède le même flux que `oldfd`. Si `newfd` était ouvert, il est fermé.

Exemple "`dup2(1,0)`" Les descripteurs 0 et 1 accèdent le même flux.



# Flux : Duplication de descripteurs



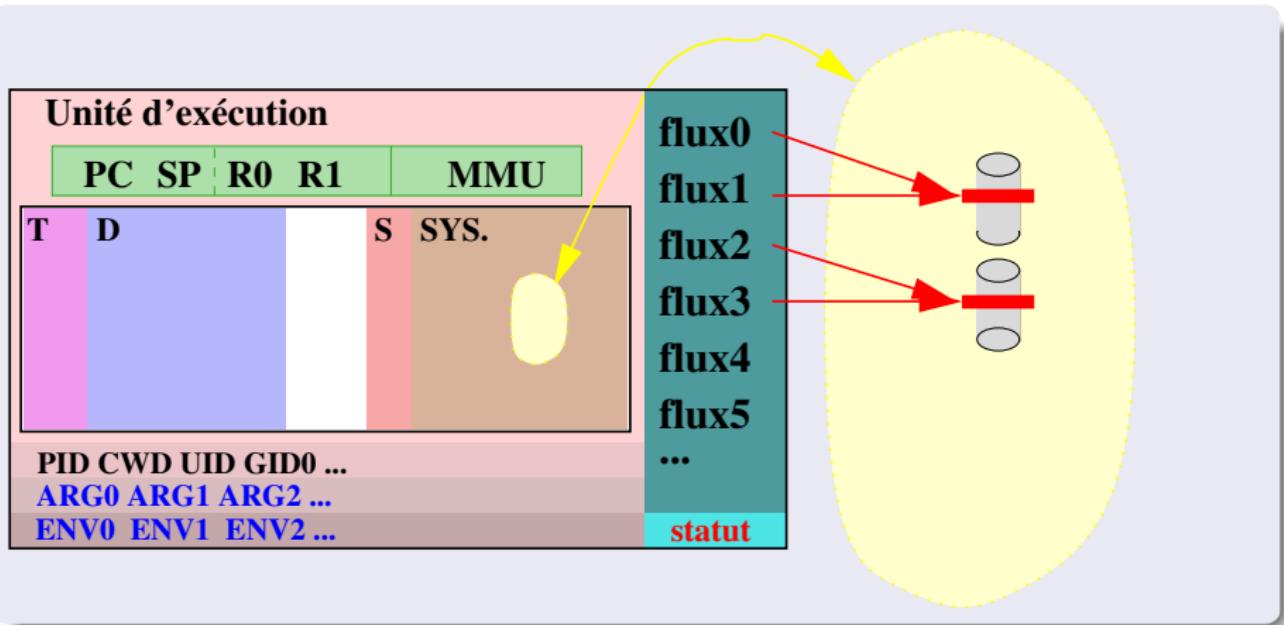
**Synopsis** `int dup2(int oldfd, int newfd)`

**Fonction** Fait que le descripteur de flux `newfd` accède le même flux que `oldfd`. Si `newfd` était ouvert, il est fermé.

**Exemple** "dup2(1,0)" Les descripteurs 0 et 1 accèdent le même flux.



# Flux : Duplication de descripteurs



Synopsis `int dup2(int oldfd, int newfd)`

Fonction Fait que le descripteur de flux newfd accède le même flux que oldfd. Si newfd était ouvert, il est fermé.

Exemple "dup2(1,0)" Les descripteurs 0 et 1 accèdent le même flux.



# Exemple

Écrivez un programme à deux arguments src et dest. Il considère src et dest comme 2 chemins de fichiers.

Il crée ou érase le fichier dest avec au plus les  $n*10^{i\text{ème}}$  octets de src, n allant de 0 à 9 inclus.

# Exemple : Header et teste du nombre d'arguments

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <string.h>
7 #include <errno.h>
8
9 int main(int argc, char*argv[])
10 {
11     int i;
12     if ( argc!=3 ) {
13         fprintf( stderr,
14             "%s: Fatal : %d mauvais nb d'args (2 attendus)\n",
15             argv[0], argc);
16         return 1; // ou exit(1)
17 }
```

## Exemple : Ouverture des flux

```
20 int fsrc;
21 if ( (fsrc=open(argv[1],O_RDONLY))==-1 ) {
22     fprintf(stderr,
23             "%s : Fatal : pb ouverture %s en lecture : %s\n",
24             argv[0], argv[1], strerror(errno));
25     return 1;
26 }
27 int fdes;
28 if ( (fdes=open(argv[2],
29                  O_WRONLY|O_TRUNC|O_CREAT,
30                  0666))==-1 ) {
31     fprintf(stderr,
32             "%s : Fatal : pb ouverture %s en ecriture : %s\n",
33             argv[0], argv[2], strerror(errno));
34     return 1;
35 }
```

## Exemple : Lecture écriture, méthode 1

```
38     for ( i=0 ; i<10 ; i+=1) {
39         char c;
40         int status = read( fsrc ,&c ,1 );
41         if ( status == -1 ) {
42             fprintf( stderr ,
43                 "%s : Fatal : pb lecture %s : %s \n" ,
44                 argv [0] , argv [1] , strerror( errno ) );
45             return 1 ;
46         }
47         if ( status == 0 ) break ;
48
49         status = write( fdes ,&c ,1 );
50         if ( status == -1 || status == 0 ) {
51             fprintf( stderr ,
52                 "%s : Fatal : pb ecriture %s : %s \n" ,
53                 argv [0] , argv [2] , strerror( errno ) );
54             return 1 ;
```

## Exemple : Lecture écriture, méthode 1

```
56     if ( lseek( fsrc ,9 ,SEEK_CUR)==-1 ) {  
57         fprintf( stderr ,  
58             "%s : Fatal : pb\u00e0 avancee\u00e0 ds\u00e0 : %s\n" ,  
59             argv[0] ,argv[1] ,strerror(errno));  
60         return 1;  
61     }  
62 }
```

# Exemple : Lecture écriture, méthode 2

```
38 for (i=0 ; i<10 ; i+=1) {  
39     char c;  
40     if ( lseek(fsrc , i*10 ,SEEK_SET)==-1 ) {  
41         fprintf( stderr ,  
42             "%s : Fatal : pb lseek %s : %s" ,  
43             argv[0] , argv[1] , strerror(errno));  
44         return 1;  
45     }  
46     int status = read(fsrc ,&c,1);  
47     if (status == -1) {  
48         fprintf( stderr ,  
49             "%s : Fatal : pb lecture %s : %s\n" ,  
50             argv[0] , argv[1] , strerror(errno));  
51         return 1;  
52     }  
53     if (status == 0) break;  
54  
55     status = write(fdes ,&c,1);  
56     if (status == -1 || status == 0) {  
57         fprintf( stderr ,  
58             "%s : Fatal : pb ecriture %s : %s\n" ,  
59             argv[0] , argv[2] , strerror(errno));  
60         return 1;  
61     }  
62 }
```

## Exemple : Lecture écriture, méthode 3

```
38  for ( i=0 ; i<10 ; i+=1) {
39      char t[10];
40      int status = read(fsdc,t,10);
41      if (status == -1) {
42          fprintf(stderr,
43                  "%s : Fatal : pb lecture %s : %s\n",
44                  argv[0],argv[1],strerror(errno));
45          return 1;
46      }
47      if (status == 0) break;
48
49      status = write(fdes,&t[0],1);
50      if (status == -1 || status == 0) {
51          fprintf(stderr,
52                  "%s : Fatal : pb ecriture %s : %s\n",
53                  argv[0],argv[2],strerror(errno));
54          return 1;
```

## Exemple : Terminaison

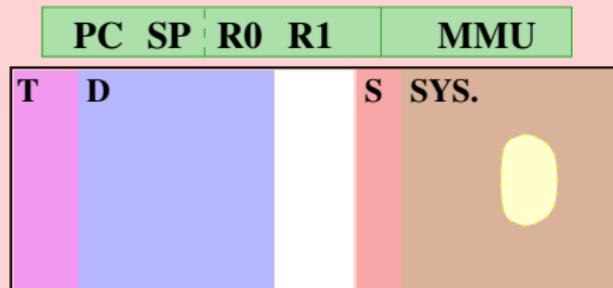
```
57     close( fsrc );  close( fdes );
58     return 0;
59 }
```

## 5 Flux

- Algorithmes
- Les flux noyau
- **Les flux libc**
- Mapping
- Comparaison

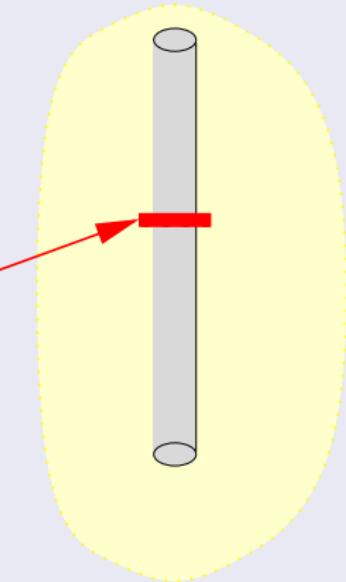
# Flux libc : Type FILE et principe

## Unité d'exécution



**flux0**  
**flux1**  
**flux2**  
**flux3**  
**flux4**  
**flux5**  
...

**statut**



type FILE\* : int fd, void\* buf, ...

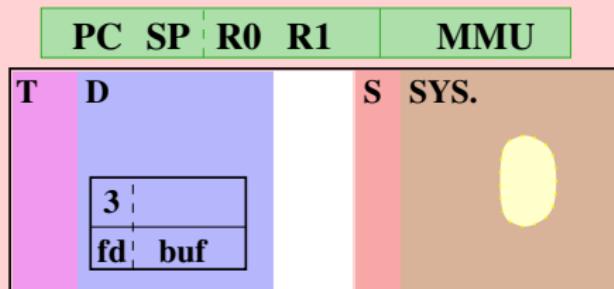
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

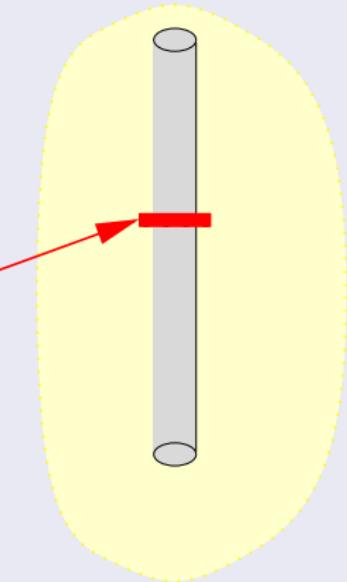
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

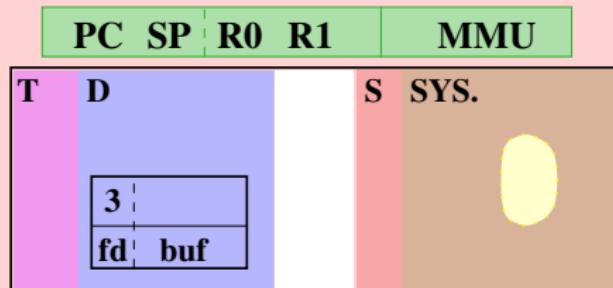
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

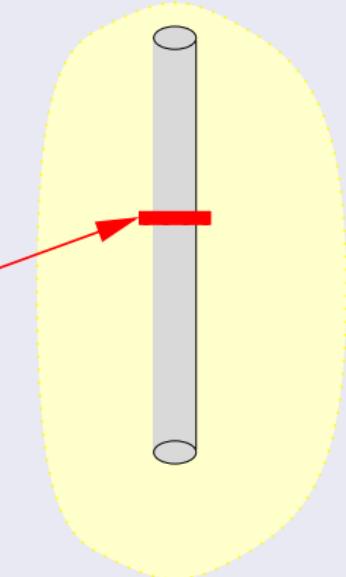
# Flux libc : Type FILE et principe

## Unité d'exécution



**flux0**  
**flux1**  
**flux2**  
**flux3**  
**flux4**  
**flux5**  
...

**statut**



type FILE\* : int fd, void\* buf, ...

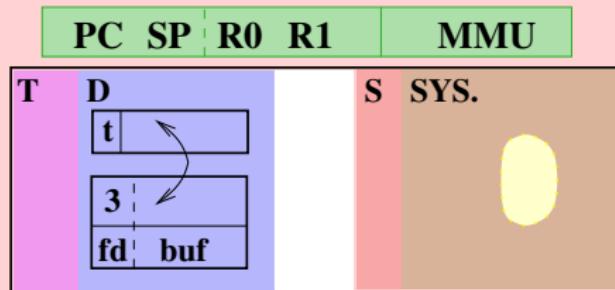
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

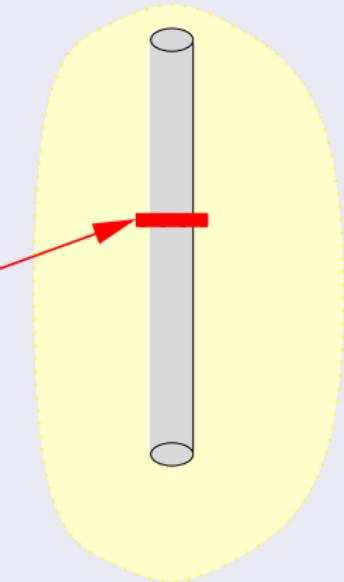
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

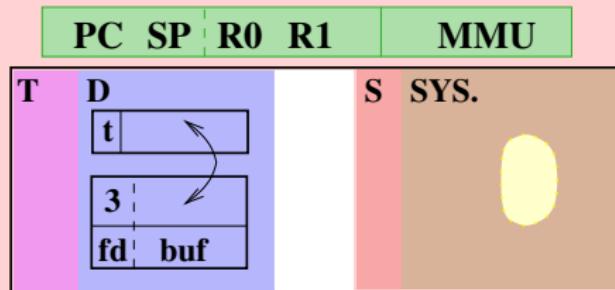
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

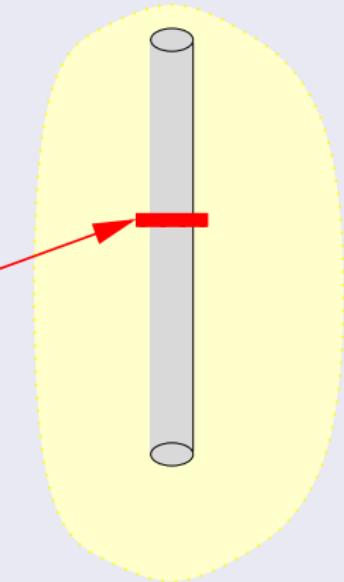
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

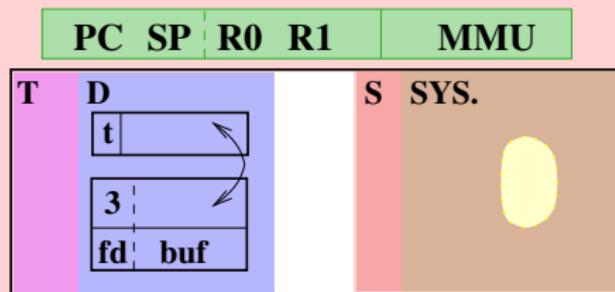
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

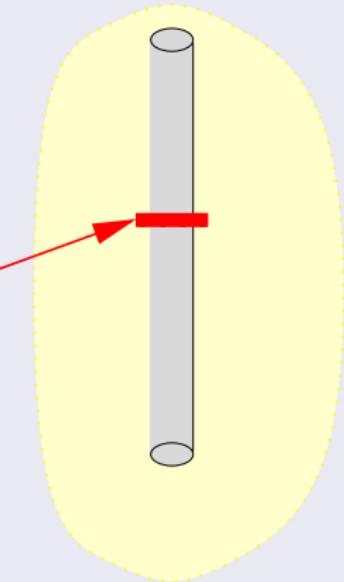
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

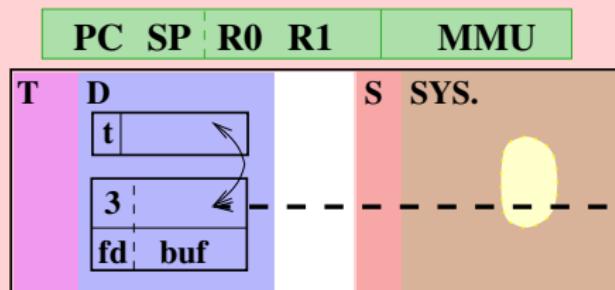
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

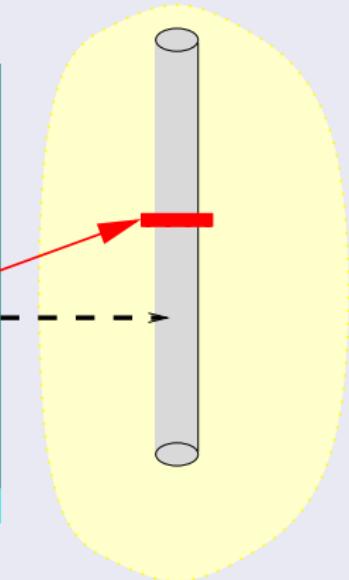
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



**flux0**  
**flux1**  
**flux2**  
**flux3**  
~~**flux4**~~  
**flux5**  
...  
**statut**



type FILE\* : int fd, void\* buf, ...

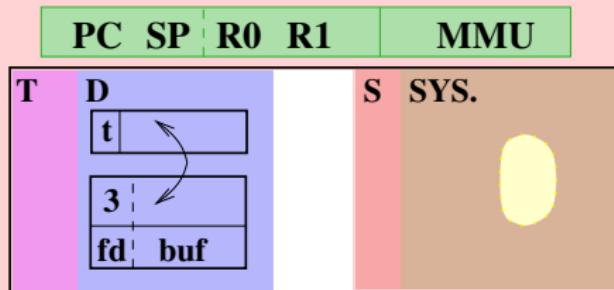
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

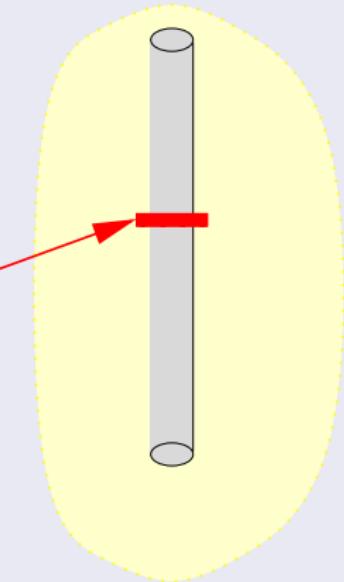
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

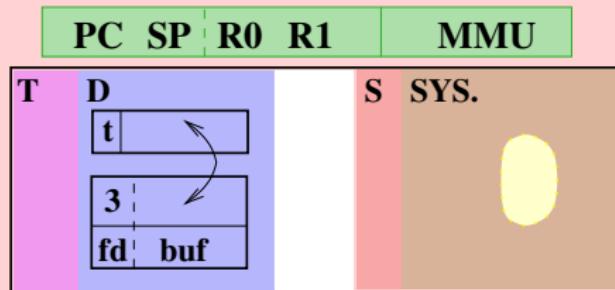
localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

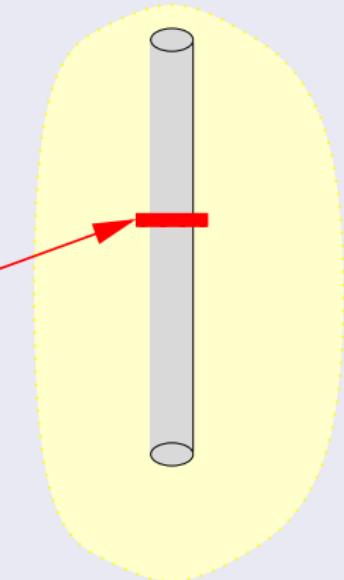
fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Type FILE et principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



type FILE\* : int fd, void\* buf, ...

localisation Espace virtuel utilisateur.

fonction 1 Formatage des entrées/sorties  $\Rightarrow$  facilité d'utilisation.

fonction 2 Minimiser le nombre d'appels système  $\Rightarrow$  performance

# Flux libc : Définitions et opérations sur FILE

extern FILE \*stdin, \*stdout, \*stderr ;

Variables globales pointant les descripteurs des flux standard d'entrée, de sortie et d'erreur.

#define EOF ...

constante indiquant fin de fichier.

int fileno(FILE \*f)

Renvoie le descripteur de flux noyau associé au flux libc f.

int feof(FILE\* f)

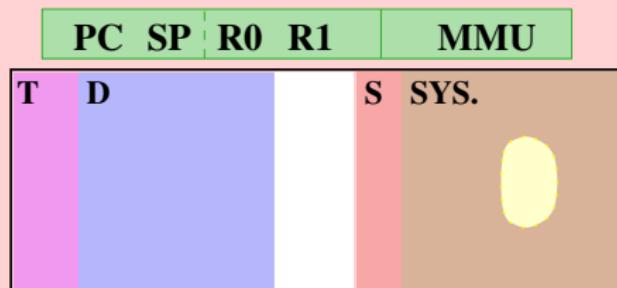
Renvoie 0 si le flux libc f est en fin de fichier.

int fflush(FILE \*f)

Fonction Sauve si besoin le tampon associé au flux f (en utilisant write).

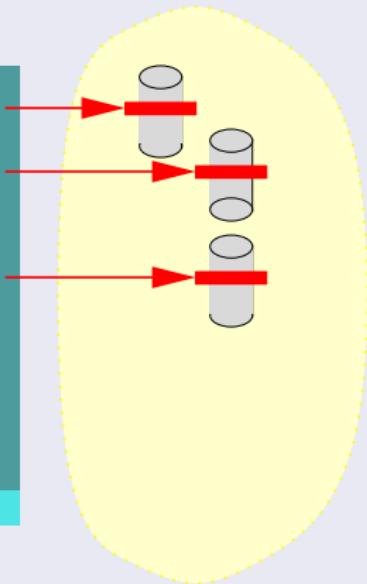
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



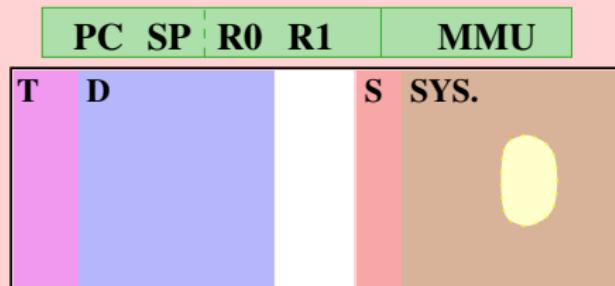
Synopsis FILE\*fopen(const char\* fn, const char\*flag)

Synopsis FILE\*fdopen(int fd, const char\*flag)

Fonction Associe un descripteur de flux au fichier f ou à fd et le renvoie.

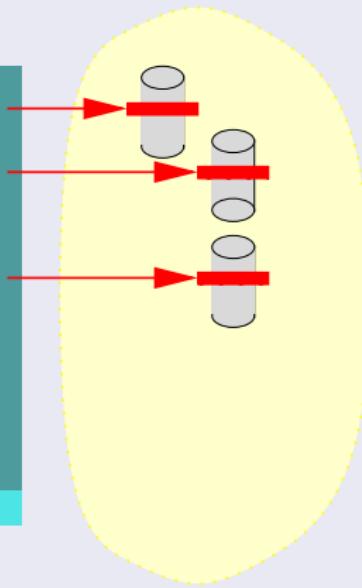
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

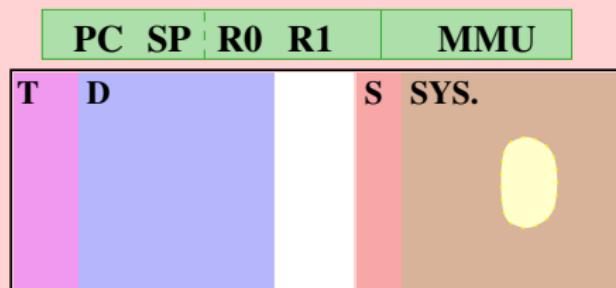
flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
**statut**



Exemple "fopen("f1","r")" crée un flux noyau (2) attaché au fichier f1, alloue un FILE\* et l'associe au flux noyau.

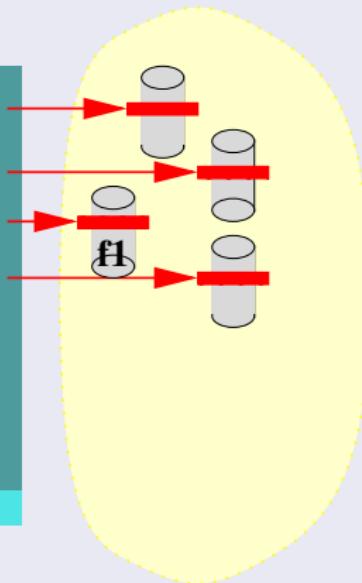
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

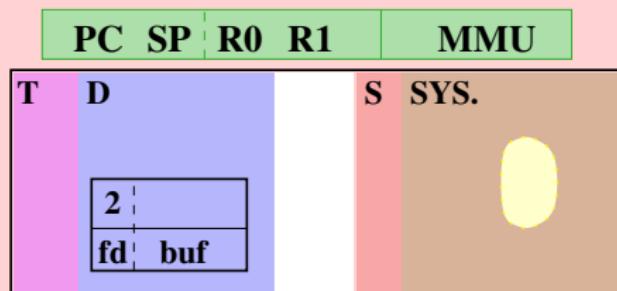
flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
**statut**



Exemple "fopen("f1","r")" crée un flux noyau (2) attaché au fichier f1, alloue un FILE\* et l'associe au flux noyau.

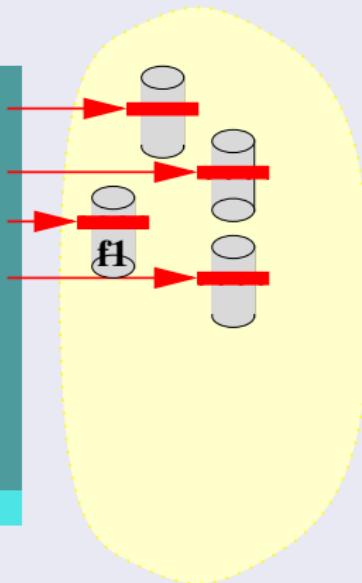
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

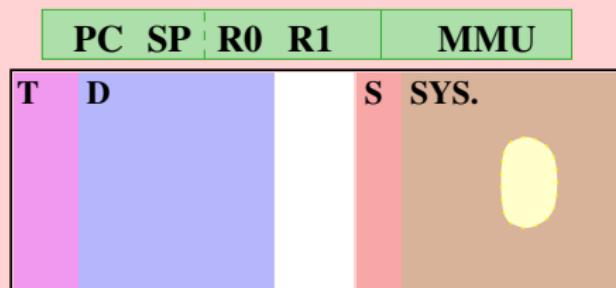
flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
**statut**



Exemple "fopen("f1","r")" crée un flux noyau (2) attaché au fichier f1, alloue un FILE\* et l'associe au flux noyau.

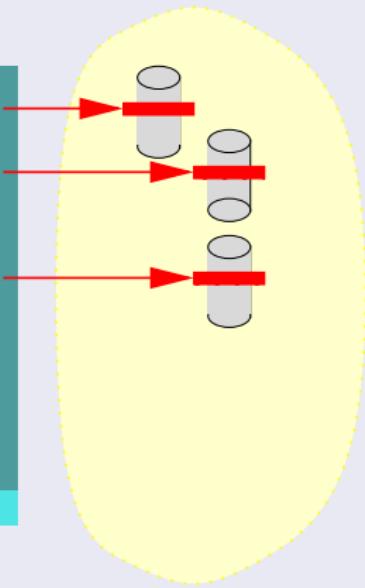
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

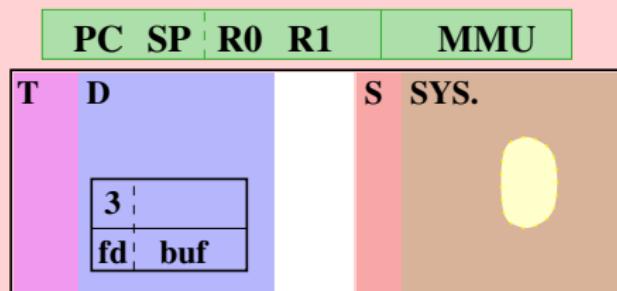
flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
**statut**



Exemple "fdopen(3, "rw")" alloue un FILE\* et l'associe au flux noyau (3).

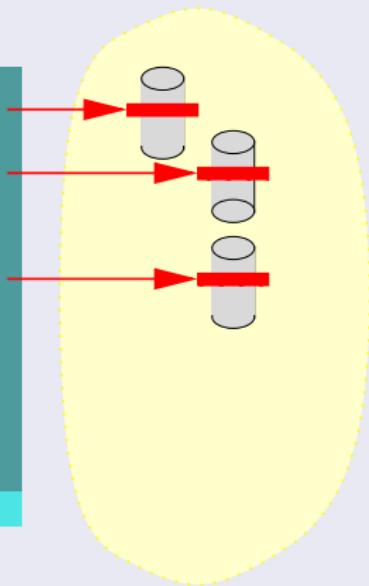
# Flux libc : Ouverture

## Unité d'exécution



PID CWD UID GID0 ...  
ARG0 ARG1 ARG2 ...  
ENV0 ENV1 ENV2 ...

flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
**statut**



Exemple "fdopen(3, "rw")" alloue un FILE\* et l'associe au flux noyau (3).

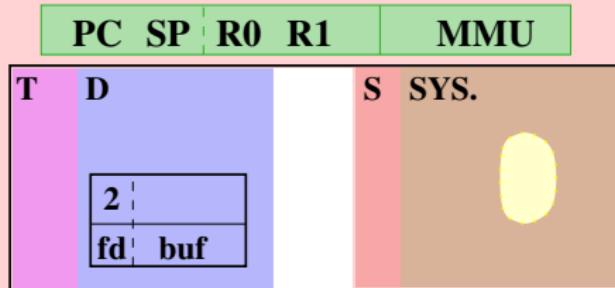
# Flux libc : Ouverture

## Modes

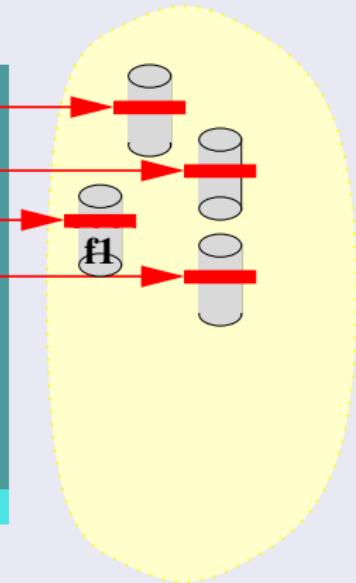
flag	accès	pos	tronqué	création
r	ro	début	non	jamais
r+	rw	début	non	jamais
w	wo	début	oui	si besoin
w+	rw	début	oui	si besoin
a	w	fin	non	si besoin
a+	rw	fin	non	si besoin

# Flux libc : Fermeture

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



Synopsis `int fclose(FILE* f)`

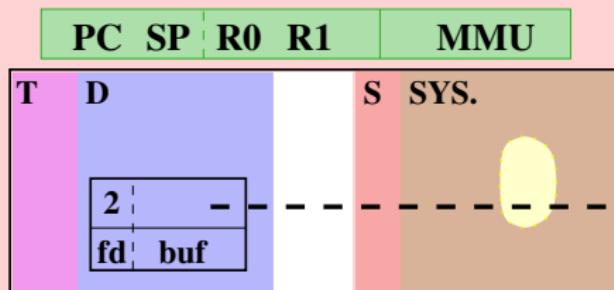
Fonction Ferme le flux f.

Retour  $\emptyset$  si pas d'erreur.

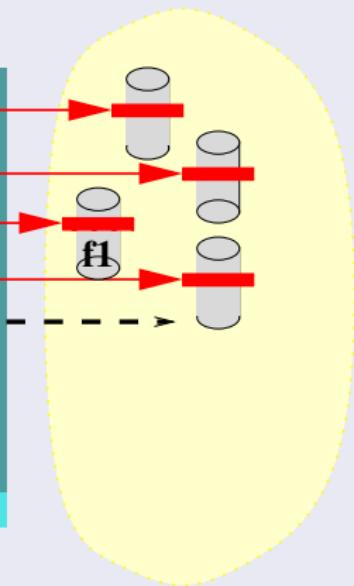
Exemple "`fclose(f)`" : Écriture du tampon si besoin, libère le flux

# Flux libc : Fermeture

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



Synopsis `int fclose(FILE* f)`

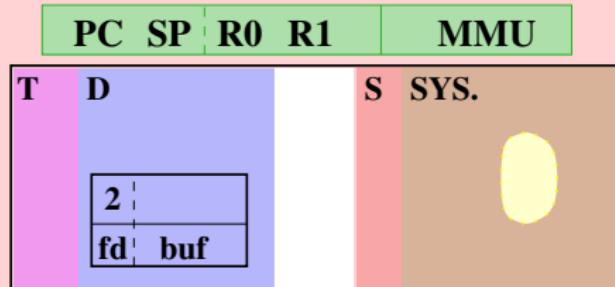
Fonction Ferme le flux f.

Retour  $\emptyset$  si pas d'erreur.

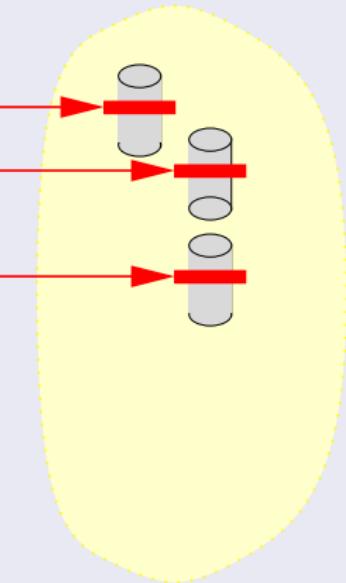
Exemple "`fclose(f)`" : Écriture du tampon si besoin, libère le flux

# Flux libc : Fermeture

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



Synopsis `int fclose(FILE* f)`

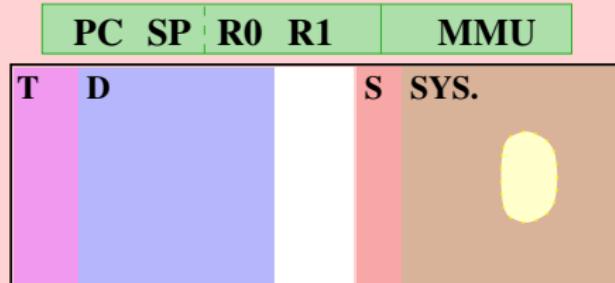
Fonction Ferme le flux f.

Retour  $\emptyset$  si pas d'erreur.

Exemple "`fclose(f)`" : Écriture du tampon si besoin, libère le flux

# Flux libc : Fermeture

## Unité d'exécution



flux0

flux1

flux2

flux3

flux4

flux5

...

statut

Synopsis `int fclose(FILE* f)`

Fonction Ferme le flux f.

Retour  $\emptyset$  si pas d'erreur.

Exemple "`fclose(f)`" : Écriture du tampon si besoin, libère le flux

# Flux libc : E/S non formatées (fonctions) I

`size_t fread(void *buf, size_t sze, size_t nbe, FILE *f)`

**Fonction** Essaye de lire nbe éléments de taille sze ( $nbe*sze$  octets) du flux f et les range dans le tampon buf.

**Retour** Le nombre d'éléments transférés. Ø indique soit E.O.F soit une erreur.

**E.O.F** Est indiquée par la fonction `feof(f)`.

`size_t fwrite(void *buf, size_t sze, size_t nbe, FILE *f)`

**Fonction** Essaye de d'écrire les nbe premiers éléments de taille sze ( $nbe*sze$  octets) du tampon buf dans le flux f.

**Retour** Le nombre d'éléments transférés. Ø indique une erreur.

## Flux libc : E/S non formatées (exemple)

```
1 FILE *f;
2 while ( (n=fread( buf ,
3                   16,5,f)) ) {
4     // traiter n elements
5 }
6 if ( !feof(f) ) {
7     // erreur lecture
8 }
9 // E.O.F
```

# Flux libc : E/S formatées

Ces fonctions sont réservées à la lecture ou l'écriture de fichiers texte.

```
int fscanf(FILE *f, const char *fmt, ...)  
int fprintf(FILE *f, const char *fmt, ...)  
int scanf(FILE *f, const char *fmt, ...)  
int printf(const char *fmt, ...)  
char* fgets(char *l, int size, FILE *f)  
int sscanf(const char *str, const char *fmt, ...)  
int sprintf(char *str, const char *fmt, ...)
```

# Flux libc : En pratique I

## Règle d'or

Lorsque qu'on travaille sur **1** flux il faut choisir d'utiliser l'interface noyau ou l'interface libc. Par contre on peut très bien lire un flux avec l'interface noyau et un autre avec l'interface libc.

## stderr

Le flux stderr (2) n'est pas tamponné.

## flux tty

En écriture, ils sont tamponné par ligne.

```
printf("hello world"); // tamponné  
printf("hello world\n"); // non tamponné
```

## EOF et erreur de lecture

Les fonctions de lecture des flux libc retournent la même valeur pour erreur de lecture et E.O.F. Les cas d'erreurs de lecture sont rares une fois que le flux est ouvert avec succès :

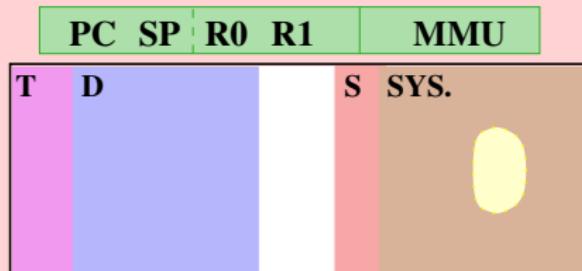
- Fichier régulier local  $\Rightarrow$  défaillance matérielle.
- Fichier régulier réseau  $\Rightarrow$  le noyau bloque la lecture jusqu'à ce que le réseau revienne.
- Fichier tty ou FIFO, c'est impossible.

## 5 Flux

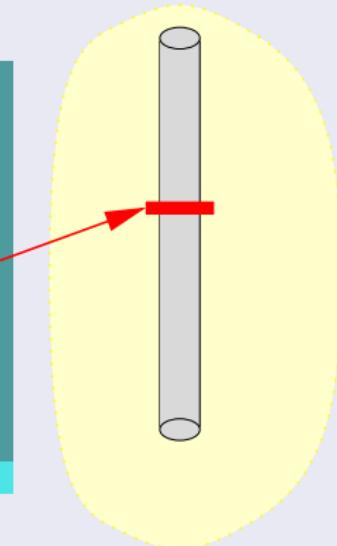
- Algorithmes
- Les flux noyau
- Les flux libc
- **Mapping**
- Comparaison

# MMAP : Principe

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut

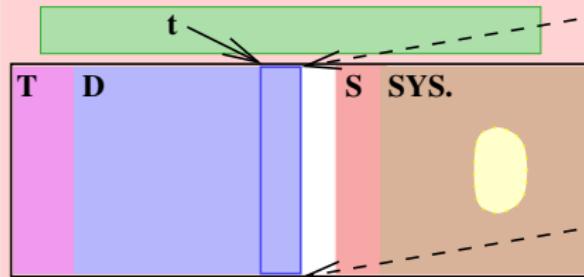


Synopsis `void* mmap((void*)0, size_t len, int prot, MAP_SHARED, int fd, off_t offset)`

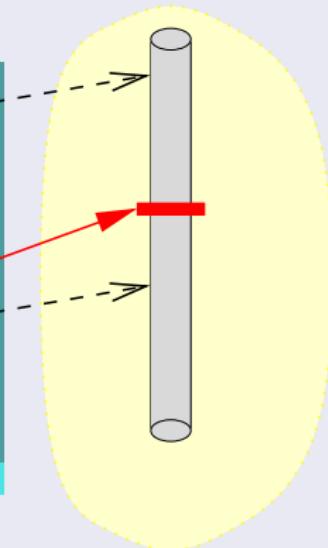
Fonction Mappe les octets [offset : offset+len-1] du flux décrit par fd dans l'espace utilisateur. Renvoie l'adresse du mapping.

# MMAP : Principe

## Unité d'exécution



`flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut`

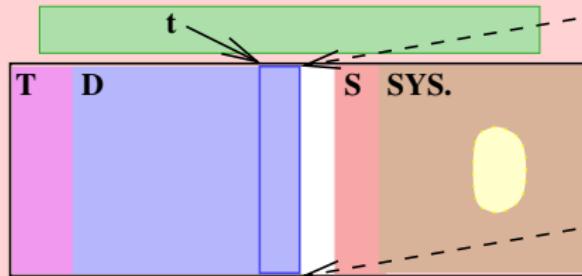


Synopsis `void* mmap((void*)0, size_t len, int prot, MAP_SHARED, int fd, off_t offset)`

Fonction Mappe les octets [offset : offset+len-1] du flux décrit par fd dans l'espace utilisateur. Renvoie l'adresse du mapping.

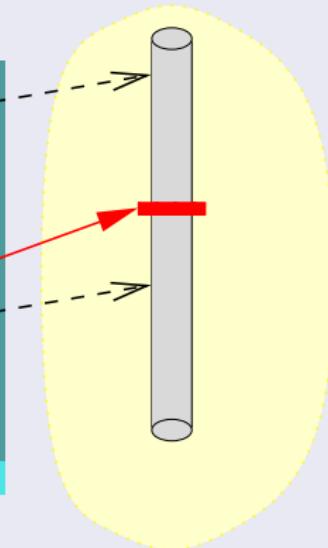
# MMAP : Principe

## Unité d'exécution



`flux0`  
`flux1`  
`flux2`  
`flux3`  
`flux4`  
`flux5`  
...

`statut`

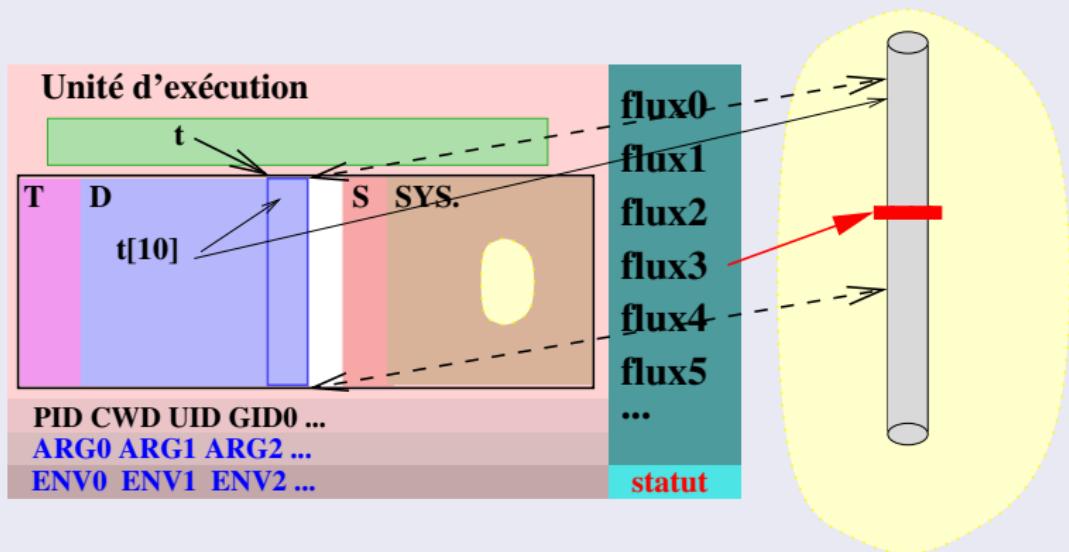


**Retour** L'adresse du mapping en cas de succès, sinon

MAP\_FAILED et errno est mis à jour.

**prot** PROT\_READ pour accès en lecture, PROT\_WRITE pour accès en écriture, PROT\_READ|PROT\_READ pour accès en lecture et écriture.

# MMAP : Principe

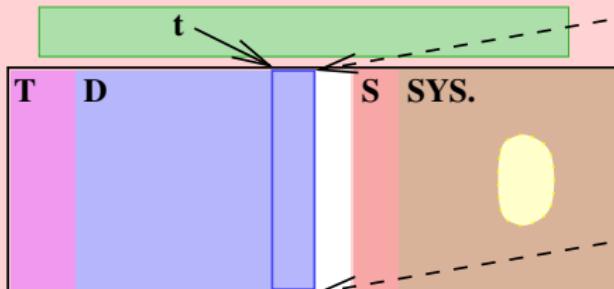


## Exemple

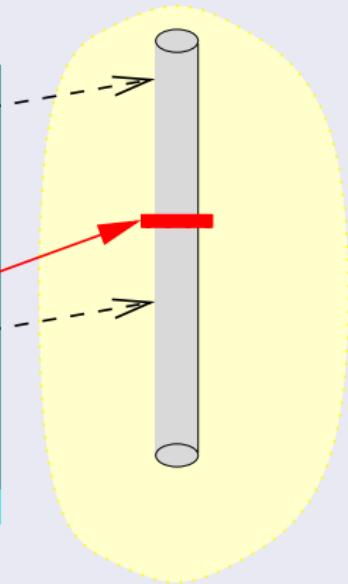
```
char* t=mmap(...);  
c=t[10];  
t[10]='A';
```

# MMAP : Munmap

## Unité d'exécution



flux0  
flux1  
flux2  
flux3  
flux4  
flux5  
...  
statut



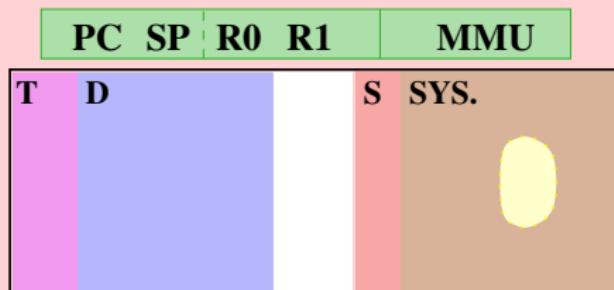
Synopsis `int munmap(void* adr, size_t len)`

Fonction Unmap la zone mémoire [adr :adr+len-1] de l'espace virtuel utilisateur. Le flux associé n'est pas fermé.

Retour  $\emptyset$  en cas de succès, sinon -1 et errno est mis à jour.

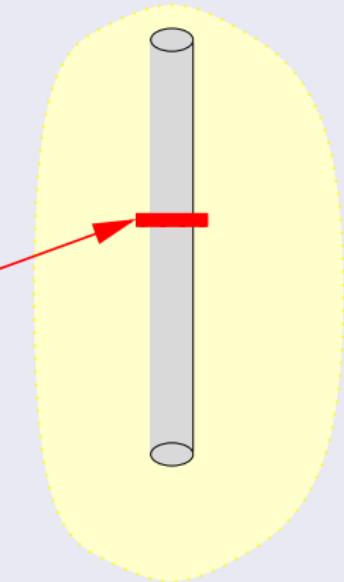
# MMAP : Munmap

## Unité d'exécution



**flux0**  
**flux1**  
**flux2**  
**flux3**  
**flux4**  
**flux5**  
...

**statut**



Synopsis `int munmap(void* adr, size_t len)`

Fonction Unmap la zone mémoire [adr :adr+len-1] de l'espace virtuel utilisateur. Le flux associé n'est pas fermé.

Retour  $\emptyset$  en cas de succès, sinon -1 et errno est mis à jour.

# MMAP : Exemple

```
10 int main( int argc , char* argv [] )
11 {
12     int len;
13     if ( (len=lseek(STDIN_FILENO,0,SEEK_END)) == -1 ) {
14         fprintf( stderr , "%s : lseek failed : %s\n" ,
15             argv[0] , strerror(errno));
16         return 1;
17     }
18     char *p = mmap(0,len,PROT_READ,
19                     MAP_SHARED,STDIN_FILENO,0);
20     if ( p==MAP_FAILED ) {
21         fprintf( stderr , "%s : mmap failed : %s\n" ,
22             argv[0] , strerror(errno));
23         return 1;
24     }
25     write (STDOUT_FILENO,p,len );
26     return 0;
```

## 5 Flux

- Algorithmes
- Les flux noyau
- Les flux libc
- Mapping
- Comparaison

# Flux : Comparaison

## Efficacité théorique

1/2/3 copies pour mmap/flux noyau/libc.

## Efficacité pratique

Mal utilisés, les flux noyau peuvent être catastrophiques.

Mal utilisés, mmap peut coûter cher.

⇒ Les flux libc donnent une efficacité acceptable.

## Facilité d'utilisation

Les flux libc sont faciles à utiliser surtout si il y a des E/S formatées.

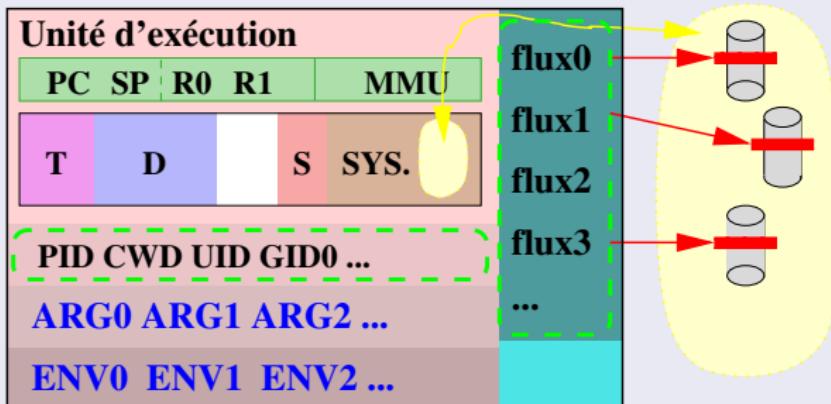
Mmap est le plus complexe (alignement + agrandissement).

si on a pas de contraintes d'efficacité sur les E/S et que le tampon ne pose pas de problème ⇒ flux libc.

## 6 Quelques fonctions système

- Exec
- Exit
- Environnement
- Divers

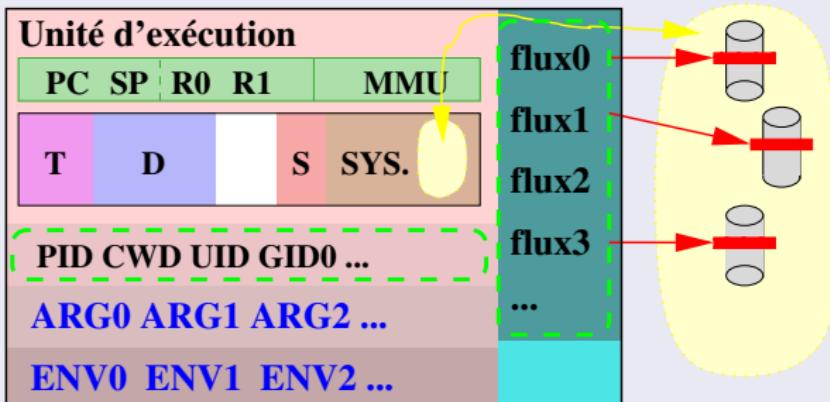
# Exec : L'appel système execve



Synopsis `int execve(const char *path, char *const argv[], char *const envp[])`

Fonction Exécute le programme path dans le processus courant.  
Seuls les identifiants (PID, UID\*, ...) et les flux sont conservés.  
Le programme lancé commence par la fonction main\* avec argv et envv comme arguments.

# Exec : L'appel système execve

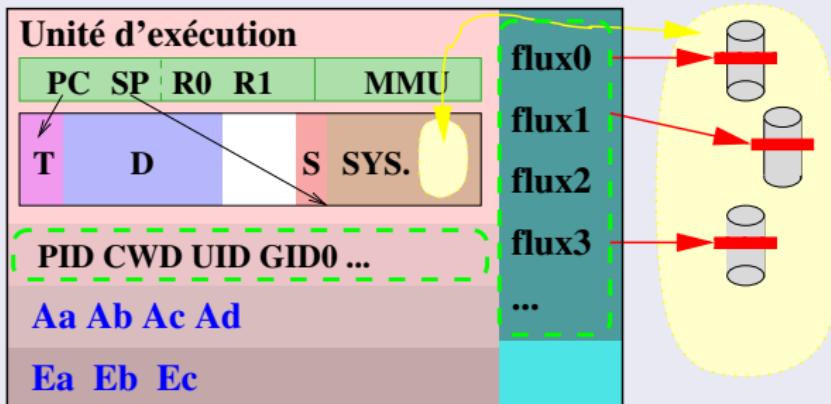


**Retour** En cas de succès, **il n'y a pas de retour**, et en cas d'échec, -1 et errno est mis à jour.

**Exemple**

```
char* a[]={ "Aa", "Ab", "Ac", "Ad", \0};  
char* e[]={ "Ea", "Eb", "Ec", \0};  
execve("./a.out",a,e);
```

# Exec : L'appel système execve



**Retour** En cas de succès, **il n'y a pas de retour**, et en cas d'échec, -1 et errno est mis à jour.

**Exemple**

```
char* a[]={ "Aa", "Ab", "Ac", "Ad", \0};  
char* e[]={ "Ea", "Eb", "Ec", \0};  
execve("./a.out",a,e);
```

# Interface libc

## Synopsis

```
int execlp(const char *path, const char * a0, ... , (char*)[])
int execvp(const char *path, char *const arg[])
```

## Retour

En cas de succès, **il n'y a pas de retour**, et en cas d'échec, -1 et errno est mis à jour.

## Fonction

Ces 2 fonctions appellent execve.

- path est cherché avec la variable d'environnement PATH.
- L'environnement utilisé pour execve est l'environnement courant.

## 6 Quelques fonctions système

- Exec
- Exit
- Environnement
- Divers

Synopsis void \_exit(int statut);

Retour Pas de retour

## Fonction

- Termine le processus et libère tout ce qui était alloué par le processus (E.V, unmapping, fermeture des descripteurs de fichiers ouverts).
- La valeur statut est envoyé au père du processus comme "Statut de fin du processus".
- Le signal SIGCHLD est envoyé au processus père (voir chapitre suivant).
- Le processus 1 devient le père des processus fils.

Synopsis void exit(int statut) ;

Retour Pas de retour

Fonction

- Libère toutes les allocations système faites par la libc (flux libc, suppression des fichiers temporaires, ...).
- Puis appel de \_exit(statut).

## 6 Quelques fonctions système

- Exec
- Exit
- **Environnement**
- Divers

## Synopsis

```
char* getenv(const char *name)
```

```
int setenv(const char *name, const char *value, int overwrite)
```

```
int unsetenv(const char *name)
```

**Retour** getenv renvoie la valeur de la variable d'environnement name ou (char\*)Ø si elle n'existe pas.

setenv et unsetenv renvoient Ø en cas de succès et -1 en cas d'échec.

**Fonction** Ces fonctions permettent de récupérer la valeur d'une variable d'environnement, d'ajouter ou modifier une variable d'environnement et de supprimer une variable d'environnement.

**Note** On peut récupérer les variables d'environnement dans le main :

```
int main(int argc, char *argv[], char **envv[])
```

## 6 Quelques fonctions système

- Exec
- Exit
- Environnement
- Divers

## Synopsis

```
int getpid();  
int getppid();  
char* getcwd(char*buf, size_t bufsz);  
int chdir(const char*path);  
unsigned int sleep(unsigned int sec);  
int usleep(useconds_t usec);  
int system(const char* cmd);
```

## Fonction/Retour

getpid renvoie le PID du processus, getppid renvoie le PID du processus père.

getcwd et chdir permettent d'obtenir ou de changer le CWD.

sleep (usleep) suspend le processus pendant au moins sec secondes (usec  $\mu$ s).

system lance un Shell (/bin/sh) qui exécute la commande

## 7 Communication inter-processus

- Signaux
- FIFO
- SHM et Sémaphore

# Signaux : Définition

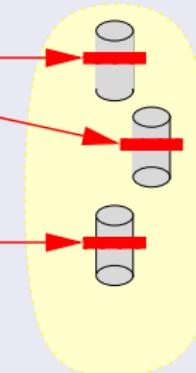
Un signal est un événement (élément dans un ensemble prédéfini) que l'on peut envoyer à un processus.

# Signaux : Définition

Il y a 4 traitements possibles pour un processus qui reçoit un signal :

- ① Ignorer le signal.
- ② Se terminer.
  - ① Interrompre l'exécution en cours.
  - ② Générer un core du processus (facultatif).
  - ③ Terminer le processus.
- ③ Se suspendre.
  - ① Interrompre l'exécution en cours.
  - ② Mettre le processus en mode "endormi".
- ④ Traitement spécifique.
  - ① Interrompre l'exécution en cours
  - ② Exécuter une fonction gestionnaire (même E.V.)
  - ③ Reprendre l'exécution en cours

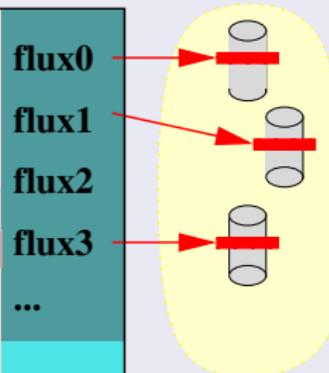
# Signaux : Implémentation



- Une table indiquant pour chaque signal le traitement associé.
- Émettre un signal à un processus P
  - ⇒ marquer le signal comme reçu,
  - ⇒ le réveiller s'il est suspendu\*.
- Que ce passe-t-il si on réenvoie le même signal ?

# Signaux : Implémentation

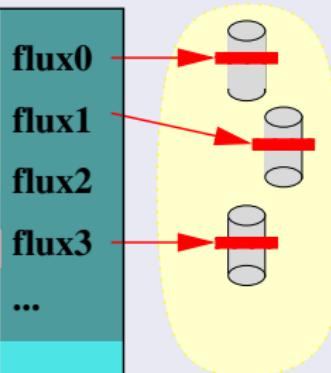
Unité d'exécution				MMU		
PC	SP	R0	R1	MMU		
T	D	S	SYS.			
PID CWD UID GID0 ...						
ARG0	ARG1	...	ENV0	ENV1	...	
SIG	3	9	10	11	15	19 17
REC						
ACT	C	T	T	C	h1	S I



- Une table indiquant pour chaque signal le traitement associé.
- Émettre un signal à un processus P
  - ⇒ marquer le signal comme reçu,
  - ⇒ le réveiller s'il est suspendu\*.
- Que ce passe-t-il si on réenvoie le même signal ?

# Signaux : Implémentation

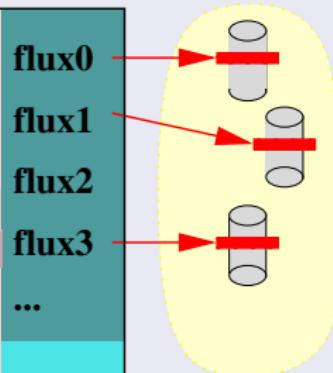
Unité d'exécution							
PC	SP	R0	R1	MMU			
T	D	S	SYS.				
PID CWD UID GID0 ...							
ARG0	ARG1 ...	ENV0	ENV1 ...				
SIG	3	9	10	11	15	19	17
REC			*				
ACT	C	T	T	C	h1	S	I



- Une table indiquant pour chaque signal le traitement associé.
- Émettre un signal à un processus P
  - ⇒ marquer le signal comme reçu,
  - ⇒ le réveiller s'il est suspendu\*.
- Que ce passe-t-il si on réenvoie le même signal ?

# Signaux : Implémentation

Unité d'exécution				MMU							
PC	SP	R0	R1	MMU							
T	D	S	SYS.								
PID CWD UID GID0 ...											
ARG0 ARG1 ... ENV0 ENV1 ...											
SIG	3	9	10	11	15	19	17				
REC			*								
ACT	C	T	T	C	h1	S	I				



- Une table indiquant pour chaque signal le traitement associé.
- Émettre un signal à un processus P
  - ⇒ marquer le signal comme reçu,
  - ⇒ le réveiller s'il est suspendu\*.
- Que ce passe-t-il si on réenvoie le même signal ?

# Signaux : L'ensemble des signaux I

NAME	NUM	Def.	Act.	comment
SIGHUP	1	Term		Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term		Interrupt from keyboard
SIGQUIT	3	Core		Quit from keyboard
SIGILL	4	Core		Illegal Instruction
SIGABRT	6	Core		Abort signal from abort(3)
SIGBUS	7	Core		Bus error (bad memory access)
SIGFPE	8	Core		Floating point exception
SIGKILL	9	Term		Kill signal
SIGSEGV	11	Core		Invalid memory reference

# Signaux : L'ensemble des signaux II

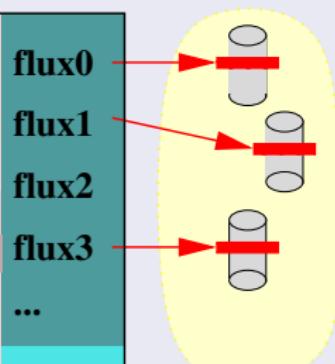
SIGPIPE	13	Term	Broken pipe : write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	User-defined signal 1
SIGUSR2	12	Term	User-defined signal 2
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18	Cont	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

# Signaux : L'ensemble des signaux III

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Signaux : P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)

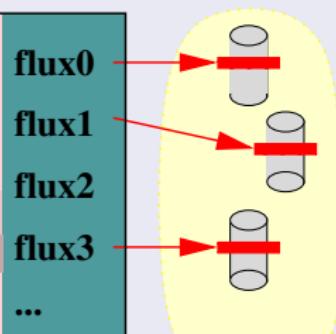
Unité d'exécution							
PID CWD UID GID0 ...							
SIG	3	9	10	11	15	19	17
REC	C	T	T	C	h1	S	I
ACT	C	T	T	C	h1	S	I



- S'il est suspendu, il n'a pas de processeur.
- Il reçoit le signal, il est réveillé (éligible)
- Un jour ou l'autre il prend un processeur

Signaux : P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)

Unité d'exécution							
PID CWD UID GID0 ...							
SIG	3	9	10	11	15	19	17
REC		*					
ACT	C	T	T	C	h1	S	I

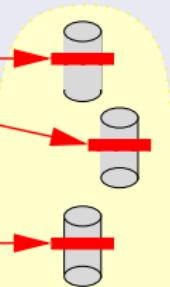


- S'il est suspendu, il n'a pas de processeur.
- Il reçoit le signal, il est réveillé (éligible)
- Un jour ou l'autre il prend un processeur

Signaux : P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)

Unité d'exécution					
PC	SP	R0	R1	MMU	
T	D	S	SYS.		
PID CWD UID GID0 ...					
ARG0	ARG1	...	ENV0	ENV1	...
SIG	3	9	10	11	15
REC		*			19
ACT	C	T	T	C	h1

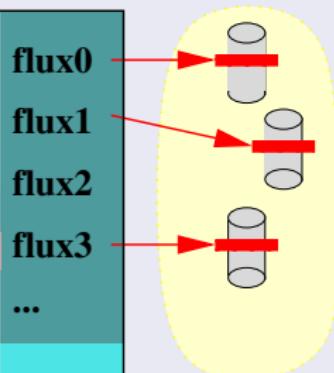
flux0  
flux1  
flux2  
flux3  
...



- S'il est suspendu, il n'a pas de processeur.
- Il reçoit le signal, il est réveillé (éligible)
- Un jour ou l'autre il prend un processeur

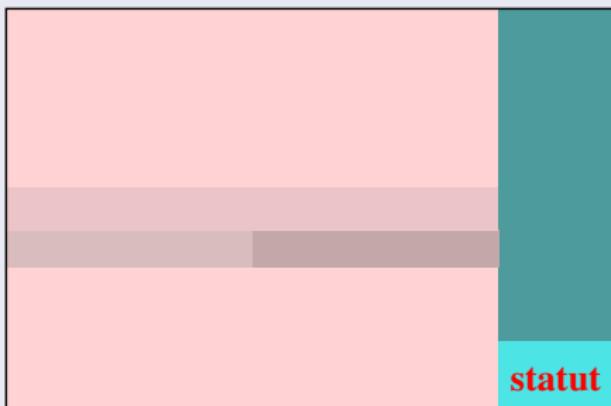
Signaux : P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)

Unité d'exécution										
PC SP		R0 R1	MMU							
T	D	S	SYS.	...						
PID CWD UID GID 0 ...										
ARG0 ARG1 ... ENV0 ENV1 ...										
SIG	3	9	10	11	15	19	17			
REC		*								
ACT	C	T	T	C	h1	S	I			



- Il termine son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGUSR1 et action Terminaison)

Signaux : P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)



- Exécute la routine système de terminaison `_exit`.
- Donne la main.

# Signaux : P est en mode utilisateur et reçoit SIGUSR1 (10 avec Term)

- Est-ce possible ?
- S'il est en mode utilisateur, il a le processeur.
- Il continue de tourner tranquillement
- Il passe en mode système (appel système ou interruption)
- Il fait son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGUSR1 et action Terminaison)
- Exécute la routine système de terminaison `_exit`.
- Donne la main.

# Signaux : P est en mode user et reçoit SIGTERM (15 avec h1)

- S'il est en mode utilisateur, il a le processeur.
- Il continue de tourner tranquillement
- \* Il passe en mode système (Appel système ou interruption)
- Il fait son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGTERM et action h1())
- Contexte1= contexte retour normal
- Change le contexte pour lancer h1 (pc=h1 et sp=zone vierge, et retour h1 déclenche l'appel système "retour de gestionnaire")
- Passe en mode utilisateur
- h1() s'exécute
- En mode système "retour de gestionnaire"
- Contexte=contexte1
- Retour en mode utilisateur (où il avait quitté en \*)

# Signaux : Conclusion

- La durée entre l'envoi d'un signal (quasi instantané) et son traitement est très variable.
- Elle dépend de plein de paramètres (de ce que fait le processus, charge de la machine, ...)
- Les signaux sont très loin du temps réels.
- Lorsqu'un processus s'envoie un signal à lui-même, cette durée peut elle être longue ?

# Signaux : Envoyer un signal

Synopsis `int kill(pid_t pid, int sig);`

Fonction Envoie le signal `sig` au processus `pid`.

Retour  $\emptyset$  en cas de succès, -1 en cas d'échec et `errno` est mis à jour.

Exemple

```
kill(getpid(),SIGKILL);
printf("Je me suis tué\n");
// verra-t-on ce printf?
```

# Signaux : Fixer le gestionnaire d'un signal

## Synopsis

```
typedef void (*sighandler_t)(int);  
sighandler_t signal(int sig, sighandler_t handler);
```

**Fonction** Met le gestionnaire du signal `sig` à `handler`. `Handler` est soit  
une [adresse en E.V. utilisateur](#) la fonction gestionnaire.  
`SIG_IGN` Ce signal sera ignoré.  
`SIG_DFL` Remet le gestionnaire par défaut.

**Retour** Le gestionnaire précédent en cas de succès, `SIG_ERR` en  
cas d'échec et `errno` est mis à jour.

## Exemple

```
// désactive le <CTL-C>  
signal(SIGQINT,SIG_IGN);
```

# Signaux : Autres

## Synopsis

```
int pause(void);  
unsigned int alarm(unsigned int durée);  
useconds_t ualarm(useconds_t durée, Ø);
```

## Fonction

Pause suspend le processus jusqu'à l'arrivée d'un signal non ignoré.

Alarm (resp : ualarm) indique au noyau d'envoyer un signal SIGALRM au processus après au moins durée secondes (resp :  $\mu$ -secondes).

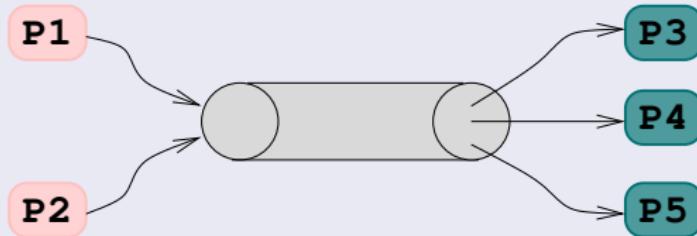
**Retour** Pause renvoie toujours -1.

Alarm et ualarm renvoie Ø si il n'y a pas d'alarme en cours, sinon la durée restante pour atteindre l'alarme en cours.

## 7 Communication inter-processus

- Signaux
- FIFO
- SHM et Sémaphore

# FIFO : Définitions



**FIFO** First In First Out (file d'attente à un guichet).

**Canal de communication** Il a une taille maximale et 2 états :  
vide Il n'y a aucune donnée dans le canal.

plein Il y a "taille maximale" données dans le canal.

**Producteurs/Écrivains** Ceux qui écrivent des données dans la FIFO.

**Consommateurs/Lecteurs** Ceux qui lisent les données de la FIFO.

**Canal de synchronisation**

- ⇒ Un consommateur est bloqué si la FIFO est vide.
- ⇒ Un producteur est bloqué si la FIFO est pleine.

# FIFO : Les différentes FIFO

tty N<->N, même machine, flux d'octets

pipe N<->N, même machine, flux d'octets, processus parenté

pipe nommé N<->N, même machine, flux d'octets

message IPC N<->N, même machine, flux de messages

unix socket stream 1<->1, même machine, flux d'octets

unix socket datagram N->1, même machine, flux de messages

socket TCP 1<->1, inter machine, flux d'octets

socket UDP N->1, inter machine, flux de messages

# FIFO : Accès aux flux des pipes

## Création d'un pipe non nommé

```
int pipe(int fd[2]);
```

- $fd[0]$   $\Rightarrow$  sortie de la FIFO, lecture
- $fd[1]$   $\Rightarrow$  entrée de la FIFO, écriture

## Création d'un pipe nommé

```
sh> mkfifo path
```

ou

```
sh> mknod path p
```

- Crée le fichier spécial path correspondant à une FIFO.
- Pour écrire ou lire la FIFO il suffit d'ouvrir le fichier path.

## Lecture/écriture d'un pipe

Une fois que l'on a le descripteur de flux ([Unix](#) ou [libc](#)), il suffit d'utiliser les primitives d'E/S standard.

# FIFO : Accès aux flux des pipes I

## Spécificité ouverture

- Ouverture RO est bloquante si il n'y a pas d'écrivains.
- Ouverture WO est bloquante si il n'y a pas de lecteurs.

## Spécificité lecture

- Un `read(pipefd,buf,n)` peut retourner une valeur positive ( $> \emptyset$ ) et inférieure à  $n$  sans que l'on soit en fin de flux.
- Un `read(pipefd,buf,n)` est bloquant si le pipe est vide et qu'il y a des écrivains potentiels.
- Un `read(pipefd,buf,n)` renvoie  $\emptyset$  si le pipe est vide et qu'il n'y a pas d'écrivains.

## Spécificité écriture

Une écriture dans un pipe sans lecteur génère un signal SIGPIPE.

- Terminaison du programme si le gestionnaire de SIGPIPE est SIG\_DFL (Terminaison).
- Renvoie -1 si le gestionnaire de SIGPIPE est SIG\_IGN ou une fonction. Dans ce cas errno vaut EPIPE.

# FIFO : Exemple

Soit fifo1 et fifo2 deux pipes nommées, écrire les programmes ho et ell dont les comportements sont donnés ci-dessous :

- ho écrit h, o et '\n' sur le flux standard de sortie.
- ell écrit e, l et l sur le flux standard de sortie.
- Lancés dans n'importe quel ordre, ils écrivent "hello" sur le flux standard de sortie.

```
sh> ./ho & # ou ./ell  
sh> ./ell      # ou ./ho  
hello  
sh>
```

# FIFO : Solution 1

```
10 // ho
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_WRONLY);
16     int s2m=open(
17         "fifo2",O_RDONLY);
18
19     write(1,"h",1);
20     write(m2s,&c,1);
21
22     read(s2m,&c,1);
23     write(1,"o\n",2);
24
25     return 0;
```

```
10 // ell
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_RDONLY);
16     int s2m=open(
17         "fifo2",O_WRONLY);
18
19
20
21
22
23
24
25
26
27
28     read(m2s,&c,1);
29     write(1,"ell",3);
30     write(s2m,&c,1);
31
32
33     return 0;
```

# FIFO : Solution 1

```
10 // ho
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_WRONLY);
16     int s2m=open(
17         "fifo2",O_RDONLY);
18
19     write(1,"h",1);
20     write(m2s,&c,1);
21
22     read(s2m,&c,1);
23     write(1,"o\n",2);
24
25     return 0;
```

```
10 // ell
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_RDONLY);
16     int s2m=open(
17         "fifo2",O_WRONLY);
18
27
28     read(m2s,&c,1);
29     write(1,"ell",3);
30     write(s2m,&c,1);
31
32
33     return 0;
```

# FIFO : Solution 1

```
10 // ho
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_WRONLY);
16     int s2m=open(
17         "fifo2",O_RDONLY);
18
19     write(1,"h",1);
20     write(m2s,&c,1);
21
22     read(s2m,&c,1);
23     write(1,"o\n",2);
24
25     return 0;
```

```
10 // ell
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_RDONLY);
16     int s2m=open(
17         "fifo2",O_WRONLY);
18
27
28     read(m2s,&c,1);
29     write(1,"ell",3);
30     write(s2m,&c,1);
31
32
33     return 0;
```

# FIFO : Solution 1

```
10 // ho
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_WRONLY);
16     int s2m=open(
17         "fifo2",O_RDONLY);
18
19     write(1,"h",1);
20     write(m2s,&c,1);
21
22     read(s2m,&c,1);
23     write(1,"o\n",2);
24
25     return 0;
```

```
10 // ell
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_RDONLY);
16     int s2m=open(
17         "fifo2",O_WRONLY);
18
27
28     read(m2s,&c,1);
29     write(1,"ell",3);
30     write(s2m,&c,1);
31
32
33     return 0;
```

# FIFO : Solution 1

```
10 // ho
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_WRONLY);
16     int s2m=open(
17         "fifo2",O_RDONLY);
18
19     write(1,"h",1);
20     write(m2s,&c,1);
21
22     read(s2m,&c,1);
23     write(1,"o\n",2);
24
25     return 0;
```

```
10 // ell
11 int main()
12 {
13     char c;
14     int m2s=open(
15         "fifo", O_RDONLY);
16     int s2m=open(
17         "fifo2",O_WRONLY);
18
27
28     read(m2s,&c,1);
29     write(1,"ell",3);
30     write(s2m,&c,1);
31
32
33     return 0;
```

# FIFO : Solution 2

```
10 // ho
11 int main()
12 {
13     char c;
14
15     write(1,"h",1);
16     int s2m=open(
17         "fifo",O_RDONLY);
18
19     read(s2m,&c,1);
20     write(1,"o\n",2);
21
22     return 0;
23 }
```

```
10 // ell
11 int main()
12 {
13     char c;
14
24
25     int s2m=open(
26         "fifo",O_WRONLY);
27     write(1,"ell",3);
28     write(s2m,&c,1);
29
30
31     return 0;
32 }
```

# FIFO : Solution 2

```
10 // ho
11 int main()
12 {
13     char c;
14
15     write(1,"h",1);
16     int
17         "fifo",O_RDONLY);
18
19     read(s2m,&c,1);
20     write(1,"o\n",2);
21
22     return 0;
23 }
```

```
10 // ell
11 int main()
12 {
13     char c;
14
24
25     int
26         "fifo",O_WRONLY);
27     write(1,"ell",3);
28     write(s2m,&c,1);
29
30
31     return 0;
32 }
```

# FIFO : Solution 2

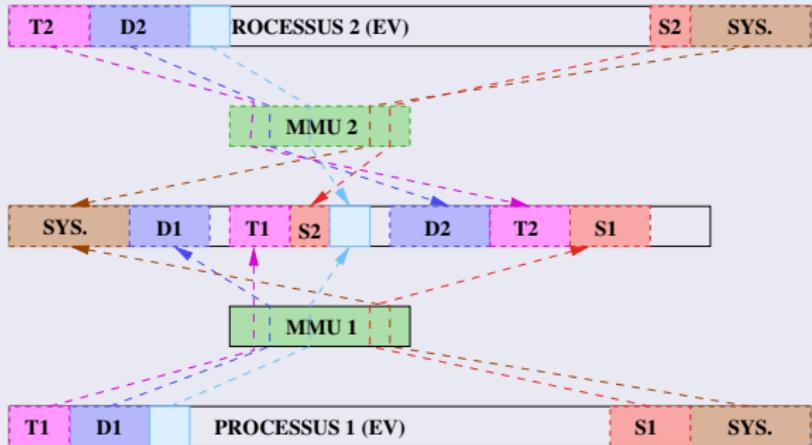
```
10 // ho
11 int main()
12 {
13     char c;
14
15     write(1,"h",1);
16     int s2m=open(
17         "fifo",O_RDONLY);
18
19     read(s2m,&c,1);
20     write(1,"o\n",2);
21
22     return 0;
23 }
```

```
10 // ell
11 int main()
12 {
13     char c;
14
24
25     int s2m=open(
26         "fifo",O_WRONLY);
27     write(1,"ell",3);
28     write(s2m,&c,1);
29
30
31     return 0;
32 }
```

## 7 Communication inter-processus

- Signaux
- FIFO
- SHM et Sémaphore

# SHM : Principes



En jouant avec les MMU, on peut accrocher le même espace de mémoire physique aux segments de données de 2 processus.

⇒ On appelle un tel segment , un segment de mémoire partagée.

- Ils ont généralement des adresses virtuelles différentes.
- Les 2 processus peuvent s'échanger des données au travers de ce segment.

# SHM : Les différents outils

Thread	Le segment données est partagé.
mmap	Permet de créer des segments de mémoire partagée pour des processus parentés.
IPC System V	voir « <code>sh&gt; man svipc</code> » (sv=System V).
POSIX Shared memory	voir « <code>sh&gt; man shm_overview</code> »

# SEM : Problème

Soit "int\*p ;" un pointeur dans un segment partagé par 3 processus qui pointe sur la même case mémoire physique.

P1 :  $*p += 1;$

P2 :  $*p += 3;$

P3 :  $*p += 5;$

On aimerait que quand les 3 processus ont fait leurs modifications  
 $*p$  soit incrémenté de 9.

# SEM : Problème

"`*p += n;`" est traduit en assembleur par plusieurs instructions par exemple : "`r = *p; r += n; *p = r`" où `r` est un registre du processeur.

Séquencement 1

P1	P2	P3
<code>r = *p</code>		
<code>r += 1</code>		
<code>*p = r</code>		
	<code>r = *p</code>	
	<code>r += 3</code>	
	<code>*p = r</code>	
		<code>r = *p</code>
		<code>r += 5</code>
		<code>*p = r</code>
		<code>*p + 9</code>

Séquencement 2

P1	P2	P3
<code>r = *p</code>		
	<code>r = *p</code>	
	<code>r += 3</code>	
	<code>*p = r</code>	
		<code>r = *p</code>
	<code>r += 1</code>	
	<code>*p = r</code>	
		<code>r = *p</code>
		<code>r += 5</code>
		<code>*p = r</code>
		<code>*p + 6</code>

Séquencement 3

P1	P2	P3
<code>r = *p</code>		
<code>r += 1</code>		
	<code>r = *p</code>	
	<code>r += 3</code>	
	<code>*p = r</code>	
		<code>r = *p</code>
		<code>r += 5</code>
		<code>*p = r</code>
		<code>*p + 1</code>

Suivant le séquencement des instructions assembleur `*p` peut avoir toutes les valeurs suivantes :

`*p+1, *p+3, *p+5, *p+4, *p+6, *p+8 et *p+9.`

# SEM : Sémaphore d'exclusion mutuelle

Un sémaphore simple est une entité ayant un état binaire (LIBRE, BLOQUÉ), une file de processus et 2 fonctions.

- P()
  - si l'état est BLOQUÉ, enfiler le processus et le suspendre.
  - si l'état est LIBRE, mettre l'état à BLOQUÉ.
- V()
  - si l'état est LIBRE, erreur.
  - si l'état est BLOQUÉ et la file est vide, mettre l'état à LIBRE.
  - si l'état est BLOQUÉ et la file est non vide, défiler le 1<sup>er</sup> processus de la file et le réveiller.

Ainsi si mutex est un sémaphore initialisé à {LIBRE,  $\emptyset$ }, ces codes

P1

```
P(mutex);  
*p += 1;  
V(mutex);
```

P2

```
P(mutex);  
*p += 3;  
V(mutex);
```

P3

```
P(mutex);  
*p += 5;  
V(mutex);
```

# SEM : Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg : fixer des points de rendez-vous).

```
10 | S1 <- {BLOQUE, vide}
11 | S2 <- {BLOQUE, vide}
12 | ...
13 | V(S2);
14 | P(S1);
15 | RDV
16 | ...
```

```
10 | S1 <- {BLOQUE, vide}
11 | S2 <- {BLOQUE, vide}
12 | ...
13 | P(S2);
14 | V(S1);
15 | RDV
16 | ...
```

# SEM : Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg : fixer des points de rendez-vous).

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   V(S2);
14 |   P(S1); P(S1);
15 |   RDV RDV
16 |   ...
```

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   P(S2);
14 |   V(S1); V(S1);
15 |   RDV RDV
16 |   ...
```

# SEM : Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg : fixer des points de rendez-vous).

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   V(S2);
14 |   P(S1); P(S1);
15 |   RDV RDV
16 |   ...
```

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   P(S2);
14 |   V(S1); V(S1);
15 |   RDV RDV
16 |   ...
```

# SEM : Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg : fixer des points de rendez-vous).

```
10   S1 <- {BLOQUE, vide}
11   S2 <- {BLOQUE, vide}
12   ...
13   V(S2);
14   P(S1);
15   RDV
16   ...
```

```
10   S1 <- {BLOQUE, vide}
11   S2 <- {BLOQUE, vide}
12   ...
13   P(S2);
14   V(S1);
15   RDV
16   ...
```

# SEM : Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg : fixer des points de rendez-vous).

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   V( S2 );
14 |   P( S1 );
15 |   RDV
16 |   ...
```

```
10 |   S1 <- {BLOQUE, vide}
11 |   S2 <- {BLOQUE, vide}
12 |   ...
13 |   P( S2 );
14 |   V( S1 );
15 |   RDV
16 |   ...
```

# SEM : Les différents outils

futex                    Sémaaphore rapide (Fast Mutex).

Ils ne sont utilisables qu'entre threads.

POSIX thread        Inclus une API de sémaphores.

Ils ne sont utilisables qu'entre threads.

IPC System V        Voir « `sh> man svipc` » (sv=System V).

Ils sont utilisables sans restriction.

8

## Processus

- Processus Unix
- Thread POSIX
- Fonction réentrant et thread-safe

# fork : Syntaxe

Synopsis pid\_t fork(void)

Fonction Crée un clone du processus courant. Ce clone est un fils du processus courant.

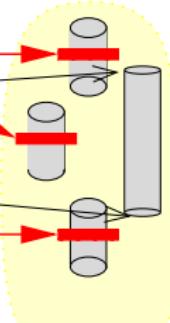
Retour En cas de succès 0 dans le fils et PID du fils dans le père. En cas de d'échec -1 et errno est mis à jour (dans le père seulement).

Exemple

```
1 ... // le père tourne
2 int pid =fork();
3 if ( pid==0 ) { // fils
4     execvp( "ls" , "ls" , "-l" , NULL );
5     ...
6     exit(1);
7 }
8 // le père continue ici
9 if ( pid<0 ) { ... ; exit(1); }
```

# fork : Principe

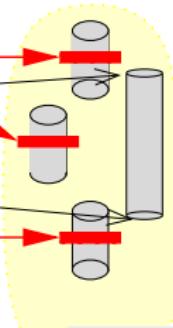
Unité d'exécution				MMU
PC	SP	R0	R1	MMU
T	D	S	SYS.	
PID CWD UID GID0 ...				
ARG0 ARG1 ...		ENV0 ENV1 ...		
REC		*		
ACT	C	T	T	C h1 S I



PID : Nouvelle valeur.  
E.V. Utilisateur : cloné.  
MMU : Pointe sur le clone.  
Mémoire partagée : Conservée.  
Flux : Conservés (mêmes curseurs)  
Signaux : Reçus effacés,  
gestionnaires conservés  
Registres : Identiques, sauf un  
(retour du fork)

# fork : Principe

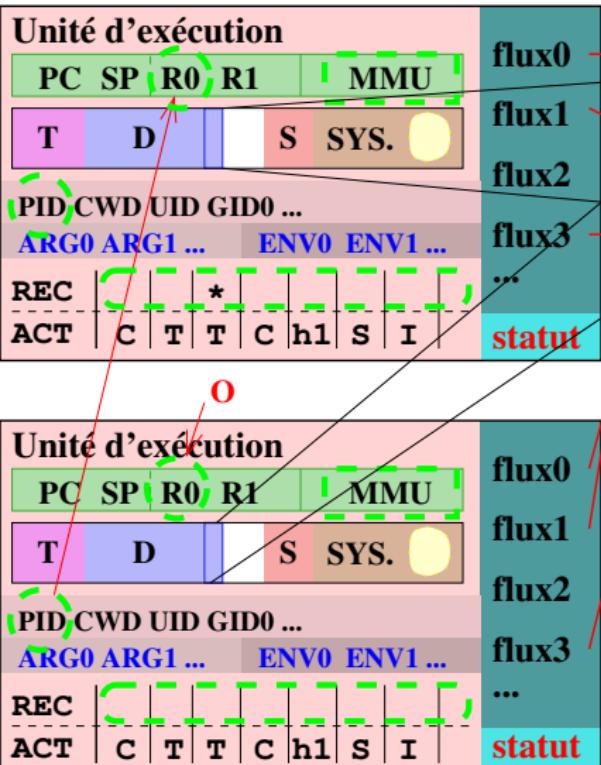
Unité d'exécution				MMU
PC	SP	R0	R1	MMU
T	D	S	SYS.	
PID CWD UID GID0 ...				
ARG0 ARG1 ...		ENV0 ENV1 ...		
REC		*		
ACT	C	T	T	Ch1 S I
				statut



Unité d'exécution			

PID : Nouvelle valeur.  
E.V. Utilisateur : cloné.  
MMU : Pointe sur le clone.  
Mémoire partagée : Conservée.  
Flux : Conservés (mêmes curseurs)  
Signaux : Reçus effacés,  
gestionnaires conservés  
Registres : Identiques, sauf un  
(retour du fork)

# fork : Principe



PID : Nouvelle valeur.  
E.V. Utilisateur : cloné.  
MMU : Pointe sur le clone.  
Mémoire partagée : Conservée.  
Flux : Conservés (mêmes curseurs)  
Signaux : Reçus effacés,  
gestionnaires conservés  
Registres : Identiques, sauf un  
(retour du fork)

# fork : Principe

Fils clone parfait du père à part le PID et R0.

Père et fils reviennent en mode utilisateur et exécutent le même code mais dans des espaces physiques différents.

Sont ils indépendants pour :

- ① Le déroulement du code ?
- ② La lecture et l'écriture mémoire dans leur E.V ?
- ③ La fermeture et l'ouverture de flux ?
- ④ La lecture et l'écriture des flux ?
- ⑤ La gestion des signaux ?

# Proc. léger : Syntaxe

**Synopsis** pid\_t sys\_clone(int flag, void\* stack, ...)

**Fonction** Crée un clone du processus courant. Ce clone est un fils du processus courant. flags indique ce qui est partagé entre les 2 processus, Le fils aura son pointeur de pile initialisé à stack.

**Retour** En cas de succès 0 dans le fils et PID du fils dans le père. En cas de d'échec -1 et errno est mis à jour (dans le père seulement).

# Proc. léger : Syntaxe

## Exemple

```
1   ... // le père tourne
2   sys_clone_asm(pid,
3     CLONE_VM|CLONE_FILE|CLONE_SIGHAND|SIGCHLD,
4     ((uchar*)malloc(SZ))+SZ);
5   if ( pid==0 ) {
6     // le fils continue ici
7     execlp("ls","ls","-l",NULL);
8     ...
9     exit(1);
10 }
11 // le père continue ici
12 if ( pid<0 ) { ... ; exit(1); }
```

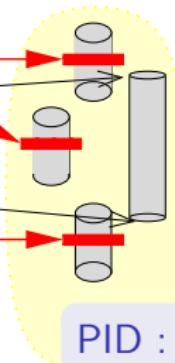
## Proc. léger : Syntaxe

**Remarque** Le wrapper direct à l'appel système `sys_clone` n'existe pas dans la libc, car le changement de pile rend son implémentation impossible, syscall n'est d'aucun secours, il faut le faire en assembleur.

Dans la libc il existe une fonction `clone` qui donne la main au fils dans une fonction (voir `man clone`).

# Proc. léger : Principe

Unité d'exécution				MMU
PC	SP	R0	R1	MMU
T	D	S	SYS.	
PID CWD UID GID0 ...				
ARG0 ARG1 ... ENV0 ENV1 ...				
REC		*		
ACT	C	T	T	C h1 S I
				statut



PID : Nouvelle valeur.

E.V. Utilisateur : Partagé.

MMU : non modifiée.

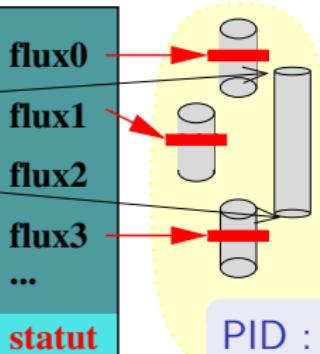
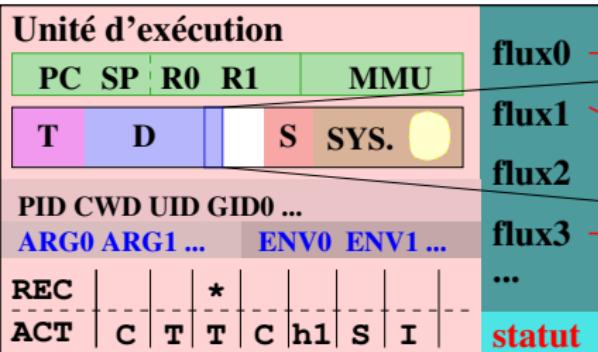
Mémoire partagée : Conservée.

Flux : Conservés (mêmes flux)

Signaux : Reçus effacés,  
gestionnaires partagés

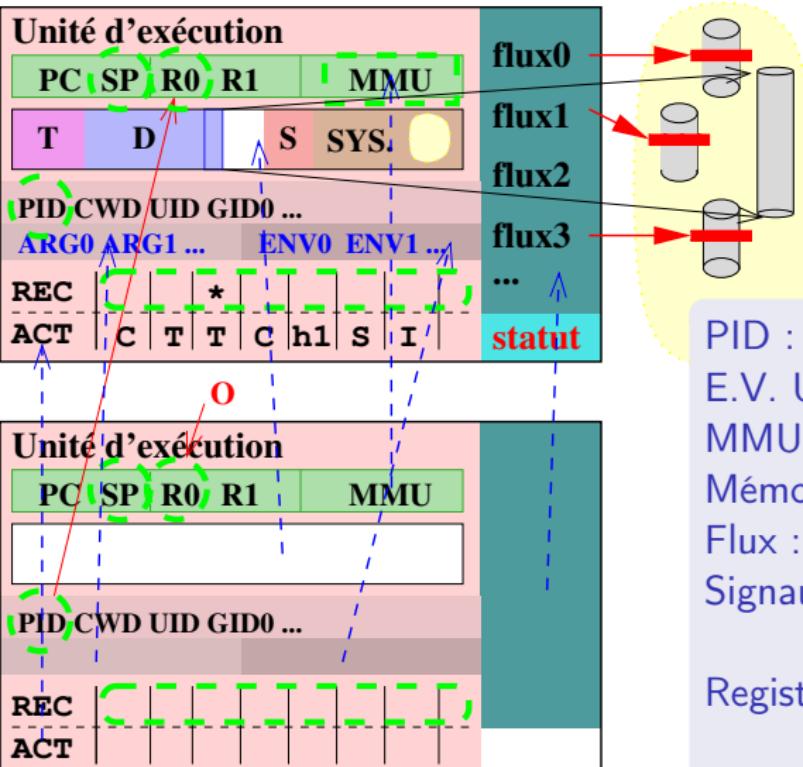
Registres : Identiques, sauf SP (pile  
vierge) et un autre (retour du  
clone)

# Proc. léger : Principe



PID : Nouvelle valeur.  
E.V. Utilisateur : Partagé.  
MMU : non modifiée.  
Mémoire partagée : Conservée.  
Flux : Conservés (mêmes flux)  
Signaux : Reçus effacés,  
gestionnaires partagés  
Registres : Identiques, sauf SP (pile  
vierge) et un autre (retour du  
clone)

# Proc. léger : Principe



PID : Nouvelle valeur.

E.V. Utilisateur : Partagé.

MMU : non modifiée.

Mémoire partagée : Conservée.

Flux : Conservés (mêmes flux)

Signaux : Reçus effacés,  
gestionnaires partagés

Registres : Identiques, sauf SP (pile  
vierge) et un autre (retour du  
clone)

# Proc. léger : Principe

Fils clone parfait du père à part le PID, SP, et R0.

Père et fils reviennent en mode utilisateur et exécutent le même code dans le même espace physique.

Sont ils indépendants pour :

- ① Le déroulement du code ?
- ② Que se passe-t-il si le fils atteint la fin de la fonction (instruction C return) qui a appelé sys\_clone ?
- ③ La lecture et l'écriture mémoire dans leur E.V ?
- ④ La fermeture et l'ouverture de flux ?
- ⑤ La lecture et l'écriture des flux ?
- ⑥ La gestion des signaux ?

# Attente : Principe

Un processus père peut se mettre en attente d'événements sur ses processus fils :

- terminaison du fils
- suspension du fils
- réactivation du fils

# Attente : Syntaxe

Synopsis pid\_t wait(int\* status)

Synopsis pid\_t waitpid(-1,int\* status,  
WUNTRACED|WCONTINUED)

Fonction Attend un événement sur un processus fils, et le code dans status. wait ne traque que la terminaison d'un fils. waitpid traque la terminaison, la suspension ou l'activation du fils.

Retour Le pid du processus fils qui a subit l'événement, sinon -1 et errno est mis à jour.

# Attente : Syntaxe

## Exemple

```
1 int status , pid ;  
2 ...  
3 pid = wait(&status);  
4 if ( pid == -1 )  
5     fprintf( stderr , "%d : pas de fils \n" , getpid () );  
6 else if ( WIFEXITED( status ) )  
7     fprintf( stderr ,  
8             "%d : child %d exited with status %d \n" ,  
9             getpid () , pid , WEXITSTATUS( status ) );  
10 else if ( WIFSIGNALED( status ) )  
11     fprintf( stderr ,  
12             "%d : child %d exited due to signal %d \n" ,  
13             getpid () , ret , WTERMSIG( status ) );  
14 else  
15     fprintf( stderr , "%d : cas inattendu \n" , getpid () );  
16 ...
```

# Processus zombie

Unité d'exécution	statut
PID CWD UID GID ...	

Un processus qui se termine doit délivrer sa terminaison à son père.

Si son père ne *mange* pas sa terminaison, le noyau libère toutes ses allocations et ne conserve que son PID et sa terminaison.

Que se passe-t-il si le père meurt avant son fils ?

# Processus zombie

Les différentes façons pour un père de *manger* la terminaison d'un fils sont :

- Positionner le gestionnaire du signal SIGCHLD.
- Faire un wait ou waidpid qui renvoie le PID du fils.

# Processus : Exemple

```
8 int main(int argc, char* argv[])
9 {
10     int status, pid;
11
12     pid=fork();
13
14     if ( pid==0 ) { // fils
15         write(1,"hel",3);
16         exit(0);
17     }
18     // père
19     wait(&status);
20     write(1,"lo\n",3);
21
22     return 0;
23 }
```

# Table des matières

## 8 Processus

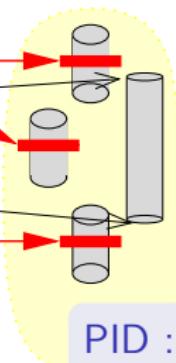
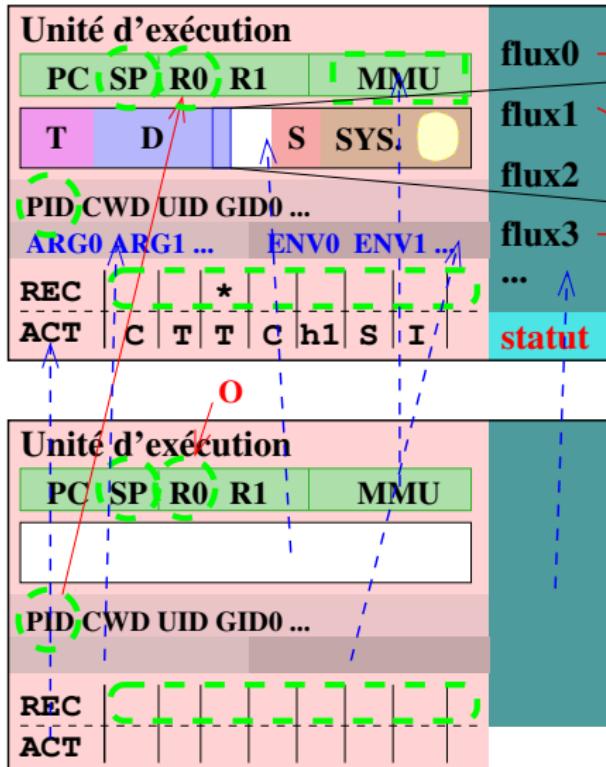
- Processus Unix
- Thread POSIX
- Fonction réentrant et thread-safe

# Pthread : Introduction

Les threads POSIX ou Pthread sont une API (publiée en 1995) pour le développement d'applications parallèles partageant les mêmes données.

- Gestion de processus légers (création, attente fin, ...).
- Synchronisation (sémaphore).
- Gestion de signaux.
- Gestion d'une zone locale de storage (TLS)
- Gestion de l'ordonnancement.

# Pthread : Introduction



- PID : Nouvelle valeur.
- E.V. Utilisateur : Partagé.
- MMU : non modifiée.
- Mémoire partagée : Conservée.
- Flux : Conservés (mêmes flux)
- Signaux : Reçus effacés,  
gestionnaires partagés
- Registres : Identiques, sauf SP et un autre (retour du clone)

# Pthread : Introduction

POSIX y ajoute à (ou réserve une partie de) l'espace virtuel à la pile et à la TLS du nouveau processus. La TLS contient des variables :

- propres au thread pour sa gestion interne (ex : état du thread, valeur de retour, ...).
- utilisateur qui ne peuvent pas être partagées (ex : errno qui devient une fonction, les variables marquées `__thread` en C++).
- une table d'action de fin de thread. Pour l'utilisateur sa structure est (clé, pointeur, function). Au départ d'un thread cette table est vide. API propose des fonctions pour ajouter, rechercher, enlever des éléments à la table. En fin de thread tous les "function(pointeur)" de la table sont appelés dans l'ordre inverse d'ajout.

Attention : les piles et les TLS sont dans le même espace virtuel  $\Rightarrow$  tout thread peut modifier la pile ou le TLS de ses collègues.

# Pthread : Création I

## Synopsis

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  void *(*func) (void *), void *arg);  
  
void pthread_exit(void *ret);
```

## Fonction

`pthread_create` crée un nouveau thread et son point d'entrée est la fonction `func` avec l'argument `arg`.

`pthread_exit` termine le thread avec le statut `val`.

Un "return `x`;" dans `func` est équivalent à `pthread_exit(x)`.

**Retour** En cas de succès 0 sinon un numéro d'erreur (équivalent à `errno`).

# Pthread : Création II

## Exemple

```
9 void * print(void *str)
10 { printf((char*)str); return NULL; }
11
12 int main(int argc, char*argv) {
13     pthread_attr_t att;
14     pthread_attr_init(&att);
15
16     pthread_t th;
17     pthread_create(&th,&att,print,"hel");
18
19     sleep(1);
20     printf("lo\n");
21
22     return 0;
```

# Pthread : Création III

23 | }

# Pthread : Attente I

## Synopsis

```
int pthread_join(pthread_t th, void**statut);
```

## Fonction

Attend la fin « `pthread_exit(x)` » du thread `th` et délivre son statut (`x`) `*statut`.

**Retour** En cas de succès 0 sinon un numéro d'erreur (équivalent à `errno`).

## Exemple

# Pthread : Attente II

```
9 void * print(void *str)
10 { printf((char*)str); return NULL; }
11
12 int main(int argc, char*argv) {
13     pthread_attr_t att;
14     pthread_attr_init(&att);
15
16     pthread_t th;
17     pthread_create(&th,&att,print,"hel");
18
19     pthread_join(th,NULL);
20     printf("lo\n");
21
22     return 0;
23 }
```

# Pthread : Sémaphore d'exclusivité mutuelle I

## Synopsis

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

## Fonction

Crée un sémaphore mutex avec les fonction P (lock) et V (unlock).

**Retour** En cas de succès 0 sinon un numéro d'erreur (équivalent à errno).

## Exemple

# Pthread : Sémaphore d'exclusivité mutuelle II

```
9 #ifndef NOMUTEX
10
11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12 #define P() pthread_mutex_lock(&mutex)
13 #define V() pthread_mutex_unlock(&mutex)
14
15 #else
16
17 #define P()
18 #define V()
19
20#endif
```

# Pthread : Sémaaphore d'exclusivité mutuelle III

```
22 int a; // init. à 0 par défaut
23 void * add(void*signe)
24 {
25     int i;
26     for (i=0 ; i<(1<<23) ; i++) {
27         P();
28         int x=a;
29         if (signe!=0)
30             x = x + -2;
31         else
32             x = x + +2;
33         a=x;
34         V();
35     }
36     return NULL;
37 }
```

# Pthread : Sémafore d'exclusivité mutuelle IV

```
39 int main( int argc , char*argv ) {
40     pthread_attr_t att;
41     pthread_attr_init(&att );
42
43     pthread_t tha , thb ;
44     pthread_create(&tha ,&att , add , ( void* )0 );
45     pthread_create(&thb ,&att , add , ( void* )1 );
46
47     pthread_join( tha ,NULL );
48     pthread_join( thb ,NULL );
49
50     printf( "a=%d\n" ,a );
51
52     return 0;
53 }
```

# Pthread : Sémaphore d'exclusivité mutuelle V

## Expérimentation

```
sh> gcc -DNOMUTEX mutex.c -lpthread && ./a.out  
a=-7197708
```

```
sh> gcc -DNOMUTEX mutex.c -lpthread && ./a.out  
a=-7081580
```

```
sh> gcc mutex.c -lpthread && ./a.out  
a=0
```

```
sh> gcc mutex.c -lpthread && ./a.out  
a=0
```

```
sh>
```

Sur un PC Linux bi-processeurs, La version avec le sémaphore est environ 25 fois plus lente.

⇒ Quel rapport serait attendu ?

⇒ À quoi est dû le surplus ?

# Pthread : Threads POSIX sous Linux I

## Thread = Processus

Processus léger créé avec l'appel système `sys_clone`.

## Processus regroupés

Les threads partageant le même E.V sont regroupés dans un groupe.  
Appelons  $T_m$  le processus initial et  $T_a$  les autres.

## `exit(s)`

Dans un thread termine tous les thread du group.

## `getpid()` et `getppid()`

Dans un thread  $T_i$  donnent celui de  $T_m$ .

⇒ Les processus threads sont masqués

# Pthread : Threads POSIX sous Linux II

`syscall(SYS_gettid)`

Renvoie le vrai PID du processus.

⇒ Pour un  $T_m$ , `syscall(SYS_gettid)=getpid()`

`ps -Af`

Affiche tous les processus  $T_m$ .

`ps -L Af`

Affiche tous les processus  $T_m$  et  $T_a$ .

Gestionnaire de signal

Partagé par les  $T_i$ .

## Envoi d'un signal

Il faut l'envoyer à `syscall(SYS_gettid)`.

## Fork dans un $T_i$

Le processus  $T_i$  uniquement est cloné, le père du clone est  $T_m$ .

- ⇒ Tous les  $T_i$  peuvent faire un wait sur ce fils.
- ⇒ Tous attendront la mort du clone.
- ⇒ Tous sauf 1 recevront -1 avec errno "no child".

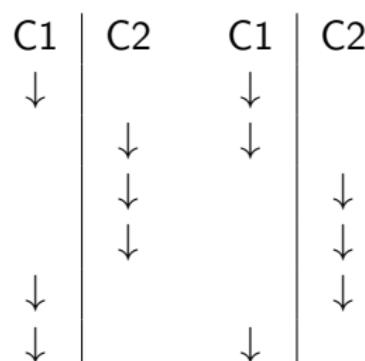
## 8 Processus

- Processus Unix
- Thread POSIX
- Fonction réentrant et thread-safe

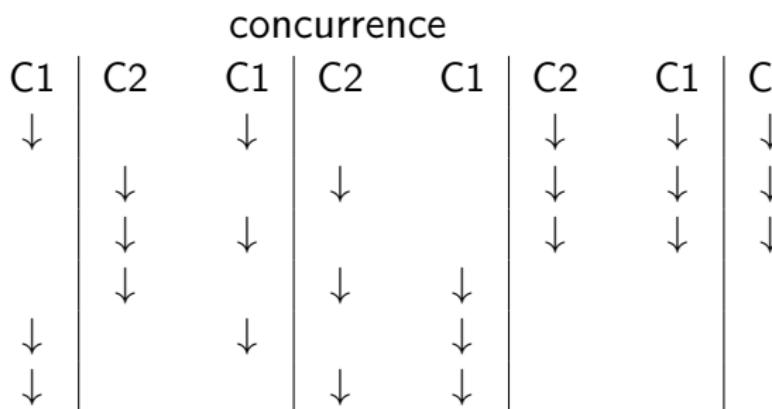
# Problème 1

Soit un code  $C_1$  manipulant des données  $D_1$ , et un code  $C_2$  manipulant des données  $D_2$ .  $C_1$  et  $C_2$  peuvent s'exécuter en :

mode interruption



$C_1$  s'interrompt,  $C_2$  s'exécute complètement puis  $C_1$  reprend.



$C_1$  et  $C_2$  s'exécutent sans ordre préétabli, et éventuellement en même temps.

## Problème II

Il faut que quelque soit le scénario à la fin les données  $D_1$  et  $D_2$  contiennent les bons résultats des 2 codes et soient dans un état cohérent.

Si  $D_1$  et  $D_2$  sont disjoints les codes sont résistants à une exécution concurrente et en mode interruption.

# Définition I

## Fonction réentrant

f est réentrant si un second appel à f se déroulant pendant le premier appel donne un résultat correct pour les 2 appels.

Par exemple, f se déroule, un signal est attrapé et le gestionnaire du signal appelle f.

## Fonction thread-safe

f est thread-safe si deux appels en parallèle donnent un résultat correct pour les deux appels.

Par exemple, 2 threads exécutent une fonction f en même temps,

# Appels système et fonctions de la libc I

## Appels système

Les appels système sont threadsafe.

Au niveau utilisateur, ils n'ont pas besoin d'être réentrant car si le gestionnaire de signal est appelé, il n'y a pas d'appel système en cours.

## Fonctions de la libc

Les fonctions de la libc sont réentrant et threadsafe sauf mention contraire dans la page de man.

Dans ce cas il existe une fonction équivalente qui l'est.

# Exemple 1

Les fonctions de lecture et écriture de la libc sont thread-safe.

Elles fonctionnent ainsi :

- Posent un verrou
- Font leur job
- Relâchent le verrou

De ce fait le code ci-contre pose autant de verrous qu'il lit de caractères.

```
1 #include <stdio.h>
2
3 int main( int argc , char*argv[] )
4 {
5     char c; int len=0;
6     while ( fread(&c,1,1,stdin ) ==
7             len += 1;
8     printf( "len=%d\n" , len );
9     return 0;
10 }
```

Toutes les fonctions de lecture et écriture des flux libc ont leurs équivalents sans verrou (ex : printf  $\Rightarrow$  printf unlocked).

## Exemple II

```
sh> gcc test.c && time ./a.out < 10m
real 0m0.233s
sh> gcc -Dfread=fread_unlocked test.c && \
    time ./a.out < 10m
real 0m0.180s
sh>
```

La capture d'écran ci-dessus montre que les poses et les relâchements du verrou prennent 1/3 du temps d'exécution.

De plus en optimisant, gcc inline fread\_unlocked, ce qui donne :

```
sh> gcc -O2 -Dfread=fread_unlocked test.c && \
    time ./a.out < 10m
real 0m0.031s
sh>
```