La mémoire, les primitives exec et dup

Luigi Santocanale

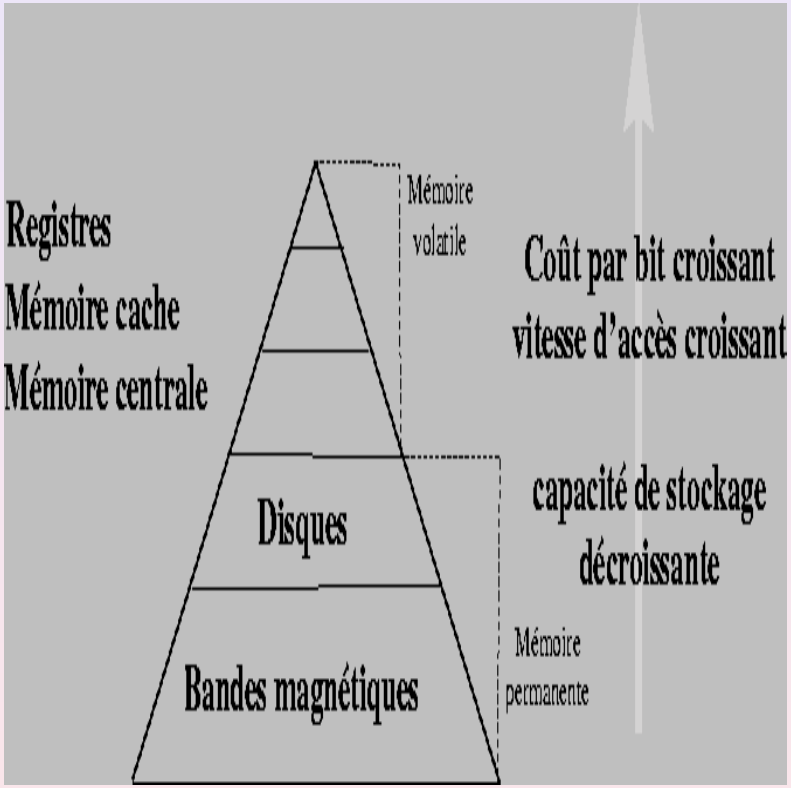
Laboratoire d'Informatique Fondamentale,
Centre de Mathématiques et Informatique,
39, rue Joliot-Curie - F-13453 Marseille

24 novembre 2004

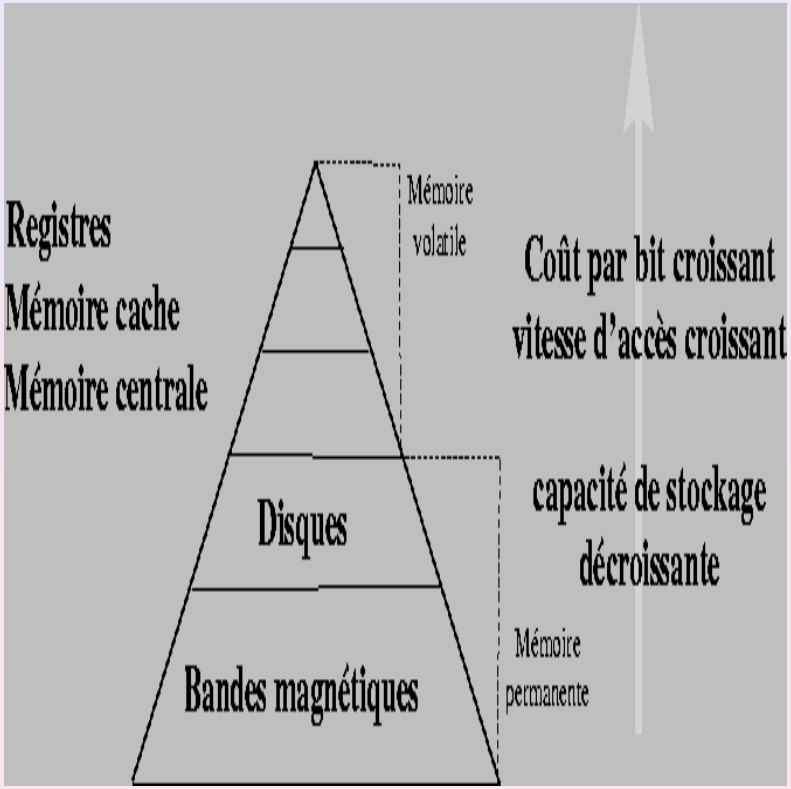
Plan

Plan

Les mémoires



Les mémoires



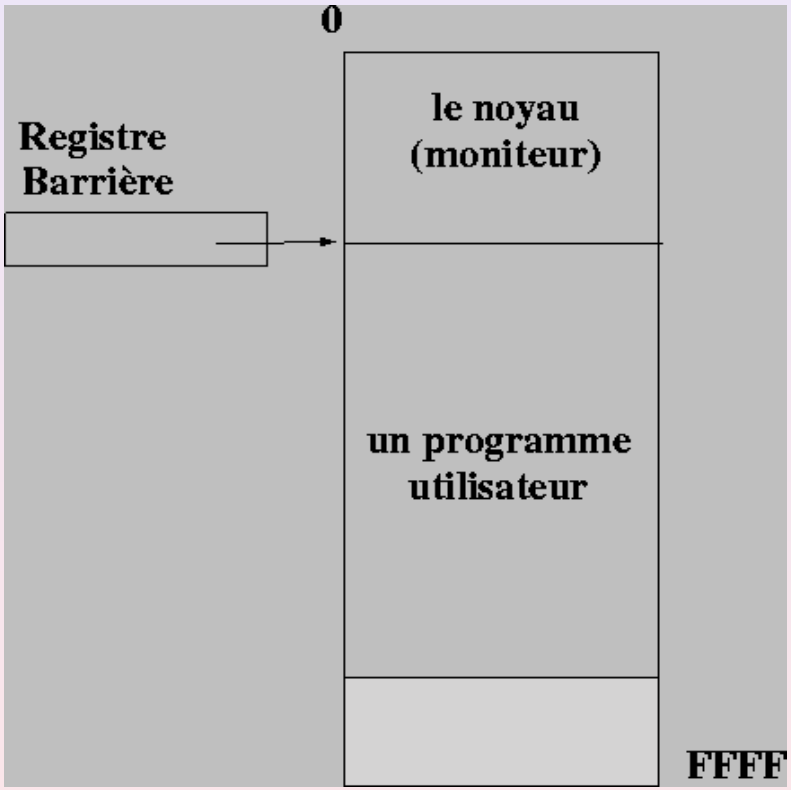
CARACTERISTIQUES DES TYPES DE MEMOIRES

TYPE DE MEMOIRE	TAILLE (Octets)	TEMPS D'ACCES (secondes)	COUT RELATIF PAR BIT
CACHE	10^3-10^4	10^{-8}	10
MEMOIRE CENTRALE	10^6-10^7	10^{-7}	1
DISQUE	10^8-10^9	10^{-3}-10^{-2}	10^{-2}-10^{-3}
BANDE	10^8-10^9	10-10^2	10^{-4}

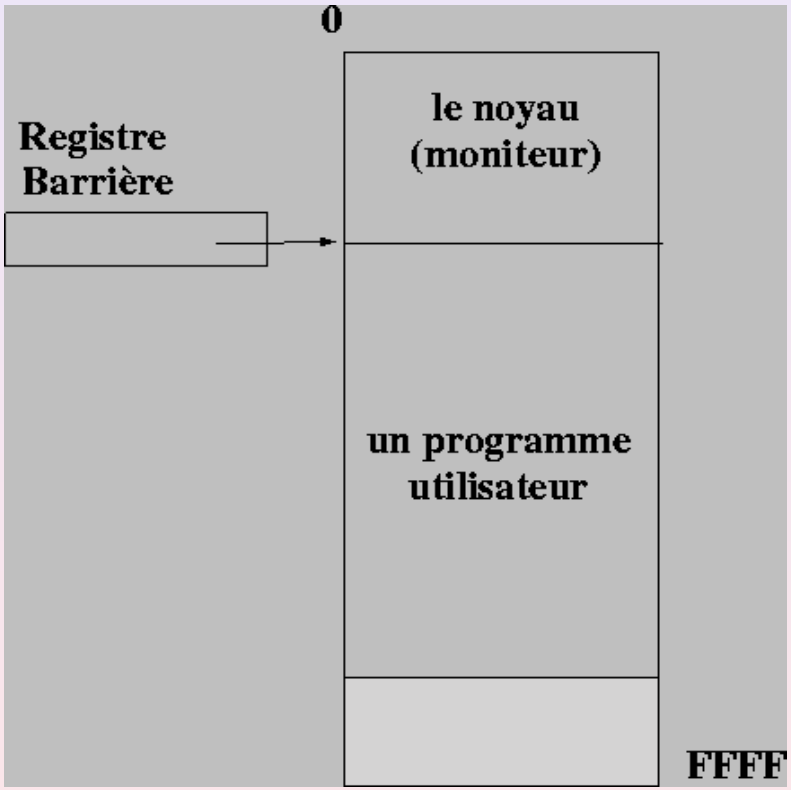
Plan

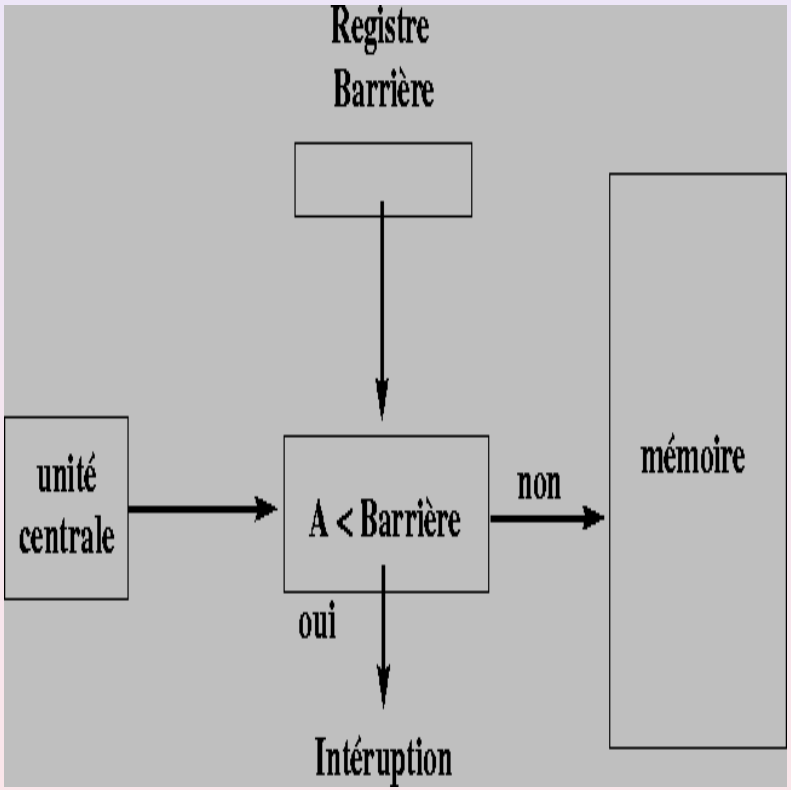
Protection du noyau : le registre barrière

Protection du noyau : le registre barrière

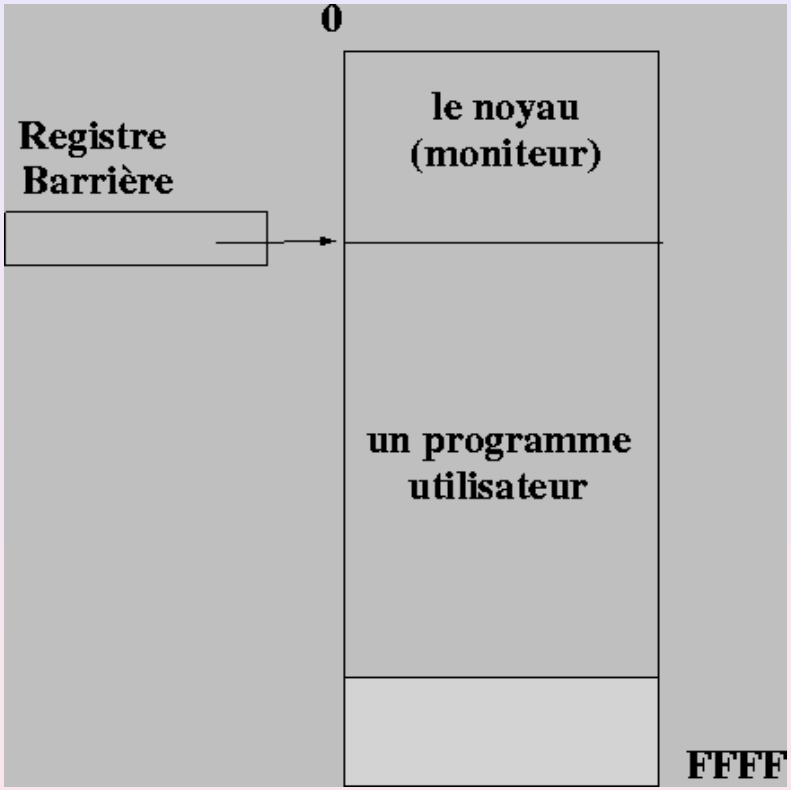


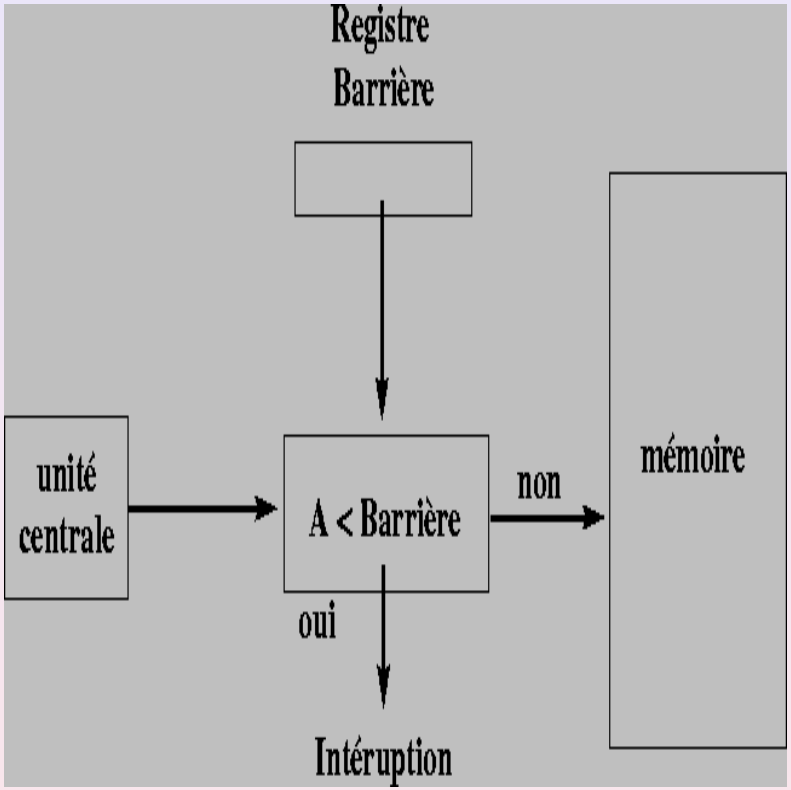
Protection du noyau : le registre barrière





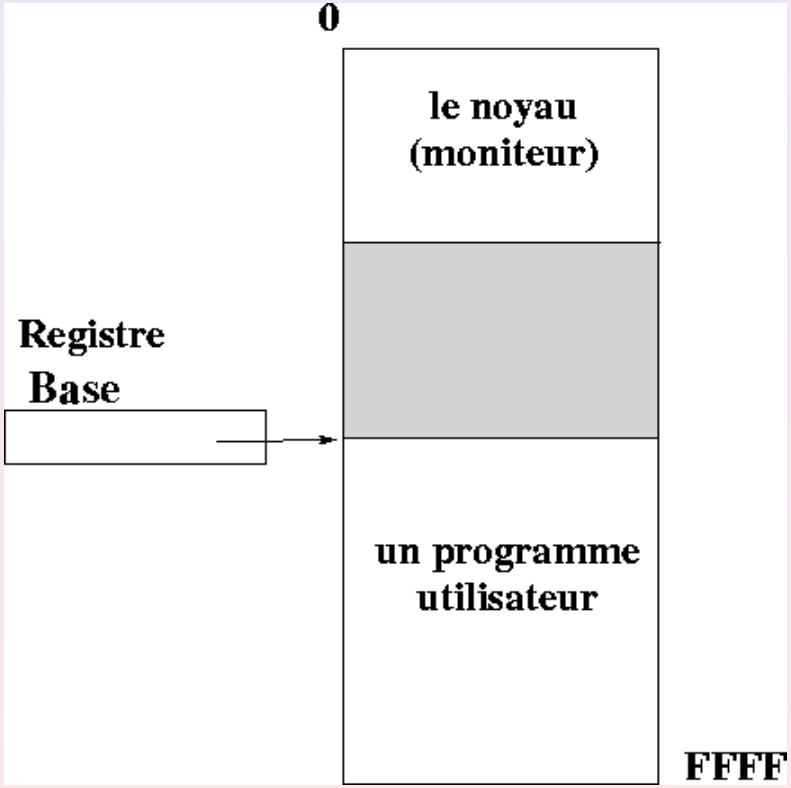
Protection du noyau : le registre barrière



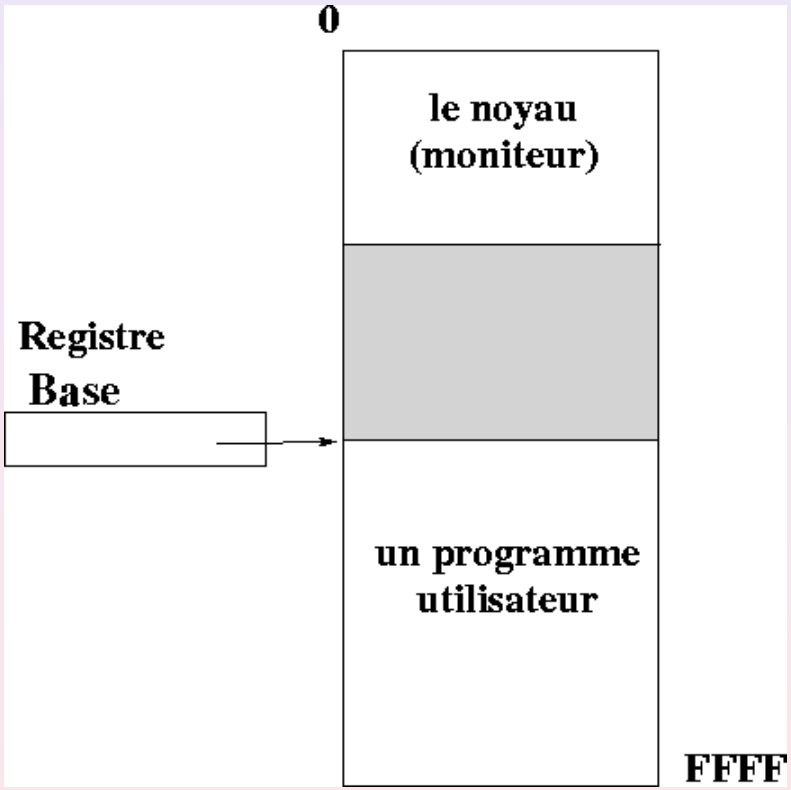


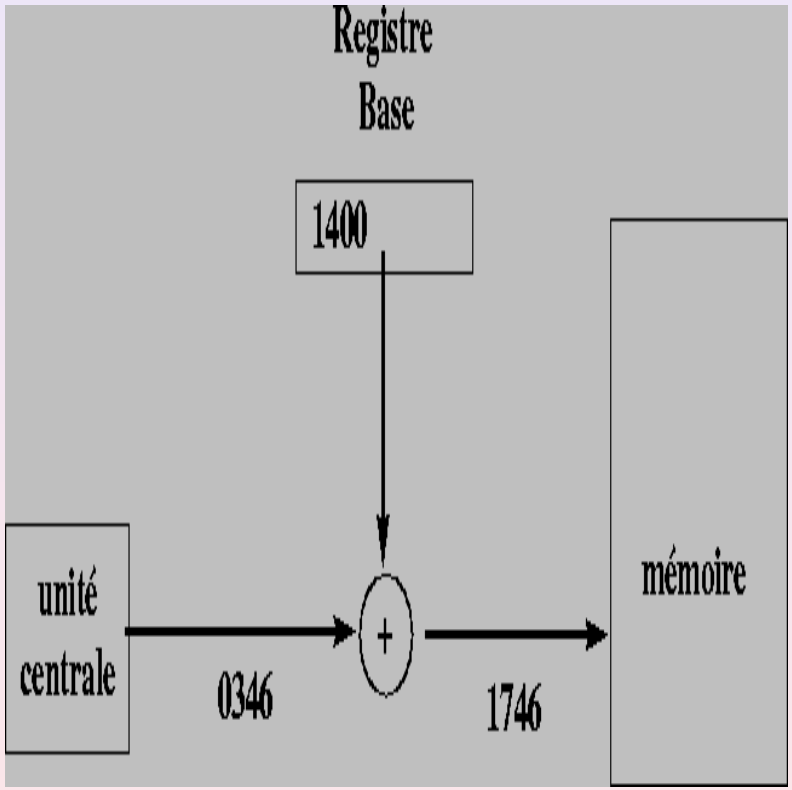

```
si ( adresse <= Barriere )
    lever exception
sinon
    utiliser adresse
```

Protection du noyau : le registre base

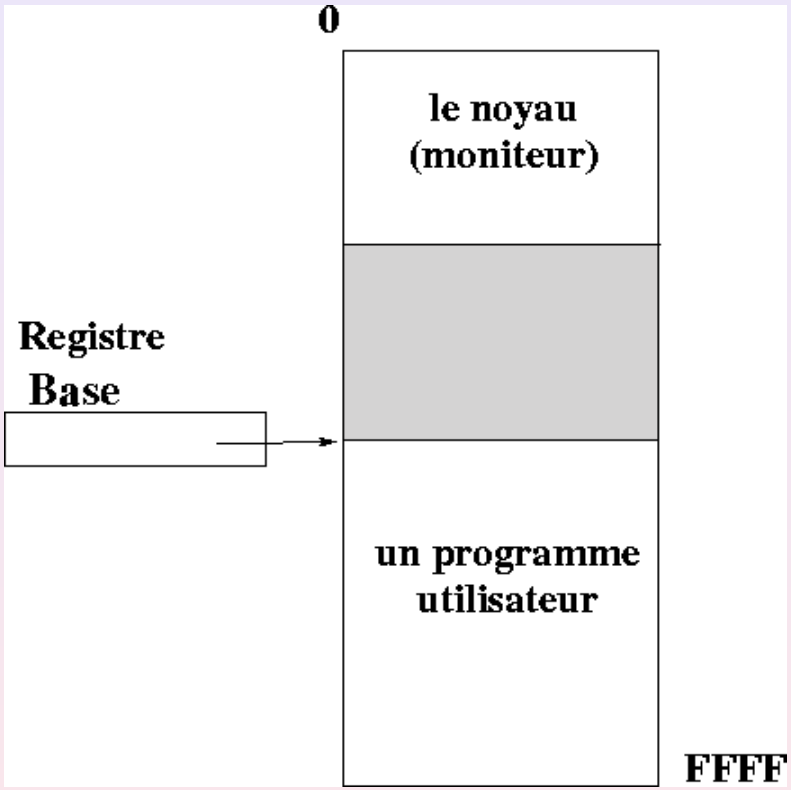


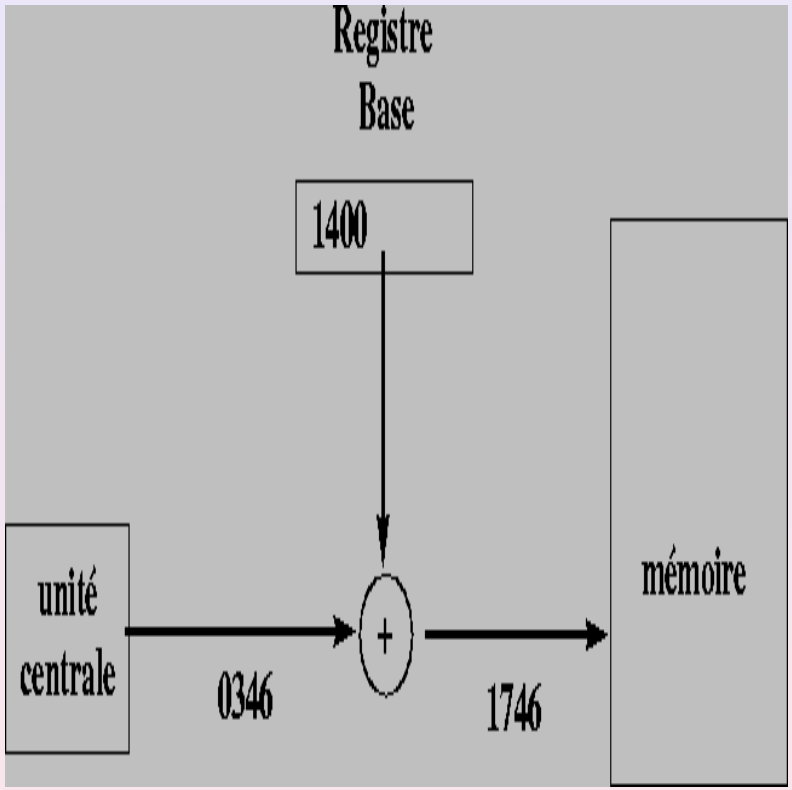
Protection du noyau : le registre base





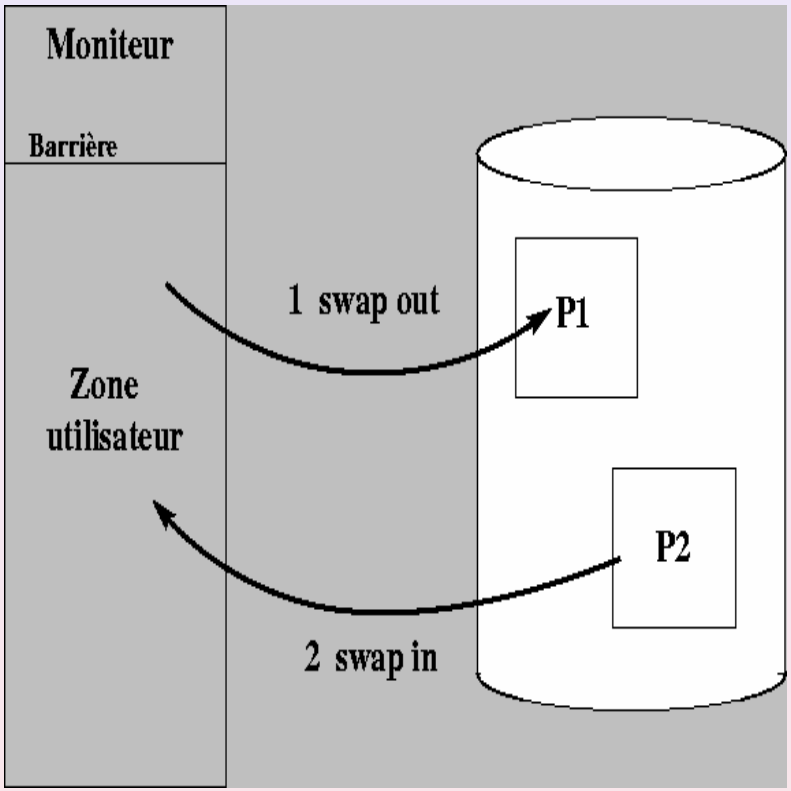
Protection du noyau : le registre base





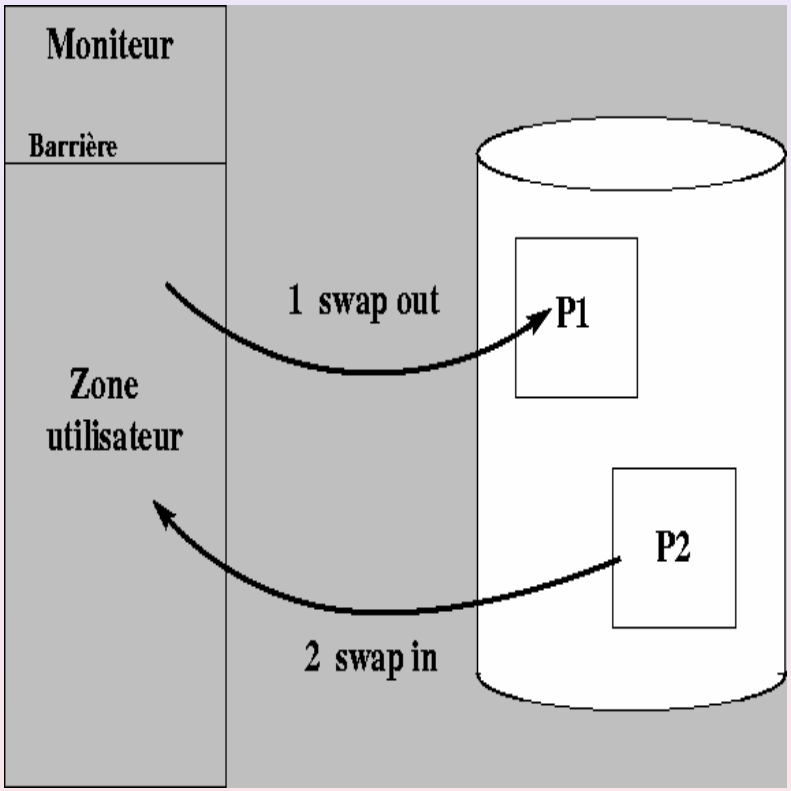

```
adressephysique =
    adresselogique + Base
```

Le partage de la mémoire : le swap



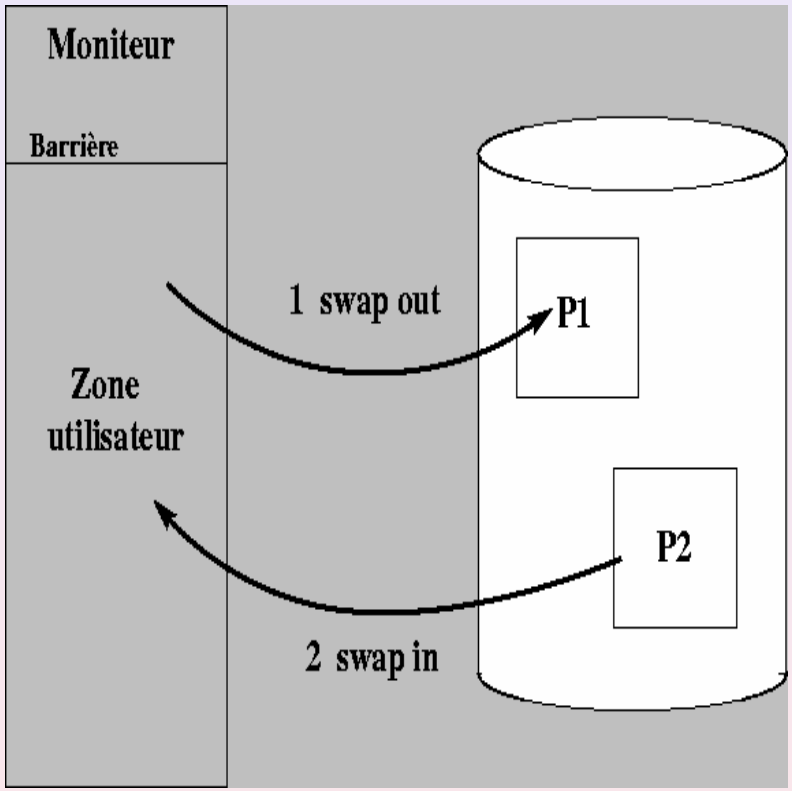
- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en attente de E/S
  - réaliser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : le swap



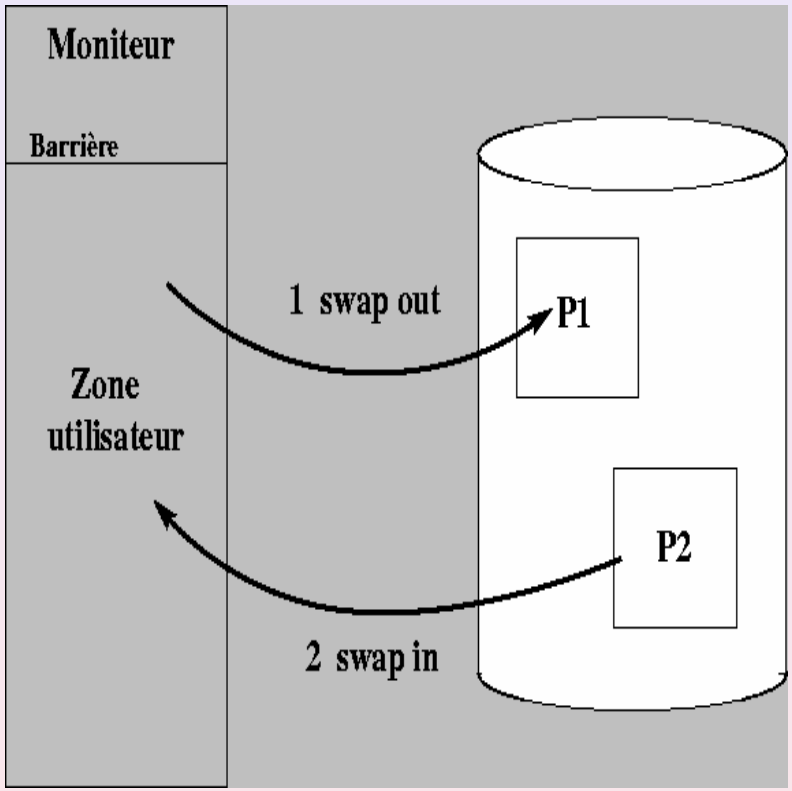
- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en en attante de E/S
  - realiser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : le swap



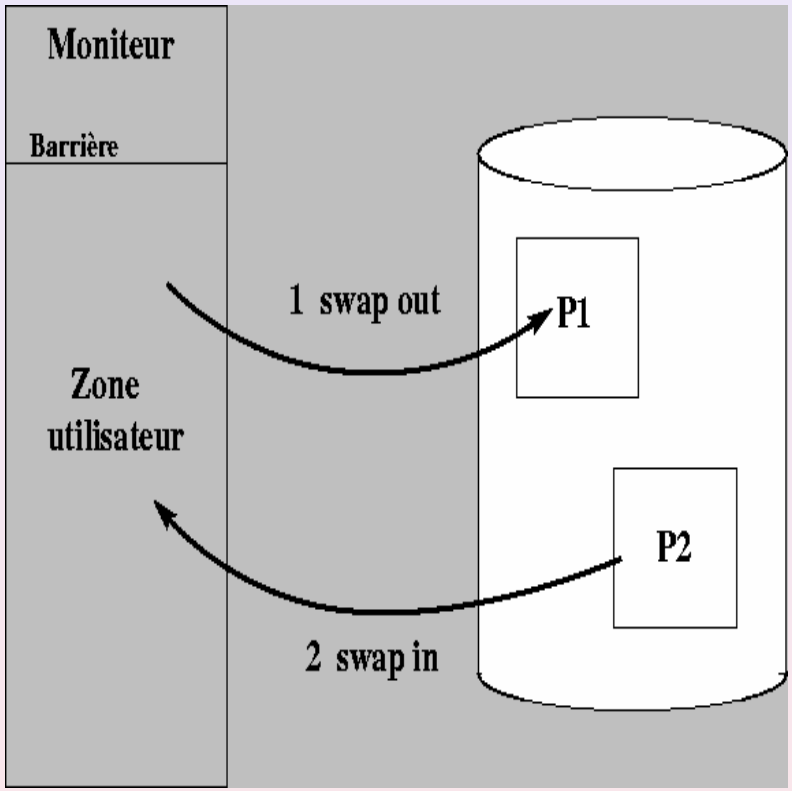
- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en en attante de E/S
  - réaliser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : le swap



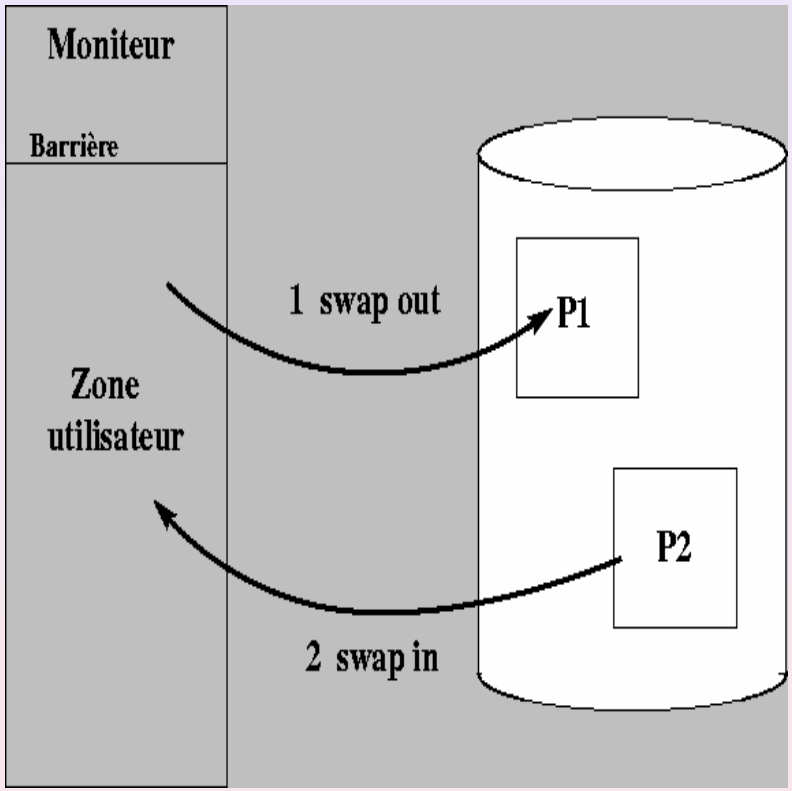
- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en en attante de E/S
  - réaliser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : le swap



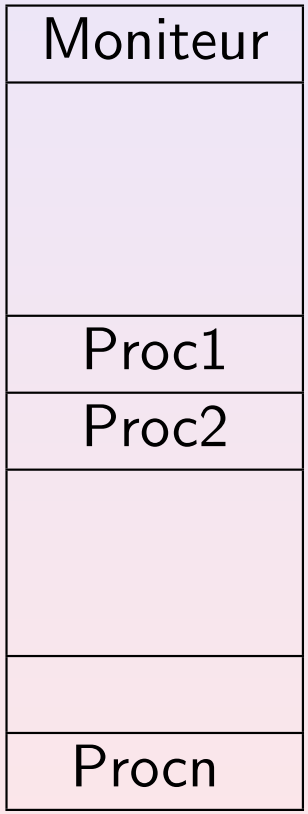
- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en en attante de E/S
  - réaliser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : le swap



- Coût
- Taille des processus
- Contraintes sur le E/S :
  - ne pas swapper les processus en en attante de E/S
  - réaliser des buffers de E/S dans le noyau (voir UNIX).

Le partage de la mémoire : plusieurs processus en mémoire

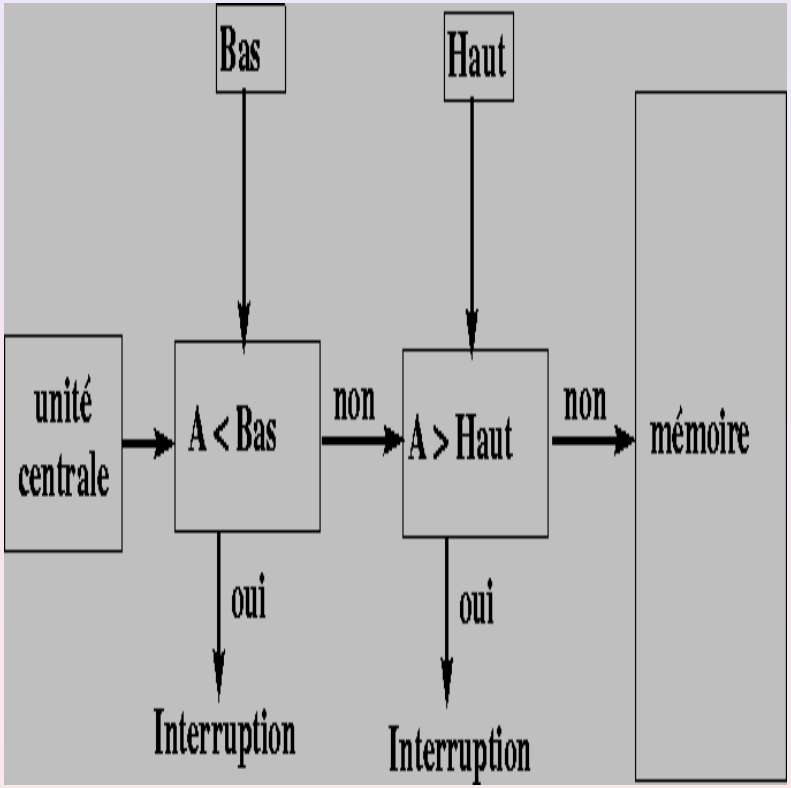


Problèmes :

- Protection entre processus
- Allocation de la mémoire aux processus (ordonnancement)

La protection entre processus : deux registres

Barrière et Haut:



Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

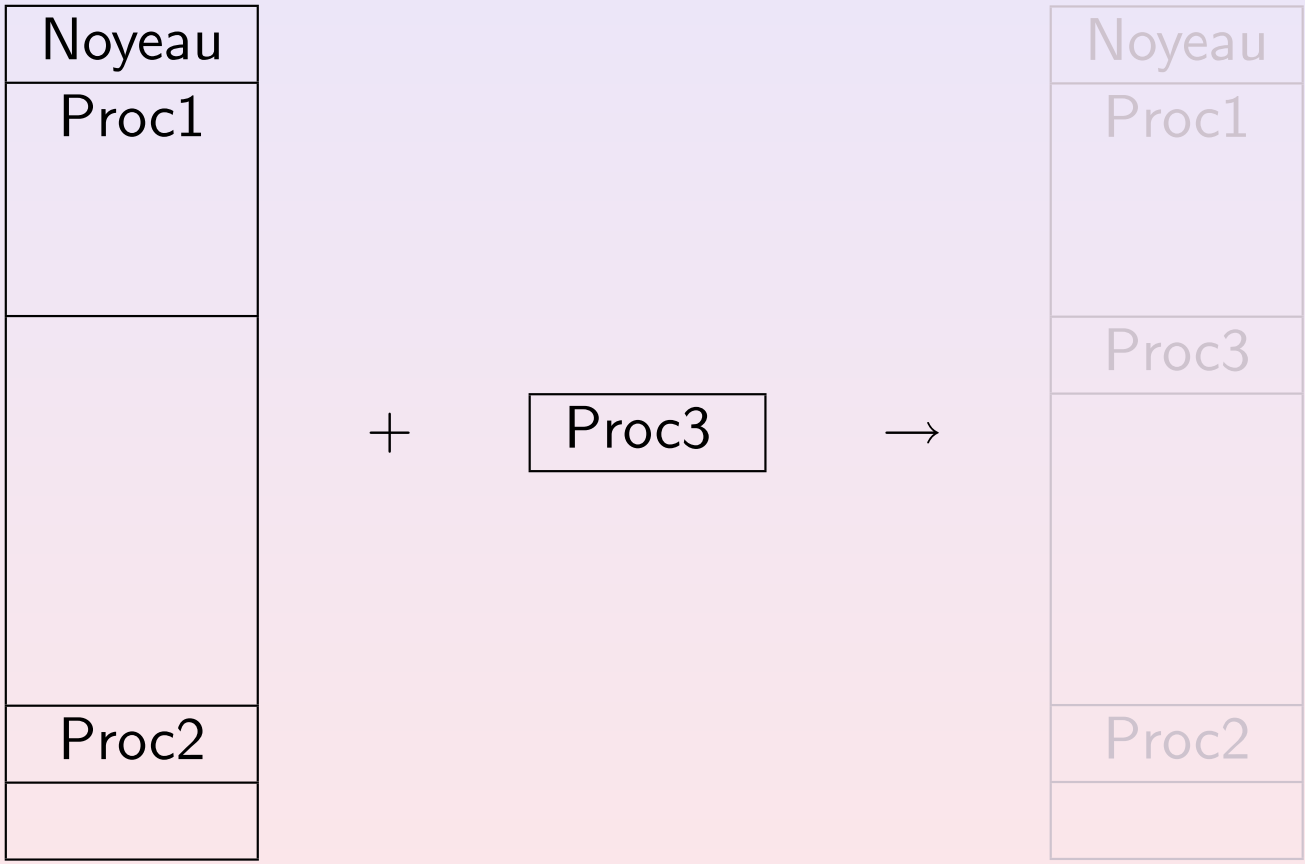
Ordonnancement en mémoire

Comment choisir l'endroit où charger un nouveaux processus :

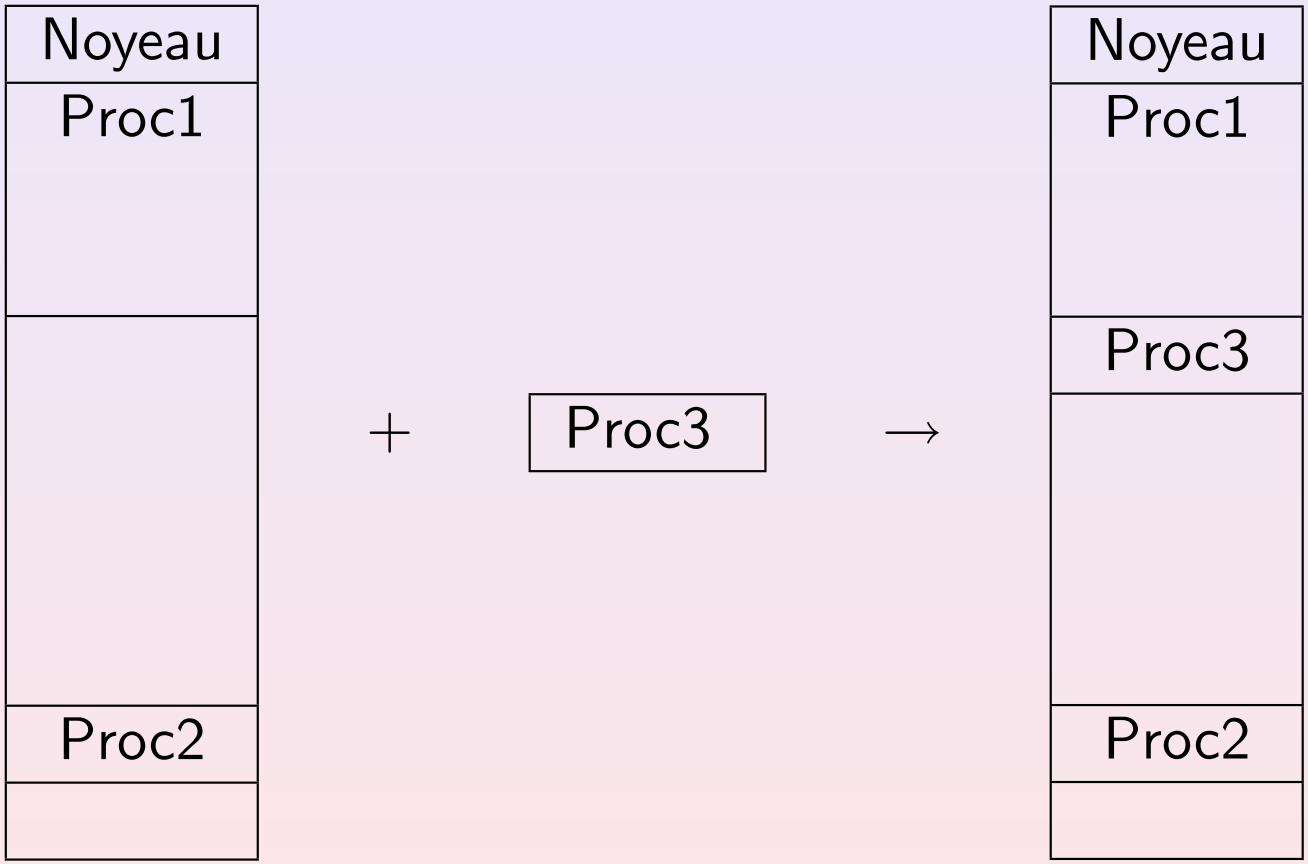
- First-fit : premier bloc suffisamment grand.
- Best-fit : plus petit bloc suffisamment grand.
- Worst-fit : le bloc qui nous laisse le plus grand bloc libre (le plus grand bloc).

Problème : la fragmentation.

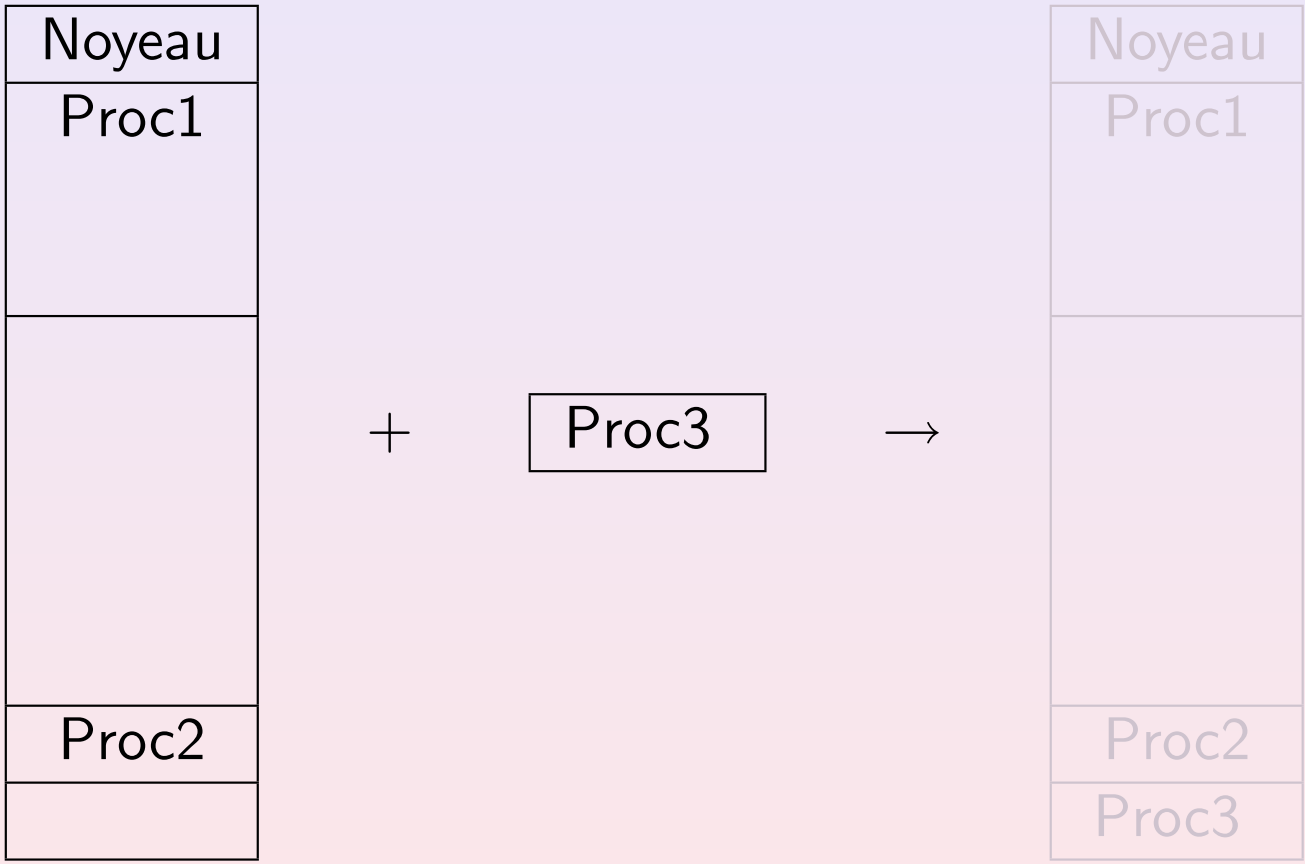
First fit



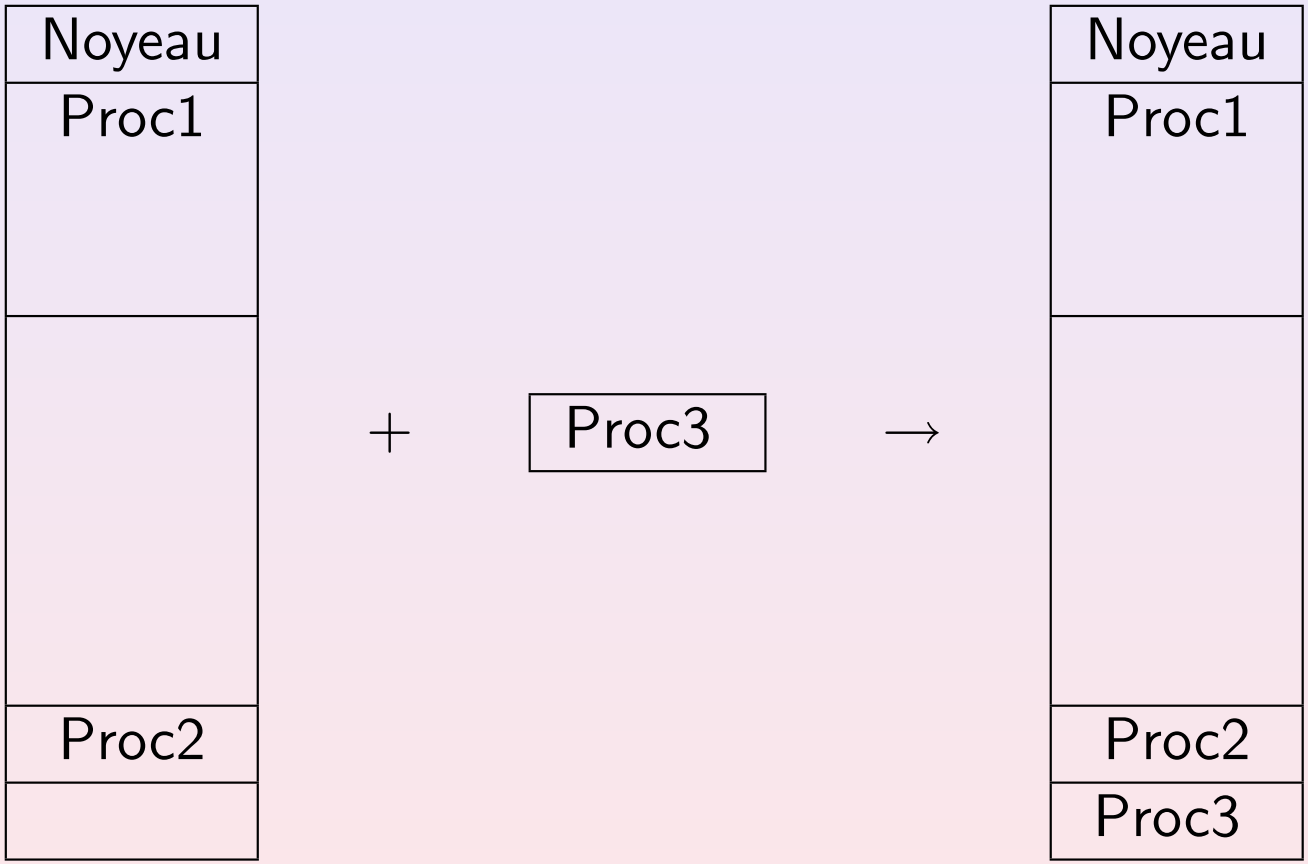
First fit



Best & worst fit

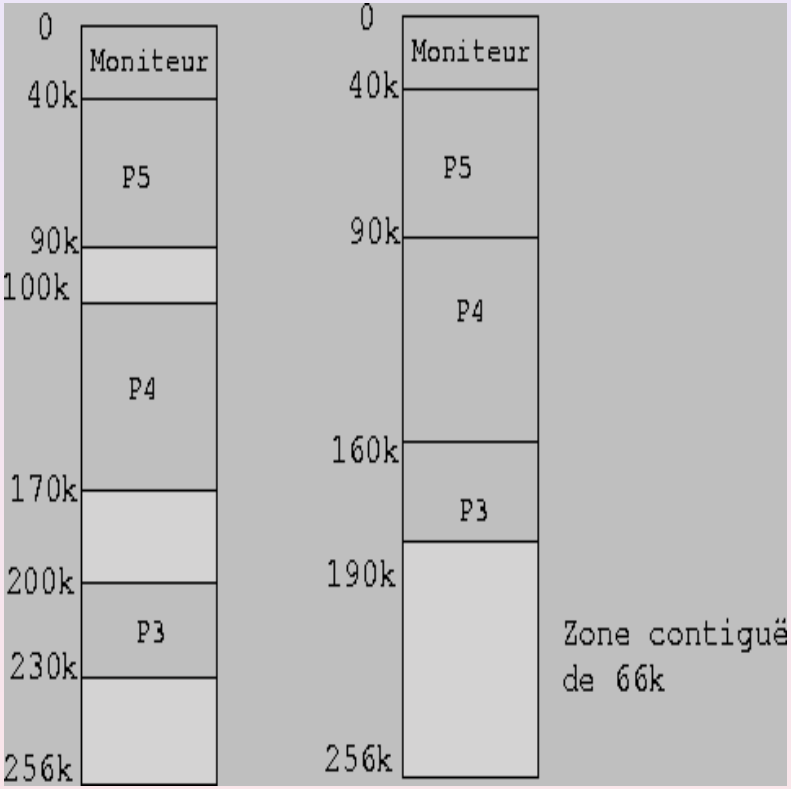


Best & worst fit



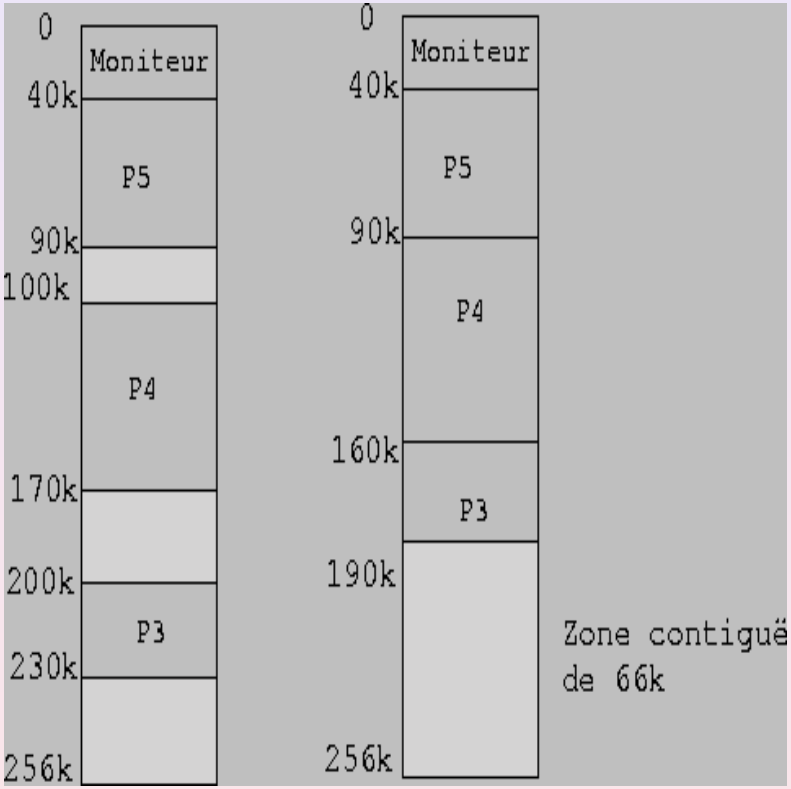
Le compactage

Solution à la fragmentation.

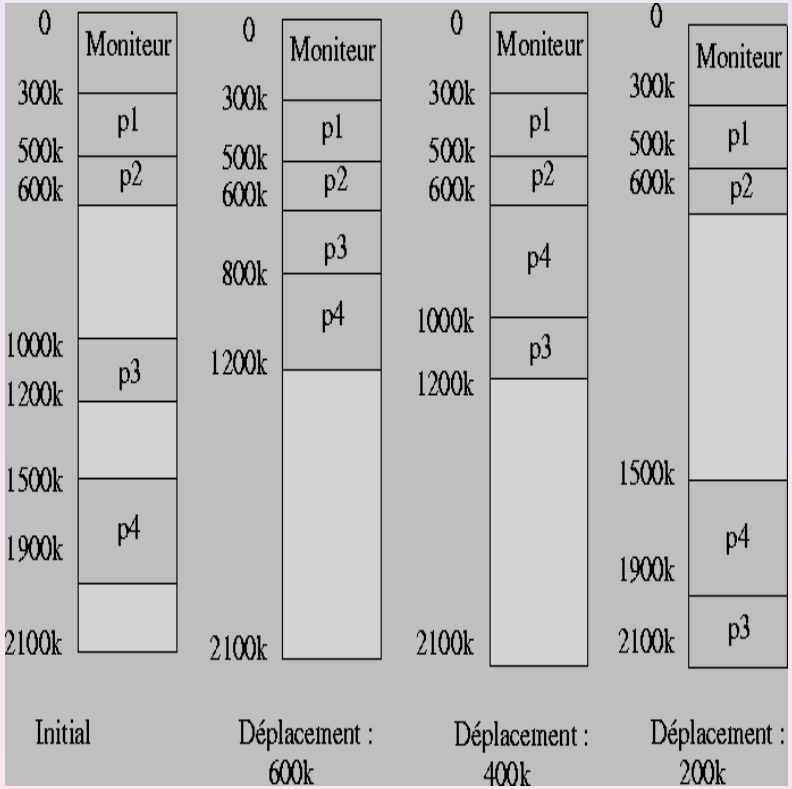


Le compactage

Solution à la fragmentation.



Exemples :



Plan

La pagination

But : solution à la fragmentation,
Cf. la gestion de la mèmoire sur disque.

- La mémoire est découpée en pages.
- Une adresse logique est découpée en une couple :
(no. de page, position dans la page)

Si

A — adresse logique, T — taille de page

alors

no de page $= A/T$, position $= A\%T$

La pagination

But : solution à la fragmentation,
Cf. la gestion de la mèmoire sur disque.

- La mémoire est découpée en pages.
- Une adresse logique est découpée en une couple :
(no. de page, position dans la page)

Si

$A =$ adresse logique, $T =$ taille de page

alors

no de page $= A/T$, position $= A\%T$

La pagination

But : solution à la fragmentation,
Cf. la gestion de la mèmoire sur disque.

- La mémoire est découpée en pages.
- Une adresse logique est découpée en une couple :
(no. de page, position dans la page)

Si

$$A = \text{adresse logique}, \qquad T = \text{taille de page}$$

alors

$$\text{no de page} = A/T, \qquad \text{position} = A\%T$$

La pagination

But : solution à la fragmentation,
Cf. la gestion de la mèmoire sur disque.

- La mémoire est découpée en pages.
- Une adresse logique est découpée en une couple :
(no. de page, position dans la page)

Si

$$A = \text{adresse logique}, \qquad T = \text{taille de page}$$

alors

$$\text{no de page} = A/T, \qquad \text{position} = A\%T$$

La pagination

But : solution à la fragmentation,
Cf. la gestion de la mèmoire sur disque.

- La mémoire est découpée en pages.
- Une adresse logique est découpée en une couple :
(no. de page, position dans la page)

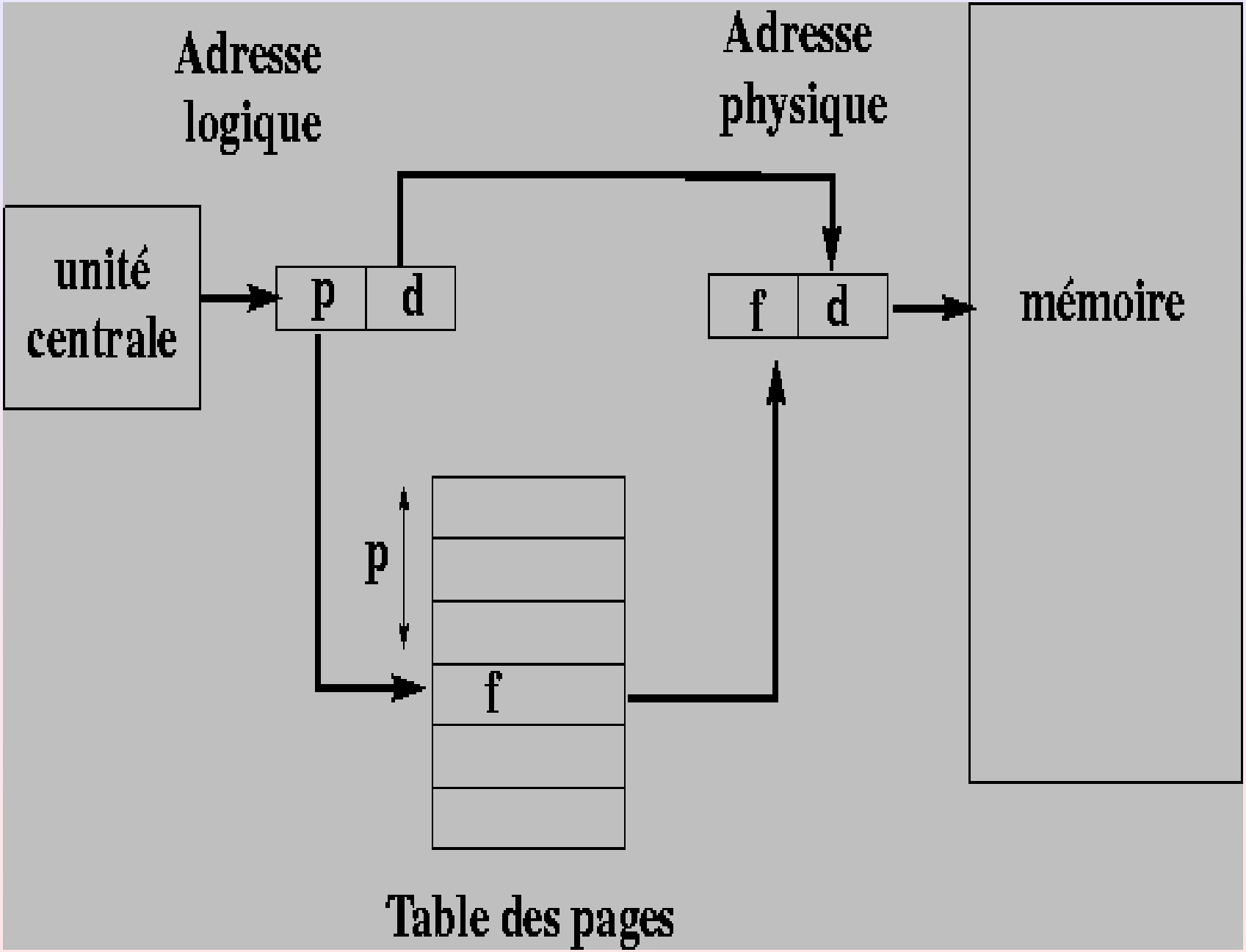
Si

$$A = \text{adresse logique}, \qquad T = \text{taille de page}$$

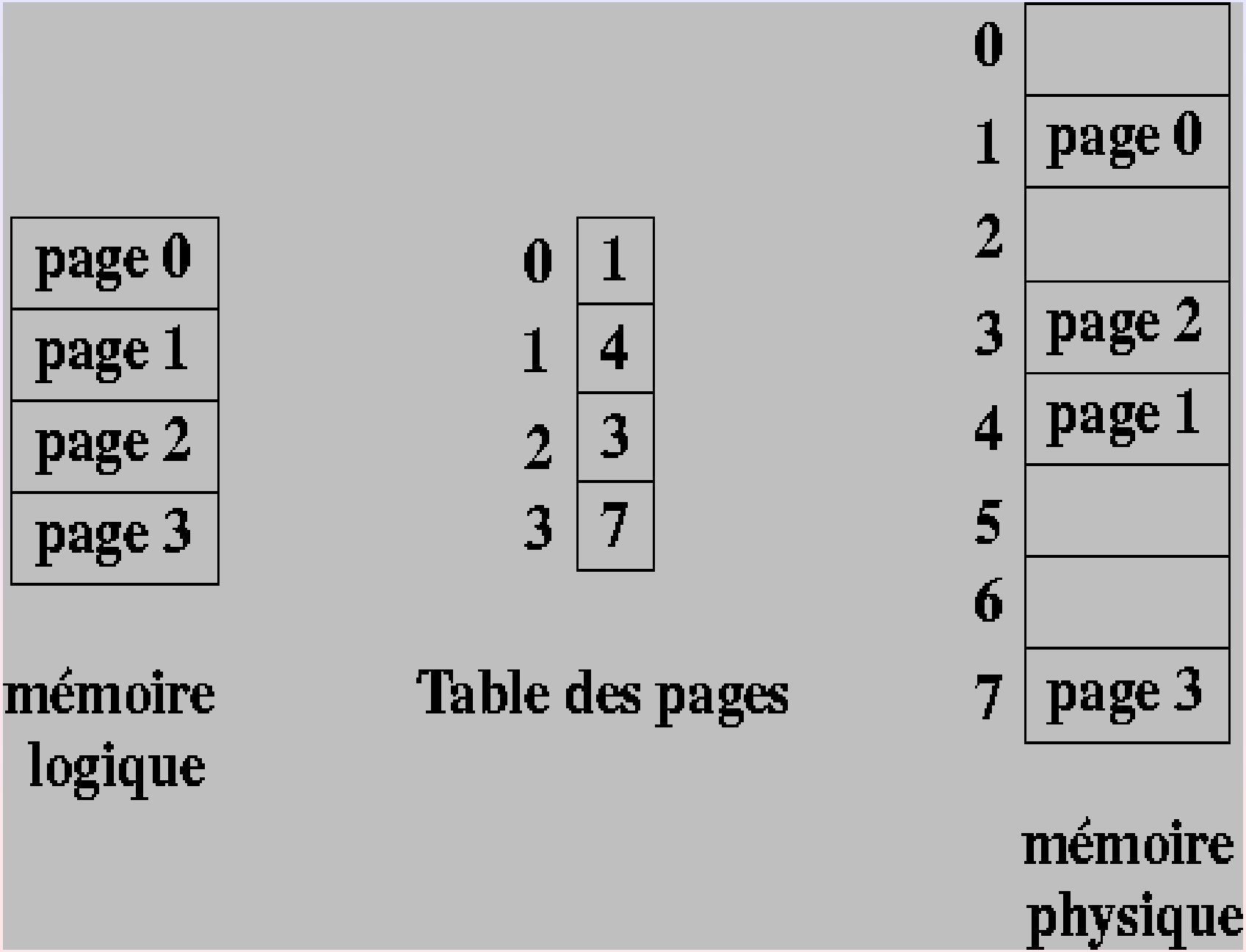
alors

$$\text{no de page} = A/T, \qquad \text{position} = A\%T$$

Traduction des adresses logiques en adresses physiques



Exemple


page 0
page 1
page 2
page 3
mémoire logique
0 1
1 4
2 3
3 7
Table des pages
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
mémoire physique

La mémoire segmentée

But :

- Partage des ressources (le code) entre plusieurs programmes.
- Organisation de la mémoire en unités logiques :
  - code (TEXT),
  - données statiques initialisés (DATA),
  - données statiques non initialisés (BSS),
  - données dynamiques,
  - pile d'exécution.

Pour ne pas avoir des problèmes de fragmentation il faut avoir la mémoire paginée.

La mémoire segmentée

But :

- Partage des ressources (le code) entre plusieurs programmes.
- Organisation de la mémoire en unités logiques :
  - code (TEXT),
  - données statiques initialisés (DATA),
  - données statiques non initialisés (BSS),
  - données dynamiques,
  - pile d'exécution.

Pour ne pas avoir des problèmes de fragmentation il faut avoir la mémoire paginée.

La mémoire segmentée

But :

- Partage des ressources (le code) entre plusieurs programmes.
- Organisation de la mémoire en unités logiques :
  - code (TEXT),
  - données statiques initialisés (DATA),
  - données statiques non initialisés (BSS),
  - données dynamiques,
  - pile d'exécution.

Pour ne pas avoir des problèmes de fragmentation il faut avoir la mémoire paginée.

Implémentation

GDT : Global descriptor table. Description des adresses et attributs des tous les segments.
Appartient aux noyau.

LDT : Local descriptor table. Description des adresses et attributs des segments appartenant à un processus donné.
Appartient au processus.

`gdtr`, `ldtr` : Registres pour repérer la GDT et la LDT courante.

Implémentation

GDT : Global descriptor table. Description des adresses et attributs des tous les segments.
Appartient aux noyau.

LDT : Local descriptor table. Description des adresses et attributs des segments appartenant à un processus donné.
Appartient au processus.

`gdtr`,
`ldtr` : Registres pour repérer la GDT et la LDT courante.

Implémentation

GDT : Global descriptor table. Description des adresses et attributs des tous les segments.
Appartient aux noyau.

LDT : Local descriptor table. Description des adresses et attributs des segments appartenant à un processus donné.
Appartient au processus.

`gdtr`,

`ldtr` : Registres pour repérer la GDT et la LDT courante.

Implémentation

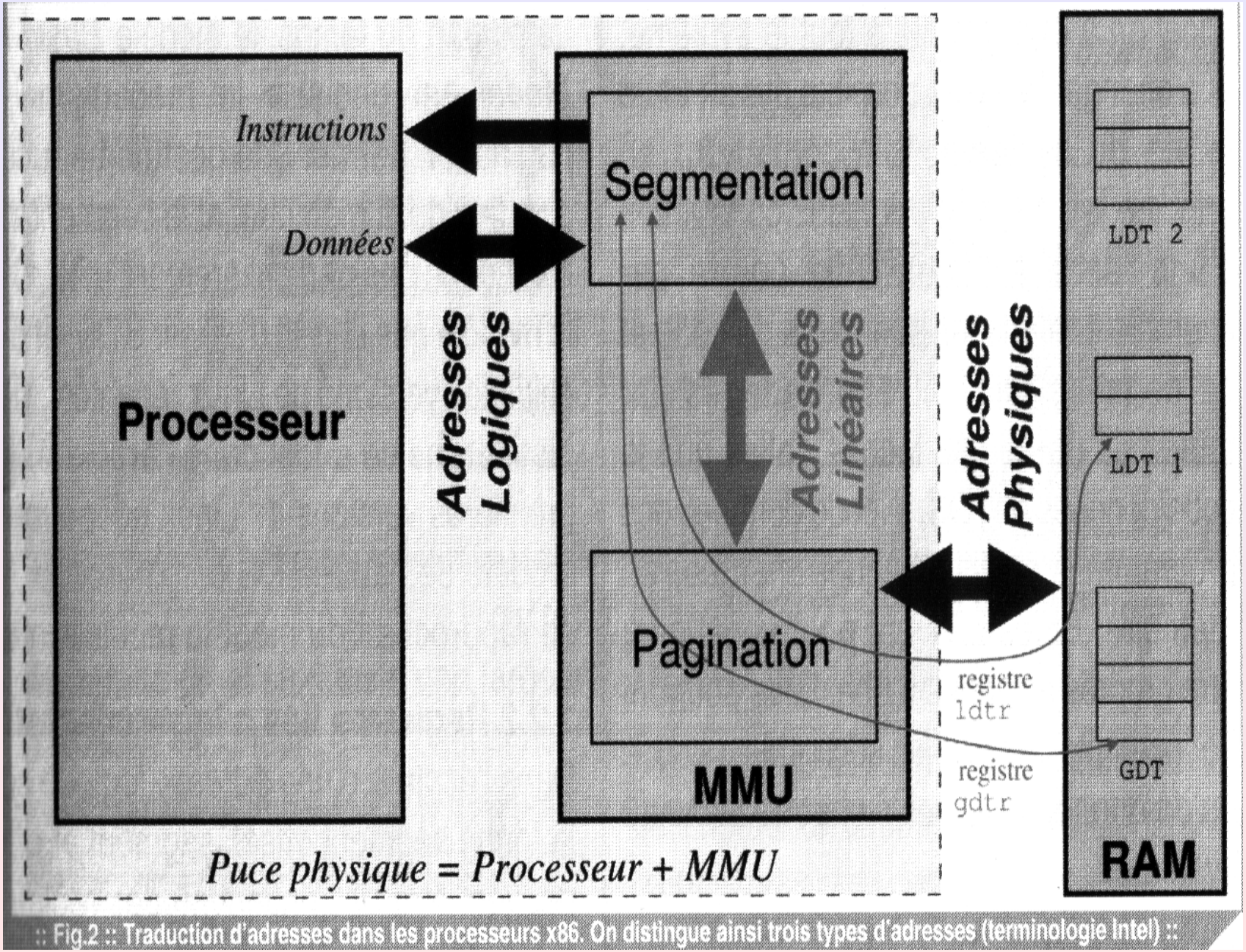
GDT : Global descriptor table. Description des adresses et attributs des tous les segments.
Appartient aux noyau.

LDT : Local descriptor table. Description des adresses et attributs des segments appartenant à un processus donné.
Appartient au processus.

`gdtr`,

`ldtr` : Registres pour repérer la GDT et la LDT courante.

Exemple gestion mémoire sur un x86 (Linux magasine)



:: Fig.2 :: Traduction d'adresses dans les processeurs x86. On distingue ainsi trois types d'adresses (terminologie Intel) ::

Plan

Exécuter un fichier exécutable

- But : lancer (exécuter) un fichier exécutable.
- Remplacement en mémoire du processus par un nouveaux exécutable.
- On ne revient pas de cet exécutable.

Algorithme exec (Bach 1989)

```
algorithme exec
entrée: (1) nom d'un fichier
        (2) liste de paramètres
        (3) liste des variables d'environnement
sortie: néant
{
    accéder à l'i-noeud du fichier (algorithme iput);
    vérifier que le fichier est exécutable et que l'utilisateur a la
                                                permission de l'exécuter;
    lire l'en-tête du fichier, vérifier que c'est un module chargeable;
    copier les paramètres de l'exec de l'ancien espace
                                            d'adressage dans l'espace système;
    for (chaque région attachée au processus)
        détacher toutes les anciennes régions (algorithme detachreg);
    for (chaque région spécifiée dans le module chargeable)
    {
        s'allouer de nouvelles régions (algorithme allocreg);
        attacher les régions (algorithme attachreg);
        charger la région en mémoire si cela est approprié
                                                (algorithme loadreg);
    }
    copier les paramètres de l'exec dans la nouvelle région
                                                de pile de l'utilisateur;
    traitement spécial pour les programmes "setuid" ou mis au point;
    initialiser la zone de sauvegarde des registres  utilisateur de
                        telle façon à pouvoir retourner en mode utilisateur;
    libérer l'i-noeud du fichier (algorithme iput);
}
```

main

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

argc : nombre d'arguments

argv[] : tableau de chaînes de caractères contenant la liste des paramètres – argv[0] la commande donné. Le tableau est terminé par NULL

arge[] : tableau de chaînes de caractères permettant l'acces à l'environnement

Remarques : *arge[]* style ancien C, on peut utiliser getenv.

main

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

argc : nombre d'arguments

argv[] : tableau de chaînes de caractères contenant la liste des paramètres – argv[0] la commande donné. Le tableau est terminé par NULL

arge[] : tableau de chaînes de caractères permettant l'acces à l'environnement

Remarques : *arge[]* style ancien C, on peut utiliser getenv.

main

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

argc : nombre d'arguments

argv[] : tableau de chaînes de caractères contenant la liste des paramètres – `argv[0]` la commande donné. Le tableau est terminé par `NULL`

arge[] : tableau de chaînes de caractères permettant l'acces à l'environnement

Remarques : *arge[]* style ancien C, on peut utiliser `getenv`.

main

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

argc : nombre d'arguments

argv[] : tableau de chaînes de caractères contenant la liste des paramètres – `argv[0]` la commande donné. Le tableau est terminé par `NULL`

arge[] : tableau de chaînes de caractères permettant l'acces à l'environnement

Remarques : *arge[]* style ancien C, on peut utiliser `getenv`.

main

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

argc : nombre d'arguments

argv[] : tableau de chaînes de caractères contenant la liste des paramètres – `argv[0]` la commande donné. Le tableau est terminé par `NULL`

arge[] : tableau de chaînes de caractères permettant l'acces à l'environnement

Remarques : *arge[]* style ancien C, on peut utiliser `getenv`.

getenv

```
#include <stdlib.h>
...
const char *name = "HOME";
char *value;


value = getenv(name);
```

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - l : liste explicite de paramètres
  - v : tableau de paramètres
- passer les arguments en argv
  - l : liste explicite de paramètres
  - v : tableau de paramètres
- utiliser
  - e : passer un nouveau environnement
  - p : l'environnement courant

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'environnement courant
  - e : passer un nouveau environnement

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'environnement courant
  - e : passer un nouveau environnement

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'environnement courant
  - e : passer un nouveau environnement

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'environnement courant
  - e : passer un nouveau environnement

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'environnement courant,
  - e : passer un nouveau environnement.

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'enivrommenemnr courant,
  - e : passer un nouveau environnement.

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - `p` : aussi dans le `PATH`
- passer les arguments en tant que
  - tableau de paramètres
  - `l` : liste explicite de paramètres
- utiliser
  - l'enivrommenemnr courant,
  - `e` : passer un nouveau environnement.

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - p : aussi dans le PATH
- passer les arguments en tant que
  - tableau de paramètres
  - l : liste explicite de paramètres
- utiliser
  - l'enivrommenemnr courant,
  - e : passer un nouveau environnement.

Les cousins de la famille

Quand on exécute un main on peut :

- chercher l'exécutable
  - relativement au répertoire de travail
  - `p` : aussi dans le `PATH`
- passer les arguments en tant que
  - tableau de paramètres
  - `l` : liste explicite de paramètres
- utiliser
  - l'enivrommenemnr courant,
  - `e` : passer un nouveau environnement.

exec avec listes

```
#include <unistd.h>
```

int execl (const char * *ref*, const char * *arg[0]*, ..., *NULL*);

- *ref* : le chemin à l'executable
- *arg[0]* : le nom qu'on veut donner au processus
- ... : les arg[i], pour $i \geq 1$
- *NULL* : terminateur

int execlp (const char * *ref*, const char * *arg*, ... , NULL);

int execle (const char * *ref*, const char * *arg*, ... , NULL , const char ** *arge*);

exec avec listes

```
#include <unistd.h>
```

int execl (const char * *ref*, const char * *arg[0]*, ..., *NULL*);

ref : le chemin à l'executable

arg[0] : le nom qu'on veut donner au processus

... : les arg[i], pour $i \geq 1$

NULL : terminateur

int execlp (const char * *ref*, const char * *arg*, ... , NULL);

int execle (const char * *ref*, const char * *arg*, ... , NULL , const char ** *arge*);

exec avec listes

```
#include <unistd.h>
```

int execl (const char * *ref*, const char * *arg[0]*, ..., *NULL*);

ref : le chemin à l'executable

arg[0] : le nom qu'on veut donner au processus

... : les arg[i], pour $i \geq 1$

NULL : terminateur

int execlp (const char * *ref*, const char * *arg*, ... , NULL);

int execle (const char * *ref*, const char * *arg*, ... , NULL , const char ** *arge*);

exec avec listes

```
#include <unistd.h>
```

int execl (const char * *ref*, const char * *arg[0]*, ..., *NULL*);

ref : le chemin à l'executable

arg[0] : le nom qu'on veut donner au processus

... : les arg[i], pour $i \geq 1$

NULL : terminateur

int execlp (const char * *ref*, const char * *arg*, ... , NULL);

int execle (const char * *ref*, const char * *arg*, ... , NULL , const char ** *arge*);

exec avec tableaux

```
#include <unistd.h>

int execv ( const char * ref, const char * argv[] );

int execvp ( const char * ref, const char * argv[] );

int execve ( const char * ref, const char * argv[], const
char ** arge );
```

exec avec tableaux

```
#include <unistd.h>

int execv ( const char * ref, const char * argv[] );

int execvp ( const char * ref, const char * argv[] );

int execve ( const char * ref, const char * argv[], const
char ** arge );
```

exec avec tableaux

```
#include <unistd.h>

int execv ( const char * ref, const char * argv[] );

int execvp ( const char * ref, const char * argv[] );

int execve ( const char * ref, const char * argv[], const
char ** arge );
```

exec avec tableaux

```
#include <unistd.h>

int execv ( const char * ref, const char * argv[] );

int execvp ( const char * ref, const char * argv[] );

int execve ( const char * ref, const char * argv[], const
char ** arge );
```

system

```
#include <stdlib.h>
int system(const char * comm);
```

comm : un commande (contenant pararemètres et options) en forme en forme de chaîne de caractères.

Retourne : La valeur de retour du shell.

Sommaire : La commande *commande* est passée au shell.

Remarques : Fonction de la bibliothèque standard C.

system

```
#include <stdlib.h>
int system(const char * comm);
```

comm : un commande (contenant pararemètres et options) en forme en forme de chaîne de caractères.

Retourne : La valeur de retour du shell.

Sommaire : La commande *commande* est passée au shell.

Remarques : Fonction de la bibliothèque standard C.

system

```
#include <stdlib.h>
int system(const char * comm);
```

comm : un commande (contenant pararemètres et options) en forme en forme de chaîne de caractères.

Retourne : La valeur de retour du shell.

Sommaire : La commande *commande* est passée au shell.

Remarques : Fonction de la bibliothèque standard C.

system

```
#include <stdlib.h>
int system(const char * comm);
```

comm : un commande (contenant pararemètres et options) en forme en forme de chaîne de caractères.

Retourne : La valeur de retour du shell.

Sommaire : La commande *commande* est passée au shell.

Remarques : Fonction de la bibliothèque standard C.

Plan

dup

```
#include <unistd.h>
int dup(int desc);
```

desc : le descripteur qu'on veut dupliquer

Retourne : un nouveau descripteur (-1 si erreur)

Sommaire : duplication d'un descripteur

Remarques : le nouveau descripteur est le premier disponible

dup

```
#include <unistd.h>
int dup(int desc);
```

desc : le descripteur qu'on veut dupliquer

Retourne : un nouveau descripteur (-1 si erreur)

Sommaire : duplication d'un descripteur

Remarques : le nouveau descripteur est le premier disponible

dup

```
#include <unistd.h>
int dup(int desc);
```

desc : le descripteur qu'on veut dupliquer

Retourne : un nouveau descripteur (-1 si erreur)

Sommaire : duplication d'un descripteur

Remarques : le nouveau descripteur est le premier disponible

dup

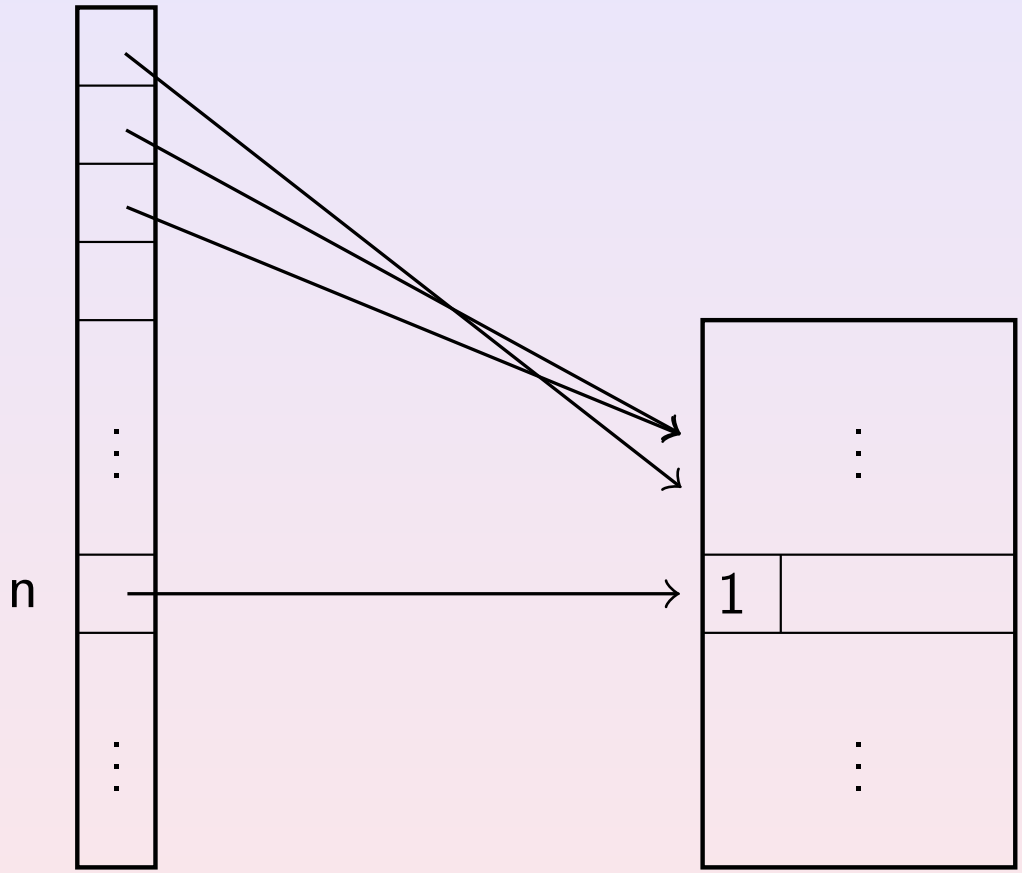
```
#include <unistd.h>
int dup(int desc);
```

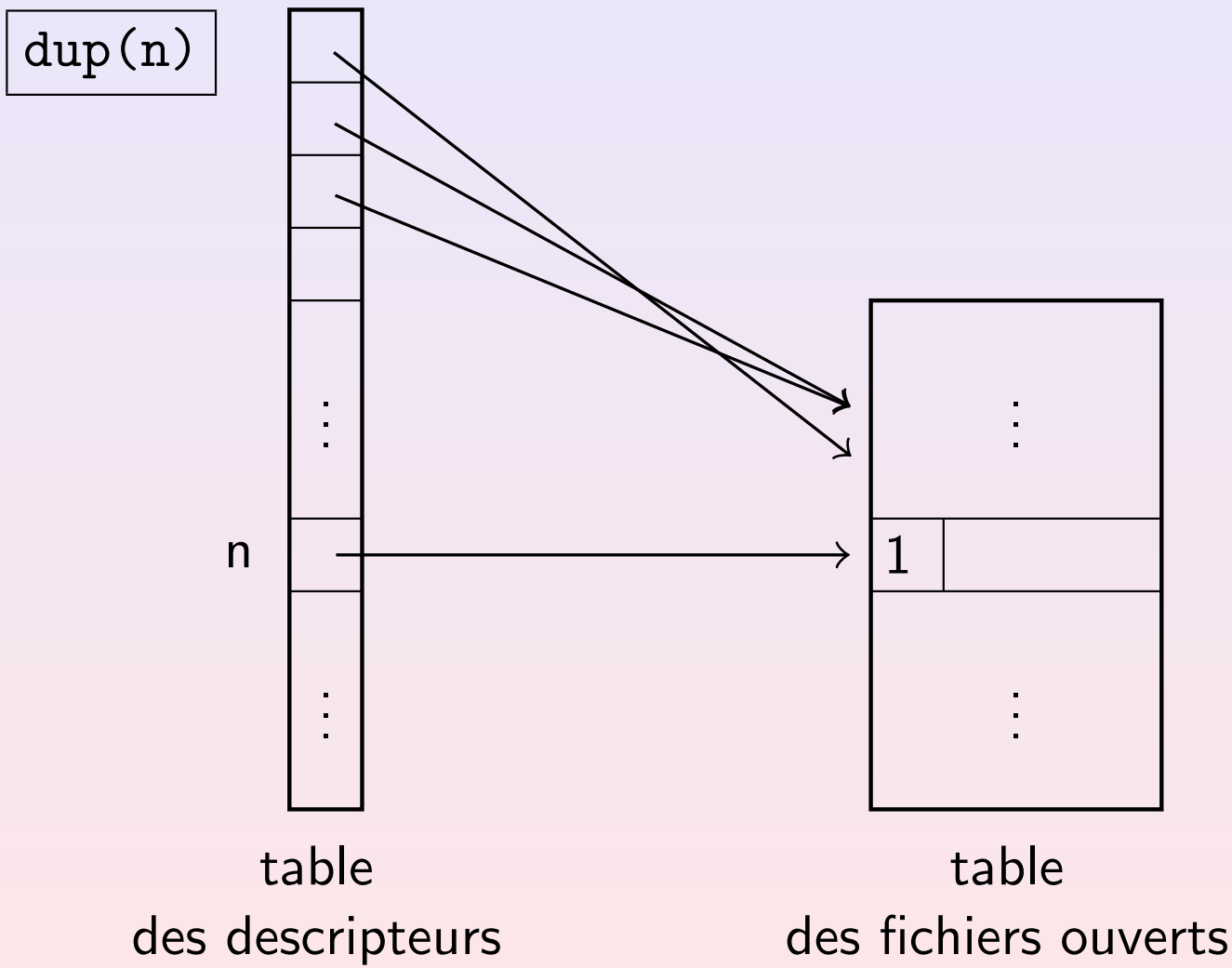
desc : le descripteur qu'on veut dupliquer

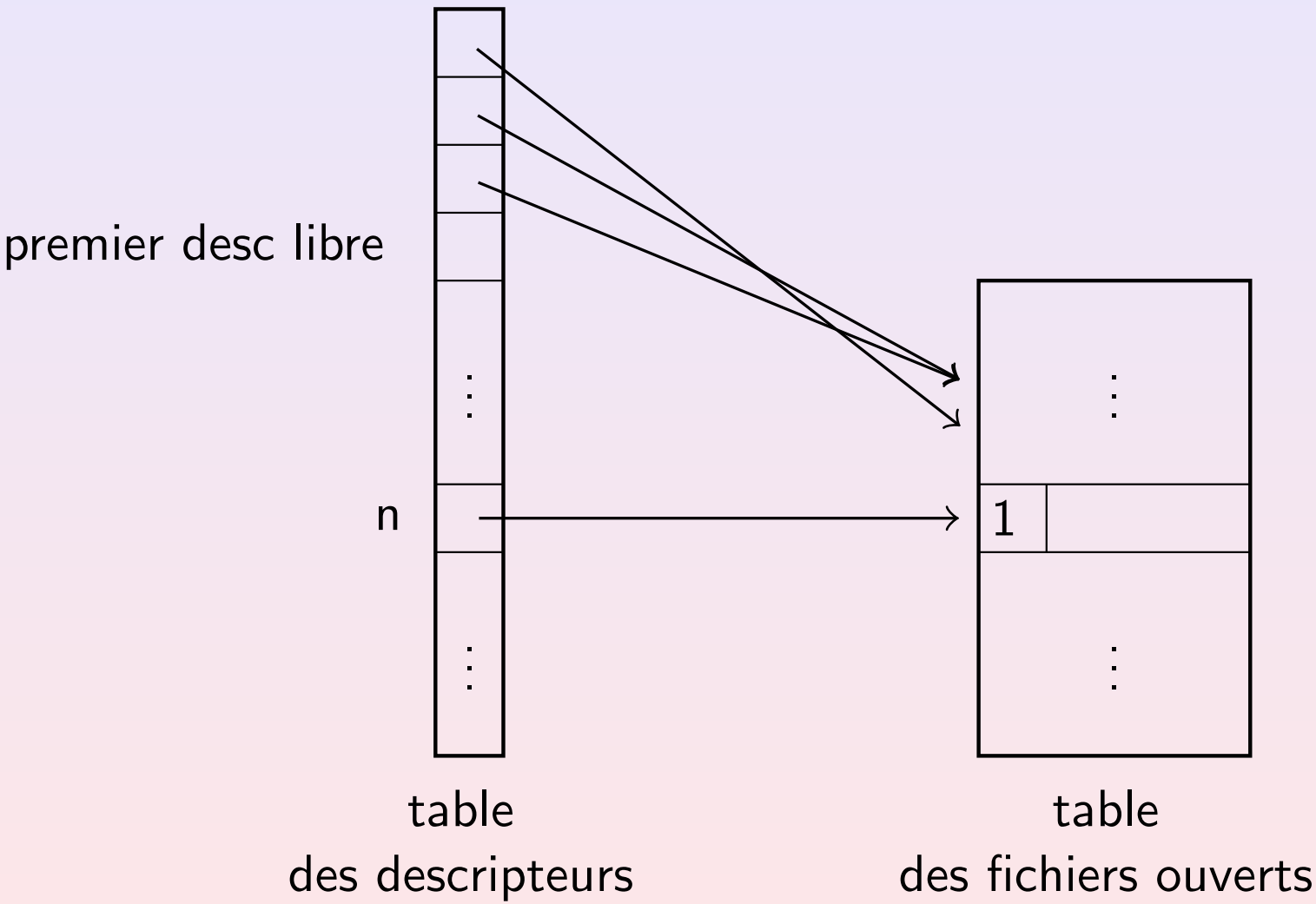
Retourne : un nouveau descripteur (-1 si erreur)

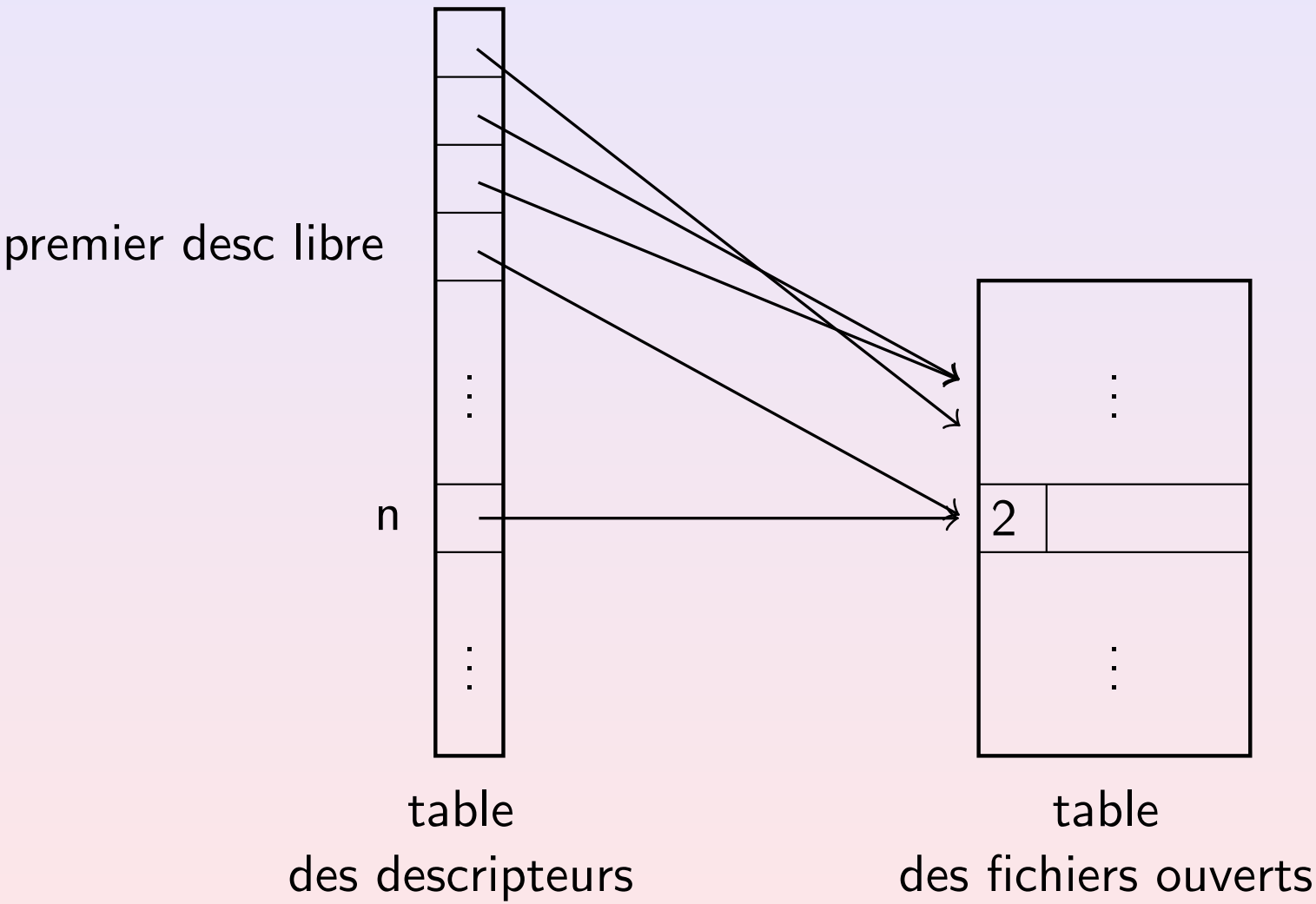
Sommaire : duplication d'un descripteur

Remarques : le nouveau descripteur est le premier disponible


n
1
table
des descripteurs
table
des fichiers ouverts

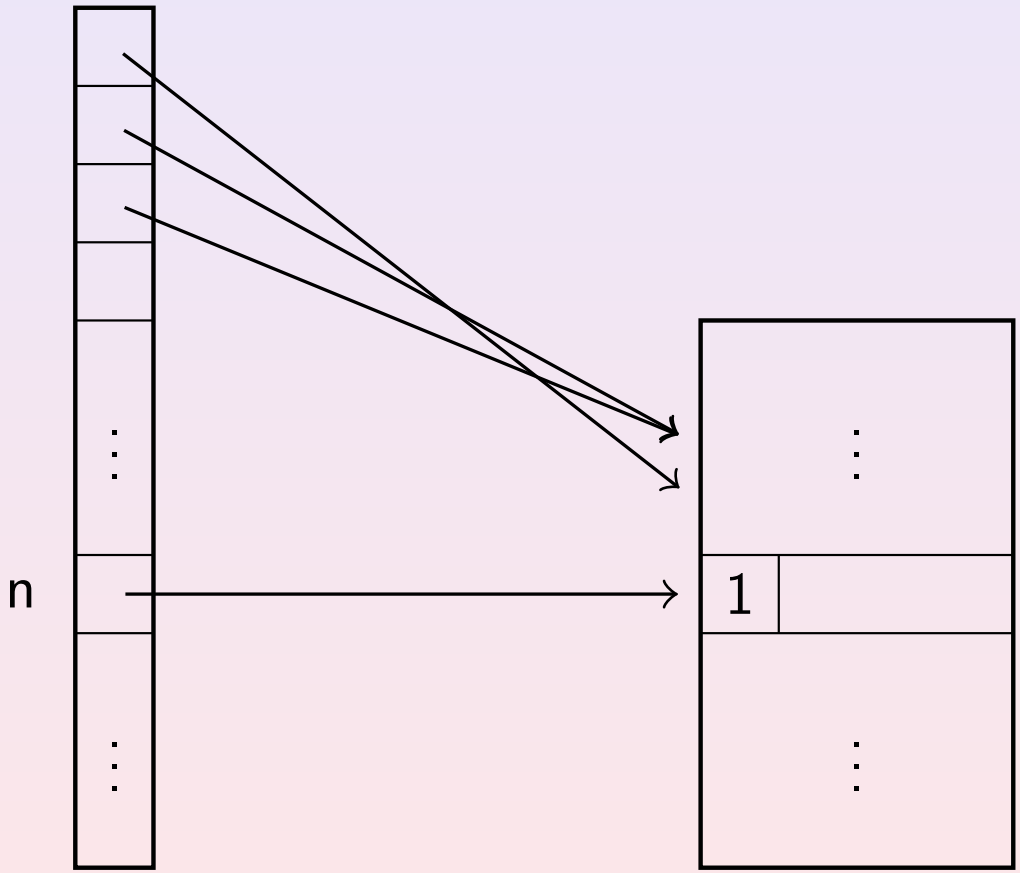

dup(n)
n
1
table
des descripteurs
table
des fichiers ouverts


premier desc libre
n
1
table
des descripteurs
table
des fichiers ouverts

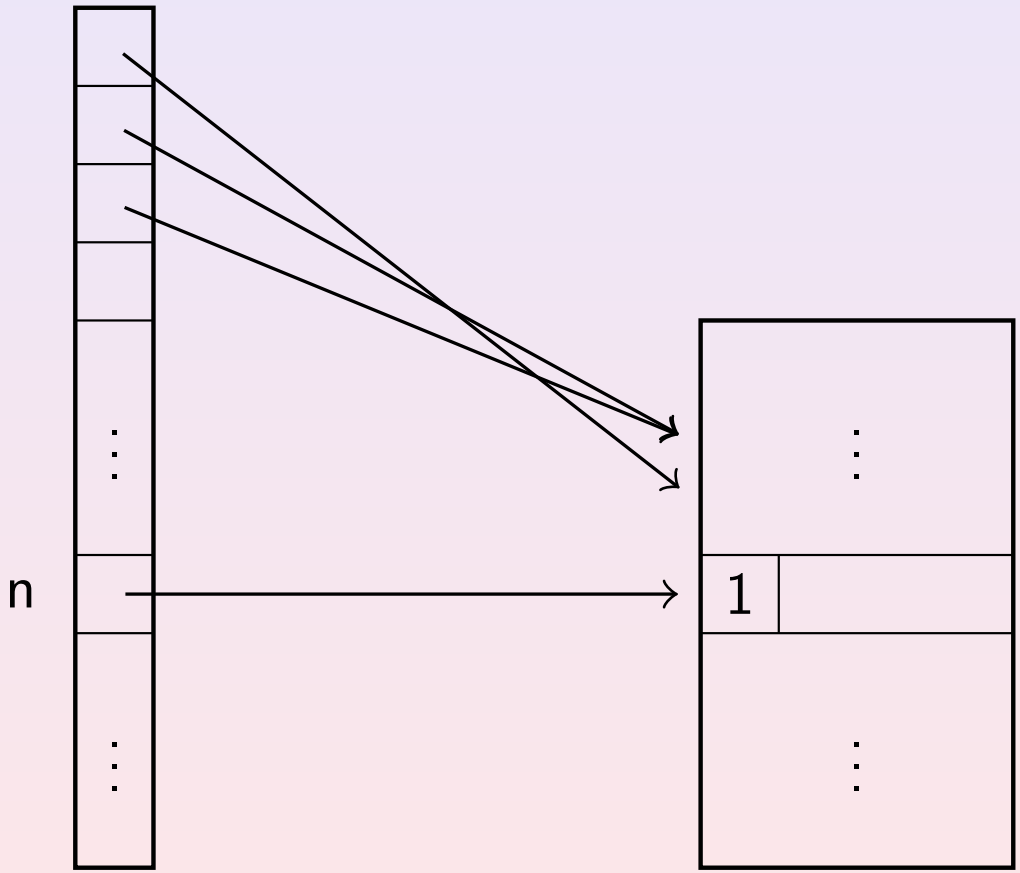

premier desc libre
n
2
table
des descripteurs
table
des fichiers ouverts

Redirection entrée

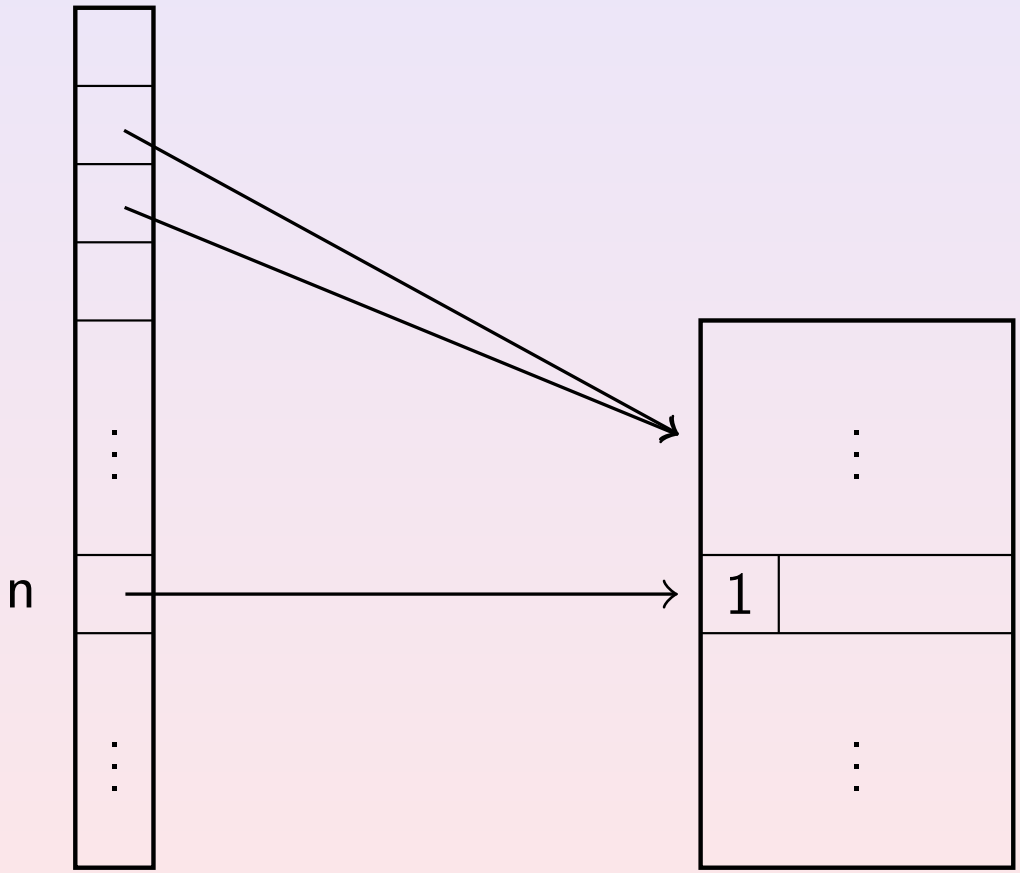
```
{
    ...
    close(STDIN_FILENO);
    dup(n);
    close(n);
    ...
}
```


n
1
table
des descripteurs
table
des fichiers ouverts

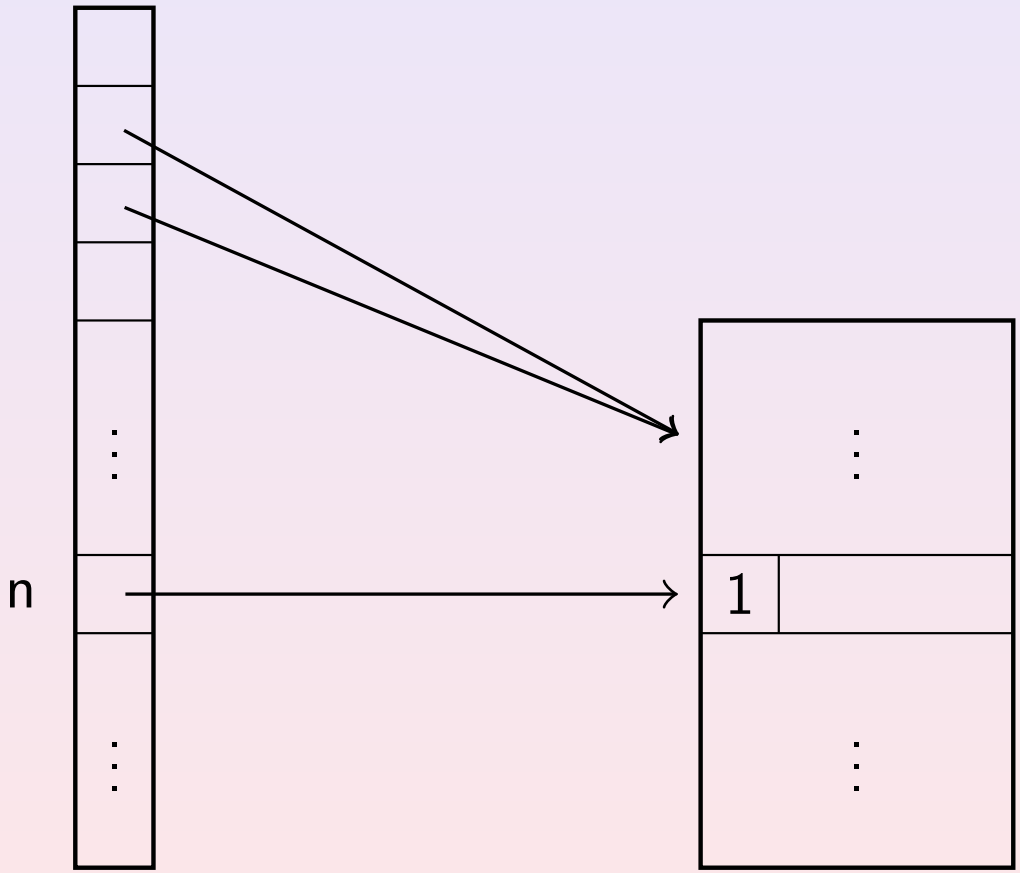
```
close(SDTIN_FILENO);
```



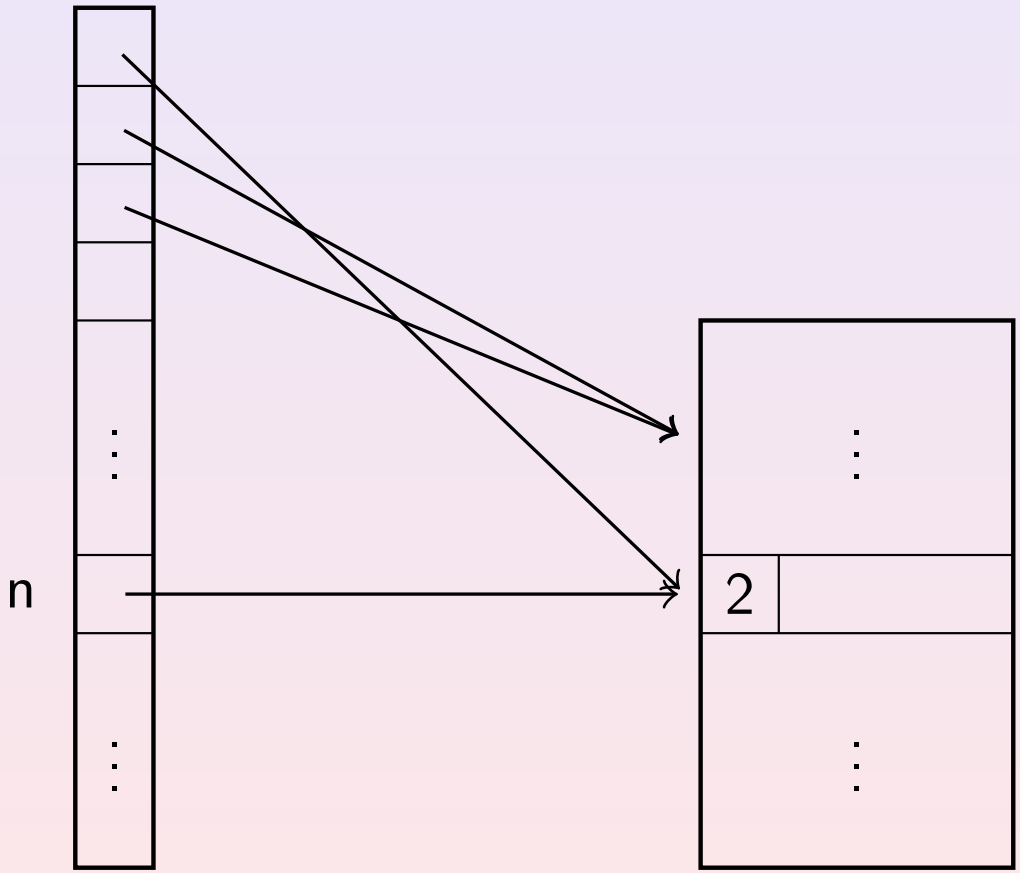
```
close(SDTIN_FILENO);
```



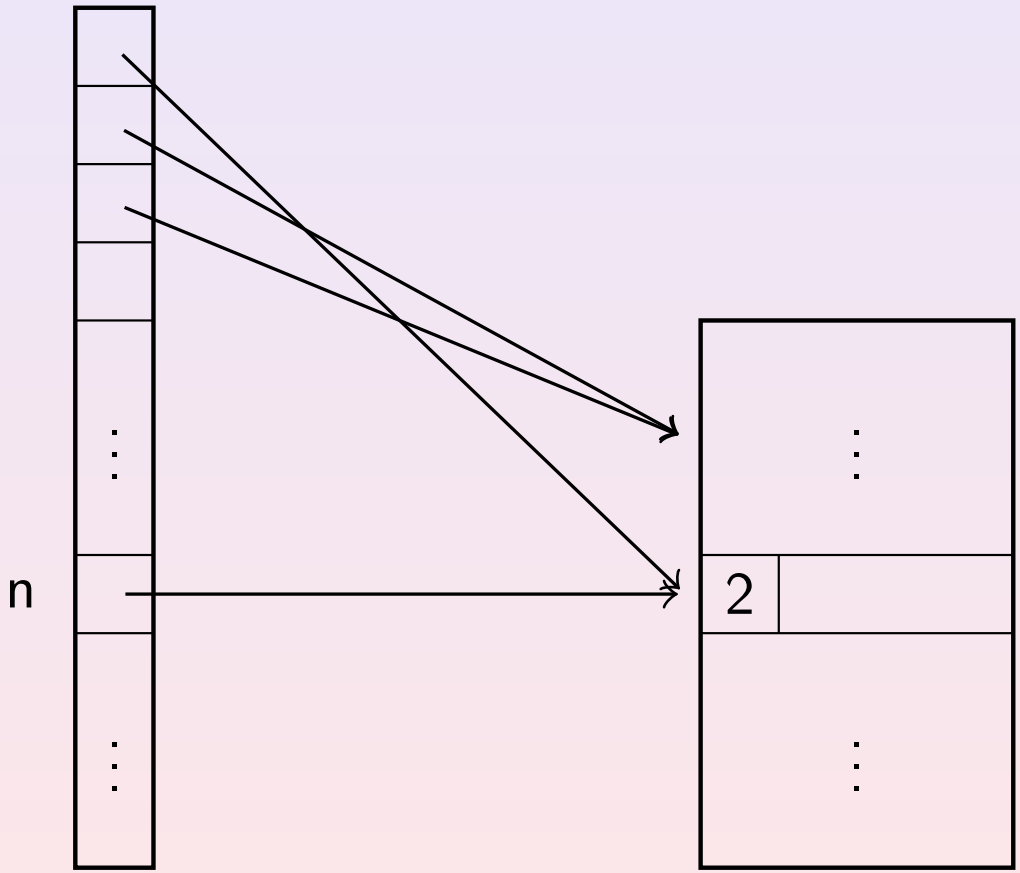
```
dup(n);
```



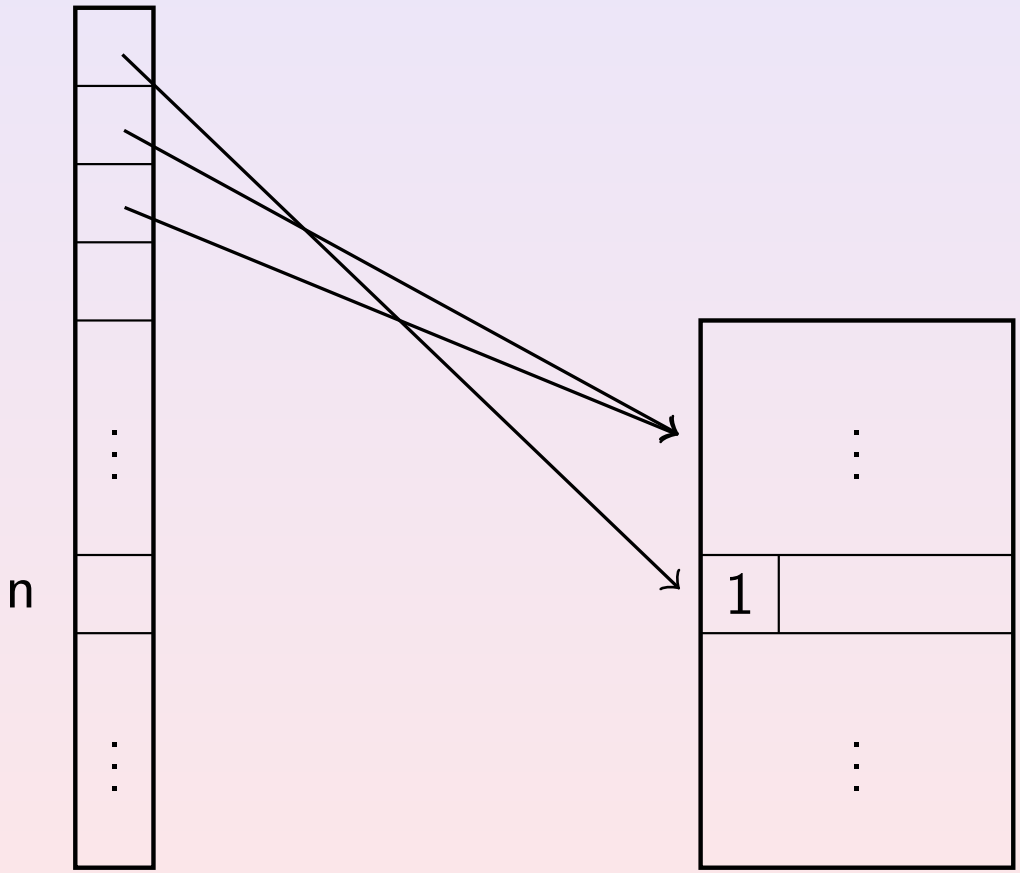
```
dup(n);
```



```
close(n);
```



```
close(n);
```



dup2

```
#include <unistd.h>
int dup2(int desc1, int desc2);
```

desc1 : le descripteur à dupliquer
desc2 : le descripteur qu'on aimerait utiliser. S'il est déjà utilisé, le système opère un fermeture préliminaire : `close(desc2)`

Retourne : le descripteur dupliqué (-1 si erreur)

dup2

```
#include <unistd.h>
int dup2(int desc1, int desc2);
```

desc1 : le descripteur à dupliquer

desc2 : le descripteur qu'on aimerait utiliser. S'il est déjà utilisé, le système opère un fermeture préliminaire : `close(desc2)`

Retourne : le descripteur dupliqué (-1 si erreur)

dup2

```
#include <unistd.h>
int dup2(int desc1, int desc2);
```

desc1 : le descripteur à dupliquer

desc2 : le descripteur qu'on aimerait utiliser. S'il est déjà utilisé, le système opère un fermeture préliminaire : `close(desc2)`

Retourne : le descripteur dupliqué (-1 si erreur)

dup2

```
#include <unistd.h>
int dup2(int desc1, int desc2);
```

desc1 : le descripteur à dupliquer
desc2 : le descripteur qu'on aimerait utiliser. S'il est déjà utilisé, le système opère un fermeture préliminaire : `close(desc2)`

Retourne : le descripteur dupliqué (-1 si erreur)

Programme : exempledup

```
 1 : #include <unistd.h>
 2 : #include <stdio.h>
 3 : #include <stdlib.h>
 4 : #include <ctype.h>
 5 :
 6 : void fils()
 7 : {
 8 :     printf("%s", "abcde\n");
 9 :     exit(EXIT_SUCCESS);
10 : }
11 :
12 : void pere()
13 : {
14 :     char tampon[100];
15 :     fgets(tampon, sizeof(tampon), stdin);
16 :     printf("%s", tampon);
17 :     exit(EXIT_SUCCESS);
18 : }
19 :
```

Programme : exempledup

```
20 : int main(void)
21 : {
22 :     int tube[2], desc;
23 :     if (pipe(tube) == -1)
24 :         exit(EXIT_FAILURE);
25 :
26 :     switch (fork())
27 :     {
28 :     case -1:
29 :         exit(EXIT_FAILURE);
30 :     case 0:
31 :         close(tube[0]);
32 :         /* Redirection stdout vers tube[1] */
33 :         close(STDOUT_FILENO);
34 :         desc = dup(tube[1]);
35 :         fprintf(stderr, "[Fils] Nouveau descripteur : %d\n", desc);
36 :         close(tube[1]);
37 :         /* Fin redirection */
38 :         fils();
39 :     default:
40 :         close(tube[1]);
41 :         /* Redirection stdin vers tube[1] */
42 :         close(STDIN_FILENO);
43 :         desc = dup(tube[0]);
44 :         fprintf(stderr, "[Père] Nouveau descripteur : %d\n", desc);
45 :         close(tube[0]);
46 :         /* Fin redirection */
47 :         pere();
48 :     }
49 :     exit(EXIT_FAILURE);
50 : }
```

Programme : exempledup2

```
20 : int main(void)
21 : {
22 :     int tube[2], desc;
23 :     if (pipe(tube) == -1)
24 :         exit(EXIT_FAILURE);
25 :
26 :     switch (fork())
27 :     {
28 :     case -1:
29 :         exit(EXIT_FAILURE);
30 :     case 0:
31 :         close(tube[0]);
32 :         /* Redirection stdout vers tube[1] */
33 :         desc = dup2(tube[1], STDOUT_FILENO);
34 :         fprintf(stderr, "[Fils] Nouveau descripteur : %d\n", desc);
35 :         close(tube[1]);
36 :         /* Fin redirection */
37 :         fils();
38 :     default:
39 :         close(tube[1]);
40 :         /* Redirection stdin vers tube[1] */
41 :         desc = dup2(tube[0], STDIN_FILENO);
42 :         fprintf(stderr, "[Père] Nouveau descripteur : %d\n", desc);
43 :         close(tube[0]);
44 :         /* Fin redirection */
45 :         pere();
46 :     }
47 :     exit(EXIT_FAILURE);
48 : }
```

Programme : exemple implementation de « ls | wc -l »

```
 1 : #include <unistd.h>
 2 : #include <stdlib.h>
 3 :
 4 : int main(void)
 5 : {
 6 :     int tube[2];
 7 :     if (pipe(tube) == -1)
 8 :         exit(-1);
 9 :
10 :     switch (fork())
11 :     {
12 :     case -1:
13 :         exit(-1);
14 :     case 0:
15 :         close(tube[0]);
16 :         dup2(tube[1], STDOUT_FILENO);
17 :         close(tube[1]);
18 :         execlp("ls", "Faire la liste", NULL);
19 :         break;
20 :     default:
21 :         close(tube[1]);
22 :         dup2(tube[0], STDIN_FILENO);
23 :         close(tube[0]);
24 :         execlp("wc", "Compter lignes", "-l", NULL);
25 :     }
26 :     exit(-1);
27 : }
```