Cours de Réseau et communication Unix n°3

Benjamin MONMEGE

Faculté des Sciences

Aix-Marseille Université (AMU)

2016-2017

Les transparents de ce cours sont téléchargeables ici :
https://ametice.univ-amu.fr/course/view.php?id=24506

Transparents en partie empruntés à Edouard Thiel

Plan du cours précédent

1. Les entrées-sorties
2. Les i-nodes, ou nœuds d'index
3. Les tables du système au niveau 1
4. Ouverture et fermeture au niveau 1
5. Lecture et écriture de fichiers
6. Les tubes

Plan du cours n°3

1. Duplication de descripteurs
2. Scrutation de plusieurs descripteurs
3. Exemple d'utilisation de `select`
4. Alternatives à `select`

1 - Duplication de descripteurs

Acquisition d'un descripteur par un processus

Duplication avec dup

La duplication de descriteur permet à un processus d'acquérir un nouveau descripteur dans sa TD, synonyme d'un ancien descripteur qu'il possède déjà :

- ancien descripteur d'un fichier déjà ouvert
- n'ouvre pas de nouveau fichier

Primitive `dup` :

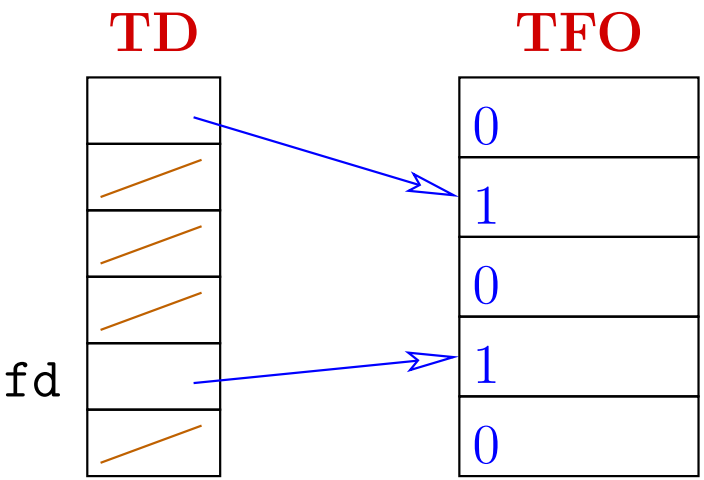
```
#include <unistd.h>
int dup (int oldfd);
```

Renvoie le nouveau descripteur $\geqslant 0$, sinon -1 erreur

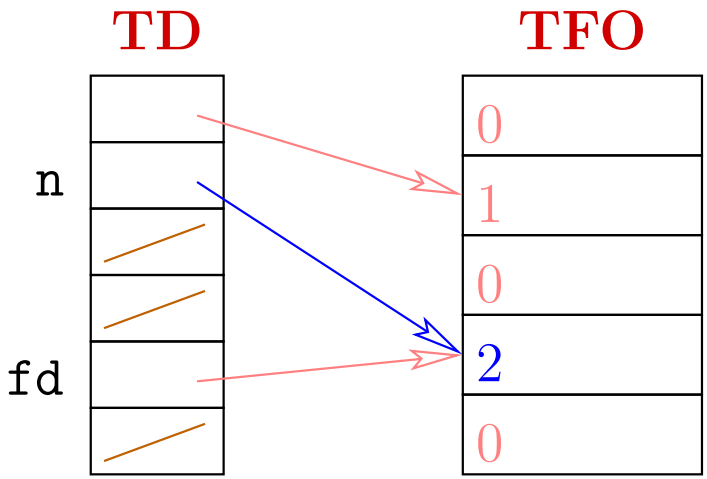
Important : le nouveau descripteur est l'indice de la première case disponible dans TD.

Effet de dup

Avant :



`n = dup(fd);`



On peut ensuite indifféremment utiliser `n` ou `fd` pour manipuler le fichier.

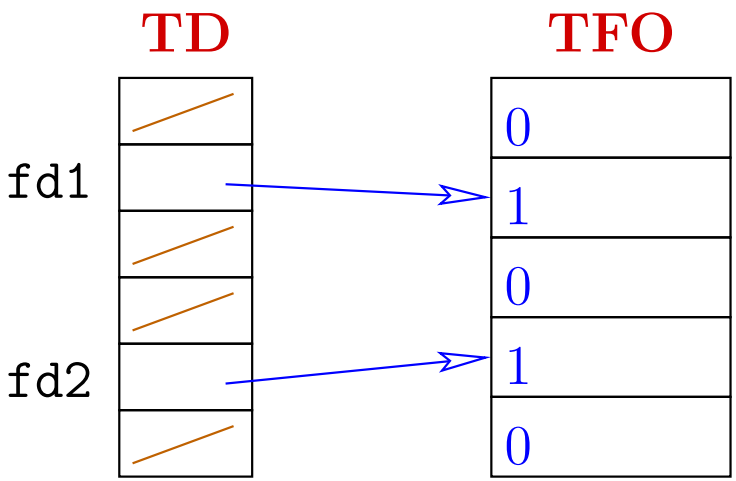
Duplication avec dup2

```
#include <unistd.h>
int dup2 (int oldfd, int newfd);
```

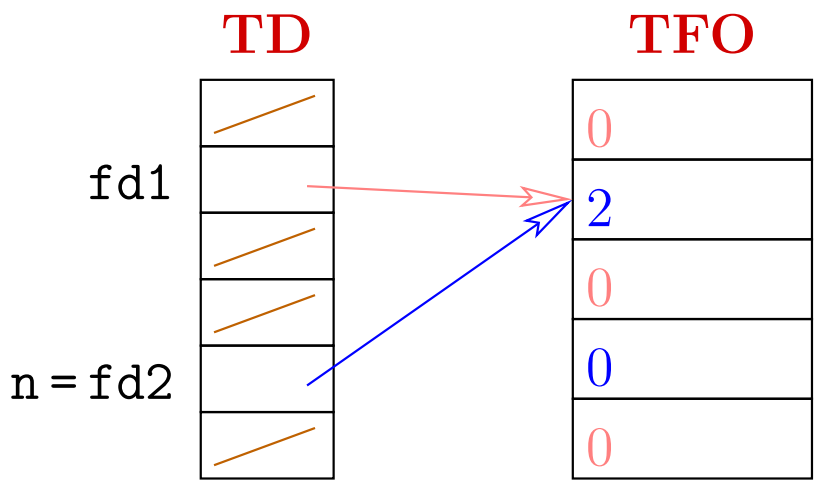
- Duplique le descripteur `oldfd` dans la case `newfd`
- Ferme au préalable `newfd` si déjà ouvert

Renvoie `newfd` $\geqslant 0$ ou -1 erreur

Avant :



`n = dup2(fd1,fd2);`



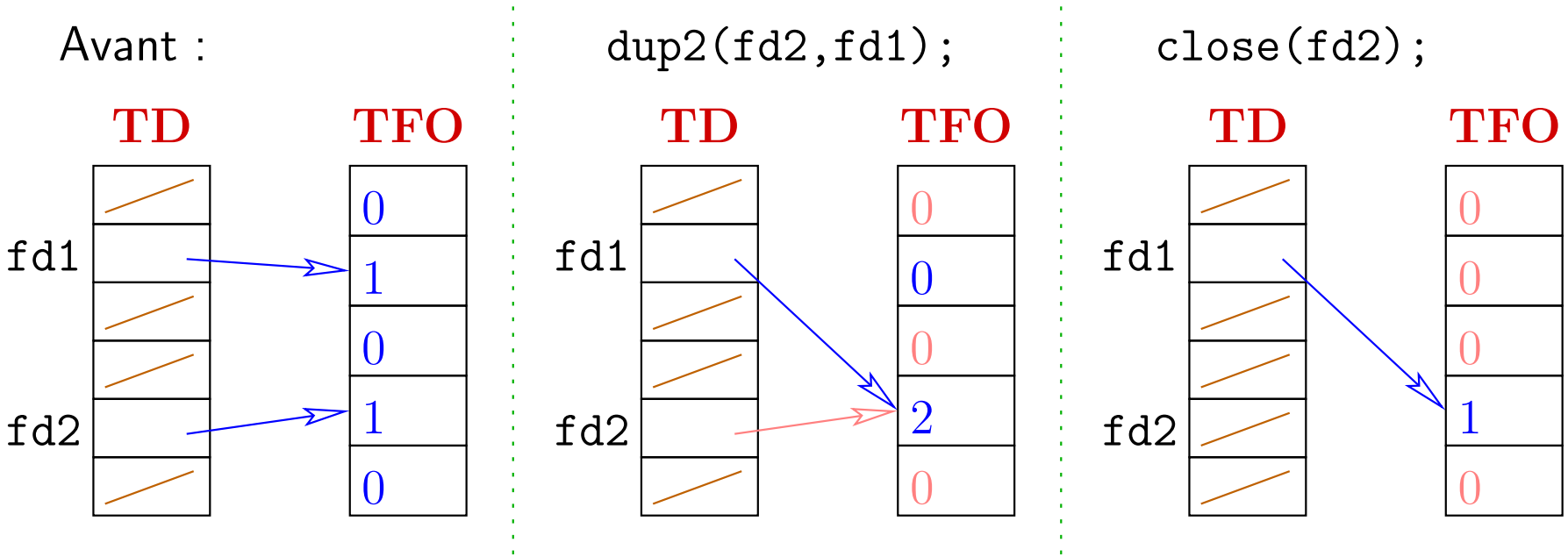
Équivalence entre dup et dup2

Si tous les descripteurs $<$ `fd2` sont ouverts, alors

```
n = dup2(fd1,fd2);      ⇔      close(fd2); n = dup(fd1);
```

Redirection avec dup2

Soit `fd1` et `fd2` deux descripteurs ouverts.
Pour rediriger `fd1` sur `fd2` :



Intérêt

Héritage de la table des descripteurs lors de `fork` ou conservation lors de `exec`

$\rightarrow$ permet les redirections d'un programme vers :

- fichiers
- terminaux
- tubes anonymes
- tubes nommés
- socket connectées

2 - Scrutation de plusieurs descripteurs

Solution efficace pour dialoguer sur plusieurs descripteurs.

Descripteur éligible

Un descripteur (ouvert) est éligible si une opération bloquante sur ce descripteur peut être effectuée immédiatement.

Exemples :

- un tube non plein est éligible en écriture
- un tube non vide est éligible en lecture

Problème : un programme qui manipule plusieurs descripteurs peut rester (indéfiniment) bloqué sur un descripteur X, alors que

- un autre descripteur Y est éligible ;
- agir sur Y pourrait indirectement débloquer X.

$\rightarrow$ Risque de blockage ou ralentissement

Solution 1

Rendre toutes les opérations non bloquantes :

`open` ou `fcntl`, flags `O_NONBLOCK` ou `O_NDELAY`

Inconvénients :

- impacte toutes les fonctions sur les fichiers
- attente active coûteuse

Solution 2

Utiliser des pthreads

Ce sont des processus légers, qui partagent les données

Inconvénients :

- Mise au point complexe ; si pose de verrous (mutex) nécessaire, risque d'interblockages
- un pthread par descripteur = coût élevé (kernel, 1 stack/pthread, cache cpu)

Bonne utilisation des pthreads : pthreads "métier" (GUI, réseau, BD, etc)

Solution 3

Solution efficace : par scrutation

La fonction `select` est bloquante ; elle scrute (surveille) plusieurs descripteurs, et revient dès que l'un des descripteurs est éligible.

$\rightarrow$ elle garanti que la prochaine opération (lecture, écriture, etc) sur ce descripteur sera immédiate.

Fonction select

```
/* POSIX.1-2001 */
#include <sys/select.h>

/* Standards antérieurs */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

Listes des descripteurs à scruter

```
int select (int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

`readfds` liste de descripteurs en lecture

`writefds` liste de descripteurs en écriture

`exceptfds` liste de descripteurs en "lecture urgente" (très rarement employé)

`nfds` Max { descripteurs des 3 listes } + 1
permet de borner les listes

`fd_set` : type bitmap

Manipulation des listes de descripteurs

```
void FD_ZERO (fd_set *set);
```
vide la liste `set`

```
void FD_SET (int fd, fd_set *set);
```
ajoute `fd` dans la liste `set`

```
int FD_ISSET (int fd, fd_set *set);
```
est vrai si `fd` est dans la liste `set`

```
void FD_CLR (int fd, fd_set *set);
```
supprime `fd` de la liste `set`

Timeout

```
int select (int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

`timeout` est le délai maximum d'attente pour `select`.

Cas particuliers :

- si `timeout` est 0, `select` revient immédiatement
- si `timeout` est NULL, il n'y a plus de limite de temps d'attente

La structure de `timeout` est définie dans `<sys/time.h>` :

```
struct timeval {
   long    tv_sec;         /* secondes */
   long    tv_usec;        /* microsecondes */
};
```

Micro sleep portable

La fonction `sleep` fait une attente en secondes.
Il existe des fonctions avec une résolution plus élevée :

- `usleep` en microsecondes
- `nanosleep` en nanosecondes (POSIX.1)

mais leur portabilité est limitée.

Une solution portable : utiliser `select` avec

- `nfds` = 0
- les 3 listes à NULL
- un timeout non-NULL

Exemple :

```
struct timeval t;
t.tv_sec = 0; t.tv_usec = 200000;
select (0, NULL, NULL, NULL, &t);
```

Au retour de `select`

`select` renvoie :

- > 0 nombre de descripteurs éligibles
- $= 0$ timeout expiré sans aucun descripteur éligible
- $=$ -1 une erreur, décrite par `errno` :
  - `errno = EBADF` descripteur invalide dans liste
  - `errno = EINTR` signal délivré
  - `errno = EINVAL` `nfds` < 0 ou mauvais timeout
  - `errno = ENOMEM` plus assez de mémoire

`errno = EBADF`

Un descripteur fermé est dans la liste

$\rightarrow$ Retour immédiat de `select`

 Il faut retirer ce descripteur de la liste car `select` ne peut plus scruter.

Comment détecter le mauvais descripteur ?
pour chaque descripteur `fd`, appeler `select` avec timeout à 0

errno = EINTR

Un signal a été capté.

Si on place un handler pour un signal X avec `sigaction` et l'option `SA_RESTART`,

lorsque le signal X est délivré, tous les appels bloquants sont silencieusement repris ... sauf `select`

> Le but : être informé dans l'appel de `select` de la survenue d'un signal

Exception : sauf sur certains Unix (SGI) où `select` est silencieusement repris.

(Solution : wake-up pipe ou `pselect`)

Au retour de `select`

Les paramètres de `select` contiennent :

- `nfds` inchangé (par valeur !)
- Les 3 listes ne contiennent plus que les descripteurs éligibles des listes originales.
- Le `timeout` contient le temps restant par rapport à l'échance du timeout initial (selon système).

Modification des listes :

- Pour savoir si un descripteur est éligible, tester sa présence dans la liste avec `FD_ISSET`.
- Si on boucle sur `select`, il faut chaque fois reconstruire les listes.

Descripteurs éligibles

Lorsque un descripteur est éligible,

- `select` garantit que la prochaine opération bloquante sera immédiate ;
- mais il faut faire cette opération :

Si on ne la fait pas, le prochain appel à `select` reviendra immédiatement

$\rightarrow$ plus de scrutation efficace

$\rightarrow$ un descripteur peut "effacer" les autres.

3 - Exemple d'utilisation de `select`

Exemple : programme qui lit une ligne au clavier, ou s'arrête au bout de 4,5 secondes si rien n'a été tapé entre temps.

Avant l'appel

```
fd_set set1;
struct timeval t;
int r;

/* Init set1 */
FD_ZERO (&set1);
FD_SET (0, &set1);

/* Init t */
t.tv_sec = 4;
t.tv_usec = 500000;

r = select (0+1, &set1, NULL, NULL, &t);
```

Après l'appel

```
if (r < 0) {
  if (errno == EINTR)
       printf ("Un signal a été capté\n");
  else perror ("select");

} else if (r == 0) {
  printf ("Délai écoulé\n");

} else {
  if (FD_ISSET (0, &set1)) {  // inutile ici ..
    char s[1000]; int k;
    k = read (0, s, sizeof(s)-1);
    if (k < 0) { perror ("read"); exit (1); }
    s[k] = 0;
    printf ("Lu %s\n", s);
  }
}
```

4 - Alternatives à `select`

En savoir plus : `man select`
`man select_tut`

`select` apparu dans BSD 4.2 (1983) ; normalisé POSIX

Portable sauf certains détails $\rightarrow$ utiliser `pselect`

Équivalent natif de `select` sur Unix SysV : `poll`

Évolution de `select` et `poll` : `epoll`

pselect

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

int pselect(int nfds, fd_set *readfds,
            fd_set *writefds, fd_set *exceptfds,
            const struct timespec *timeout,
            const sigset_t *sigmask);
```

Identique à `select` sauf

- `pselect` a un timeout en nanosecondes
- `pselect` ne modifie pas le timeout
- `select` ne masque pas les signaux
- `pselect` évite une course sur les signaux

Problème de scalabilité

Pour un petit nombre de descripteurs scrutés (< 100), `select` est parfait : efficace, simple, portable.

Une application peut avoir à gérer

- un grand nombre de connexion simultanées ($> 100\,000$)
- avec un nombre faible de connexions actives (< 100).

`select` passe mal à cette échelle :

- limites intrinsèques
- problèmes de design $\rightarrow$ coût élevé en cycles CPU

Limitations de `select`/`pselect`

1. Type `fd_set` : bitmap de taille fixe `FD_SETSIZE` (1024).
2. L'application doit réécrire `fd_set` avant chaque appel.
3. Le noyau doit parcourir entièrement `fd_set` à chaque appel.
4. Si aucun descripteur n'est éligible, le kernel doit mettre en place des handlers internes pour chaque descripteur.
5. L'application doit parcourir entièrement `fd_set` après chaque appel.

poll et ppoll

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

   struct pollfd {
       int   fd;         /* file descriptor */
       short events;     /* requested events */
       short revents;    /* returned events */
   };
```

`timeout` en millisecondes (-1 pour désactiver)

Même retour que `select`

Dans un champ `event` on met ce qu'on attend (`POLLIN`, `POLLOUT`, etc) ; après l'appel, `revents` contient les événements effectifs.

`ppoll` est à `poll` ce que `pselect` est à `select`.

`poll` vs `select`

`poll` plus paramétrable que `select`, mais plus compliqué.

Scalabilité : problèmes 1. et 2. corrigés

1. Nombre de `fd` non limité.
2. Plus besoin de refaire la liste `fds` avant chaque appel.
3. Le noyau doit parcourir entièrement `fds` à chaque appel.
4. Si aucun descripteur n'est éligible, le kernel doit mettre en place des handlers internes pour chaque descripteur.
5. L'application doit parcourir entièrement `fds` après chaque appel.

Solution au problème de scalabilité

Fondements : article Banga et al, 1999

Implémentations :

- `epoll` spécifique à Linux, introduit à la version 2.5.44 (2002)
- `kqueue` propre à BSD (FreeBSD 4.1, 2000) ; serait supérieur à `epoll`
- I/O Completion Port sur Solaris, Windows

Idée : stateless $\rightarrow$ statefull

créer un objet persistant dans le noyau qui décrit un ensemble d'éléments à observer ; surveiller cet objet.

epoll

`epoll` sépare la déclaration et le contrôle des éléments, de l'obtention des évènements :

- on crée une instance `epoll` avec `epoll_create`, qui renvoie un descripteur `epollfd` ;
- on enregistre les descripteurs à surveiller avec `epoll_ctl` ;
- dans la boucle d'évènements, on scrute avec `epoll_wait`.

Réduction possible du nombre d'évènements :
le mode "edge-triggered" (vs "level-triggered") ne signale que les changements d'états.

Bilan : lorsque `nfds` est grand, `select` et `poll` en O(`nfds`), `epoll` en 0(1) [Gammo et al, 2004].

Voir aussi `man epoll`

Évènements graphiques

Applications réseau dotées d'une interface graphique : gestion unifiée des évènements réseau et graphiques

Unix/X11 : `select` sur `ConnexionNumber(display)`

Windows : scrutation avec `MsgWaitForMultipleObjects`

Autre approche : thread graphique + thread réseau