Programmation système II

Socket UNIX, Terminaux, Async IO, Mémoire, ELF

Timothée Ravier

LIFO, INSA-CVL, LIPN

1re année cycle ingénieur STI
2013 – 2014

Plan global

Plan

Socket du domaine UNIX

- Communication inter processus sur un même système ;
- Exemple : `/dev/log/`, socket UNIX utilisée pour envoyer des logs au démon syslog (et maintenant journald avec systemd) ;
- Fonctionnement similaire aux sockets de type AF_INET, sauf pour l'adressage :

```
struct sockaddr_un {
    sa_family_t sun_family; /* Toujours AF_UNIX */
    char sun_path[108];     /* Chemain vers la socket: */
};                          /* char[] terminé par '\0' */
```

Socket UNIX : binding

```
const char *SOCKNAME = "/tmp/mysock";
int sfd, ret;
struct sockaddr_un addr;
sfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sfd == -1)
    exit_with_error("socket");
memset(&addr, 0, sizeof(struct sockaddr_un));
addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path)-1);
/* Attention au code de retour de strncpy ! */
ret = bind(sfd, (struct sockaddr *) &addr,
           sizeof(struct sockaddr_un));
if (ret == -1)
    exit_with_error("bind");
```

Échanges sur une socket UNIX

- Transport fiable : les messages sont livrés dans l'ordre et sans duplication ;
- Taille de paquet potentiellement importante : supérieur à 200ko par défault sous Linux ;
- `ss -x` pour lister ces sockets sur un système (anciennement `netstat -x`).

Messages particuliers

- On peu envoyer des éléments supplémentaires sur une socket UNIX en utilisant `sendmsg` et `recvmsg` :
  - `SCM_RIGHTS` : envoie ou réception d'un descripteur de fichiers. Se comporte comme si on avait effectué un `dup`.
  - `SCM_CREDENTIALS` : envoie le PID, UID, GID du processus. Potentiellement utilisé pour l'authentification.
- Cf man unix(7).

Socket UNIX abstraites

- Spécifique à Linux ;
- Pas de nom dans le système de fichier donc pas de collisions ;
- Pas besoin de créer un chemin dans le système de fichier ;
- Socket détruite lorsque qu'elle n'est plus nécessaire (unlink pour les sockets UNIX classiques) ;
- Assez peu utilisé.

Plan

Terminaux

- Interface de communication avec un programme. Historique en [1] ;
- Auparavant liés à des éléments matériels, ils sont maintenant la plupart du temps virtuels ;
- Chaque "console" a son propre terminal : `tty` ;
- Plusieurs processus peuvent être rattaché à un terminal mais un seul reçoit les entrées clavier, le processus en avant plan.
- Les opérations d'entrée sortie sont bufferisées lorsque l'on utilise bibliothèque C standard :
  - Affichage ligne par ligne pour les terminaux ;
  - Buffer de taille variable sinon (de 4ko à 64ko).

Plan

Contexte

- Jusqu'à présent, toutes les opérations d'entrée / sortie étaient bloquantes ;
- Ces opérations pouvaient durer un court instant (`write()` dans un fichier sur le disque) ou très longtemps (`read()` dans un pipe vide) ;
- On ne pouvait donc travailler qu'avec un seul descripteur de fichier à la fois dans un même processus ou thread ;
- Chaque connexion crée à partir d'une socket devait être gérée dans un thread ou un processus distinct.

Objectifs

- Éviter d'avoir à créer un thread ou processus par connexion ou fichier lu ;
- Gérer un ensemble d'événements avec un seul thread ;
- Pouvoir vérifier si une opération est possible sans bloquer l'exécution du programme (`poll` en anglais) ;
- Optimiser le comportement d'un programme effectuant autant d'accès disques ou de transferts sur le réseau que possible.

Attribut O_NONBLOCK

- Attribut passé à `open` pour indiquer que l'on souhaite effectuer des opérations d'entrée sortie non bloquantes ;
- Attribut associé au descripteur de fichier (affecte tous les threads et les processus fils) ;
- `fcntl(int fd, int commande)` : pour changer cette attribut sur un descripteur de fichier déjà ouvert :

```
int ret = fcntl(fd, F_GETFL);
if (ret != -1) fcntl(fd, F_SETFL, ret|O_NONBLOCK);
```

- Des versions spécifiques Linux de certains appels systèmes (socket) permettent aussi de préciser cette option.

Utilisation

- Toutes les opérations de lecture et écriture ne seront plus bloquantes et peuvent donc échouer pour diverses raisons ;
- Il faut donc impérativement vérifier le code de retour de tous ces appels systèmes ;
- Valeur d'`errno` : `EAGAIN` et `EWOULDBLOCK`.

Implémentation naïve

- On boucle sur les descripteurs de fichiers disponibles et l'on effectue nos opérations ;
- Peu efficace car consomme du temps CPU pour rien.

Alternatives

- Multiplexage des entrées / sorties : `select()` et `poll()` permettent de surveiller un ensemble de descripteurs pour déterminer si une opération est possible.
- Entrées sorties guidée par signaux : demande au noyau d'indiquer à un processus par un signal qu'une opération est possible sur un descripteur de fichier. Meilleures performances que `select()` et `poll()` pour une grande quantité de descripteurs de fichiers ;
- `epoll()` : spécifique Linux, pas de signaux, configuration plus fine, performant.

Polling vs non bloquant

- Lorsque l'on utilise `select()`, `poll()` et `epoll()`, il n'est pas nécessaire de rendre les descripteurs de fichiers non bloquants ;
- En revanche, ces deux modes d'opération sont souvent combinés.

select

- `int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict exceptfds, struct timeval *restrict timeout);`
- Surveille les évènements se produisant sur un ensemble de descripteurs de fichiers ;
- Possibilité d'attendre indéfiniment ou de s'arrêter après un certain temps (timeout) ;
- readfds : ensemble de descripteurs de fichiers sur lesquels on surveille l'arrivée d'input ;
- writefds : ensemble de descripteurs de fichiers sur lesquels on surveille la possibilité d'output ;
- exceptfds : ensemble de descripteurs de fichiers sur lesquels on surveille si une condition

select

- `int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds, fd_set *restrict exceptfds, struct timeval *restrict timeout);`
- On utilise FD_ZERO, FD_SET, FD_CLR, FD_ISSET pour manipuler les fd_set ;
- nfds doit être d'au moins la valeur du plus grand descripteur de fichier + 1 ;
- Lorsque `select` retourne la main (avec un code de retour positif), les fd_set ne contiennent plus que les fd qui peuvent être manipulés ;
- Il faut alors tester chaque fd que l'on veut suivre pour regarder si il est dans le fd_set ;
- Exemple en TD.

poll

- `int poll(struct pollfd fds[], nfds_t nfds, int timeout);`
- On remplace les sets de fd par un seul tableau avec sa taille (nfds) ;
- Chaque élément du tableau est une structure qui indique quels événements on souhaite suivre ;
- timeout :
  - -1 : bloque indéfiniment ;
  - 0 : ne bloque pas, effectue juste la vérification ;
  - n > 0 : bloque pendant n millisecondes.
- Le code de retour correspond au nombre de fd qui sont prêts.

Constat

- `select` et `poll` sont portables, mais lents et peu pratiques à utiliser dès que l'on travaille avec beaucoup de descripteurs de fichiers ;

Gestion avec les signaux

- Être prévenu par un signal qu'une opération est possible ;
- Mise en place d'un gestionnaire de signal (généralement temps réel) ;
- On associe un processus (ou thread) à un descripteur de fichier :
  `fctnl (fd, F_SETOWN, getpid());`
- On définit un signal à envoyer :
  `fcntl (fd, F_SETSIG,numerosignal);`
- A éviter (Rappel : écrire un gestionnaire de signal correct est dur).

epoll

- La liste des descripteurs de fichiers à suivre est maintenant stockée dans le noyau ;
- On s'y réfère à l'aide d'une instance sous forme de descripteur de fichier : `int epoll_create(int size);`
- On manipule la liste avec : `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);`
- On attends un événement avec : `int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents, int timeout);`

libevent, libev...

- Plusieurs bibliothèques en C propose une interface de plus haut niveau pour gérer les entrées sorties sur les descripteurs de fichier ;
- Abstrait aussi les différences entre les UNIX et Windows ;
- Elles ajoutent généralement la gestion des « timers » ;
- Simplifie notablement la gestion des entrées sorties.

signalfd

- La gestion des signaux est fastidieuse et écrire un gestionnaire de signal correct est très difficile ;
- `int signalfd(int fd, const sigset_t *mask, int flags);`
- Crée un descripteur de fichier pour recevoir les signaux de façon asynchrone ;
- Il faut bloquer les signaux (`sigprocmask`) et les traiter ensuite en lisant les informations depuis le descripteur de fichier de `signalfd`.

signalfd : utilisation

- Rappel : SIGKILL et SIGSTOP ne peuvent pas être gérés ;
- SIGBUS, SIGFPE, SIGILL, et SIGSEGV sont générés à la suite d'une erreur d'exécution d'une instruction et ne peuvent pas être gérés par signalfd ; Ils doivent être traités sur l'instant.
- Il faut donc :
  - écrire un gestionnaire de signaux minimal pour SIGBUS, SIGFPE, SIGILL, et SIGSEGV ;
  - gérer les autres signaux avec signalfd.
- Exemple complet en TD.

Async IO : Linux AIO / POSIX AIO

- Peu utilisé, mais potentiellement très performant (voir [2] et aio(7)) ;
- Principe : Soumettre des opérations à effectuer, laisser le noyau les traiter de façon asynchrone, vérifier que les opérations ont été correctement effectuées ;
- On peut être notifié soit à l'aide de signaux, par l'exécution d'un thread, ou rien du tout.

Async IO : Linux AIO / POSIX AIO

1. Ouverture d'un contexte pour soumettre et valider la complétion d'une requête d'entrée / sortie ;
2. Création d'une ou plusieurs requêtes et configuration pour indiquer quelles opérations doivent être effectuées ;
3. Soumission de ces requêtes au contexte d'entrée / sortie, qui va les envoyer au noyau pour qu'elles soient effectuées ;
4. Vérifier la complétion des requêtes d'un contexte ;
5. Retour en 2 si nécessaire.

Attribut O_DIRECT, O_DSYNC, O_SYNC

- Attributs passés à `open` ou appliqués sur un descripteur de fichier avec `fcntl()` ;
- `O_DIRECT` : indique au noyau qu'il doit minimiser les effets de cache liés à un fichier. Dans la plupart des cas, cela **réduit** les performances. Aucune garantie n'est donnée lors de l'écriture de données sur le disque (les données ne sont pas toujours écrites directement comme l'option pourrait laisser l'entendre) ;
- `O_DSYNC` et `O_SYNC` : assure que les données sont écrites sur le disque lorsque l'appel système `write` se termine. Différence mineure entre `O_DSYNC` et `O_SYNC` (cf open(2)). Privilégier `O_DSYNC`.

Plan

Mémoire physique

- La mémoire physique est composée de mots contigus repérés par une adresse physique ;
- Ces mots peuvent être de longueur différente : 8, 16, 32, 64, 128 bits ;
- Généralement, l'octet est la plus petite unité adressable par un processeur.

Mémoire cache

- Les accès direct à la mémoire principale depuis le processeur sont longs par rapport à la vitesse d'exécution des instructions du processeur ;
- Présence des mémoires caches ;
- On cherche les informations en premier dans les caches et ensuite dans la mémoire centrale.

Gestion du cache

- Principe de la localité pour déterminer si un mot va être mis dans le cache :
  - Localité spatiale : si un mot est demandé, les mots voisins ont une forte chance d'être demandés ;
  - Localité temporelle : si un mot est demandé à l'instant t, il est probable qu'il le soit à nouveau à l'instant t+1...

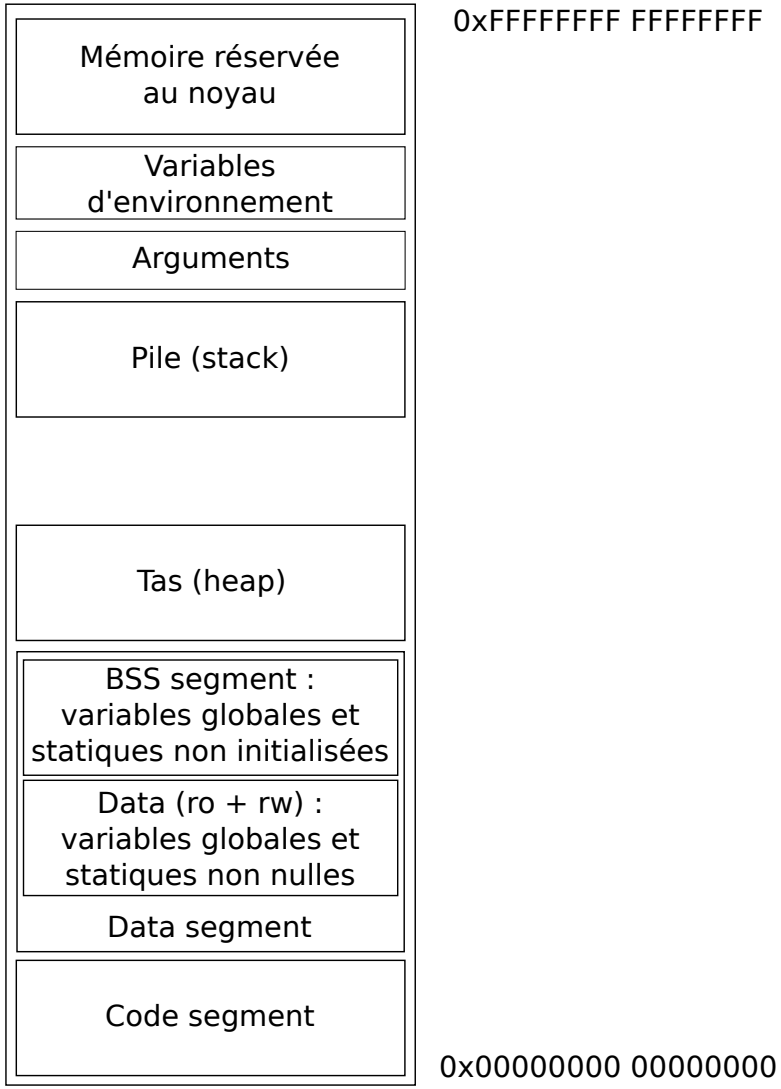
Gestion de la mémoire cache

- Lorsque le contenu d'un mot est modifié dans la mémoire cache une opération est prévue pour qu'il soit aussi modifié dans la mémoire centrale :
  - Soit immédiatement : « write-through » ;
  - Soit lorsque l'on retire ce mot de la mémoire cache : « write-back ».

Espace d'adressage d'un processus

- Les processus ne travaillent pas directement avec les adresses mémoires correspondants à l'emplacement des données dans la mémoire physique ;
- On alloue à chaque processus un espace d'adressage virtuel ;
- Celui-ci est de taille variable suivant l'architecture :
  - Intel i386 ou x86 : 32 bits = 2$\hat{}$32 (4GiB) ;
  - AMD64 ou x86-64 : 64 bits = 2$\hat{}$64 (16 EiB).
- Les implémentation matérielles sont encore limitées à 2$\hat{}$48 octets pour l'adressage physique (256 TiB).

Espace d'adressage virtuel d'un processus



Gestion de la mémoire

- L'espace d'adressage virtuel est découpé en « segments » ou « pages », qui font généralement 4kiB ;
- Pour accéder à une donnée dans une page, il faut indiquer son décalage ou « offset » ;
- L'espace d'adressage physique est aussi découpé en morceaux : les cadres (frames) ;
- Pour accéder aux données dans la mémoire, il faut faire correspondre une adresse virtuelle utilisée par un programme avec une adresse physique (donc faire correspondre les pages avec les cadres) ;

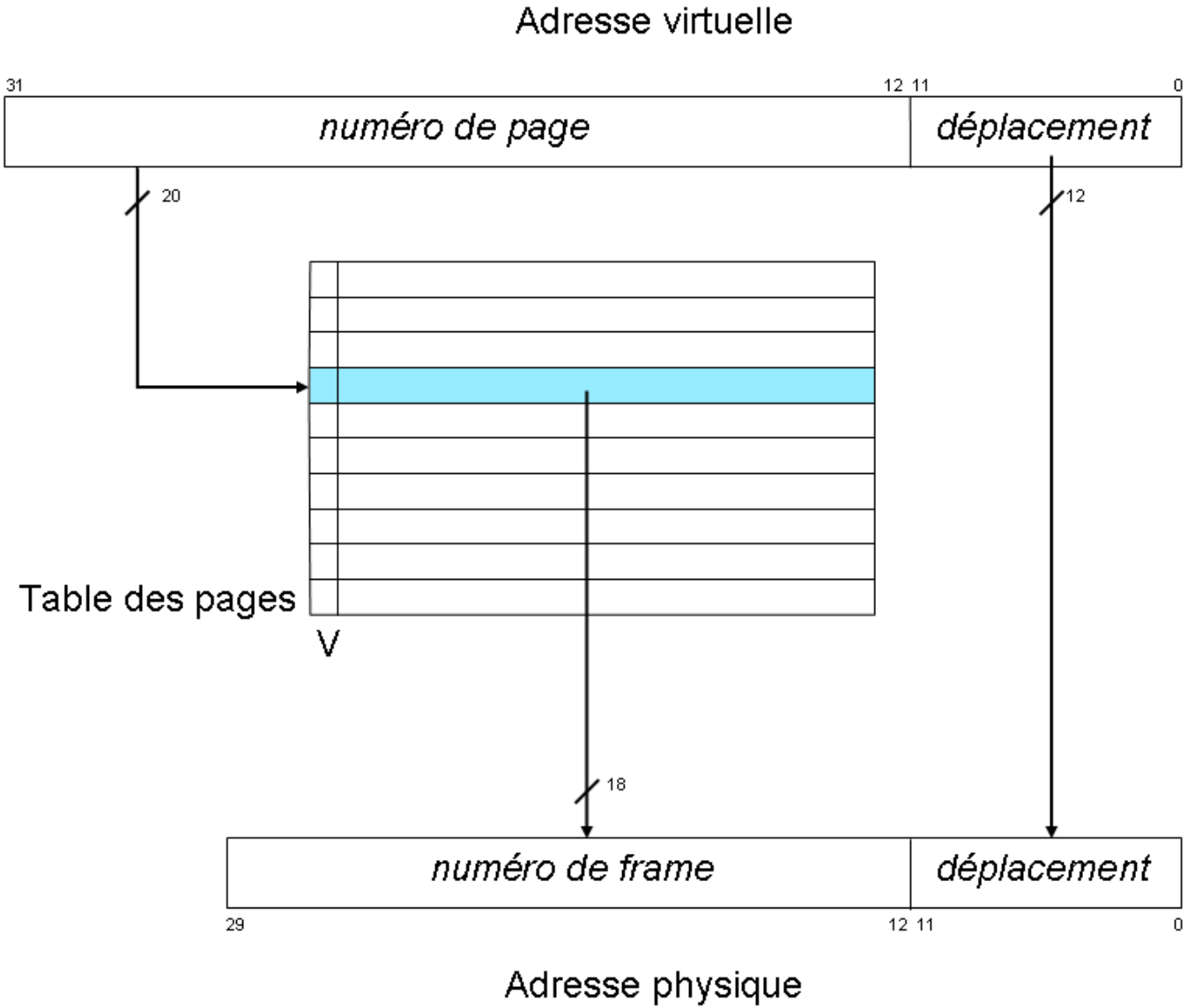
Memory Management Unit

- Pour cela le noyau maintient une table de pages par processus (avec plusieurs niveaux d'indirection) ;
- Il y maintient l'association entre cadre et pages ;
- Cette table est chargée dans la Memory Management Unit (MMU) avant l'exécution de chaque processus ;
- La MMU contient un registre qui référence l'ensemble des zones mémoire associées à un processus ;
- C'est un élément matériel pour avoir de meilleures performances.

Accès et changement de contexte

- Des bits de protection (RWX) sont associés à chaque page ;
- Lorsqu'un processus veut accéder à une valeur, la MMU s'assure que la zone est bien enregistrée pour ce processus, que les restrictions sont respectées et effectue la correspondance pour que le processeur utilise l'adresse physique ;
- À chaque changement de contexte (changement de processus en cours d'exécution), il faut changer la table dans la MMU puisque chaque table est propre à un processus.

Conversion des adresses virtuelles



Allocation (version simplifiée)

- Lorsque qu'un programme fait appel à `malloc`, deux cas peuvent se produire :
  - Il n'y a pas de pages avec de l'espace disponible pour ce processus. `malloc` demande au noyau de lui allouer plus d'espace mémoire et retourne un emplacement valide dans cette nouvelle page ;
  - Il reste de la place dans les pages allouées au processus : `malloc` retourne une adresse qui pointe vers une zone libre de taille au moins égale à la taille demandée (généralement arrondie à un multiple de 2 supérieur) ;

Pagination et SWAP

- La quantité de mémoire présentée aux processus est plus importante que la quantité réelle de mémoire disponible ;
- Lorsqu'il n'y a plus de place dans la mémoire, les pages les moins utilisées sont déplacées dans la SWAP. L'état de chaque page est stocké dans la table des pages ;
- Si un accès est effectué à une page stockée dans la SWAP, le noyau demande le chargement dans la RAM de cette page ;
- Si il n'y a plus de place dans la RAM, alors il faut au préalable déplacer une autre page ou si la SWAP est pleine, tuer un processus pour faire de la place.

Plan

Executable and Linking Format (ELF)

- Format de fichier binaire standard pour stocker les exécutables, les bibliothèques dynamiques, le code objet (fichiers `.o`) et les core dumps ;
- Format couramment utilisé sous les systèmes UNIX et Linux ;
- `file` : utilitaire permettant de déterminer le type d'un fichier ;
- `readelf`, `objdump` : outils pour lire les informations d'un fichier ELF.

Executable and Linking Format (ELF)

```
$ readelf -a /usr/bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 ...
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x40487f
  Start of program headers:          64 (bytes into file)
  Start of section headers:          108312 (bytes into file)
  ...
```

Plan

Gestion du signal SIGPIPE I

- Se produit lorsque l'on essaie d'écrire dans un pipe ou une socket qui n'est plus associé à un processus en lecture ;
- Exemples : terminaison abrupte ou volontaire du lecteur (`close(pipe);`), déconnexion abrupte ou volontaire...
- Comportement par défaut : Terminaison du programme ;
- Le code de retour de `write` est alors SIGPIPE ;
- Il est toujours plus facile de travailler avec le code de retour d'une fonction qu'avec des signaux ;
- On ignorera ou bloquera donc toujours SIGPIPE.

Gestion du signal SIGPIPE II

- Deux choix possibles suivant le contexte :
  - Si l'on peut changer la gestion des signaux : ignorer le signal et traiter correctement le code de retour de `write`, `send` ou `sendmsg` ;
  - Sinon (si l'on écrit une bibliothèque par exemple) : utiliser l'option `MSG_NOSIGNAL` avec `send` ou `sendmsg` et ne pas utiliser directement `write`. Traiter le code de retour proprement ;
  - L'option `SO_NOSIGPIPE` pour les sockets n'est pas supportée sous Linux.

Code propre : gestion des erreurs I

- Il est impératif de vérifier tous les codes de retour de toutes les fonctions en C ;
- Il faut traiter tous les cas précautionneusement ;
- Si l'on souhaite se débarrasser de certains cas de façon systématique, on peut écrire :

```
ssize_t Read(int fd, void *buf, size_t count)
{
    ssize_t ret = read(fd, buf, count);
    if ((ret == -1) && (errno == EIO)) {
        perror("read");
        exit(EXIT_FAILURE);
    } else { return ret; }
}
```

Code propre : gestion des erreurs II

- Il est parfois judicieux d'arrêter l'exécution du programme rapidement en cas d'erreur pour ne pas entraîner de bugs difficiles à comprendre plus tard.

Code propre : read/write

```
ssize_t taille_lue = read(STDIN_FILENO, buffer, buffer_size)
if (taille_lue == -1) {
  perror("Erreur read STDIN"); exit(EXIT_FAILURE);
} else if (taille_lue == 0) { break; } /* EOF */
else {
  ssize_t taille_ecrite = 0;
  while (taille_ecrite < taille_lue) {
    retour_write = write(tube, buffer + taille_ecrite,
                         taille_lue - taille_ecrite);
    if (retour_write == -1) {
      perror("Erreur write tube"); exit(EXIT_FAILURE);
    }
    taille_ecrite += retour_write;
} }
```

Code propre : utiliser les bons types pour les tailles

- Evitez :

```
int i;
for (i = 0; i < z; i++) { // code }
```

- Préférez :

```
size_t i;
for (i = 0; i < z; ++i) { // code }
```

- Le dépassement de la taille d'un entier signé est une opération indéfinie ;
- Le dépassement de la taille d'un entier non signé est une opération complètement définie : la valeur boucle à zéro.

Code propre : limiter la complexitée du code I

- Il est très facile de faire des erreurs en C :
  - = au lieu de == dans un if ;
  - oubli des { } pour une boucle for avec plusieurs instructions ;
  - ...
- Il faut donc essayer de limiter la complexité du code en réduisant chaque ligne à une expression la plus simple possible ;
- Il ne faut pas avoir peur d'ajouter des variables intermédiaires si cela facilite la lecture ;
- Le compilateur se chargera d'optimiser l'usage de la mémoire à ce niveau.

Code propre : limiter la complexitée du code II

- Evitez :

```
if (read(fd, buff, size) < 0) { // erreur }
else {
    // code
}
```

- Préférez :

```
ssize_t ret = read(fd, buff, size);
if (ret > 0) {
    // ok
} else if (ret == 0) {
    // EOF
] else {
    // erreur
}
```

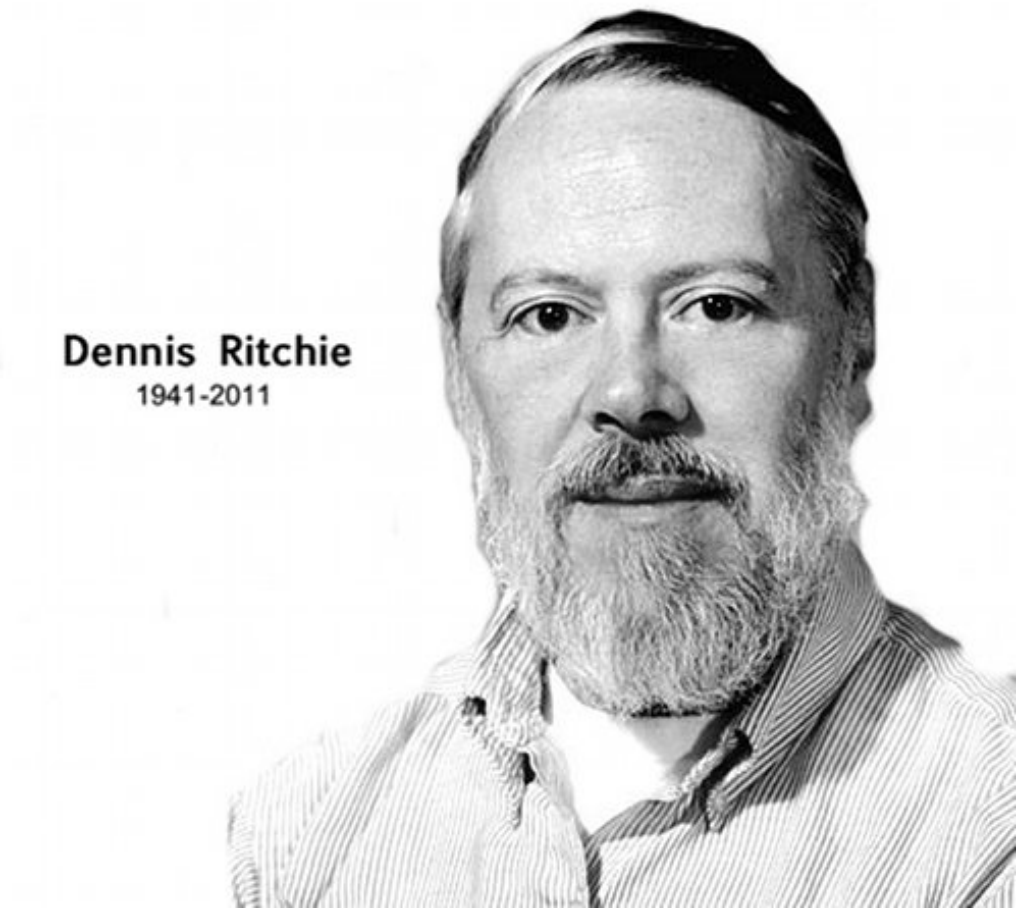
Code propre : limiter la complexitée du code II

- Evitez :

```
for (i = 0; i < z; z++) {
    tab[i++][a--][x] = a;
    *(--b) = a++;
}
```

- Vous risquez de tomber sur des cas non définis dans le standard C (undefined behavior) ce qui peut mener à des bugs difficiles à trouver ;
- Préférez la version longue avec les variables intermédiaires ;
- Le compilateur optimisera sans aucun soucis les codes source de ce type.

Petite pensée pour le créateur du C et d'UNIX : Dennis Ritchie



Références I

1. The TTY demystified : `http://www.linusakesson.net/programming/tty/` ;
2. AIO User Guide : `https://code.google.com/p/kernel/wiki/AIOUserGuide`.
3. Linux Kernel Development (3rd Edition) by Robert Love.
4. ELF 101 posters : `https://code.google.com/p/corkami/wiki/ELF101`
5. Schéma de traduction des adresses virtuelles en adresses physiques : `http://fr.wikipedia.org/wiki/M%C3%A9moire_virtuelle`