Tuning de code
optimisation d'un filtre

Yann Guidon

EN DEUX MOTS **Cela commence par « Est-ce que ce code simple est aussi rapide que possible ? » et nous embarque dans « Quelle est la méthode la plus performante ? ».**

Cet intermède dans la série sur la compression de données n'est pas seulement un exercice d'optimisation ou une caractérisation du comportement d'un noyau particulier (un vieux 2.4.19pre7), nous dégagerons aussi quelques points permettant d'écrire des programmes un peu plus efficaces.

1) La question

Dans le premier article d'introduction à la compression des signaux (GLMF numéro 73, juin 2005), je mentionne qu'on peut traiter des fichiers au moyen de programmes « filtres » qui peuvent avoir la forme générale suivante :

```
typedef signed short int SAMPLE;
SAMPLE droite, gauche;
FILE entree, sortie;

entree = fopen(nom_fichier_entree, "rb");
sortie = fopen(nom_fichier_sortie, "wb");

while ( fread(&gauche, 1, sizeof(SAMPLE), entree)
      + fread(&droite, 1, sizeof(SAMPLE), entree)
        == sizeof(SAMPLE) * 2) {

/* traitement des échantillons ici */

 fwrite(&gauche, 1, sizeof(SAMPLE), sortie);
 fwrite(&droite, 1, sizeof(SAMPLE), sortie);
}

fclose(entree);
fclose(sortie);
```

Cependant, les fichiers à traiter sont *très* gros et les 618Mo de notre fichier de référence ne sont qu'un avant-goût de certaines applications envisagées pour la suite de la série sur la compression. Une petite modification du code source pourrait entraîner une variation de plusieurs secondes sur le temps total d'exécution. L'appel aux fonctions `fread()` et `fwrite()` pour *chaque* échantillon me semble être un premier point à éviter. J'avais écrit le code ci-dessus dans un souci de concision et de simplicité, mais est-ce adapté aux situations plus tendues, telles que dans une application « temps réel » comme l'enregistrement ou la lecture d'un fichier *très* long ? L'idée de cet article est d'examiner le comportement du noyau et de coder un autre filtre plus performant, même s'il doit être plus compliqué. Cependant, comme le résultat doit être exécutable sur presque n'importe quel système Linux, l'éventail des techniques possibles est restreint. En raison de cette généralité, utiliser le mot « optimisation » dans le titre de cet article serait presque un abus de langage s'il n'y avait pas toutes les mesures. Tout bon codeur sait qu'optimiser sans mesurer conduit à de pires résultats qu'au départ, et ce qui suit en est un exemple flagrant.

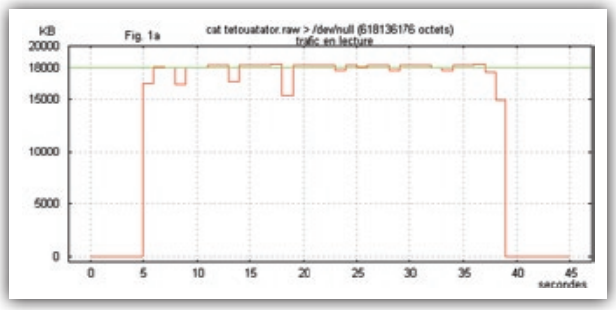
2) Les premières mesures

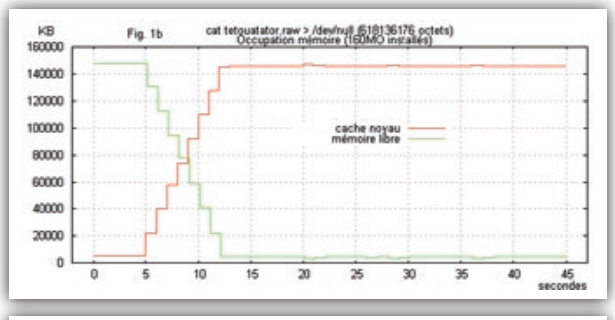
Pour commencer, intéressons-nous aux cas les plus simples : la lecture et l'écriture des données sur le disque. Tout le reste n'est qu'une combinaison des effets de ces deux opérations.

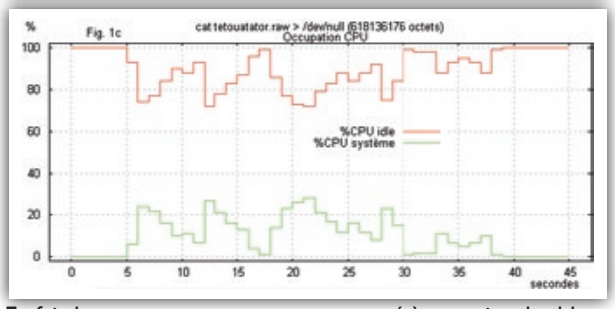
2.1) Lecture

La première mesure concerne la lecture de notre fichier de référence (fourni avec GLMF numéro 73 de juin 2005, mais tout fichier de 618136176 Ko fait aussi bien l'affaire). La commande `cat tetouatator.raw > /dev/null` est expédiée en 34 secondes. La *figure 1a* montre que le débit moyen tourne autour de 18Mo/s (618136176/18000 Ko/s =~ 34s : le compte y est). Un paramètre important pour ce type de mesure est la mémoire installée. Le noyau Linux utilise presque toute la RAM disponible pour y loger les derniers morceaux de fichiers accédés. Ainsi, tout accès ultérieur sera presque instantané. Cette mémoire cache en lecture peut interférer avec les mesures : s'il reste un petit bout de fichier en RAM, l'exécution suivante sera anormalement plus courte. Le moyen d'éviter cela (pour des mesures répétitives) est d'opérer sur des fichiers au moins deux fois plus gros que la mémoire installée dans l'ordinateur. Ici, il dispose de 160Mo de RAM et `tetouatator.raw` occupe 618Mo, ce qui ne pose pas de problème (618/160=3,86 > 2).

La *figure 1b* montre l'état de la cache de lecture. Au rythme de 18Mo par seconde, il faut 8 secondes pour épuiser toute la RAM libre. Encore 8 secondes plus tard, toute la mémoire cache a été réécrite. On observe même une légère oscillation sur une période d'environ 8 secondes. La *figure 1c* présente la charge du processeur durant la lecture. Le disque dur tourne à plein régime mais grâce au mode DMA, le processeur n'est occupé qu'au quart de ses possibilités dans le pire des cas. On observe aussi la même oscillation que sur la *figure 1b*, mais amplifiée. Je n'ai aucune explication sur la raison de telles variations.







En fait, le processeur est surtout occupé à recopier des blocs de données en RAM :

- Une fois qu'un bloc est lu en provenance du disque, il est recopié de la cache du noyau vers l'espace utilisateur (*userland*) du programme `cat`.
- `cat` envoie ce bloc (par `write()`) en sortie standard, où le *shell* le lit : c'est au moins une copie supplémentaire, de userland(cat) à userland(shell), qui passe probablement aussi par le noyau.
- Le shell va écrire ce bloc vers `/dev/null`. Il y a fort à parier qu'une copie supplémentaire de userland(shell) à kernel est aussi effectuée par `write()`, qui ignore probablement que le bloc sera ignoré.

Il y a plusieurs spéculations, mais il ressort que passer par le shell induirait une occupation CPU plus grande (à cause de la sur-duplication des données) que si on avait écrit un simple programme en C pour lancer uniquement `read()`.

Il reste cependant assez de temps CPU pour faire d'autres choses à côté (c'est un pauvre Celeron Coppermine rev.6 à 550MHz pour donner une vague idée).

NOTE

Permettre à un programme de lire ses données dans `stdin` est une fonction très pratique et facile à coder. Cela évite de perdre du temps dans les cas où le flux à traiter doit passer par plusieurs filtres, obligeant sinon à écrire le flux transformé sur le disque avant de le relire.

Toutefois, si la vitesse importe vraiment et si le programme reçoit les données directement d'un fichier sans passer par d'autres programmes à travers un *pipe* du shell, il vaut mieux effectuer « soi-même » la lecture pour réduire les duplications en RAM (et encore, on ne parle pas de `fread()`/`fwrite()`).

Un programme doit donc utiliser ces deux techniques pour être efficace dans les deux situations.

2.2) Réduction des copies

Pour vérifier que les duplications peuvent être réduites, il suffit d'écrire un petit programme, tel que `lecture.c`, ignorant simplement le résultat de la lecture.

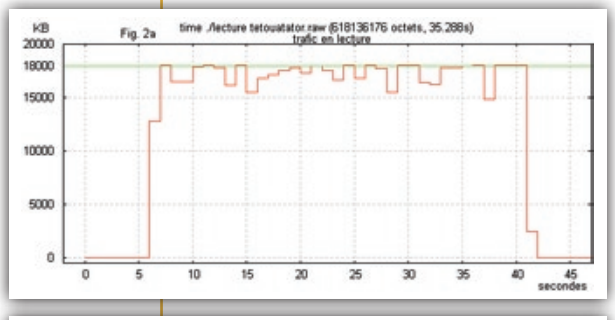
```
/* fichier lecture.c
créé par Yann GUIDON (whygee@f-cpu.org)
Fri Jul 29 10:46:21 CEST 2005
 fonction :
   lire un fichier le plus vite possible
 compilation :
   gcc -Wall -g -o lecture lecture.c
 invocation :
   lecture nom_du_fichier
*/
#define _FILE_OFFSET_BITS 64
#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>  /* open() */
#include <errno.h>  /* perror() */
#include <stdio.h>  /* fprintf(), fflush() */
#include <stdlib.h> /* malloc() */
#include <unistd.h> /* exit(), read() */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE (64*1024)
#endif
int main (int argc, char *argv[]) {
  off_t total_lu=0, buffsize=BUFFER_SIZE;
  int fd_in;
  ssize_t t=0;
  char *buffer;
  fd_in = open(argv[1], O_RDONLY);
  if (fd_in == -1) {
    perror("Erreur à l'ouverture de la source");
    exit(EXIT_FAILURE);
  }
  buffer=malloc(buffsize);
  if (buffer == NULL) {
    perror("Erreur de malloc() ");
    exit(EXIT_FAILURE);
  }
  while ( (t = read(fd_in, buffer, buffsize)) > 0 ) {
    total_lu+=t;
    fprintf(stderr,"%lld%c", total_lu, 0xD);
    fflush(NULL);
  }
  if (t < 0) {
    perror("Erreur de lecture ");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

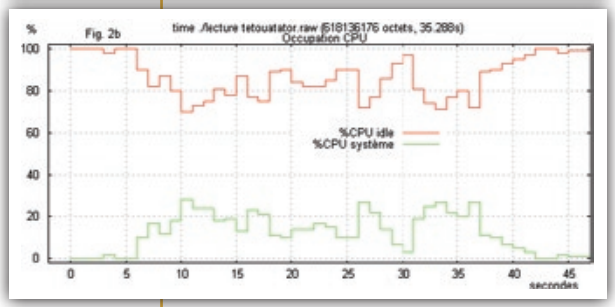
NOTE

L'objectif initial était de tester l'option `READ_O_DIRECT` mais celle-ci ne fonctionne pas sur les fichiers normaux (sur ce système-là, en tout cas). Voir GLMF numéro 68 de janvier 2005 pour une discussion à ce sujet.

Sans l'option `READ_O_DIRECT`, on peut tout de même observer la différence d'utilisation du processeur. Ici, la taille du tampon de

lecture a été choisie pour être la moitié de la taille de la mémoire cache L2, afin de réduire l'impact de la duplication du tampon vers userland. La figure *figure 2a* montre que le débit et le profil sont tout à fait comparables. La *figure 2b* suggère qu'il n'y a pas de différence dans l'occupation totale du processeur, malgré une forme un peu différente.





On remarque aussi que le temps d'exécution augmente d'une seconde. L'hypothèse sur l'impact des copies internes n'est donc pas vérifiée ici.

2.3) Importance de la taille du tampon

Voici encore un sujet de discussion interminable. On peut s'attendre à ce que le système (librairies et noyau) se débrouille correctement dans une grande variété de situations pour réconcilier tout le monde. Pourtant, il doit bien y avoir un point de fonctionnement meilleur que les autres, même s'il dépend du système ?

Le code du filtre original repose sur la librairie d'entrée/sortie standard, qui s'occupe de gérer des tampons avant de les envoyer au noyau. Mais cela ajoute des instructions qui ne contribuent pas directement à remplir la fonction du programme. En gérant nous-mêmes nos tampons, nous nous assurons qu'il n'y a pas trop d'instructions superflues.

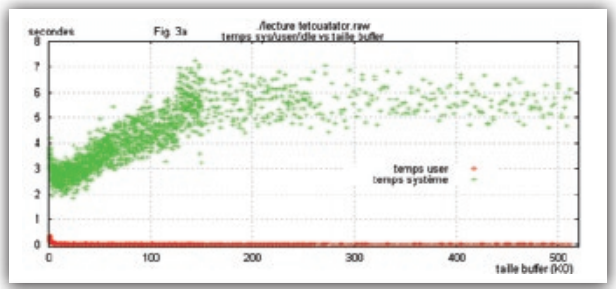
De toute façon, si vous lisez cet article, c'est que vous n'avez pas l'intention de passer par les fonctions `fwrite()` et `fread()`. Les programmes que nous codons devant manipuler des gigaoctets de données, on peut faire l'effort de coder notre propre code de gestion de *buffers* pour les accéder directement, sans avoir à appeler une librairie qui dupliquera encore des données. Mais on ne peut pas faire n'importe quoi :

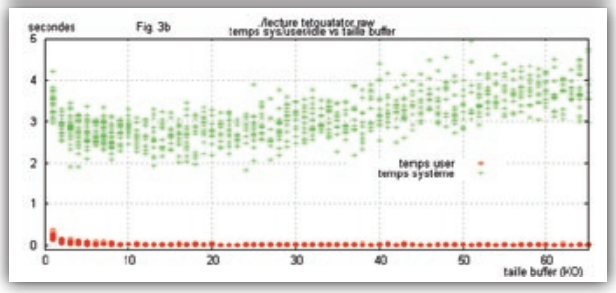
- Si les blocs sont trop courts, le processeur perd son temps à changer de contexte (passer du mode *kernel* au mode *user*, et vice versa, prend à chaque fois un certain temps).
- Si les blocs sont trop longs, le temps de réponse du programme risque de souffrir. Par exemple, si le bloc fait 16Mo, il faudra une seconde pour le lire (sur cet ordinateur), pendant laquelle le programme n'aura pas la main pour répondre à d'autres événements ou effectuer des opérations utiles.

Le programme précédent utilisait un bloc de la moitié de la taille de la mémoire cache, mais est-ce la meilleure taille ? Pour le savoir, il suffit encore de le mesurer. Les *figure 3a* et *figure 3b* ont été obtenues au moyen de scripts du genre :

```
for i in $(seq 64 -1 1)
do
  echo $i:
  gcc -DBUFFER_SIZE=($i*1024) -o lecture lecture.c
  echo $i $(/usr/bin/time -f '%e %U %S' \
    ./lecture /common/tetouatator.raw 2>&1 ) >> bufsz.2.out
done
```

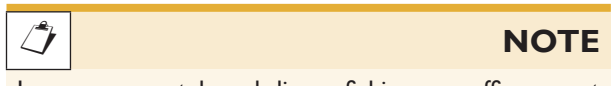
La boucle est dans l'ordre décroissant pour réduire autant que possible les erreurs de mesure dues au démarrage du script et je voudrais savoir exactement ce qui se passe à 1Ko. J'ai aussi enlevé le code d'affichage de la progression, aussi bien pour ne pas perdre de temps que pour ne pas gonfler le fichier de sortie avec des nombres sans intérêt. Avec plus de 1900 points sur la *figure 3a* et 35s par point, je vous laisse calculer le temps qu'il a fallu pour obtenir un des graphes les plus intéressants de ces pages, puisqu'on y voit enfin une *singularité*.





Le temps total ne varie quasiment pas (34,5s, plus ou moins un quart de seconde) car c'est l'accès au disque qui est prépondérant. De plus, les mécanismes de préchargement spéculatif dans le noyau et le disque dur masquent presque complètement l'influence des blocs très courts. On voit que ce n'est pas le temps *user* qui varie, mais le temps *systeme*, à part pour les très petits blocs. Les très gros blocs ne semblent

pas les plus efficaces, puisqu'ils prennent le plus de temps à traiter. La situation change brusquement à 128Ko, c'est-à-dire la taille de la mémoire cache L2 : la pente suggère que cet cache accélère un peu les copies internes. Le creux se situe vers 16Ko, c'est-à-dire la taille de la mémoire cache L1. Entre ces deux extrémités, le temps utilisé par le noyau change du simple au double. Je me suis donc concentré sur la zone entre 1Ko et 64Ko sur la *figure 3b*. L'imprécision du chronométrage vient certainement de la manière dont le temps est mesuré, en *tics* du système au lieu de microsecondes ou autre. La durée d'exécution du programme étant comptée avec une très grosse louche, il y a beaucoup de variations d'une mesure à l'autre. De plus, Linux n'étant pas conçu pour être ultraprévisible (du moins pour la branche 2.4), les appels système peuvent suivre des chemins d'exécution différents si ça les arrange.



NOTE

Le noyau permet donc de lire un fichier assez efficacement avec une grande gamme de taille de blocs. Les facteurs limitant aux deux extrêmes sont :

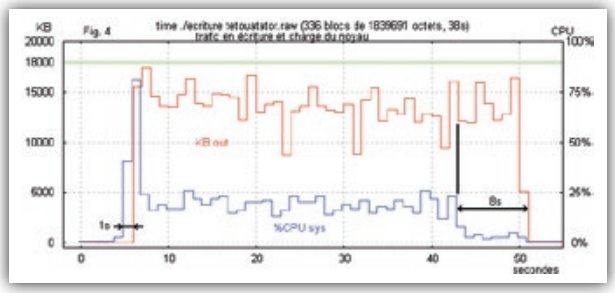
- Le temps de l'appel système, qui déclenche un changement de contexte relativement coûteux, pénalise les blocs très courts (< 4K0).
- La cache de lecture du noyau entraîne un doublement de la taille effective (l'empreinte en mémoire) d'un bloc. Si la taille d'un bloc est supérieure à la moitié de la mémoire totale, le noyau va devoir faire de la place pour la cache, ce qui entraîne la mise en *swap* du tampon en userland et du programme. Je vous laisse imaginer le bruit du crépitement du disque dur.

Des blocs de quelques dizaines de kilooctets représentent un bon compromis entre encombrement, latence, renouvellement de la mémoire cache du processeur... De plus, c'est proche de la taille des tampons de préchargement spéculatif (*prefetch*) du noyau et du disque dur (voir `man hdparm`).

2.4) Ecriture

L'objet de cette mesure est simplement de chronométrer le temps d'écriture pour la suite, puisque le comportement est probablement très proche de la lecture.

A ceci près que je désactive traditionnellement la cache d'écriture interne au disque dur, mais est-ce bien nécessaire sur un ordinateur portable, où les batteries prennent le relais en cas de coupure de courant ? De toute façon, la cache d'écriture du noyau prend le relais. La *figure 4* représente le débit, seconde par seconde, de l'écriture d'un fichier « vide » sur le disque, au moyen d'un petit programme ad hoc dérivé de `lecture.c`.



En voulant écrire un fichier de la taille exacte de `tetouatator.raw` pour obtenir des nombres aussi précis que possible, je me suis aperçu qu'on pouvait la factoriser pour utiliser des blocs de taille raisonnable, ce qui a conduit aux lignes suivantes :

```
>8 snip-snip >8
#define BUFFER_SIZE (1839691)
#define BUFFER_NB  (7*3*2*2*2*2)
>8 snip-snip >8
for (i=0; i<BUFFER_NB; i++) {
  t=write(fd_out, buffer, BUFFER_SIZE);
  if (t < BUFFER_SIZE) {
    perror("Erreur d'écriture");
    exit(EXIT_FAILURE);
  }
}
>8 snip-snip >8
```

Le temps d'exécution, variant entre 37,2s et 39s, est de 38,2s en moyenne. En raison de la grande taille du tampon et de la concision extrême du cœur de la boucle, le temps utilisé par le programme est tellement court que `time` indique 0. Le kernel occupe entre 6,7s et 7s, le processeur est donc exploité en moyenne à 18%.

Le débit est un peu moins rapide et le profil est plus saccadé que la lecture, puisqu'il faut en plus allouer de nouveaux *inodes*. On voit aussi que la cache d'écriture fonctionne à fond et on voit son effet :

- Au début, lorsque le programme la remplit à craquer (notez aussi le décalage d'une seconde) ;
- Et à la fin, lorsque l'écriture continue au moins 7 secondes après l'arrêt du programme (ce qui correspond bien à la taille de la RAM utilisable).

Le pic d'activité initial génère le même volume de données.

3) Effets combinés

Nos programmes filtres font appel à la lecture et à l'écriture presque simultanément. Mais il est physiquement impossible pour le disque dur d'effectuer ces deux opérations en même temps, en particulier parce que les adresses sont différentes (à moins de disposer de deux disques durs sur des interfaces différentes, ce qui n'est pas le cas le plus courant).

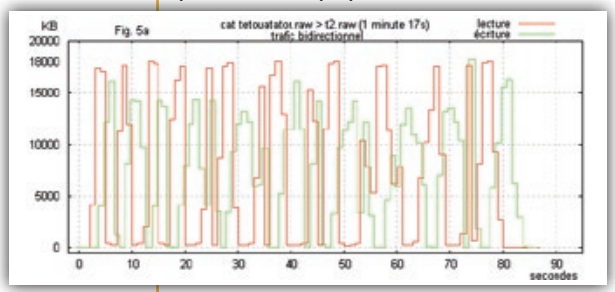
Dans les mesures effectuées ici, la partition contenant les données est relativement petite (10% de la taille totale du disque), ce qui réduit les mouvements de la tête et donc le temps perdu à la repositionner lors du passage du mode lecture au mode écriture et vice versa. Mais il faut bien alterner à un moment ou un autre.

Dans le cas présent, cette alternance est dictée par le système de fichier : `ext3` force une resynchronisation (*commit*) toutes les 5 secondes. La lecture va donc être interrompue par le noyau pour vider son cache d'écriture.

3.1) Alternance simple

La *figure 5a* trace le trafic en lecture et en écriture lors de la recopie du fichier en passant par le shell. Le temps total (indiqué par time, donc sans tenir compte du dernier commit mais ce n'est pas important) est de 77s. Normalement, 35s pour la lecture et 38s pour l'écriture donnent un total de 72s, il reste donc environ 5s perdues dans les « collisions » entre l'écriture et la lecture.

En regardant la *figure 5a*, on a l'impression que les accès sont un peu chaotiques. En toute logique, si on remet un peu d'ordre dedans, on devrait peut-être grappiller une partie du temps perdu.



Cette réflexion est donc suivie de la rédaction d'un autre programme qui va se servir d'un maximum de mémoire pour charger une partie du fichier, avant de le réécrire par gros morceaux. Le tout est de trouver la bonne taille du tampon pour ne pas entrer en conflit avec ceux du kernel. La meilleure taille pour lire les données ayant été déterminée autour de 16Ko, il faut effectuer de nombreuses opérations d'écriture et de lecture pour remplir un tampon beaucoup plus gros, supérieur à la moitié de la RAM. Le tout est de trouver comment empêcher le kernel de mettre le programme en swap pour libérer de la mémoire pour son propre cache (désactiver la swap n'étant pas une solution satisfaisante). Autre question : est-il nécessaire de « chauffer la mémoire » ? Sachant que Linux a une allocation « optimiste » (selon `man malloc`), la mémoire allouée ne sera même pas prête à l'utilisation au retour de `malloc()`. C'est la première référence (écriture ou lecture) sur *chaque* page qui va déclencher la traduction d'adresse. L'idée serait d'effectuer un premier accès à chaque page du bloc alloué pour forcer la translation avant la lecture. Mais il y a deux facteurs limitant :

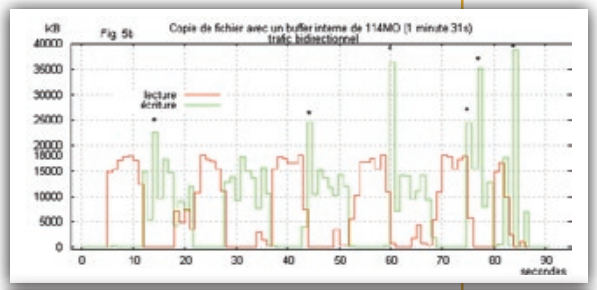
- D'abord, sur x86, les pages font 4Ko (le support des grandes pages n'étant pas disponible partout), il y a un nombre limité d'entrées dans la TLB et elle ne peut pas contenir tout le *mapping* de la RAM, ni de notre immense tampon. Le temps d'accéder à la fin du tampon, la TLB aura été remplacée plusieurs fois.
- La lecture est assez lente pour permettre aux autres opérations internes de s'exécuter « en même temps » (heureusement).

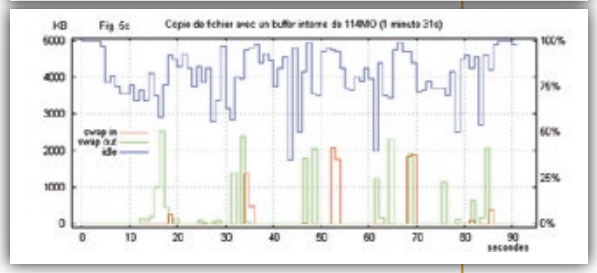
Sans oublier qu'une procédure de chauffe serait une étape « sérialisante » pour le programme, c'est-à-dire un retard inutile pour passer à la lecture proprement dite. Par contre, il y a une autre solution intermédiaire : utiliser le temps *idle* pour chauffer les prochains blocs. Mais on ne peut pas encore utiliser ce temps car `read()` et `write()` sont bloquants et ne redonnent la main au programme que lorsqu'ils ont terminé leur travail. Il faudrait du code *multithread* ou bien faire appel à `select()`. Et encore, il reste à prouver que cette méthode permet réellement de gagner du temps. De plus, le temps « libre » ne sera plus disponible lorsque le filtre remplira sa fonction, au moyen de code qui prendra probablement plus de temps à exécuter que `read()` et `write()` réunis. Nous voilà donc avec `copie.c`, un programme encore simple, constitué essentiellement d'une paire de boucles.

```
/* fichier copie.c
créé par Yann GUIDON (whygee@f-cpu.org)
version Mon Aug  1 13:14:06 CEST 2005
 fonction :
   copie un fichier le plus vite possible
   au moyen d'un gigantesque buffer !
 compilation :
   gcc -Wall -g -o copie copie.c
 invocation :
   copie nom_src nom_dest
*/
#define _FILE_OFFSET_BITS 64
#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>  /* open() */
#include <errno.h>  /* perror() */
#include <stdio.h>  /* fprintf(), fflush() */
#include <stdlib.h> /* malloc() */
#include <unistd.h> /* exit(), read() */
#ifndef BLOCK_SIZE
#define BLOCK_SIZE (16*1024)
#endif
/* 7000*16K=114MB */
#ifndef BLOCK_NUMBER
#define BLOCK_NUMBER (7000)
#endif
#define BUFFER_SIZE (BLOCK_SIZE*BLOCK_NUMBER)
int main (int argc, char *argv[]) {
  int fd_in, fd_out, i, j,
     t, u, v; /* taille du dernier bloc */
  char *buffer;
  if (argc<3) {
    printf("Erreur d'argument\n\
 Spécifier un nom de fichier source et un nom de fichier
destination.\n");
    exit(EXIT_FAILURE);
  }
  fd_in=open(argv[1], O_RDONLY);
```

```
  if (fd_in==-1) {
    perror("Erreur à l'ouverture de la source");
    exit(EXIT_FAILURE);
  }
  fd_out=open(argv[2], O_WRONLY|O_CREAT);
  if (fd_out==-1) {
    perror("Erreur à l'ouverture de la destination");
    exit(EXIT_FAILURE);
  }
  buffer=malloc(BUFFER_SIZE);
  if (buffer==NULL) {
    perror("Erreur de malloc() ");
    exit(EXIT_FAILURE);
  }
  while(1) {
/* remplissage du buffer */
    i=0;
    do {
      t=read(fd_in, buffer+(i*BLOCK_SIZE), BLOCK_SIZE);
      if (t<0) {
        perror("Erreur de lecture");
        exit(EXIT_FAILURE);
      }
    } while ((t==BLOCK_SIZE)
        && (++i<BLOCK_NUMBER));
       /* i n'est incrémenté que si (t==BLOCK_SIZE) */
/* vidange */
    j=0;
    do {
      /* détermine la taille du bloc à écrire */
      if (j==i) { /* fin de la boucle interne */
        if (t==0)
          exit(EXIT_SUCCESS);
        else {
          u=t;
          t=BLOCK_SIZE; /* force la sortie de cette boucle */
/* t=0 serait plus radical (entrainant
 la sortie directe du programme à la prochaine itération)
 mais man read dit que ce n'est pas nécessairement
 la fin d'un fichier si read retourne un bloc plus petit que
demandé. */
        }
      }
      else
        u=BLOCK_SIZE;
      v=write(fd_out, buffer+(j*BLOCK_SIZE), u);
      if (u!=v) {
        perror("Erreur d'écriture");
        exit(EXIT_FAILURE);
      }
      j++;
    }
    while ((j<i) || (t!=BLOCK_SIZE));
     /* engendre un "off-by-one" s'il reste un morceau */
  }
}
```

Avec un tel programme, on se dit que ça devrait aller. En fait, pas du tout, c'est même la catastrophe. Le temps total d'exécution est 1min.31s, le résultat est donc défavorable. Mais surtout, la *figure 5c* montre que le système déplace des blocs en provenance et vers le swap ! Ce qui explique une partie du temps perdu. Le programme alloue juste les deux tiers de la mémoire disponible, ce devait être suffisant pour le reste mais Linux n'aime vraiment pas que l'on se mêle de ses tampons. Le recours au swap a un effet marqué sur le temps de réponse du système : sur la *figure 5b*, `vmstat` « saute » quelques secondes, comme le montrent des pics d'écriture que le disque ne peut fournir (indiqués par des astérisques, expliqués par l'accumulation du trafic sur deux ou trois secondes). Le temps *idle* montre aussi des pics peu encourageants. Ce n'est donc par la bonne direction à prendre.



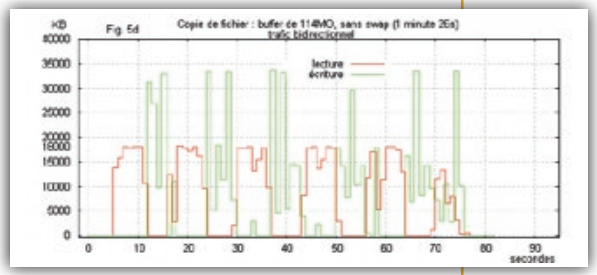


3.3) Je suis root chez moi, après tout !

Comment empêcher Linux de swapper ? J'ai tenté de lui enlever son swap chéri pour voir.

```
$ swapoff -a
```

et le temps écoulé descend un peu à 1:26.91s. Pourtant ça ne l'empêche pas de « perdre les pédales » comme le montrent encore les pics de la *figure 5d*, impossibles à atteindre en réalité.



Ce n'est donc pas la bonne voie, alors

```
$ swapon -a
```

3.4) Un peu plus en douceur ?

Réduction du tampon à 64Mo, moins de la moitié de la mémoire installée. Le temps d'exécution total reste à 1min. 26s. Le kernel swappe toujours un petit peu mais `vmstat` n'a apparemment plus le hoquet. Linux semble continuer à lire le disque « au compte-gouttes » pendant qu'il écrit, ce qui ralentit un peu l'ensemble.

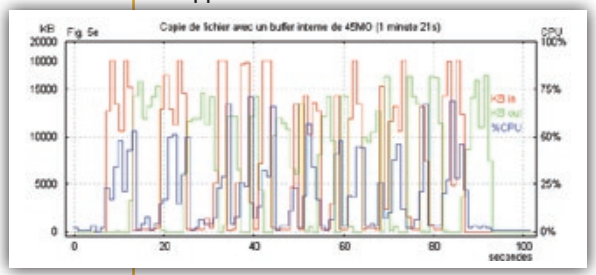
3.5) Il faut lui dire avec des fleurs, alors ?

Je me suis peut-être trompé sur la variable à synchroniser avec le noyau récalcitrant.

Puisque Linux semble décidé à vouloir faire papillonner la tête du disque dur, c'est peut-être avec cela qu'il faut compter. L'alternance principale durant 5s et chaque seconde permettant de transférer 18Mo, chaque demi-période (lecture ou écriture) va contenir 18×5/2=45Mo. J'ai donc tenté un essai avec 2746 blocs de 16Ko.

```
$ /usr/bin/time ./copie tetouatator.raw t.raw
0.06user 14.51system 1:21.77elapsed 17%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (75major+10992minor)pagefaults 0swaps
```

C'est un peu plus rapide de 5s mais le processeur a des pics d'activité très importants (voir la tracé de l'occupation CPU sur la *figure 5e*) et il a quand même swappé 2Mo...



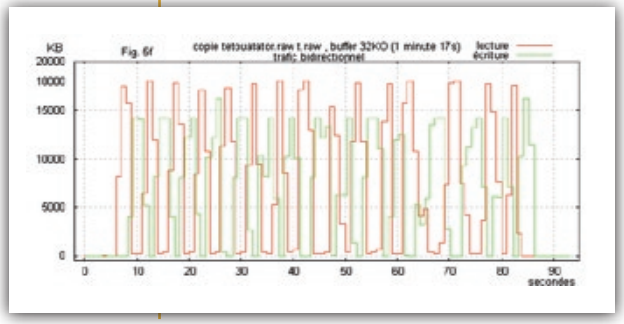
3.6) Retour à la case départ

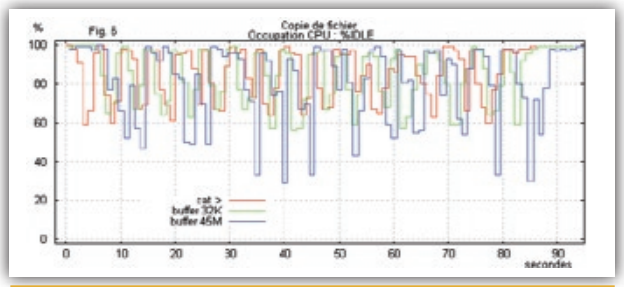
Il semble bien que l'impression de départ soit mauvaise, puisque le temps d'exécution diminue avec la taille du tampon. Un dernier essai s'impose donc avec une taille beaucoup plus réduite, 2 blocs de 16Ko feront l'affaire.

```
$ gcc -DBLOCK_NUMBER=2 -Wall -g -o copie copie.c
$ /usr/bin/time copie tetouatator.raw t.raw
0.09user 11.07system 1:17.43elapsed 14%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (76major+15minor)pagefaults 0swaps
```

Aucune activité de swap n'est à déplorer et nous voici revenus dans les 77s de temps d'exécution. La *figure 5f* ressemble un peu à la *figure 5a* mais avec une fréquence d'alternance un peu plus rapide et moins désordonnée.

La *figure 6* compare le temps CPU disponible selon le type d'approche. L'utilisation d'un grand buffer, en plus d'allonger l'exécution, entraîne des pics d'activité qui peuvent être préjudiciables pour certaines applications.





NOTE

Le noyau Linux s'occupe tout seul de déterminer la taille des transferts avec le disque dur, il est donc inutile d'utiliser un grand tampon de copie. Linux alloue un maximum de place pour ces derniers, il vaut donc mieux garder la taille de nos programmes aussi petite que possible si on ne veut pas que le système se mette à swapper.

4) Influence de la charge

Un troisième élément, et non des moindres, vient alors s'ajouter : il faut bien que notre programme-filtre remplisse une fonction. Pour l'instant, pas besoin d'une utilité particulière, il faut juste voir comment le système se comporte quand on ajoute des instructions de traitement.

4.1) Un filtre simple

La « charge » doit répondre à quelques contraintes : elle doit être facilement et finement réglable, très facile à réaliser, proche d'une utilisation réelle et solliciter uniquement le processeur pour ne pas faire intervenir d'autres paramètres. En plus, le compilateur ne doit pas pouvoir l'« optimiser » pour éviter que le programme n'ait aucun effet. La solution retenue est d'effectuer un XOR sur chaque octet du buffer, ce qui fait en plus travailler le système de mémoire cache intégré. Le programme `filtre_simple.c` est une modification du source précédent, à la différence que le corps de la boucle principale a la forme suivante :

```
  while ( (t = read(fd_in, buffer, BUFFER_SIZE)) > 0 ) {
/* ne fait absolument rien mais grille du temps CPU */
    for (j=0; j<charge; j++)
      for (i=0; i<t; i++)
        buffer[i]^=j;
    write(fd_out, buffer, t);
  }
```

Chaque fois que la boucle interne est exécutée, tout le buffer d'entrée est balayé et chaque octet est légèrement modifié. Le XOR est effectué avec le compteur de boucle externe pour empêcher qu'un compilateur trop malin ne simplifie le code. En fait, ce n'est pas le XOR qui prend du temps, mais l'accès à la mémoire et le découpage en octets (cela nécessite des conversions internes sur les processeurs PII et supérieurs). La « charge » est le nombre de passes et on peut la définir au démarrage du programme pour éviter de le recompiler à chaque fois. On peut remarquer que si la charge est une puissance de deux (et supérieure à deux), la sortie est identique à l'entrée (exercice pour les débutants en programmation : expliquez pourquoi). Pour commencer, un premier essai est effectué avec une charge de 4. Le temps d'exécution moyen est de 113s ou plus exactement :

```
$ rm -f t.raw && sync && /usr/bin/time -f '%e %U %S %P \
 ./filtre_simple tetouatator.raw t.raw
```

résultats de 5 exécutions consécutives :

```
total  user   sys  CPU
113.98 59.46 10.90 61%
111.99 58.33 11.42 62%
113.57 59.44 9.89  61%
113.12 58.97 10.68 61%
113.34 59.24 10.25 61%
```

Si le temps *user* est de presque 60s pour 4 passes, alors chaque passe occupe environ 15s de temps CPU (soit, rapporté au fichier entier, environ 41 millions de XOR par seconde). On pourrait affiner la durée de chaque passe en sautant des octets, mais la bande passante de la mémoire cache atteindrait sa limite. La *figure 7a* montre les débits en lecture et en écriture. Pour une fois, la lecture est plus « lente » que l'écriture, ce qui est un premier signe qu'il se passe quelque chose de différent. Les alternances sont bien régulières avec une période de 5s. L'écriture atteint des pics de 16Mo/s, signifiant que le noyau fonctionne à fond.

Mais le plus intéressant est sur la *figure 7b*, qui montre un détail de l'activité en lecture et en écriture. Essentiellement, le processeur est au repos lors de l'écriture et hyperactif lors de la lecture, où le programme occupe environ 60% de la machine et le noyau 40%.

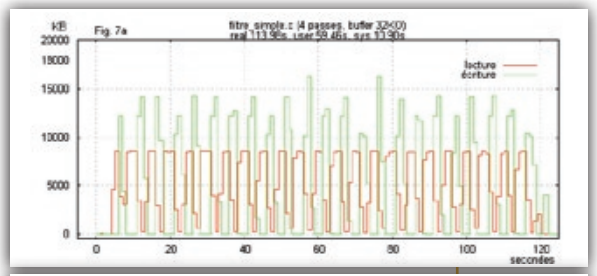
La raison est simple : le kernel retourne les données de lecture immédiatement au programme mais il attend de remplir son buffer ou qu'un commit automatique se déclenche pour écrire. Le programme utilisateur, en particulier sa charge, est donc dépendant de la lecture et n'effectue rien pendant que le noyau vide son cache d'écriture. Voici un nouvel axe d'amélioration !

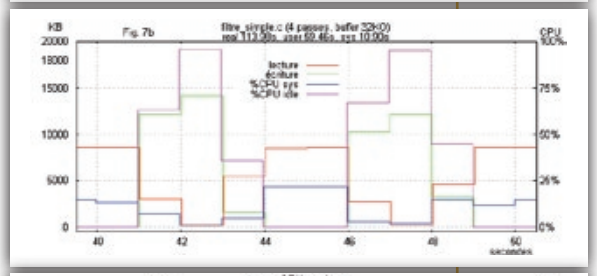
La *figure 7c* synthétise ce comportement avec, d'une part, les temps totaux et *user* et, d'autre part, l'occupation du CPU.

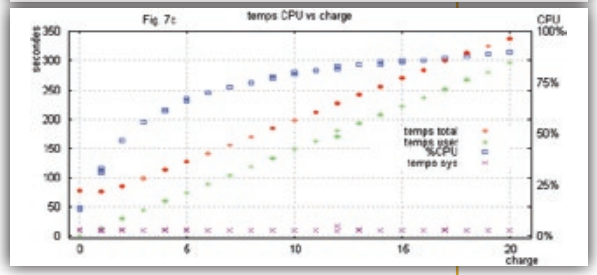
- La charge est totalement linéaire et le temps *user* est affiché par une belle droite qui part de zéro.
- Le temps système est constant (à environ 10s).
- Le temps total dépend de la somme du temps système et du temps *user*, avec un minimum défini par le temps d'accès au disque (mesuré auparavant à 77s). Le tracé du temps total est donc défini par deux droites (une constante et une linéaire).
- La différence entre le temps total et la somme système+*user* est le temps d'écriture, mesuré à environ 40s dans la deuxième partie (ce qui correspond à l'écartement entre les deux droites).
- Le CPU n'est donc jamais totalement occupé comme le montre la belle courbe en `C - 1/x`. L'occupation du processeur ne sera jamais totale à cause des 40 secondes d'écriture.

Au passage, ce graphe valide aussi le principe du système de charge et nous donne une première mesure des caractéristiques du système.

Normalement, le processeur devrait être totalement occupé (*idle* à 0%) avec une charge à 4 pour cet ordinateur (cela varie d'un système à l'autre).







4.2) Même charge mais programme différent

La solution est claire : il faut « découpler » la partie lecture du couple traitement-écriture. Il existe dans les systèmes UNIX des fonctions de gestion des E/S asynchrones au moyen de `select()` ou `poll()` mais cela ne permet pas de découpler totalement les deux parties, car il y en aura toujours une qui devra décider quelle fonction exécuter. Nous avons vu qu'il faut laisser le kernel décider seul de cela, au risque de subir de lourdes pénalités. En fait, mon choix se porte directement sur le modèle « *multithread* » en raison de précédentes expériences « quasi-temps réel » et de la disponibilité des fonctions POSIX. Dans notre cas, cela permet de simplifier et raccourcir le code tout en le gardant portable et parallélisable dans le futur.

Le fichier `filtre_thread.c`, dont les lignes importantes sont données ci-dessous, utilise une paire de *sémaphores* pour gérer la ressource du tampon de mémoire global. Dans le cas le plus simple (avec juste un seul bloc de 16Ko), les sémaphores implémentent un protocole de « poignée de main » (*handshaking*) qui empêche un des *threads* d'utiliser l'unique bloc pendant que l'autre l'accède. Pourquoi alors ne pas utiliser

un *mutex* ? Dans notre cas, la ressource ne comprend pas un, mais de nombreux sous-blocs. De plus, un sémaphore ne fait pas que bloquer une ressource, il indique aussi la taille de cette ressource. Donc, quand il y a de nombreux blocs à gérer, la paire de sémaphores remplace de nombreux mutex (surtout que nous devons les accéder dans un ordre particulier) et le kernel fait en sorte de débloquer un thread en attente de ressources dès que ces dernières deviennent disponibles. Si on avait utilisé `select()`, il aurait fallu coder en plus une partie qui sélectionne que faire à chaque moment.

```
/* fichier filtre_thread.c
>8 snip-snip >8
 compilation :
   gcc -D_REENTRANT -lpthread \
       -o filtre_thread filtre_thread.c
>8 snip-snip >8
#include <pthread.h>   /* threads */
#include <semaphore.h> /* semaphores */
/* permet de moduler la charge CPU */
#ifndef CHARGE
#define CHARGE (4)
#endif
#ifndef BLOCK_COUNT
#define BLOCK_COUNT (16)
#endif
#ifndef BLOCK_SIZE
#define BLOCK_SIZE (16*1024)
#endif
#define BUFFER_SIZE (BLOCK_COUNT*BLOCK_SIZE)
char *buffer;
int charge = CHARGE, fd_in, fd_out;
ssize_t t = 0;
ssize_t byte_count[BLOCK_COUNT];
sem_t sem_lu, sem_ecrit;
pthread_t thr_lecture;
/* Cette partie du code se charge de remplir
*buffer avec des
  blocs, tant qu'il y reste de la place et des
octets à lire : */
void * thread_lecture (void * arg) {
  ssize_t t;
  int index=0, index_block=0;
  do {
    sem_wait(&sem_ecrit);
    t = read(fd_in, &buffer[index_block],
BLOCK_SIZE);
    byte_count[index] = t;
    if (t < 0)
      perror("Erreur de lecture");
    else {
      /* renvoie le résultat et incrémente les
compteurs locaux */
      index_block += BLOCK_SIZE;
      if (++index >= BLOCK_COUNT)
        index = index_block = 0;
    }
    sem_post(&sem_lu);
  } while (t > 0);

  /* mort naturelle sans conséquence : */
  return NULL;
}
/* En plus d'effectuer toute l'initialisation,
le main() contient l'autre
 partie du programme qui effectue le "vrai
travail" et écrit le résultat : */
int main (int argc, char *argv[]) {
  unsigned long int i;
  int index=0, index_block=0, j;
>8 snip-snip >8
  /* initialisation des sémaphores */
  sem_init(&sem_ecrit, 0, BLOCK_COUNT);
  sem_init(&sem_lu, 0, 0);
  /* création du thread de lecture */
  if (pthread_create (&thr_lecture, NULL, thread_lecture, NULL)) {
    perror("Erreur à la création du thread de lecture");
    exit(EXIT_FAILURE);
  }
  /* boucle d'attente-xoriture-écriture */
  while(1) {
    sem_wait(&sem_lu);
    t = byte_count[index];
    if (t > 0) {
      /* charge le CPU */
      for (j = 0; j < charge; j++)
        for (i = 0; i < t; i++)
          buffer[i+index_block] ^= j;
      write(fd_out, &buffer[index_block], t);
      index_block += BLOCK_SIZE;
      if (++index >= BLOCK_COUNT)
        index = index_block = 0;
      sem_post(&sem_ecrit);
    }
    else {
      pthread_join(thr_lecture, NULL);
      exit(EXIT_SUCCESS);
    }
  }
}
```

Avec le système des doubles sémaphores, le codage est relativement simple (même si l'expérience aide toujours et je vous invite à lire les pages `man` de toutes les fonctions nouvelles pour vous). La paire fonctionne un peu comme des vases communicants et si on regarde la succession des valeurs des sémaphores, on s'aperçoit que leur somme est toujours égale (à une unité près) au nombre total de blocs dans le tampon. Après un peu de débuggage pour enlever les quelques bêtises de syntaxe qui m'étaient passées sous le nez, je tente un premier essai :

```
$ gcc -D_REENTRANT -lpthread -o filtre_thread filtre_thread.c
$ /usr/bin/time -f '%e %U %S %P' ./filtre_thread tetouatator.raw t.raw
129.99 72.48 14.05 66%
```

Aucune activité de swap n'est détectée, mais le processeur reste inactif durant les périodes d'écriture. Pire : le temps total d'exécution est bien supérieur aux 113s mesurées auparavant.

4.3) Le retour du gros tampon

En fait, la taille du tampon apparaît bien trop petite. Sur les informations retournées par `vmstat` sur la *figure 8a*, on voit que le kernel écrit des quantités de 20 à 25Mo à chaque fois et le petit tampon de 256Ko ne suffit pas du tout. J'augmente donc le nombre de blocs et le résultat est net :

```
95.26 72.14 15.22 91%
```

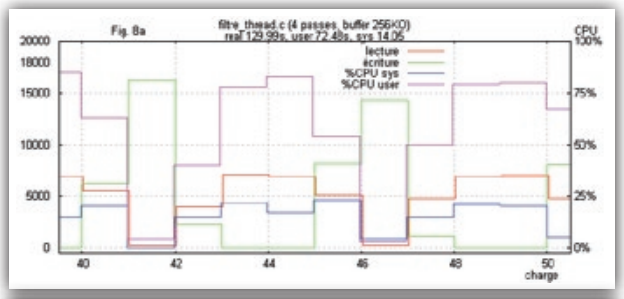
Les 113s sont oubliées ! Mais c'est encore loin des 77s chronométrés sans charge et les rafales d'écriture bloquent encore sporadiquement l'exécution du programme. Il faut encore augmenter le nombre de blocs et le prochain essai en compte 2048, soit 32Mo au total :

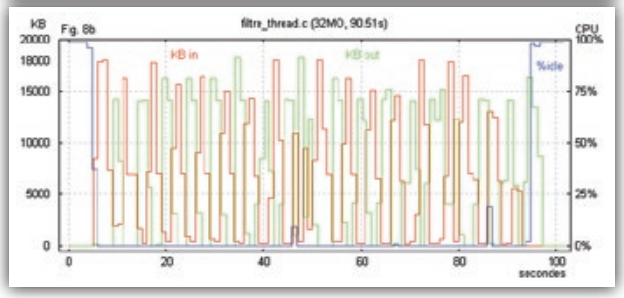
```
90.51 71.37 16.34 96%
```

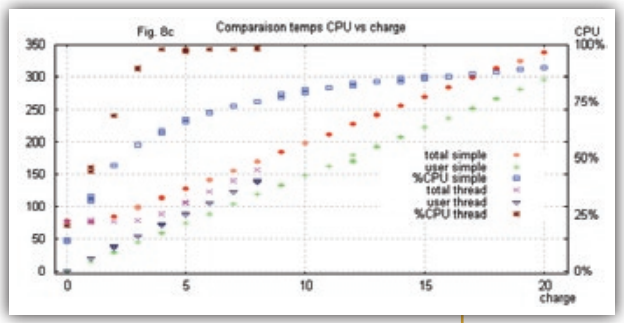
On a encore gagné 5s mais on ne peut pas faire mieux, car il n'y a presque plus de temps CPU libre. De plus, une légère modification du code de charge a un peu augmenté le temps d'exécution de chaque passe. Cela pourrait changer en jouant sur les directives d'optimisation du compilateur, mais le programme est maintenant *CPU bound* : le total des 71s de traitement plus les 10s du système est plus long que 77s sans charge. La *figure 8b* montre d'ailleurs que le temps *idle* est presque tout le temps nul. Grâce au script suivant, on peut tracer le profil comparatif de l'activité, qui est présenté sur la *figure 8c*.

```
for i in $(seq 8 -1 0)
do
  echo $i:
  for j in 1 2 3
  do
    rm -f /common/t.raw
    sync
    echo $i $(/usr/bin/time -f '%e %U %S %P' \
      ./filtre_thread tetouatator.raw t.raw $i 2>&1 ) >> thread.out
  done
done
```

On observe la légère différence de pente pour le temps *user* due à une petite modification du code (une simple histoire de mode d'adressage non optimisé). Mais on voit surtout que la pente de l'occupation du CPU est beaucoup plus raide et l'augmentation du temps total se produit sous une charge plus élevée. L'amélioration est donc significative pour des charges moyennes, telles que le traitement ou la compression/décompression de son ou d'images. L'effet est moins marqué pour les charges très fortes (bien qu'un gain de quelques dizaines de secondes soit toujours bienvenu) et la limitation de la bande passante du disque dur rend l'utilisation de threads trop lourde s'il faut faire moins de deux passes. Mais ces conclusions ne sont pas définitives car elles dépendent de l'ordinateur qui fait tourner le programme. Une accélération du processeur ou du disque dur changerait le bilan. Une chose est sûre : sans recourir à `nice`, la lecture d'un fichier MP3 simultanément à l'exécution du programme multithreadé donne une écoute très saccadée.







5) Application directe

En parlant de problèmes de lecture audio, j'avais tenté de lire directement des fichiers compressés par `bzip2` (tels ceux fournis dans GLMF numéro 73 de juin 2005) au moyen de commandes du type :

```
bzip2 -d -c -k fichier.raw.bz2 | \
  /usr/bin/nice --10 sox -t sw -r 44100 -c 2 - -t ossdsp /dev/dsp
```

Le résultat était saccadé pour une autre raison. `bzip2` travaille par à-coups avec de grands buffers (plusieurs Mo) alors que `sox` n'utilise au mieux que quelques dizaines de Ko. Lorsque `bzip2` a fini d'écrire le bloc courant (qui s'écoule lentement au travers de sox, au rythme de 176,4Ko/s), il doit lire la suite du fichier compressé puis le décompresser, ce qui prend beaucoup de temps. Le petit tampon de `sox` et du kernel a déjà expiré et la lecture est momentanément interrompue...

En modifiant le programme multithreadé précédent, on résout facilement le problème. Pour cela, il suffit de faire quelques petits aménagements :

- Permettre de lire et écrire sur `stdin`/`stdout`. C'est l'affaire de quelques lignes supplémentaires. Comme pour `sox`, j'ai choisi « `-` » pour signifier qu'il s'agit de l'entrée/sortie standard.
- Enlever le code de charge fictive.
- Permettre de spécifier la taille totale du tampon sur la ligne de commande. Il faut faire quelques petits aménagements car toutes les tailles étaient en constantes.
- Ne pas oublier de fermer les fichiers en sortant ! Sinon `sox` reste figé alors que le programme est déjà terminé et sorti de la mémoire.

Le résultat s'appelle `pb.c` pour « *Pipe Buffer* ». Les clics ont tous disparu (alors qu'il y en avait une douzaine par minute). J'en ai profité pour faire un joli script qui est logé dans `/usr/bin/playbz2` :

```
#!/bin/bash
if [ -z "$1" ]
then
  echo "Missing argument : file name"
else
```

```
  for FILENAME in "$@"
  do
    bzip2 -d -c -k $FILENAME | pb - - 1024 | \
    /usr/bin/nice --10 sox -t sw -r 44100 -c 2 - -t ossdsp /dev/dsp
  done
fi
```

`nice --10` est juste une précaution pour éviter de saccader la lecture quand d'autres programmes « intensifs » comme ceux étudiés plus haut sont lancés à pleine vitesse.

6) Améliorations possibles

▶ J'ai choisi de découper le programme en deux, car il semblait que la sérialisation de la lecture avec le traitement était le frein central. Il se peut que des systèmes éprouvent aussi des difficultés en écriture, par exemple si la mémoire est très petite ou si le noyau est trop primitif. Dans ce cas, il suffit de créer un autre thread pour l'écriture (et croiser les doigts pour qu'une forme de threads soit disponible si on ne veut pas travailler directement au niveau des interruptions).

▶ La taille du tampon a été déterminée expérimentalement pour une plate-forme particulière et ne conviendrait probablement pas si le rapport des vitesses entre le disque et le processeur change.Au lieu de le déterminer à la main pour chaque machine, il faudrait créer un algorithme *adaptatif* qui augmenterait la taille du tampon au besoin. Cet algorithme est trop compliqué pour le présent article. De plus, la plupart des applications qui utiliseront le code multithreadé seront adaptées au cas par cas et probablement même pour une machine particulière et des charges spécifiques.

▶ Nous avons étudié des cas où l'information ne subissait pas de modification de bande passante. Dans le cas d'un compresseur ou d'un décompresseur, il faut tenir compte du fait que la quantité de données lues n'est pas la même qu'en écriture. Le système de sémaphores et de gestion des blocs devrait encore fonctionner mais une vérification formelle s'impose.

7) Conclusion

Méthodologie

Cet article a présenté un cas simple d'« optimalisation » : commencer par examiner les opérations les plus simples et les plus rapides puis ajouter progressivement des éléments en essayant de quantifier et réduire leur impact sur les performances. Optimiser un programme revient classiquement à le regarder et remplacer des blocs jugés inefficaces par d'autres que l'on estime meilleurs. Par contre, reconstruire un programme en partant de sa fonction la plus fondamentale est une stratégie d'optimisation qui empêche de se faire piéger dans des conjectures (qui sont souvent fausses et ralentissent le programme au lieu de l'accélérer). Bien sûr, il ne faut jamais oublier de chronométrer le résultat pour vérifier que les modifications ont eu un effet et qu'il est positif. Nous avons vu que c'est aussi très instructif !

Linux

L'utilisation de buffers noyau pour `read()` et `write()` peut devenir une catastrophe à cause de la copie entre l'espace noyau et l'espace *user*. Ce serait acceptable, s'il ne fallait pas compter avec son envie irrépressible de swapper tout ce qu'il peut pour faire de la place pour ses tampons en mémoire, parfois inutilement immenses. Et ce n'est que la partie émergée du problème :

▶ La compatibilité `POSIX` et la base logicielle existante ne permettent pas d'éviter cette duplication qui, dans certains cas, n'a aucun sens : on ne peut pas faire un `read()` de plus de la moitié de la mémoire vive, au risque de faire swapper nos propres données *user* pour libérer de la place pour le buffer du kernel ! Et même avec un simple tampon utilisant seulement le quart de la mémoire, il trouve encore le moyen de swapper quelques morceaux (c'est à se demander s'il faut vraiment activer le swap).

▶ L'option `READ_O_DIRECT` ne fonctionne pas sur les fichiers normaux, du moins sur ma version de kernel.Affaire à suivre de près !

▶ `mmap()` est un outil indispensable, mais en userland, c'est « la cause de tous les maux » (par rapport à cela, on peut mettre des `goto` partout les yeux fermés). En plus d'être synchrone, sérialisant, et surtout avec une latence imprévisible, il faut l'utiliser avec une prudence extrême et prévoir beaucoup de code pour éviter de faire des bêtises : le mieux est l'ennemi du bien ! (Ce sont encore ces réminiscences du *benchmarking* de BLAST qui frappent encore, des zigotos n'avaient pas imaginé qu'il faudrait utiliser des bibliothèques de protéines de plus de 1,8GO sur une plate-forme Linux/ia32)

Je ne vais pas paraître très original, mais vu le comportement du kernel considéré ici, s'il fallait faire des traitements informatiques très lourds (par exemple, traiter des images gigantesques comme celles issues de grands téléscopes ou des bases de données d'entreprises), le premier moyen pour les accélérer serait d'augmenter considérablement la quantité de mémoire. Certaines distributions GNU/Linux ne veulent plus lancer leur installeur s'il n'y a pas au moins 128Mo de RAM. Maintenant je sais que ce n'est pas dû uniquement à la lourdeur de GNOME ou KDE installés par défaut ;-)

Et les mesures montrent qu'on ne peut pas faire mieux, en userland, qu'utiliser `read()` et `write()` sur des blocs de taille inférieure à celle de la mémoire cache. Mais à part ces remarques gratuites, mon vieux kernel 2.4.19pre7 s'en sort très bien dans une grande variété de situations, sans avoir à lui apprendre son travail. J'ai eu beau ressortir mes vieilles « techniques MS-DOS », il m'a longtemps tenu en échec. D'où l'importance cruciale de mesurer avant d'optimiser. Et encore, ce kernel a plus de trois ans d'âge ! Son amélioration a continué sans relâche et ce n'est que la perspective de devoir reconstruire mon LFS à partir de zéro qui me retient de passer à la branche 2.6.

Parallélisme :

Autre chose à retenir :

Un programme qui utilise `select()` ou `poll()` n'est pas parallélisable alors qu'en utilisant des threads, :

- On laisse le kernel travailler proprement (les `read()` et `write()` bloquent uniquement le thread qui les appelle, laissant les autres threads travailler quand le kernel a terminé ses devoirs).
- C'est plus flexible et extensible (même si potentiellement tout aussi spaghetti que `poll()`/`select()`),
- On peut mieux contrôler les aspects temporels (temps de réponse, etc.).
- Les sémaphores sont des outils très puissants qui peuvent simplifier beaucoup de codes à base de mutex.
- Le découpage en tâches simples permet de passer facilement en multi-CPU, NUMA, MPI, cluster... L'avenir est parallèle !

Pour la suite :

Le code multithread va servir de base pour les prochains articles sur la compression de données. Il est facilement extensible et peut même être adapté pour faire un lecteur ou un enregistreur de sons (ou les deux).

Pour expérimenter avec de nouvelles « charges », il suffit simplement de remplacer le code dans la boucle principale du `main()`. Des programmes similaires avec plus de threads pourront simuler un environnement multipiste.

Mais que l'on utilise les threads ou non, nos algorithmes de traitement vont traiter des blocs de taille variable au lieu de prendre une paire d'échantillons à la fois.

Ces algorithmes peuvent être inclus dans le code examiné ici ou dans un code plus simple sans threads et la question des détails techniques de bas niveau ne se pose plus : nous pouvons maintenant nous concentrer sur les traitements eux-mêmes.

LIEN

Miroir des sources : http://ygdes.com

Yann Guidon,

ANNEXE

Pour obtenir les graphes de cet article, il a suffi de quelques utilitaires simples comme `vmstat`, `grep`, `cut`, `awk`, `gnuplot` et de quoi éditer les images.

Collecte des statistiques

Dans un terminal séparé (alt-F1), il suffit de lancer `vmstat` et de rediriger la sortie vers un fichier. `tee` permet de conserver l'affichage en temps réel pour voir comment les choses se présentent sans attendre la fin de la mesure.

```
$ vmstat 1 |tee fichier.v
```

Normalement, `vmstat` ne devrait pas influencer les mesures, ou imperceptiblement, puisque ce programme a été conçu pour profiler le système. Puis, dans un autre terminal (alt-F2), lancer le programme à mesurer. Il faut aussi s'être assuré préalablement que le noyau n'a pas conservé des données partielles des expériences similaires précédentes. Un moyen est de remplir son cache avec des données inutiles, en lisant un fichier au moins deux fois plus gros que la mémoire de l'ordinateur. `dd if=/dev/zero of=/dev/null bs=1024K count=(2xtaille_RAM)` devrait faire l'affaire. De même, le fichier destination (quand il y en a) ne doit pas déjà exister, afin de forcer la réallocation des inodes (autant pousser le réalisme dans les moindres détails).

Traitement des données

Deux choses à savoir sur `vmstat` : la première, indiquée dans la page man, est que la première ligne est la moyenne des paramètres **depuis le démarrage de l'ordinateur**, il faut donc l'enlever. La deuxième, qui n'est pas indiquée, est que le programme peut « rater » ou « sauter » des lignes si la charge CPU est beaucoup trop élevée, il faut alors considérer le résultat comme non fiable. Ensuite il faut un peu nettoyer le fichier car il contient des lignes de rappel du nom des colonnes, utiles pour nous les humains mais indésirables pour nos autres outils.

```
$ grep -v sy fichier.v > fichier.vms
```

Cela enlève les lignes contenant la chaîne `sy`, commune aux deux lignes ajoutées automatiquement. Il faut ensuite isoler les colonnes de nombres : c'est `cut` qui va s'en charger :

```
$ cut -b 51-56 fichier.vms (retourne la colonne "bo")
```

Il faut faire attention car les colonnes peuvent se décaler lorsque des nombres dépassent la capacité que `vmstat` leur alloue (comme lors de swap in/out par exemple). Un petit contrôle visuel préliminaire s'impose donc. Pour éviter ce désagrément, il aurait fallu un script en `sed` pour enlever les espaces superflus, par exemple : `sed 's/[ ][ ]/ /'`. J'aurais ainsi pu utiliser un autre mode de `cut` (champs séparés par des tabulations), mais j'ai choisi la route directe, les commandes commençant à devenir nombreuses et illisibles :-) En plus, cela aurait détruit l'alignement des nombres. Ensuite il faut continuer la préparation car `gnuplot` a un affichage inadapté à mes mesures en mode « ligne ». L'astuce pour faire les marches d'escalier est de dupliquer chaque point mais en décalant d'un cran l'indice. On obtient donc une ligne verticale et une ligne horizontale. C'est `awk` qui va s'en charger, puisque je n'ai pas réussi à le faire avec `sed`...

```
$ cut -b 51-56 fichier.vms | \
  awk '{print NR-1 $0; print NR $0}' > fichier.bo.dat
```

Pour décoder cet uniligne, il faut savoir que `NR` est la variable du numéro de ligne courante et `$0` est son contenu. Pour chaque ligne (les accolades étant un groupe de commande sans condition), `print` affiche donc deux fois la ligne avec les deux indices différents désirés. Comme il faut découper plusieurs colonnes et créer plusieurs fichiers, toutes les commandes sont dans un *script shell*. Une variable donne le nom de base du fichier à traiter pour éviter de le modifier à chaque fois. Bien sûr, il aurait été possible d'effectuer ces opérations avec un seul langage (j'entends d'ici des Mongueurs uniligner), mais je suis pour la diversité culturelle et la fainéantise unixienne :-)

Mise en forme

Pour l'instant, je n'ai pas trouvé mieux que `gnuplot` (qui, rappelons-le, n'est pas un projet `GNU`). Il faut un peu tâtonner au début, mais on arrive rapidement à un résultat montrable. Voici par exemple le code d'une des illustrations :

```
# LECTURE+ECRITURE NORMALE
set key 83,21700
set xrange [-2:95]
set yrange [-500:20000]
set ylabel "KB" 3,25
set label "Fig. 5a" at 10,21000 c
set label "18000" at -6,18000 c
set label "trafic bidirectionnel" at 45,20500 c
set title "cat tetouatator.raw > t2.raw (1 minute 17s)"
plot "rw.bi.dat" t "lecture" with lines,\
     "rw.bo.dat" using ($1-0.3):($2) t "écriture" with lines
unset label
pause -1
```

Pour améliorer la lisibilité des lignes (qui ont tendance à se recouvrir les unes les autres), j'ai décalé certains tracés. C'est la fonction des expressions du type `($1-0.3):($2)` : on ajoute 0,3 au X et on laisse le Y tel quel.

Finition

Capture d'écran, découpage, correction des couleurs. Tout le reste a déjà été fait par les programmes précédents. A vous de jouer ! Une bonne illustration vaut mieux que des mots et vos présentations n'en seront que plus attrayantes, lisibles et crédibles !