CCIN2P3

Optimisation des entrées/sorties pour les systèmes POSIX

Loïc Tortay (& Éric Legay),

École Informatique IN2P3/Groupe Calcul,

9 février 2012

cnrs

Irfu cea saclay

Hiérarchie des mémoires (ordres de grandeur)

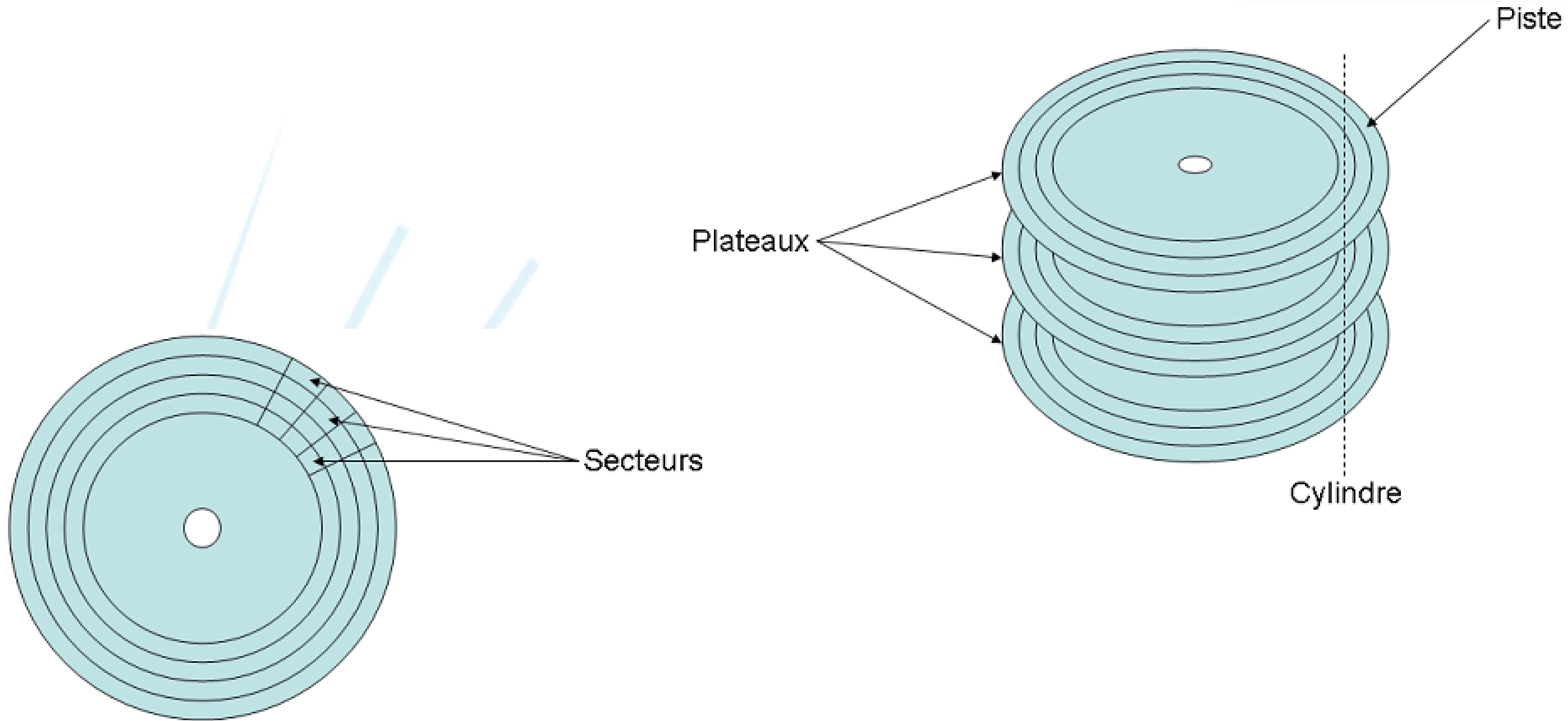
Type	Latence	Débit (utile)
Cache processeur (L1)	~nanoseconde	≫ Go/s
Cache L2, puis L3	< 10 nanosecondes	≫ Go/s
RAM	*n**10 nanosecondes	N Go/s (~12 Go/s DDR3-1600)
PCI-Express 3	*< microseconde*	*8 GT/s (1 Go/s)/lane*
PCI-Express 2	*< microseconde*	*5 GT/s (500 Mo/s)/lane*
Infiniband	*~microseconde*	*1 Go/s/link (QDR)*
Ethernet 1Gb/10Gb	*µ microsecondes*	*1 Go/s (10Gb)*
SAS 6 Gbps		*6 Gbps/channel*
SATA 2/3		*3/6 Gbps*
Flash (SSD)	*m* millisecondes	X * 100 Mo/s, kIOps
Disques durs	*M* millisecondes	100 Mo/s (séq), 80-200 IOps
Bandes	montage & positionnement : **secondes**	100-250 Mo/s (LTO-4/5 & Jaguar/T10K)

Disques durs [1/2]

CCIN2P3

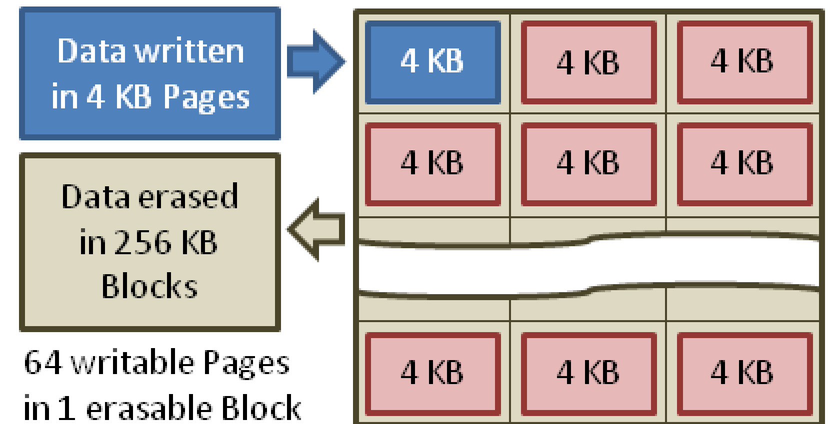
- Élement dominant du stockage
- Adressage CHS, LBA depuis ~15 ans
- Secteurs de 512 octets **utiles** (4096 pour disques *Advanced Format*), lecture & écriture minimale : 1 secteur
- Têtes solidaires et mouvement unique (par cylindre)
- Accès séquentiels faciles, accès aléatoires difficiles : ~0,66 entrée/sortie aléatoire pure par rotation (heuristique)
- Typiquement 1 plateau disques portables, plusieurs autres disques
- A vitesse de rotation égale, disques 2.5" plus rapides que 3.5" (moins de déplacements nécessaires pour accéder aux secteurs)
- File d'attente d'opérations (TCQ/NCQ)

Disques durs (2/2)



SSD (Flash) [1/2]

- Mémoire Flash & SSD disponibles depuis 20 ans, marché de niche jusqu'à il y a ~5 ans (BIOS/Firmware, NVRAM, enregistreur de vol)
- SLC et MLC (+ x3), durée de vie des cellules limitées en nombre de cycles effacement/écriture (SLC : 100k, MLC : ~3k à 10k)
- Organisé en pages (2Ko ou 4Ko), regroupées en blocs (32 à 64 pages/bloc)
- Lecture et écriture avec la granularité d'une page **mais** effacement et ré-écriture avec la granularité d'un bloc



Typical NAND Flash Pages and Blocks

SSD (Flash) [2/2]

CCIN2P3

- Adressage LBA
- Répartition transparente de l'usure (*wear-leveling*)
- *Garbage collecting* pour les pages à effacer
- *Write amplification*
- TRIM : l'OS indique au SSD quelles pages sont "vides" pour qu'il les efface (en tâche de fond) et le note dans ses propres méta-données (Linux 2.6.33+)
- Futur ? NVM Express/NVMHCI : standard pour mémoire Flash connectée directement en PCI-Express (1ère carte à ce standard : *OCZ R5 (16 To), PCI-e 3 x2: 7.2 Go/s seq I/O, 2.5 MIOps*)

Systèmes de Fichiers [1/2]

CCIN2P3

- Organisation logique pour structurer et trouver les données sur les médias de stockage permanent
- De nombreux types existent, parmi lesquels (catégories non exclusives) :
  - *block-based* (ext2/ext3, UFS, ...),
  - *extent-based* (XFS, ext4, NTFS, VxFS, ...)
  - journalisé (en général les *extent-based*, *ext3*, ...)
  - *Copy On Write* (ZFS, WAFL, BtrFS, ...)
- Méta-données : nom des fichiers/répertoires/etc, position des données sur les médias
- Fragmentation : taille des données rarement multiple de la taille de bloc ou extent, petits blocs ou blocs pas pleins
- FS distribué : système de fichiers accessible depuis plusieurs machines

Systèmes de Fichiers [2/2]

- NFS & GlusterFS vs AFS & GPFS & Lustre (formatage + tout ou partie de la sémantique POSIX)
- Cache : voracité du cache Linux, cache générique (système) vs dédié (AFS, GPFS, ZFS, ...)
- Cohérence de cache dans les systèmes de fichiers ditribués : communication réseau entre les machines pour se coordonner/échanger les contenus

Généralités [1/2]

- *I/O wait* : le système attend qu'une ou plusieurs I/O se terminent, les processus concernés sont en général bloqués (visible avec `vmstat` ou `iostat` par exemple)
- Chaque appel système implique un changement de contexte noyau/processus utilisateur, coût en cycles processeurs mais aussi pour les différents niveaux de cache
- Alignement : les médias de stockage ont une taille de secteur/page. Entrées/sorties non alignées ⇒ accéder à 2 secteurs/pages au lieu d'un
- *Read-Modify-Write* : cycle courant pour la mise à jour de données, pertinent à la fois au niveau système de fichiers (bloc ou extent), RAID (*writehole* pour RAID-5/6) et média (secteur/page)
- *Schedulers* d'I/Os : choix de la politique d'allocation des entrées/sorties pour l'ensemble du système ou un périphérique (Linux : `cfq` vs `deadline`, `/sys/block/sdX/queue/scheduler`)

Généralités [2/2]

CCIN2P3

- `ionice` (Linux) : possibilité de donner une priorité variable aux I/Os en fonction de l'activité du système
- Attributs POSIX : *Last access time* mis à jour lors de tout accès (lecture ou écriture) à un fichier/répertoire; lecture ⇒ écriture; désactiver en montant le système de fichier avec `-o noatime`
- Sémantique POSIX (écrivains multiples) : "le dernier qui écrit a raison", seules les parties effectivement modifiées doivent être changées, en particulier si un autre programme/thread modifie une partie des données (avec ou sans chevauchement) ces autres données doivent être préservées (pas de garantie : AFS & NFS, en option : Lustre, garantie : GPFS et la plupart des systèmes de fichiers locaux)
- Fichiers creux (*sparse files*) : fichiers qui occupent moins de place que leur taille apparente (affichée avec `ls`); les parties non allouées d'un fichier sont lues comme des séquences de zéros (SEEK_HOLE/SEEK_DATA)

Fonctions C [1/2]

- Flots d'octets "bufferisés" (ou pas), manipulés avec le type opaque `FILE` (`FILE*`, obtenu avec `fopen`, libéré avec `fclose`)
  - `fread/fwrite` : accès à des vecteurs *d'éléments*, retournent le nombre d'éléments lus/écrits, la norme prévoit que fread/fwrite sont équivalent à des séries de `fgetc/fputc` ; implémentations souvent basées sur `read/write` qui accèdent par défaut à des petits blocs (4 ou 8 Ko, constante `BUFSIZ`)
  - `fscanf/fprintf` : pour les usages habituels avec des chaînes de caractères
  - `fflush` : synchronise vers les disques les données écrites dans le flot
  - `fseek/fseeko` : positionne le pointeur courant dans le flot et `ftell/ftello` donnent la position du pointeur courant dans le flot
  - `flockfile` : obtenir un verrou sur un flot pour sérialiser des entrées/sorties (plusieurs threads lisent/écrivent dans le même flot)

Fonctions C [2/2]

- `setbuf/setvbuf` : modifier la politique de "bufferisation" (par ligne, pas du tout, taille de buffer)
- `feof/ferror` : fonctions de test de fin de flot et d'erreur

- Programmes portables sur toute plateforme avec une bibliothèque standard C (Windows, Unix, ...), avantage majeur : simplicité, mais attention à la gestion des erreurs

Fonctions POSIX (blocs) [1/3]

- Entrées/sorties sur des « fichiers » (fichiers, *sockets*, *pipes*, *ttys*, *devices*, etc.) identifiés par des descripteurs obtenus/libérés avec `open/close`
- Appels systèmes bloquants par défaut mais les données transitent par le cache (écritures asynchrones vers les disques)
  - `read/write` : manipulent (en mémoire) des tableaux d'octets transférés depuis/vers un fichier
  - Variantes : *scatter/gather* `readv/writev` (`IOV_MAX`) & positionnées `pread/pwrite`
  - `lseek` : positionnement dans le fichier, offset en octets
  - `truncate/ftruncate` : changent la taille d'un fichier, éventuellement augmentent la taille apparente du fichier pour en faire un fichier creux
  - `posix_fallocate` : alloue de l'espace à un fichier sans y mettre de données (+/- symétrique de `ftruncate`)

Fonctions POSIX (blocs) [2/3]

- Linux : `fallocate` peut aussi "faire des trous" (*punch holes*)
- `stat/lstat/fstat` : lire les méta-données POSIX d'un objet
- `fsync/fdatasync` : demande au système de transférer sur disque les modifications apportées à un fichier (à partir du cache)

- Verrous : POSIX définit uniquement des verrous coopératifs (*advisory*), les autres processus ne sont pas empêchés de lire ou écrire dans un fichier verrouillé
  - `lockf` : verrouillage d'une zone relative à la position courante
  - `fcntl(..., F_SETLK, ...)` : verrouillage d'une zone quelconque
- Linux : Verrous stricts disponibles en montant un système de fichiers avec `-o mand` (réputé non-fiable, cf. man page)

Fonctions POSIX (blocs) [3/3]

- Demander au système de rendre les appels systèmes non bloquants : ouvrir avec `O_NONBLOCK/O_NDELAY`, utilisé surtout les FIFO (*named pipes*) ou les sockets Unix
- Linux : `O_ASYNC`, notification asynchrone par signal (`SIGIO` par défaut), pas sur les fichiers (*ttys, sockets, pipes*)
- SUSv4 : `http://www.opengroup.org/onlinepubs/9699919799/`

- Fonctions d'usage courant, avec des avantages (par exemple *scatter/gather*), légèrement plus complexes que `fread/fwrite` mais souvent plus performant et avec plus de possibilités d'optimisation
- Certaines fonctions manquent : `readx/writex` (données non-contigües en mémoire **et** sur disque)

Fonctions POSIX (mmap) [1/2]

- Associe un zone mémoire au contenu d'un fichier (ou *device*, toujours `open/close`)
  - `truncate/ftruncate` (ou similaire) préalable nécessaire pour les fichiers mmap()és accédés en écriture (fichiers inexistants ou agrandis)
  - `mmap/munmap`, contraintes d'alignement sur l'offset (multiple taille de page mémoire)
  - `mlock/munlock` & `mlockall/munlockall` : verrouiller des pages en RAM (pas écrites sur disque en cas de pagination/swap), quantité très limitée sans privilèges
  - `posix_madvise`, similaire à `posix_fadvise`, même *hints* mais limités à la zone mémoire mappée (pas au fichier)
- `mmap` peut aussi servir à allouer de la mémoire (non standard mais omni-présent et très utilisé) : `MAP_ANONYMOUS`

Fonctions POSIX (mmap) [2/2]

CCIN2P3

- Linux (et historiquement) : `madvise`, *hints* non POSIX pour la mémoire allouée :
  - `MADV_MERGEABLE` (noyau avec option KSM, dans RHEL6/F14) indique au système que des pages anonymes (non liées à un fichier) peuvent être fusionnées entre processus (dé-duplication), prévu au départ pour la virtualisation
  - `MADV_HUGETLB` alloue des *huge pages* sans `hugetlbfs`
- `mmap` est (à peu de choses près) la manière dont le noyau accède aux fichiers, difficile de faire plus bas niveau en espace utilisateur
- Efficace mais il est assez facile de se tromper

Fonctions POSIX (AIO) [1/3]

- Entrées/sorties asynchrones vers des fichiers/devices (*seekable devices*)
- Appels systèmes non-bloquants
- *Busy-wait* ou notification par signaux (par défaut `SIGIO`) ou *callback*
  - `struct aiocb` : structure de données utilisées par les fonctions AIO

```
.aio_buf,
.aio_fildes,
.aio_offset,
.aio_nbytes
. ...
```

  - `aio_read/aio_write` : soumet une lecture/écriture
  - `aio_error` : indique si une erreur a été détectée, si la requête n'est pas finie (utile pour *polling/busy-wait*) ou a été annulée
  - `aio_return` : obtient la valeur de retour similaire à `read/write` (octets accédés ou -1)

Fonctions POSIX (AIO) [2/3]

- `aio_cancel` : annulation d'une requête
- `aio_suspend` : bloque en attendant la fin d'une requête parmis celles soumises, un signal ou une temporisation optionelle
- `aio_fsync` : force la synchronisation du cache pour un fichier (y compris I/O non asynchrones)
- `lio_listio` : soumet un ensemble de requêtes, pas nécéssairement sur une seul cible; requêtes non ordonnées et exécutées comme le système le décide;
- `lio_listio` peut être utilisé en mode bloquant (`LIO_WAIT`) ou non-bloquant avec notification par signal ou *callback*
- Le type d'opération (lecture/écriture) est spécifié dans le champ `aio_opcode` de chaque `aiocb` dans la liste fournie

Fonctions POSIX (AIO) [3/3]

Implémenté souvent avec des threads & `pread/pwrite`

- La plupart des systèmes ont une version spécifique de AIO (AIX, Solaris, `libaio` sur Linux, etc.)
- Surtout utilisé par des applications de stockage massivement multi-threadés (Oracle, HPSS, ...)
- Particulièrement pertinent pour pour gérer du stockage à plusieurs niveaux de vitesse (disques, disques *near-line*, *off-line*/bandes)
- Permet de maîtriser le nombre d'I/O en cours (*in flight*)

Fonctions POSIX (réseau) [1/3]

- Pendant des fichiers pour le réseau : sockets
- Typiquement TCP : client et serveur dans un échange actif
  - `socket`
  - serveur : `listen/accept`
  - client : `connect`
  - `read/write` : identique aux fichiers, prévoir la possibilité de lire zéro octets
  - `recv/send` (`recvfrom/sendto`) : permettent de spécifier des paramètres supplémentaires sur la livraison des messages (OOB)
  - `recvmsg/sendmsg` : similaires a `recv/send` avec du *scatter/gather* (comme `readv/writev`)
- Pas de cache pour le réseau
- Entrées/sorties bloquantes

Fonctions POSIX (réseau) [2/3]

- `select/poll` sont utilisés pour ne pas bloquer et être notifié lorsque des données sont transmises ou vont pouvoir être transmises.
- Linux (et autres) : `sendfile()`, envoie tout ou partie d'un fichier (uniquement) sur un socket (uniquement) en un seul appel système
  - Solaris, FreeBSD, ..., ont aussi une fonction `sendfile()` mais elles sont toutes différentes
- Transferts TCP performants : multi-flux, avec des *grandes fenêtres* (RFC1323) :
  - `setsockopt(..., SO_{RCV,SND}BUF, ...)`
- BDP : *Bandwith Delay Product* (unités), pouvoir stocker en mémoire assez de données pour ne pas faire attendre le réseau
- Taille des fenêtres : 2 * *BDP*
- Configuration système adéquate nécessaire (`sysctl` sur Linux, **sauf** si Autotune)

Fonctions POSIX (réseau) [3/3]

- Linux *récents* (2.6.17+, RHEL5+) : pas de configuration TCP nécessaire pour les applications (cf. `/proc/sys/net/ipv4/tcp_moderate_recvubf`) qui recoivent des données (expéditeur depuis plus longtemps)
- Utiliser `setsockopt(..., SO_RCVBUF, ...)` désactive Autotune
- De manière générale ne pas le désactiver si on travaille uniquement sur Linux
- `http://www.psc.edu/networking/projects/tcptune`

Agir sur le cache et son utilisation [1/4]

- I/O Synchrones : demande au système de ne pas attendre avant d'écrire le contenu du cache sur disque, option de `open` ou défini avec `fcntl`
- POSIX :
  - `O_SYNC`, tout synchrone (y compris mise à jour des méta-données)
  - `O_DSYNC`, écriture des données synchrone (pas les méta-données)
  - `O_RSYNC`, lectures attendent que les autres opérations sur le fichier soient sur disque (données et méta-données)
- Linux : `O_RSYNC = O_SYNC`, `O_DSYNC` différent depuis *peu*

- Synchroniser tous les buffers des écritures vers les disques **du système** pas juste du programme en cours ou des fichiers ouverts : `sync()` (POSIX, équivalent à la commande `sync`)
- Linux : `bdflush()` permet un contrôle plus fin, mais nécessite des privilèges

Agir sur le cache et son utilisation [2/4]

- Direct I/O (non POSIX, mais présent sur la plupart des Unix) : option `O_DIRECT`, `O_DIRECTIO` pour `open()` mais `directio()` sur Solaris
- Demande au système de ne **pas** mettre les données en cache
- Souvent contraintes d'alignement sur la mémoire utilisé pour les transferts (allouer avec `memalign` ou `posix_memalign`)
- Supporté par ext4, XFS, GPFS, Lustre, mais PAS ext3, AFS, NFS, ZFS
- Si implémenté, les données vont/viennent directement des disques
  - intéressant pour SGBD ou autre application avec cache interne ou pour vérifier l'intégrité des données sur disque après les avoir écrites
- Linux : évite aussi le *buffer cache* pour les devices bloc (sans système de fichiers)

Agir sur le cache et son utilisation [3/4]

- Vider le cache : pas de méthode POSIX 100% fiable (à part `shutdown`)
- Linux : `# echo X > /proc/sys/vm/drop_caches` (1 : *pagecache*, 2 : méta-données, 3 : données & méta-données), ne vide pas les *dirty buffers* (`sync` nécessaire), privilèges requis
- POSIX : **sans** garantie de succès (*hint* au système), exemple `drop-from-cache` basé sur `posix_fadvise(..., POSIX_FADV_DONT_NEED)`, faire de la place dans le cache une fois qu'on a fini d'accéder à un fichier
  - `posix_fadvise(..., POSIX_FADV_WILLNEED)` demande au système de charger tout ou partie d'un fichier en cache
  - `POSIX_FADV_NOREUSE` : **un** accès bientôt puis plus besoin des données
  - `POSIX_FADV_{NORMAL,RANDOM,SEQUENTIAL}` *hints* pour le *prefetch* : rien de spécial, pas de prefetch, un peu de prefetch
  - `SEQUENTIAL` souvent moins efficace que `WILLNEED`
- Un seul hint à la fois, pas de combinaison

Agir sur le cache et son utilisation [4/4]

CCIN2P3

- Fonctions spécifiques ou `ioctl()` pour différents systèmes de fichiers (exemple GPFS avec `gpfs_fcntl()` & `gpfsClearFileCache_t` et/ou `gpfsAccessRange_t`), toujours *hints* uniquement
- Linux : `readahead()` qui ne donne pas un *hint* mais **lance** le *prefetch* d'une partie d'un fichier (cf. `linux-src/mm/filemap.c`)
- Utile aussi pour réduire l'impact des accès aléatoires sur les disques, **sisi** on connait la zone et la quantité de données accédées aléatoirement dans un fichier accédé : charger en cache
- Quantité de mémoire limitée et applications prioritaires sur le cache

Conseils d'optimisation (et plus) [1/4]

- « Évidences » :
  - Optimiser une fois que les algorithmes/codes sont corrects et valides (*premature optimization is the root of all evil*)
  - Optimiser n'est pas une recette magique
  - Optimiser ce qui a besoin de l'être ⇒ mesurer avant de choisir sur quoi se pencher
  - Utiliser des structures de données pertinentes, pas nécessairement les mêmes en mémoire et sur disque
  - Raisonner à un niveau d'abstraction élevé mais ne pas ignorer les limites technologiques durables (exemple : tailles de secteur/page)
  - Choisir si la portabilité est un objectif
  - Minimiser le nombre d'ouvertures d'un même fichier :

```
for i in 1..50; do for f in *.log; do
          grep toto$i $f > toto${i}.txt
```

⇒ 50 lectures de chaque fichier, très rapidement très inefficace

Conseils d'optimisation (et plus) [2/4]

- Objets :
  - Limiter le nombre d'objets/répertoire
  - Taille des fichiers : > secteur, mais doit être gérable et transférable, dizaines de Go pas centaines ou To

- Taille de bloc d'entrée/sortie idéalement 2^n, $\gg$ 512 octets, voir/voire *preferred block size* dans `struct stat`
- Minimiser le nombre d'I/Os : maximiser la taille de bloc (sans être excessif, Mo pas Go)
- Faire attention aux accès concurrents aux mêmes portions d'un fichier
- Ouvrir les fichiers avec le bon mode : utiliser `O_RDONLY`, `O_WRONLY`
- N'utiliser des verrous sur des fichiers que si c'est absolument nécessaire

Conseils d'optimisation (et plus) [3/4]

- Ne pas ignorer les erreurs et les gérer de manière sensée :
  - arrêter d'écrire après `ENOSPC` ou `EDQUOT`
  - ré-essayer (2 ou 3 fois) après `EIO` (ou `EAGAIN`, `EINTR`)
  - vérifier la valeur de retour des `read` (*garbage in*)
- N'utiliser des écritures synchrones (`O_DSYNC`) ou Direct I/O que si c'est vraiment nécessaire (mesurer le gain réel), préferer `fsync/fdatasync` avant la fermeture du fichier
- **Tirer partie du cache**
- Connaître/déterminer le type d'entrées/sorties d'un programme
- Accès aléatoires, séquentiels, *stride* & *slide*
- Exprimer un besoin/objectif de performance en partant du besoin final :

  N objets traités par seconde ⇒ N*X octets lus, N*Y octets écrits

  ⇒ Quels types d'E/S ? Objectif en pointe ou soutenu ?

  ⇒ Différencier les E/S courantes de celles qui sont rares

Conseils d'optimisation (et plus) [4/4]

- Threads : limiter le nombre de threads qui font des I/Os simultanées, #threads $\gg$ taille de file d'attente de la cible $\Rightarrow$ *trashing*
- Avec des threads, un petit nombre de threads en charge des I/Os pour les autres threads qui calculent est souvent plus efficace que tous les threads qui font des I/Os simultanées
- Donner des *hints* au système de fichiers
- Processus dont le fonctionnement dépend de la sémantique POSIX
- Utiliser des bases de données si nécessaire (éventuellement SQLite en RAM)
- Attributs étendus au lieu de micro-fichiers ?
- Utiliser le cache quand le code ne peut pas être modifié :
  - Copier les fichiers dans `/dev/null` ou commande `readahead` (Linux)
  - Ou utiliser un système de fichier en RAM (`tmpfs` voire `/dev/shm`)

Heuristiques pour le RAID

- Performances en IOps utile :
  - RAID-1/10:
    - IOps/disque * nombre de disques pour les lectures
    - IOps/*mirroré* pour les écritures
  - RAID-0/5/6/50/60 :
    - IOps/disque * nombre de disques, si I/O > *chunk* ou *stripe*
    - IOps/disque, si I/O < *chunk/strip*
- RAID-[56] & nombre de disques vs *Uncorrectable Bit Error Rate* :
  - BER * #bits/disque * #disques dans un volume RAID

    ⇒ erreur *certaine* avec des gros disques, besoin de vérification de la parité (ou checksum) lors des lectures (DDN, ZFS, Btrfs, ...)
  - RAID-5 habituellement acceptable si nombre de disques < 8
  - RAID-6 si plus de 8 disques ou disques > 500 Go
- Essayer d'avoir une taille de *strip/chunk* et/ou de *stripe* pertinente, par exemple I/O de 1Mo ⇒ *stripe* de 1 Mo (plus facile à dire qu'à faire)

Déterminer le type d'E/S d'un programme

- `iotop` identifier les programmes qui font des I/Os
- Lire/étudier les sources
- `ltrace` si framework (HDF5, ROOT, ...)
- `strace` (`strace -e 'mmap,file,desc' -fp $PID`) en collaboration avec `lsof` (`lsof -au $user -p $PID /filesystem`)
- Outils/frameworks d'instrumentation/tracing :
  - Dtrace (Solaris, FreeBSD, NetBSD & +/- MacOS X, QNX 6+?, Oracle Unbreakable Linux, système & applications, dynamique)
  - SystemTap (Linux, FC12+ & RHEL 5.2+, système & applications explicites & utrace, dynamique)
  - LTTng (Linux, pas intégré, système & applications avec UST, +/- dynamique & statique, LTTV Viewer, utilisé surtout pour le temps réel)
  - ProbeVue (AIX 6.1+, dynamique)

Exemple iotop

```
% iotop -u $USER
Total DISK READ: 0.00 B/s | Total DISK WRITE: 25.27 M/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 4196 be/4 loic        0.00 B/s   25.27 M/s  0.00 % 65.27 % perl -le f~8576*10);
 4129 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % python /us~top -u loic
 4193 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % ksh -ue ./t-small.sh
 4195 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % time -p /u~8576*10);
 4073 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % sshd: loic@pts/0,pts/1
 4074 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % -zsh
 4075 be/4 loic        0.00 B/s    0.00 B/s  0.00 %  0.00 % -zsh
%
```

Exemple strace & lsof

```
% ps -fu user
user 30569 30000 0 14:53 ? 00:00:00 /bin/bash /usr/local/products/sge/bin/starter.sh
        /var/spool/sge/ccwsge0098/job_scripts/6119482
user 30722 30569 0 14:53 ? 00:00:00 /usr/local/bin/tcsh /var/spool/sge/ccwsge0098/job_scripts/6119482
user 30744 30722 19 14:53 ? 00:31:12 ./geant.exe
# lsof -au user -p 30744 /fs
COMMAND   PID   USER FD TYPE DEVICE SIZE/OFF NODE     NAME
geant.exe 30744 user cwd DIR 0,26   32768    3490710  /fs/user/MonteCarlo/..../geant
geant.exe 30744 user 3u  REG 0,26   13664    3490717  /fs/user/MonteCarlo/..../fort.15
geant.exe 30744 user 7w  REG 0,26   48812397600 3490715 /fs/user/MonteCarlo/..../zebradat.1.fz
geant.exe 30744 user 8u  REG 0,26   87633920 521101   /fs/user/MonteCarlo/..../data.1.ntp
geant.exe 30744 user 9u  REG 0,26   36018124 3490734  /fs/user/MonteCarlo/..../geant/fort.25
# strace -e 'trace=mmap,file,desc' -fp 30744
Process 30744 attached - interrupt to quit
lseek(8, 26562560, SEEK_SET)            = 26562560
write(8, "<...>"..., 20480) = 20480
lseek(8, 23531520, SEEK_SET)            = 23531520
write(8, "<...>"..., 20480) = 20480
lseek(8, 20520960, SEEK_SET)            = 20520960
write(8, "<...>"..., 20480) = 20480
lseek(8, 17510400, SEEK_SET)            = 17510400
^C
```

SystemTap

- *Réponse* des développeurs Linux à Dtrace
- Depuis 2005/2006 implication de Intel, RedHat, IBM, Bull, Oracle, ...
- Sondes dans le noyau (indépendantes de SystemTap)
- SystemTap permet d'interroger l'état des sondes avec un langage de scripts type Awk
- `http://sourceware.org/systemtap`, tutoriel et documentations
- Exemples sur `http://souceware.org/systemtap/examples`
- RedHat : RPM `systemtap` & `systemtap-runtime` + RPM de symboles pour les noms des fonctions/objets du noyau (`kernel-debuginfo` & `kernel-debuginfo-common`, **pas** `kernel-debug`)
- RPM `kernel-debuginfo*` pas toujours disponibles (SL, FC) très gros (~280 Mo) mais indispensables
- Noyau *custom* ⇒ régénérer les packages
- Principe : commande `stap` avec un script en argument, script traduit en C, compilé sous forme de module noyau puis chargé

Exemple SystemTap : pagecache-hit-rate.stp

```
# stap -v -m pagecache_hit_rate pagecache-hit-rate.stp `id -u loic`
Pass 1: parsed user script and 77 library script(s) using 100260virt/22952res/2936shr kb, in 0usr/280sys/304real ms.
Pass 2: analyzed script: 8 probe(s), 29 function(s), 7 embed(s), 11 global(s) using 329556virt/129032res/7828shr kb, in 3520usr/10580sys/19948real ms.
Pass 3: translated to C into "/tmp/stapUFt9Ek/pagecache_hit_rate.c" using 320548virt/126668res/7736shr kb, in 270usr/460sys/797real ms.
Pass 4: compiled C into "pagecache_hit_rate.ko" in 4790usr/5320sys/13292real ms.
Pass 5: starting run.
Starting...
  Total Bytes (KB)   Cache Bytes (KB)    Disk Bytes (KB)  Miss Rate   Hit Rate
No reads
No writes
[...]
No reads
Writes:
            167636              167636                         0       0.00%    100.00%
Reads:
            116088              116088                         0       0.00%    100.00%
No writes
```

Questions ?

Merci