Temps réel sous Linux

Programmation multitâche et multicœur

Christophe Blaess

christophe.blaess@logilin.fr

https://www.blaess.fr/christophe/
twitter: @chrisblaess

Ingénierie et formations sur Linux et les logiciels libres
https://www.logilin.fr

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

Temps réel sous Linux

v. 5.3

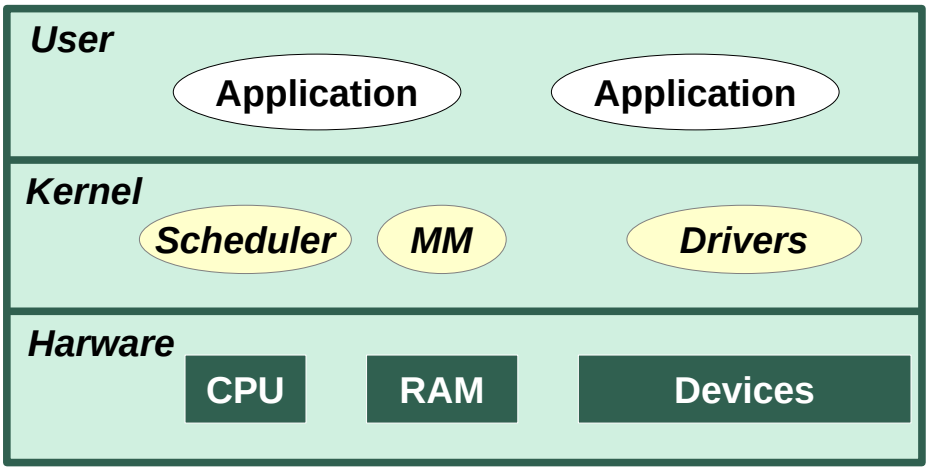
https://www.blaess.fr/christophe/

https://www.logilin.fr

Multitâche sous Linux

Linux :

- noyau de système d'exploitation compatible Unix,
- environnement constitué de logiciels libres,
- développement essentiellement bénévole.



Linux n'a jamais été prévu pour être un système industriel, temps-réel ou embarqué !

Influences de Linux :

1969 – Ken Thompson & Denis Ritchie : Unix AT&T

1978 – Université de Berkeley : Unix BSD

1984 – Richard Stallman : Projet Gnu (www.gnu.org)

1987 – Andrew Tanenbaum : Minix

Création de Linux :

1991 – Linus Torvalds : Linux 0.0.1

Voir aussi

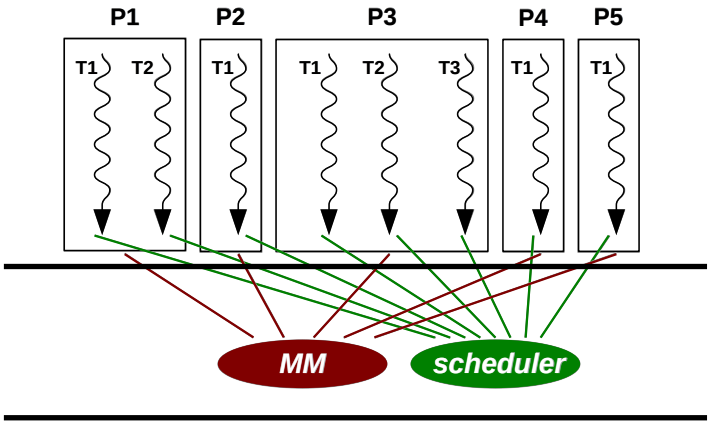
- [Tanenbaum 1997] : Andrew Tanenbaum & Albert Woodhull – O*perating Systems, Design and Implementation* – Second Edition – Prentice Hall, 1997.
- [Raymond 2004] : Eric S. Raymond – *The Art of Unix Programming* – Addison-Wesley, 2004.
- [Torvalds 2001] : Linus Torvalds & David Diamond – *Il était une fois Linux* – Osman Eyrolles Multimedia, 2001 – Titre original : *Just for fun*.

Threads et processus

Pour les premiers Unix, un **processus** représentait une tâche, s'exécutant dans son propre espace mémoire. Tout le multi-tâche était organisé autour des processus.

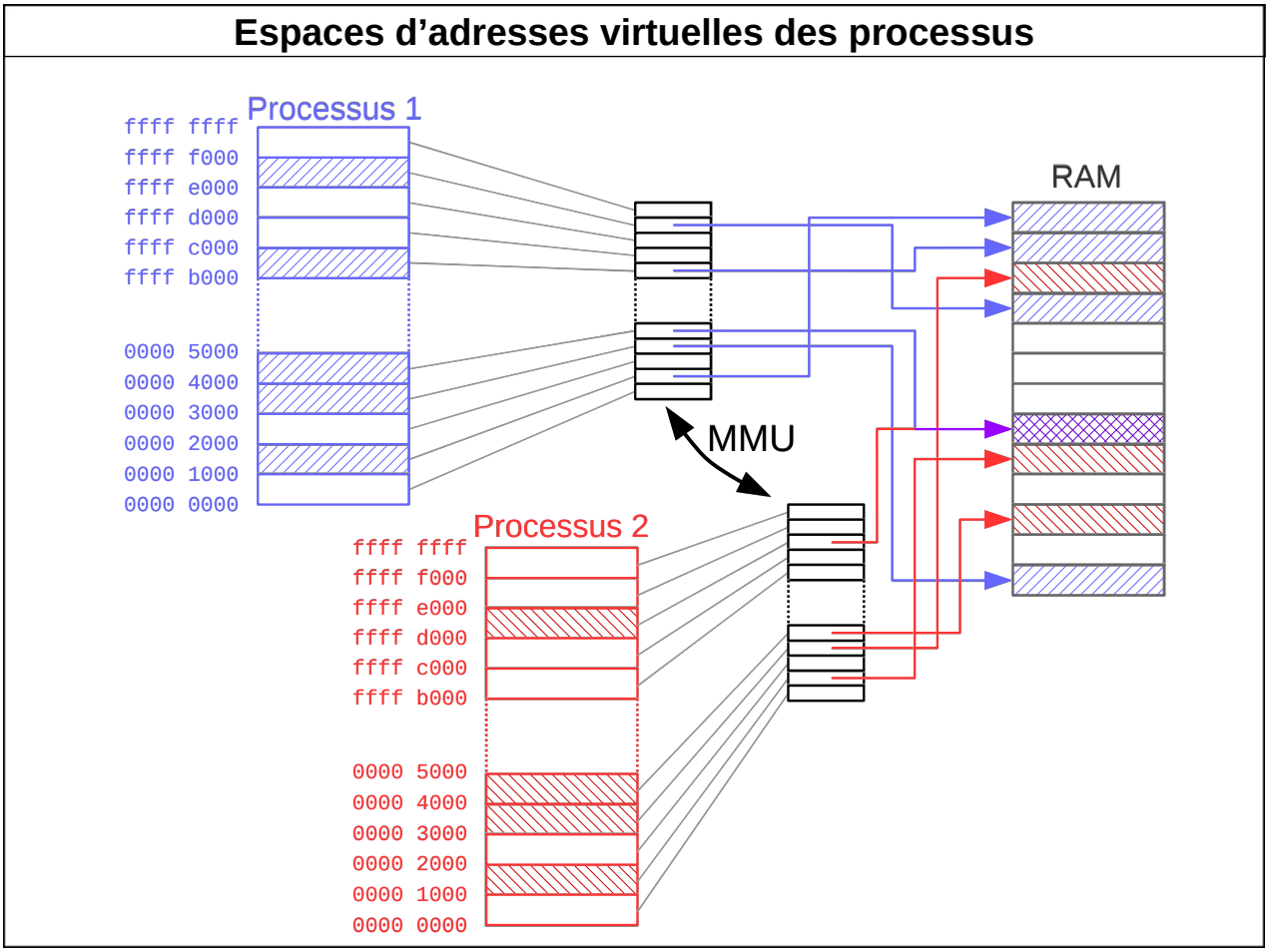
Depuis les années 1990, la notion de **thread** a modifié la situation.

Un processus est un **espace mémoire** au sein duquel s'exécute au moins une tâche.



L'ordonnanceur ne traite que des tâches, des threads. La gestion des processus relève du sous-système MM (*memory management*) du noyau.

Espaces d'adresses virtuelles des processus



La mémoire est divisée en pages (4Ko). Le CPU ne voit que des adresses dites virtuelles dans un espace linéaire de 4 Go (systèmes 32 bits) ou 16 Eo (systèmes 64 bits).

Les adresses virtuelles sont traduites en adresses physiques grâce à une table chargée dans la MMU (*Memory Management Unit*). Chaque processus dispose d'une table spécifique.

L'accès à une adresse où aucune page n'est présente dans la table de la MMU déclenche une exception (interruption) *Page Fault*. Généralement le noyau réagit en envoyant le signal `SIGSEGV` au processus, ce qui se traduit par sa mort avec le message « *Segmentation Fault* ».

Les processus Unix

Un processus dispose d'un espace mémoire totalement indépendant du reste du système. Il y a une filiation entre les processus, chacun d'eux a obligatoirement un processus parent.

Un processus est identifié par son **PID** (*Process Identifier*) entier de type `pid_t`.

```
pid_t  getpid(void);

pid_t  getppid(void);
```

Création d'un nouveau processus :

```
pid_t  fork(void);
```

Fin d'un processus :

```
void   exit(int code);
```

Attente de la fin d'un processus fils

```
pid_t  waitpid(pid_t child, int *status, int options);
```

Exécution d'un nouveau code :

```
int execve(char *file, char *argv[], char *envp[]);
```

Travaux pratiques : principes des processus

Examinez le contenu du fichier **`exemple-I-01.c`**. Compilez-le.

Testez-le en observant dans une autre console la liste des processus présents avec `ps aux`

Modifiez l'exemple pour que le processus parent s'endorme très longtemps – `sleep(1000)` par exemple – à la place du `waitpid()`. Que se passe-t-il pour le processus enfant lorsqu'il se termine ?

Essayez le « mini shell » proposé dans **`exemple-I-02.c`** en lui passant des commandes simples comme : `ls`, `date`, `who`... Fonctionne-t-il correctement ?

Essayez à présent une commande plus complète comme « `ls -l /etc` ». Que se passe-t-il ?

Les Threads Posix.1c

Les threads sont des tâches s'exécutant dans le même espace mémoire. On considère qu'un processus est constitué d'au moins un thread – celui exécutant la fonction `main()`.

Un thread est identifié par une variable de type `pthread_t` opaque (dépendant de l'implémentation).

Création d'un nouveau thread dans une application :

```
int pthread_create(pthread_t *ident, pthread_attr_t *attr,
                   void *(*function)(void *), void *arg);
```

Fin d'un thread (sans terminer l'application) :

```
void pthread_exit(void *ret_code);
```

Attente de la fin d'un thread :

```
int pthread_join(pthread_t ident, void **ret_code);
```

Pour utiliser l'API des threads, inclure le fichier d'en-tête `pthread.h` au début du fichier source.

A la compilation et à l'édition des liens, ajouter l'option `-pthread` en paramètre de `gcc`.

Les mutex Posix.1c

Les accès aux variables communes sans synchronisation peuvent mener à une corruption des données.

Pour protéger les accès on utilise des mutex (*mutual exclusion*) :

```
pthread_mutex_t   mtx  = PTHREAD_MUTEX_INITIALIZER;
```

que l'on verrouille et déverrouille avant et après l’accès aux données partagées.

```
int   pthread_mutex_lock(pthread_mutex_t *mtx);

int   pthread_mutex_unlock(pthread_mutex_t *mtx);
```

Les mutex sous Linux avec la NPTL sont implémentés de manière très efficace en s'appuyant sur les Futex (*Fast Userspace Mutex*) du kernel.

Travaux pratiques : principes des threads

Examinez le contenu du fichier **`exemple-I-03.c`**. Compilez-le et testez-le.

Les threads affichent-ils toujours leurs messages dans le même ordre ?

Lancez le programme en le préfixant de « `taskset -c 0` » afin de placer tous les threads sur le même CPU. Le comportement est-il modifié ?

Dans une autre console, voyez-vous les threads avec « `ps aux` » ?

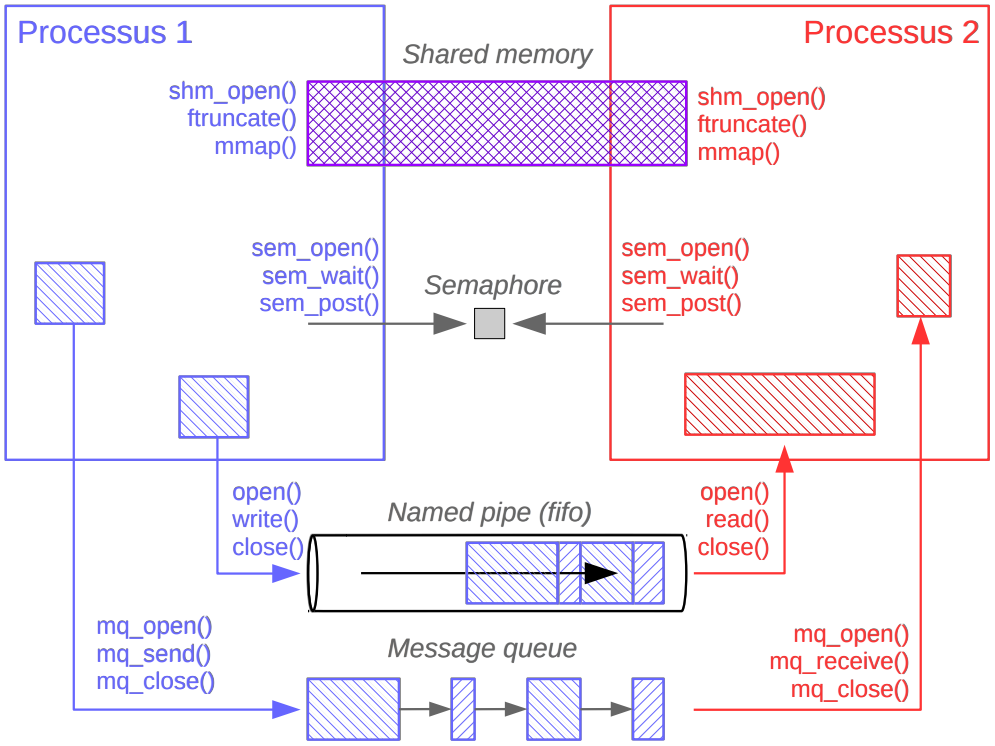
Et avec l'option « `m` » (`ps maux`) ?

Examinez le contenu du fichier **`exemple-I-04.c`**. Compilez-le.

Exécutez le programme. Son résultat est-il conforme à celui attendu (2 millions) ?

Dé-commentez les trois lignes masquées utilisant les mutex. Ré-exécutez le programme. Fonctionne-t-il correctement ? Son temps d'exécution est-il identique au précédent ?

IPC – Inter Process Communications



Le support des files de messages Posix est une option de compilation du noyau (généralement activée).

Files de messages

```
#include <mqueue.h>

mqd_t mq_open   (const char *name, int o_flag, mode_t mode,
                 struct mq_attr *attr);
int   mq_close  (mqd_t mqd);
int   mq_unlink (const char *name);
```

```
struct mq_attr {
   long mq_flags;   /* Attributs : 0 ou O_NONBLOCK */
   long mq_maxmsg;  /* Nb maxi de messages dans la file */
   long mq_msgsize; /* Taille maxi message (octets) */
   long mq_curmsgs; /* Nb messages dans la file */
};
```

```
int mq_send(mqd_t mqd, const char *message,
            size_t len, unsigned int prio);

ssize_t mq_receive(mqd_t mqd, char *buffer,
                   size_t maxlen, unsigned *prio);
```

Pour `mq_receive()`, `maxlen` doit être de taille au moins égale à `mq_msgsize` (8192 par défaut sous Linux.)

```
mqd_t mq_getattr(mqd_t mqd, struct mq_attr *attr);

mqd_t mq_setattr(mqd_t mqd, struct mq_attr *attr,
                 struct mq_attr *prev_attr);
```

À vous...

Examinez le contenu du fichier **`exemple-I-05.c`**. Compilez-le.

Exécutez le programme en lui passant en argument :

- le nom d'une file de message (choisi arbitrairement) commençant par un « slash » `/`
- une valeur de priorité du message
- un message (chaîne de caractères).

Envoyez quelques messages en variant les priorités.

Après avoir observé le contenu du fichier **`exemple-I-06.c`**, compilez-le et exécutez-le en lui passant en argument le nom de la file de message. Voyez-vous vos messages ? Dans quel ordre ?

Quel est par défaut la contenance de la file de message ?

Mémoire partagée

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

```
int shm_open(const char * name, int o_flag,
             mode_t mode);
int shm_unlink(const char * name);
```

Dimensionnement initial avec :

```
int ftruncate(int fd, off_t length);
```

Projection d'une zone en mémoire :

```
void *mmap(void *address, size_t length, int permissions,
           int flags, int fd, off_t offset);

int munmap(void *address, size_t length);
```

Utiliser l'attribut `MAP_SHARED` dans le paramètre *flags* de `mmap()`.

À vous...

Examinez le contenu du fichier **`exemple-I-07.c`**. Compilez-le.

Exécutez le programme en lui passant en argument le nom d'un partage de mémoire (choisi arbitrairement). Voyez-vous le compteur progresser ?

Arrêtez le programme. Relancez-le. Comment se comporte le compteur ?

Lancez un deuxième exemplaire dans une autre console. Comme le compteur progresse-t-il ?

Utilisez la commande « `mount` » sur votre système pour afficher les montages de systèmes de fichiers. Cherchez un montage `/dev/shm` ou `/run/shm` (suivant les distributions). Qu'y voyez-vous ?

Sémaphores

```
sem_t * sem_open(cont char *name, int o_flags,
                 mode_t mode, unsigned int value);

int     sem_close(sem_t *sem);

int    sem_unlink(const char *name);
```

Attente d'un sémaphore :

```
int  sem_wait(sem_t *sem);
int  sem_trywait(sem_t *sem);
int  sem_timedwait(sem_t *sem,
                   const struct timespec *timeout);
```

Libération d'un sémaphore :

```
int sem_post (sem_t *sem);
```

Pour compiler un programme utilisant des ressources Posix.1b, comme les sémaphores, il faut utiliser l'option « **`-lrt`** » (*library real-time*) à l'édition des liens.

À vous...

Examinez le contenu du fichier **`exemple-I-08.c`**. Compilez-le.

Passez un nom de sémaphore au lancement du programme et observez les processus créés. Vérifiez qu'à un moment donné, un seul processus tient le sémaphore.

Modifiez dans le programme le dernier argument de `sem_open()` pour accepter deux prises simultanées du sémaphore. Vérifiez le fonctionnement.

Mutex entre processus

Initialisation d'un mutex avec des attributs spécifiques :

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);


int pthread_mutex_init(pthread_mutex_t *mtx,
                       pthread_mutexattr_t *attr);
```

Attribut de partage entre processus :

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);

   pshared:
        PTHREAD_PROCESS_SHARED   <---
        PTHREAD_PROCESS_PRIVATE
```

À vous...

Examinez le contenu du fichier **`exemple-I-09.c`**. Compilez-le.

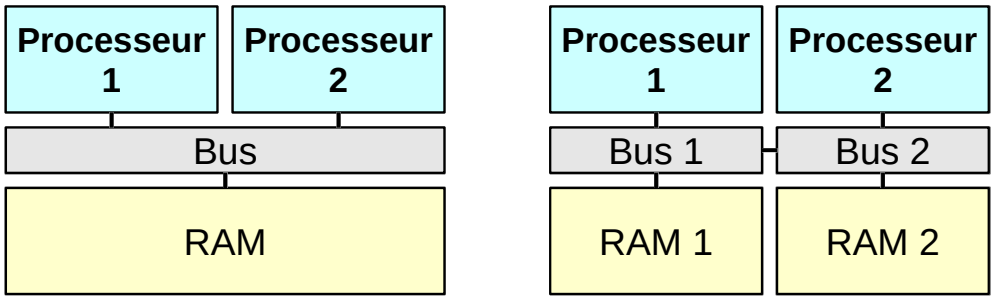
Lancez le programme en deux exemplaires dans deux terminaux différents avec le même nom de *shared memory* qui ne doit pas exister au préalable.

Démarrez le comptage simultanément (autant que possible) dans les deux terminaux, puis affichez les résultats.

Programmation multicœur

Multiprocesseur, multicœur, hyperthreading

Systèmes multiprocesseurs



Architecture SMP
(Symetrical Multi-Processing)

Architecture NUMA
(Non Uniform Memory Access)

Les processeurs (indépendants) peuvent accéder à la mémoire de manière uniforme (même temps d'accès quelque soit la zone) ou non.

Exemple de SMP classique : *Intel Xeon...*

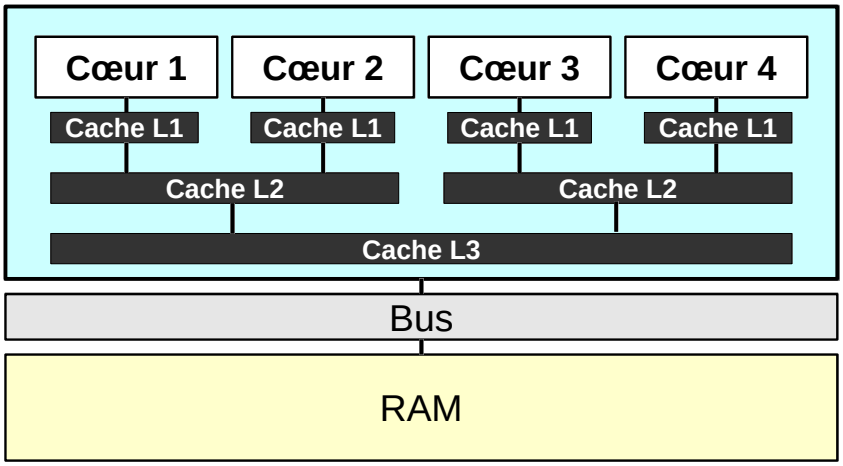
Exemple de NUMA : *Intel Itanium*, *AMD Opteron*

En savoir plus...

[DREEPER 2007] – Ulrich Drepper « *What Every Programmer Should Know About Memory* »
`http://lwn.net/Articles/250967/`

[BLAESS 2011] – Christophe Blaess – *Expériences avec le cache*
`http://www.blaess.fr/christophe/articles/` (8 et 15 avril 2011)

Systèmes multicœurs



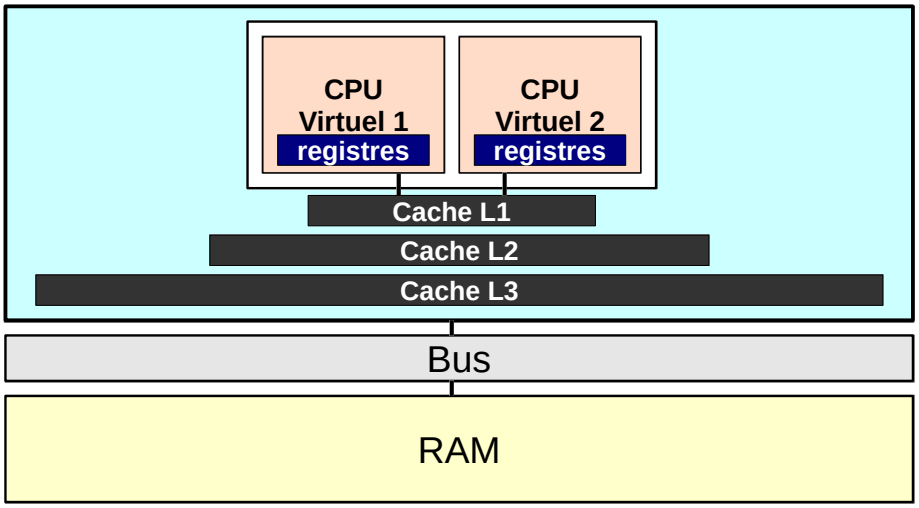
L'unité de traitement des données est multipliée (par deux ou quatre en général).

Le cache de premier niveau est spécifique à chaque cœur. Le cache de second niveau peut être partagé ou non. Lorsque le cache de niveau 2 est partagé l'accès est plus rapide, sinon il faut des messages de synchronisation lors d'accès à des zones identiques depuis les deux cœurs.

Exemples de processeurs multi-cœurs :

- *Intel Core i5*, *i7*, *i9*, etc
- *Arm Cortex A9*…
- *AMD Athlon 64*, *Opteron*, *Phenom*, *Zen*, etc.

Symetrical Multi-Threading (SMT)
Hyperthreading



En SMT, le cœur de processeur contient deux CPU virtuels, chacun disposant de ses propres registres.

Les caches sont partagés entre les CPU virtuels.

A vous...

Déterminez l'architecture de votre système (du moins tel qu'il est reconnu par Linux) avec la commande `lscpu` ou en examinant le fichier `/proc/cpuinfo` :

```
$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i7-8700T CPU @ 2.40GHz
Stepping:                        10
CPU MHz:                         3800.000
CPU max MHz:                     4000,0000
CPU min MHz:                     800,0000
BogoMIPS:                        4800.00
Virtualization:                  VT-x
L1d cache:                       192 KiB
[...]
```

Affinité et migration de tâches

Processeur sur lequel tourne une tâche :

```
#define _GNU_SOURCE
#include <sched.h>
int sched_getcpu (void);
```

Affinité d'un processus :

```
int sched_setaffinity(pid_t pid, size_t cpu_set_size,
                      cpu_set_t *cpu_set);

int sched_getaffinity(pid_t pid, size_t cpu_set_size,
                      cpu_set_t *cpu_set);
```

Manipulation des ensembles de CPU :

```
void CPU_ZERO(cpu_set_t *cpu_set);
void CPU_SET(int cpu, cpu_set_t *cpu_set);
void CPU_CLR(int cpu, cpu_set_t *cpu_set);
int  CPU_ISSET(int cpu, cpu_set_t *cpu_set);
```

Travaux pratiques : migrations de processus

Exécutez le programme **`exemple-I-10.c`** sur un système multicœur. Observez les migrations d'un CPU à l'autre en fonction de la charge système.

Conseil : l'utilitaire `gkrellm` (*Gnome Krell Monitor*) est très pratique pour surveiller les charges des différents CPU.

Examinez le fichier **`exemple-I-11.c`**. Puis exécutez-le sur un système multicœur.

Vous verrez le processus se déplacer régulièrement d'un CPU à l'autre.

Affinité d'un thread :

```
#define _GNU_SOURCE
#include <pthread.h>

int pthread_setaffinity_np(pthread_t thread,
                           size_t cpu_set_size,
                           const cpu_set_t *cpu_set);

int pthread_getaffinity_np(pthread_t thread,
                           size_t cpu_set_size,
                           cpu_set_t *cpu_set);
```

Si l'affinité est fixée avant la création du thread :

```
int pthread_attr_setaffinity_np(pthread_attr_t *attr,
                                size_t cpu_set_size,
                                const cpu_set_t *cpu_set);

int pthread_attr_getaffinity_np(pthread_attr_t *attr,
                                size_t cpu_set_size,
                                cpu_set_t *cpu_set);
```

À vous

Examinez le code source du programme **`exemple-I-12.c`**.

Lors de son exécution, observez l'utilisation des CPU avec un outil système (*GkrellM*, *System Monitor*, etc.) et vérifiez que les threads sont bien répartis sur tous les CPU.

Affinité des interruptions

On peut voir dans `/proc/interrupts` le nombre d'occurrences de chaque interruption (depuis le *boot*), CPU par CPU :

```
$ cat /proc/interrupts
           CPU0       CPU1
  0:         44         10   IO-APIC-edge      timer
  1:      12823         31   IO-APIC-edge      i8042
  7:          1          0   IO-APIC-edge
  8:          0          1   IO-APIC-edge      rtc0
  9:       8760        212   IO-APIC-fasteoi   acpi
 12:          8     347861   IO-APIC-edge      i8042
[...]
```

L'affinité des interruptions est visible dans `/proc/irq/<numero>/smp_affinity` :

```
# cat /proc/irq/9/smp_affinity
1
# cat /proc/irq/12/smp_affinity
2
#
```

NB : Il s'agit d'un masque (`1` pour CPU0, `2` pour CPU1, `4` pour CPU2, etc.)

Le démon `irqbalance` agit sur ces fichiers pour équilibrer la charge en interruption.

Travaux pratiques : migration des interruptions

Utilisez la commande `watch` pour voir évoluer les interruptions sur votre système :

```
$ watch -n 0,1  cat /proc/interrupts
```
(Suivant la localisation il faudra saisir `0,1` ou `0.1`)

Cherchez l'interruption correspondant à votre souris (sur notre exemple l'interruption 12).

Dans une autre console modifiez l'affinité de l'interruption souris. Par exemple :

```
# echo 1 > /proc/irq/12/smp_affinity
```
(pour passer sur le CPU 0)

```
# echo 2 > /proc/irq/12/smp_affinity
```
(pour revenir sur le CPU 1)

```
# echo 3 > /proc/irq/12/smp_affinity
```
(pour autoriser le traitement sur l'un des deux premiers processeurs en fonction de la charge)

Vérifiez que les interruptions soient traitées par le CPU demandé.

Activation / désactivation de CPU

En inscrivant `1` ou `0` dans

`/sys/devices/system/cpu/cpu<N>/online`

On peut « allumer » ou « éteindre » un CPU. La signification réelle de ces opérations dépend du type de processeur.

Fréquence de fonctionnement du processeur

Sur certains processeurs on peut ajuster la fréquence de fonctionnement.

Dans le répertoire `/sys/devices/system/cpu/cpu0/cpufreq` on trouve un fichier `scaling_available_governors` indiquant les comportements possibles pour gérer la fréquence :

- `performance` : la fréquence est toujours la plus élevée possible (serveur de calcul)
- `powersave` : la fréquence est toujours la plus faible (système embarqué sur batterie)
- `ondemand` : la fréquence varie de la plus faible à la plus élevée en fonction de la charge système (poste de travail)
- `conservative` : comme `ondemand`, mais la fréquence varie plus lentement, par étapes successives (serveurs).
- `userspace` : on fixe la fréquence manuellement dans le fichier `scaling_set_speed`.

Pour choisir le comportement désiré, il faut écrire son nom dans `scaling_governor`.