APPELS SYSTÈMES: INTRODUCTION

Guillaume Chanel

Remerciements à Jean-Luc Falcone

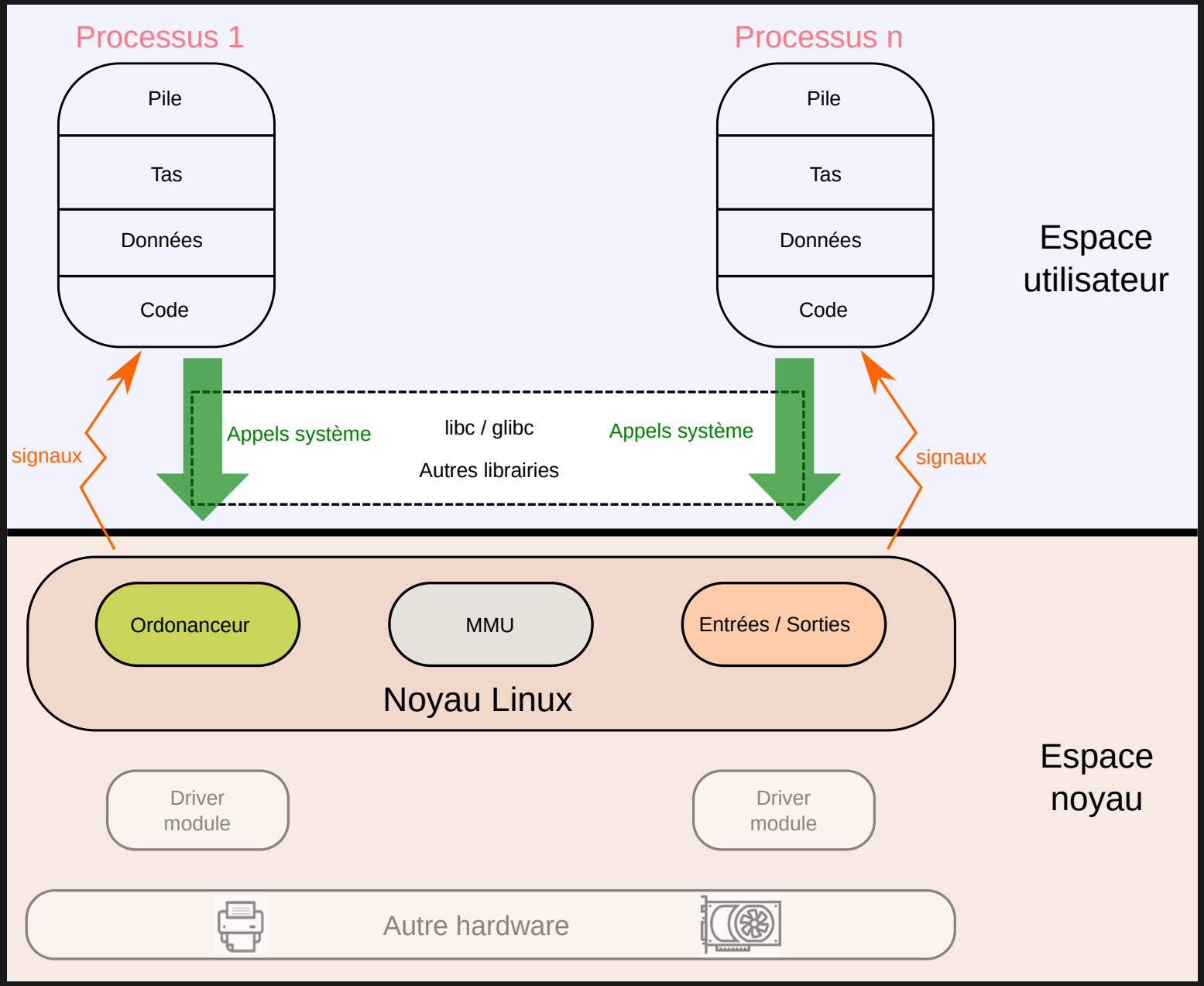
Septembre 2019

APPELS SYSTÈME

APPELS SYSTÈME – SYSTEM CALLS

Les appels systèmes sont des fonctions permettant aux programmes d'utiliser les fonctionnalités/services du noyau.

- Ils permettent notament de:
  - Intéragir avec les systèmes de fichiers
  - Intéragir avec les données des fichiers
  - Créer d'autres processus
  - Communiquer avec d'autres processus
  - …
- Cela permet plus de sécurité car les appels système imposent une interface aux ressources


Processus 1
Pile
Tas
Données
Code
Processus n
Pile
Tas
Données
Code
Espace utilisateur
signaux
Appels système
libc / glibc
Autres librairies
Appels système
signaux
Ordonanceur
MMU
Entrées / Sorties
Noyau Linux
Espace noyau
Driver module
Driver module
Autre hardware

FONCTION `syscall`

La fonction `syscall` déclenche une interruption qui change le mode du CPU (privilèges plus élevés) et passe la main au noyau pour effectuer une opération spécifique:

```
long syscall(long number, ...)
```

La fonction prend en paramètre un nombre spécifiant l'appel système, et une liste variable d'arguments dépendant de l'appel système choisi.

FONCTION `syscall`: EXEMPLE

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>

int main() {
    pid_t pid;
    pid = syscall(SYS_getpid);
    syscall(SYS_kill, pid, SIGHUP);
}
```

FONCTIONS "WRAPPER"

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char *argv[]) {
    pid_t pid;
    pid = getpid();
    kill( pid, SIGHUP);
}
```

LIBRAIRIES DE PLUS HAUT NIVEAU

Les appels système sont souvent utilisés au travers de librairies de plus haut niveau

```
// Appel systeme Posix
int open(const char *pathname, int flags, mode_t mode);


// Fonction ANSI-C
FILE *fopen(const char *path, const char *mode);
```

TYPES OPAQUES

TYPES OPAQUES

Les types utilisés peuvent être *opaques*

Définis dans time.h

```c
// Type temps
typedef /* unspecified */ time_t;

// Retourne l'heure/date actuelle
time_t now = time( NULL );

// Retourne une representation textuelle
char* str = ctime( &now );
```

FILE est aussi un type opaque

```c
typedef /* still not specified */ FILE;
FILE *f = fopen(/tmp/costs, "w");
fprintf(f, "Le cout est: %d CHF", cost);
```

TYPES OPAQUES - ORGANISATION

Les types opaques sont des structures de données (i.e. typedef) qui ne sont **pas définies dans l'interface** (i.e. fichier header). Cela permet:

- cacher les détails l'implémentation -> abstraction;
- de modifier la structure de donnée sans modifier le comportement du code utilisateur.

listeChainee.h

```c
typedef struct el* listeChainee;

listeChainee initListeChaineeVide();
void addListe(listeChainee* liste, void* contenu)
void* removeListe(listeChainee *liste);
```

ListeChainee.c

```c
#include "listeChainee.h"

/* Declaration dans le .c pour
une utilisation privée */
typedef struct el {
    struct el *suivant;
    void* contenu;
} element;

listeChainee initListeChaineeVide() {
    return NULL;
}
```

BIT FIELDS

PROBLÈME

On aimerait représenter des personnes par la structure suivante:

Définition d'une personne

- Nom : string
- Age : entier
- Marrié? : oui/non
- Enfants? : oui/non
- Permis de conduire? : oui/non
- Parle anglais? : oui/non

IMPLÉMENTATION

```c
struct person {
    char *name;
    int age;
    int isMarried;
    int hasChildren;
    int canDrive;
    int speaksEnglish;
};

typedef struct person person_t;
```

SÉLECTIONNER UNE PESONNE SELON DES CRITÈRES

```
int match( const person_t *p, int isMarried, int hasChildren, int canDrive,
  int speaksEnglish ) {
    if( isMarried && ! p->isMarried ) {
        return 0;
    }
    if( hasChildren && ! p->hasChildren ) {
        return 0;
    }

    //...

    if( speaksEnglish && ! p->speaksEnglish ) {
        return 0;
    }
    return 1;
}
```

EXEMPLE D'UTILISATION

```
person_t alice = { "Alice", 24, 0, 1, 0, 1 };
person_t bob = { "Bob", 37, 1, 0, 1, 1, };

person_t dudes[] = {alice,bob};

int i;
for( i=0; i < 2; i++ ) {
    person_t p = dudes[i];
    if( match( &p, 0, 1, 0, 1 ) ) {
        printf( "%s is selected \n", p.name );
    }
}
```

INCONVÉNIENTS

Cette solution présente de nombreux inconvénients:

- Gaspillage de mémoire
- Trop de paramètres identiques
- Lecture séquentielle
- Peu évolutif (rajout de nouvelles propriétés)

LES CHAMPS DE BITS (BIT FIELDS)

On pourrait utiliser les bits d'un entier pour contenir la même information:

`0001`	Est marrié
`0010`	A des enfants
`0100`	Permis de conduire
`1000`	Parle anglais

On peut les combiner (OR - symbole "|") pour gérer tous les cas possibles:

`0101`	Est marrié et peut conduire
`0011`	Marrié avec des enfants
…	…

DRAPEAUX (FLAGS)

- On appelle *flag* les différents champs du champ de bits.
- On les définis généralement comme des constantes ou des énumérations:

Définition

```
#define IS_MARRIED      (1 << 0)   //0001 = 1
#define HAS_CHILDREN    (1 << 1)   //0010 = 2
#define CAN_DRIVE       (1 << 2)   //0100 = 4
#define SPEAKS_ENGLISH  (1 << 3)   //1000 = 8
```

MANIPULATIONS DES FLAGS

Créer un bit-field

```
int i = HAS_CHILDREN;
int j = IS_MARRIED | CAN_DRIVE;
```

MANIPULATIONS DES FLAGS

Créer un bit-field

```
int i = HAS_CHILDREN;
int j = IS_MARRIED | CAN_DRIVE;
```

Activer les flags

```
i = i | IS_MARRIED;
j |= SPEAKS_ENGLISH | CAN_DRIVE;
```

MANIPULATIONS DES FLAGS

Créer un bit-field

```
int i = HAS_CHILDREN;
int j = IS_MARRIED | CAN_DRIVE;
```

Activer les flags

```
i = i | IS_MARRIED;
j |= SPEAKS_ENGLISH | CAN_DRIVE;
```

Désactiver les flags

```
i = i & ~IS_MARRIED;
j &= ~(CAN_DRIVE|IS_MARRIED);
```

TESTS SUR LES FLAGS

Test si possède un ou plusieurs flags

```
if( i & IS_MARRIED ) ...
if( j & (CAN_DRIVE|SPEAKS_ENGLISH) ) ...
```

TESTS SUR LES FLAGS

Test si possède un ou plusieurs flags

```
if( i & IS_MARRIED ) ...
if( j & (CAN_DRIVE|SPEAKS_ENGLISH) ) ...
```

Teste si possède tous les flags demandés

```
int mask = CAN_DRIVE | SPEAKS_ENGLISH;
if( i & mask == mask ) ...
```

IMPLÉMENTATION - BIT FIELD

```
struct person {
    char *name;
    int age;
    int properties;
};

typedef struct person person_t;
```

SÉLECTIONNER UNE PESONNE SELON DES CRITÈRES - BIT FIELD

```
int match( const person_t *p, int properties ) {
    return ( properties & (p->properties) ) == properties;
}
```

EXEMPLE D'UTILISATION - BIT FIELD

```
person_t alice = { "Alice", 24, HAS_CHILDREN | SPEAKS_ENGLISH };
person_t bob = { "Bob", 37, IS_MARRIED | CAN_DRIVE | SPEAKS_ENGLISH };

person_t dudes[] = {alice,bob};

int i;
for( i=0; i < 2; i++ ) {
    person_t p = dudes[i];
    if( match( &p, CAN_DRIVE | SPEAKS_ENGLISH ) ) {
        printf( "%s is suitable \n", p.name );
    }
}
```

ERREURS DE RETOUR

ERREUR DE RETOUR

Problème: La plupart des langages de programmation n'autorisent qu'une valeur de retour par fonction. Or on veut souvent retourner:

- Le résultat si tout se passe bien
- Un code d'erreur si quelque chose s'est mal passé

ERRNO (`errno.h`)

- En C (standards ANSI et POSIX) on utilise la variable globale `errno` pour passer un code d'erreur.
- Cette variable et les codes d'erreurs standards sont définis dans le fichier `errno.h`.

Exemples de codes standard POSIX (`man errno`)

E2BIG	Argument list too long
EACCES	Permission denied
EADDRINUSE	Address already in use
ENOSPC	No space left on device
ENOENT	No such file or directory
ETIMEDOUT	Connection timed out
EBUSY	Device or resource busy
...	...

CONVENTION DU TYPE DE RETOUR (1)

- Par convention si une fonction peut retourner une erreur, on s'arrange pour avoir des fonctions retournant:
  - soit un type entier signé (`short`, `int`, `long`)
  - soit un pointeur
- Par convention ces fonctions retournent en cas d'erreur:
  - soit -1
  - soit NULL

CONVENTION DU TYPE DE RETOUR (2)

```
/* Cherche des entrees dans une base de donnees
Retourne:
- soit le nombre de resultats trouves
- soit -1 en cas d'erreurs
Garni le tableau results avec les resultats.
*/
int lookup( DataBase *db, query_t query, int *results );
```

UTILISATION TYPIQUE

```
int results[MAX_RESULTS];
int num = lookup( myDB, q, results );
if( num == -1 ) {
    // Gerer l'erreur
} else {
    int i;
    for( i = 0; i < num; i++ ) {
        // Gerer result[i]
    }
}
```

COMPARER LE CODE D'ERREUR

- la valeur `errno` indique le type d'erreur (`#define` associés)
- Normalement, la documentation d'une fonction doit indiquer les codes d'erreurs possibles

```
if( num < 0 ) {
    switch(errno) {
        case ECONNREFUSED:
            //Gerer un refus de connection;
            break;
        case EPERM:
            //Gerer une operation non autorisée;
            break;
        case ...
    }
}
```

OBTENIR UN MESSAGE D'ERREUR (`strerror`)

La fonction `strerror` retourne un message d'erreur (chaîne de caractère)

```
if( num < 0 ) {
    printf(stderr, "An error has occured: %s\n", strerror(errno));
}
```

AFFICHER UN MESSAGE D'ERREUR (`perror`)

La fonction `perror` permet d'afficher un message d'erreur automatiquement lié a errno sur la sortie d'erreur standard:

```
//On suppose que le fichier passé en premier paramètre du program est inexistant
int main(int argc, char* argv[]) {
    char* unFichierInexistant = argv[1];
    if (open(unFichierInexistant, O_RDONLY) < 0) {
        perror(unFichierInexistant);
        return -1;
     }
}
```

Le résultat du programme pourrait donc être:

```
$ ./mon-programme /un/fichier/inexistant
/un/fichier/inexistant: No such file or directory
```

`errno` EST UNE VARIABLE GLOBALE

Mal

```c
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

`errno` EST UNE VARIABLE GLOBALE

Mal

```c
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

Bien

```c
if (somecall() == -1) {
    int errsv = errno;
    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}
```