

Appels système : Introduction

Guillaume Chanel - guillaume.chanel@hesge.ch

Florent Glück - florent.gluck@hesge.ch

May 21, 2025

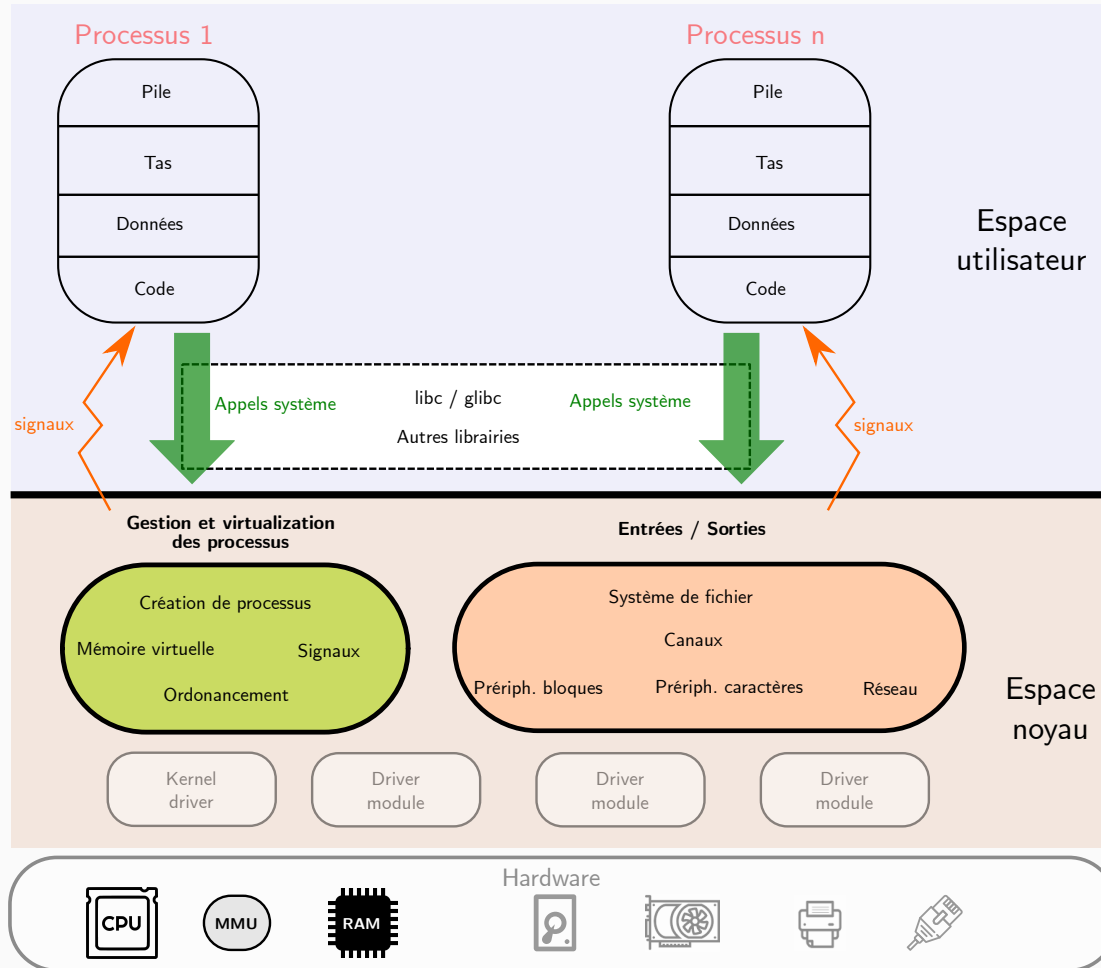
ISC - HEPIA

Appels Système

Appels système - System calls

- Les appels systèmes sont des fonctions permettant aux programmes utilisateur d'utiliser les fonctionnalités/services du noyau
- Ils permettent notamment de :
 - interagir avec les périphériques de la machine (affichage, disque, etc.)
 - interagir avec les fichiers
 - interagir avec l'affichage, clavier, souris, carte réseau, etc.
 - créer des processus, communiquer entre processus, etc.
- Les appels système permettent plus de sécurité, car ils imposent une interface aux ressources
- Le noyau gère le partage des ressources et les accès concurrents (multiplexage des ressources)

Appels Système



espace **non-privilégié**

espace **privilégié**

Fonction syscall

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

- Prend en paramètre un nombre spécifiant l'appel système, et une liste variable d'arguments dépendant de l'appel système choisi
- Déclenche une interruption logicielle qui :
 - sauvegarde l'état du CPU (registres)
 - change le niveau de privilège du CPU (non-privilégié → privilégié)
 - le noyau effectue la fonction demandée
 - restaure l'état du CPU
 - change le niveau de privilège du CPU (privilégié → non-privilégié)

Fonction syscall : exemple

```
#include <unistd.h>
#include <string.h>

int main() {
    char *msg = "syscall are awesome!\n";
    int bytes = write(1, msg, strlen(msg));
    exit(bytes);
}
```

Ré-écrit pour utiliser la fonction `syscall` :

```
#include <sys/syscall.h> // syscall number constants (SYS_xxx) can be found here
int main() {
    char *msg = "syscall are awesome!\n";
    int bytes = syscall(SYS_write, 1, msg, strlen(msg)); // SYS_write = 0
    syscall(SYS_exit, bytes);
}
```

Les n° d'appels système sont listés dans `include/x86_64-linux-gnu/asm/unistd_64.h`¹

¹Sur Ubuntu/Debian pour architecture 64 bits

Fonctions *wrapper*

- La librairie C (aussi appelée “librairie système”) contient des fonctions *wrappers* couvrant la plupart des appels système
- Exemples :

SYS_exit	exit
SYS_read	read
SYS_write	write
SYS_open	open

Attention : il n'existe pas de *wrapper* pour TOUS les appels systèmes !

Fonctions de plus haut niveau

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

- Fonction *wrapper* au dessus de l'appel système `SYS_open` :

```
// Fonction (appel système) standardisée dans POSIX.1-1988  
int open(const char *pathname, int flags, mode_t mode);
```

- Fonction de plus haut niveau que `open` (mais qui y fait appel) :

```
// Fonction standardisée dans ANSI C89  
FILE *fopen(const char *path, const char *mode);
```


Types opaques

Types opaques

Les types utilisés peuvent être opaques

- Définis dans `time.h` :

```
// Type temps
typedef /* unspecified */ time_t;
// Retourne l'heure/date actuelle
time_t now = time(NULL);
// Retourne une représentation textuelle
char *str = ctime(&now);
```

- `FILE` est aussi un type opaque :

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

Bit fields

On aimerait représenter des personnes par la structure suivante :

- Définition d'une personne
 - Nom : string
 - Âge : entier
 - Marié? : oui/non
 - Enfants? : oui/non
 - Permis de conduire? : oui/non
 - Parle anglais? : oui/non

Implémentation

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

Sélectionner une personne 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 };

for (int 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 peut utiliser les bits d'un entier pour contenir la même information :

0001 Est marié

0010 A des enfants

0100 Permis de conduire

1000 Parle anglais

On peut les combiner (avec un “or” `|`) pour gérer tous les cas possibles :

0101 Est marié et peut conduire

0011 marié avec des enfants

... ...

Flags (drapeaux)

- On appelle *flag* les différents champs d'un champ de bits
- On les définit généralement comme des constantes :

```
#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
```

ou des énumérations (**préférable**) :

```
typedef enum {
    IS_MARRIED      = 1 << 0,    // 0001 = 1
    HAS_CHILDREN    = 1 << 1,    // 0010 = 2
    CAN_DRIVE       = 1 << 2,    // 0100 = 4
    SPEAKS_ENGLISH  = 1 << 3,    // 1000 = 8
} PersonAttributes;
```

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;
```

Ajouter des 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;
```

Ajouter des flags :

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

Supprimer des flags :

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

Tests sur des 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) ...
```

Bit fields : implémentation

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

Bit fields : sélectionner une personne selon des critères

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


Bit fields : exemple d'utilisation

```
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 };

for (int 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

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 s'est bien passé
 - un code d'erreur, si quelque chose s'est mal passé

Errno (errno.h)

- En C (standards ANSI et POSIX), la variable globale `errno` est utilisée pour passer un code d'erreur
- Cette variable et les codes d'erreurs standards sont définis dans le fichier `errno.h`
- Exemples de codes standards 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`, `ssize_t`)
 - soit un pointeur
- **Par convention**, ces fonctions retournent en cas d'erreur :
 - soit `-1`
 - soit `NULL`

Convention du type de retour (2)

```
// Recherche des entrees dans une base de donnees
// Retourne:
// - soit le nombre de resultats trouves
// - soit -1 en cas d'erreurs
// Rempli 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 {
    for (int 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 caractères)

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

Afficher un message d'erreur (perror)

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

```
if (num < 0) {  
    // On suppose que le fichier passé en premier paramètre du programme 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 !

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

BIEN !

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

Exercices

Tracer des appels systèmes

Partie 1

- Créer un programme C qui lit un fichier donné en paramètre à votre programme (via `argc` et `argv`) :
 - le fichier doit être ouvert en lecture seule et en mode texte (via `fopen`)
 - le contenu du fichier doit être lu (via `fread`) et affiché à l'écran, caractère par caractère

Partie 2

- Utiliser la commande `strace` pour lister les appels systèmes effectués par votre programme :
 - identifier les appels systèmes correspondant au code de votre programme
 - à quoi correspond le code exécuté avant votre programme ?
 - identifier les *bits fields* utilisés
 - utiliser le manuel pour comprendre ces appels systèmes
 - identifier et observer les codes de retour de ces appels, notamment en cas d'erreurs
 - pourquoi n'y a-t-il qu'un appel `read` / `write` ?