Programmation système

Steve Alabi - Taha Bendjeddou - Younes Benyamna - Malek Zemni

M2 SeCReTs

21/11/2018

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

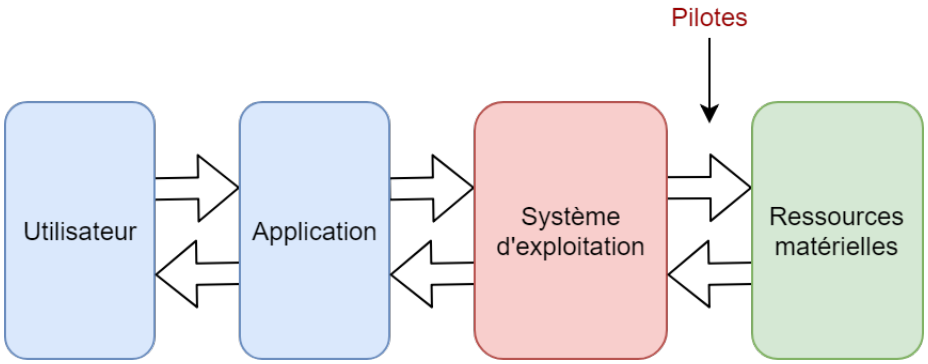
Table des matières

1 Introduction

1.1 Système d'exploitation

Un système d'exploitation (**SE** ou **OS** pour *operating system*) est un ensemble de programmes qui dirigent l'utilisation des capacités d'un ordinateur par des logiciels applicatifs (applications ou programmes directement utilisés par l'utilisateur). Il reçoit des demandes d'utilisation des capacités de l'ordinateur (capacité de stockage des mémoires et des disques durs, capacité de calcul du processeur, capacités de communication vers des périphériques ou via le réseau) de la part des logiciels applicatifs. Le système d'exploitation accepte ou refuse ces demandes, puis réserve les ressources en question pour éviter que leur utilisation n'interfère avec d'autres demandes provenant d'autres logiciels.

Lorsqu'un programme désire accéder à une ressource matérielle, il ne lui est pas nécessaire d'envoyer des informations spécifiques au périphérique, il lui suffit d'envoyer les informations au système d'exploitation, qui se charge de les transmettre au périphérique concerné via son pilote. En l'absence de pilotes il faudrait que chaque programme reconnaisse et prenne en compte la communication avec chaque type de périphérique.



Le système d'exploitation est donc chargé d'assurer la liaison entre les ressources matérielles, l'utilisateur et les applications. Il permet ainsi de dissocier les programmes et le matériel, afin notamment d'assurer la **gestion des ressources** et offrir à l'utilisateur une interface homme-machine notée IHM simplifiée lui permettant de s'affranchir de la complexité de la machine physique.

Rôles du système d'exploitation :

- ***Gestion du processeur*** **:** gère l'allocation du processeur entre les différents programmes avec un algorithme d'ordonnancement. Le type d'ordonnanceur est totalement dépendant du système d'exploitation, en fonction de l'objectif visé.
- ***Gestion de la mémoire vive*** **:** gère l'espace mémoire alloué à chaque application et, le cas échéant, à chaque usager. En cas d'insuffisance de mémoire physique, le système d'exploitation peut créer une zone mémoire sur le disque dur, appelée mémoire virtuelle. La mémoire virtuelle permet de faire fonctionner des applications nécessitant plus de mémoire qu'il n'y a de mémoire vive disponible sur le système. En contrepartie cette mémoire est beaucoup plus lente.
- ***Gestion des entrées/sorties*** **:** permet d'unifier et de contrôler l'accès des programmes aux ressources matérielles par l'intermédiaire des pilotes (appelés également gestionnaires de périphériques ou gestionnaires d'entrée/sortie).

— ***Gestion de l'exécution des applications*** **:** chargé de la bonne exécution des applications en leur affectant les ressources nécessaires à leur bon fonctionnement. Il permet à ce titre de «tuer» une application ne répondant plus correctement.

— ***Gestion des droits*** **:** chargé de la sécurité liée à l'exécution des programmes en garantissant que les ressources ne sont utilisées que par les programmes et utilisateurs possédant les droits adéquats.

— ***Gestion des fichiers*** **:** le système d'exploitation gère la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.

— ***Gestion des informations*** **:** le système d'exploitation fournit un certain nombre d'indicateurs permettant de diagnostiquer le bon fonctionnement de la machine.

Composantes du système d'exploitation :

Le système d'exploitation est composé d'un ensemble de logiciels permettant de gérer les interactions avec le matériel, parmi eux :

— ***Le noyau (kernel)*** **:** représentant les fonctions fondamentales du système d'exploitation telles que la gestion de la mémoire, des processus, des fichiers, des entrées-sorties principales, et des fonctionnalités de communication.

— ***L'interpréteur de commande (shell, traduisez "coquille" par opposition au noyau)*** **:** permettant la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes, afin de permettre à l'utilisateur de piloter les périphériques en ignorant tout des caractéristiques du matériel qu'il utilise, de la gestion des adresses physiques, etc.

— ***Le système de fichiers (en anglais «file system», noté FS)*** **:** permettant d'enregistrer les fichiers dans une arborescence.

Amorcage :

Le système d'exploitation est le premier programme exécuté lors de la mise en marche de l'ordinateur, après l'amorçage. L'amorçage ou le boot (*to pull oneself up by one's bootstrap*) est la procédure de démarrage d'un ordinateur qui comporte notamment le chargement du programme initial (le système d'exploitation).
Le problème posé par le boot est de faire démarrer un ordinateur et lui faire charger un programme alors que, a priori, il ne possède encore aucun programme dans sa mémoire. L'ordinateur exploite en fait un programme réduit, le chargeur d'amorçage, permettant d'extraire un programme accessible via un périphérique de stockage permanent ou amovible. Ce dernier est typiquement le **noyau du système d'exploitation**, qui s'installera en RAM et appellera lui-même des programmes applicatifs. On distingue :

— le démarrage à froid (*cold boot*) : allumer une machine éteinte

— le démarrage à chaud (*warm boot* ou *reboot*) : recharger le programme initial (sans coupure de l'alimentation électrique)

Au démarrage de l'ordinateur, la première chose qui s'affiche à l'écran est l'écran de boot. Cet écran varie beaucoup selon les ordinateurs parce qu'il dépend du matériel dont est constituée la machine. C'est en effet la carte mère qui affiche l'écran de boot. La carte mère est le composant fondamental de tout ordinateur, puisque c'est elle qui fait fonctionner le processeur, les disques durs, le lecteur de

CD-ROM, etc. Ensuite, le système d'exploitation se lance. C'est seulement une fois qu'il est chargé que l'on peut utiliser les programmes.

1.2 Programmation système

La ***programmation système*** est un type de programmation qui vise au développement de programmes qui font partie du système d'exploitation d'un ordinateur ou qui en réalisent les fonctions. Elle se distingue de la ***programmation des applications*** en ce qu'elle s'intéresse non pas au traitement des données, mais à la résolution des problèmes pour les humains, aux interfaces (API), aux protocoles (communication) et à la gestion des ressources.
En réalité, seuls les ***programmes d'application*** sont utilisés pas les utilisateurs. Les ***programmes système*** le sont implicitement. La programmation système inclut, en outre, l'accès aux fichiers, la gestion de la mémoire vive et des processeurs et la programmation de tous les périphériques qui font entrer ou sortir de l'information d'un ordinateur (clavier, écran, modems...). Elle permet donc de communiquer avec ces périphériques, créer des pilotes, voire même créer un système d'exploitation.

Programmation système en C sous UNIX :

La programmation système se fait généralement par le biais de langages tel que le langage ***assembleur***, et d'un langage de bas niveau. C'est le cas des systèmes d'exploitation de type **UNIX** (Linux, FreeBSD, Solaris...) dont 90% du code est écrit en ***langage C*** (qui a été spécialement créé pour le développement du système UNIX), le reste est écrit en assembleur suivant les architectures cibles (x86, SPARC...).

Les systèmes UNIX sont des systèmes d'exploitation qui sont constitués de plusieurs programmes, et chacun d'eux fournit un service au système. Tous les programmes qui fournissent des services similaires sont regroupés dans une ***couche logicielle***. Une couche logicielle qui a accès au matériel informatique s'appelle une ***couche d'abstraction matérielle***. Le ***noyau*** est une sorte de logiciel d'arrière-plan qui assure les communications entre ces programmes. C'est donc par lui qu'il va falloir passer pour avoir accès aux informations du système.

Pour accéder à ces informations du système, on utilise des fonctions qui permettent de communiquer avec le noyau. Ces fonctions s'appellent des ***appels-systèmes***. Le terme ***appel-système*** désigne l'appel d'une fonction, qui, depuis l'espace utilisateur, demande des services ou des ressources au système d'exploitation.
Chaque architecture matérielle ne supporte que sa propre liste d'appels-systèmes, c'est pourquoi les appels-systèmes diffèrent d'une machine à l'autre. Mais la plupart d'entre eux sont implémentés sur toutes les machines. Cependant, les programmes système restent très peu portables.

Pour des raisons de sécurité évidentes, les applications de l'espace utilisateur ne peuvent pas directement exécuter le code du noyau ou manipuler ses données. Par conséquent, un mécanisme de signaux a été mis en place. Quand une application exécute un appel-système, elle peut alors effectuer un ***trap*** (permutation de l'exécution de l'application en mode noyau), et peut exécuter le code, du moment que le noyau le lui autorise.

2 Gestion d'erreurs

Dans la programmation système, on manipule tout ce qui touche au système d'exploitation. Ainsi, on doit souvent faire face à des codes d'erreurs. La gestion des erreurs est donc un élément primordial dans la programmation système.

2.1 Le flux d'erreurs standard

Le flux d'erreurs standard `stderr` se modélise par un fichier dans lequel on peut écrire nos erreurs. Grâce à cette écriture séparée, l'utilisateur pourra rediriger le flux standard afin d'isoler les erreurs et ainsi mieux les traiter, en nombre moins importants, devant la masse d'informations contenues en sortie.

Les flux standards

Il existe 3 flux standards qui régissent l'utilisation d'un processus d'un programme écrit en C.

Les flux peuvent être représentés comme des canaux dans lesquels circulent des informations. Dans l'informatique moderne, ces flux sont représentés par l'abstraction de base du disque dur : le fichier. Ainsi, tout ce qui sera écrit dans un des flux sera écrit dans le fichier associé. À l'origine modélisés sur des systèmes d'exploitation de type UNIX , cette technique s'est largement généralisée auprès des systèmes d'exploitation modernes. D'une manière plus générale, on peut associer la notion de flux à d'autres flux plus renommés : flux RSS, flux de paquets, etc.

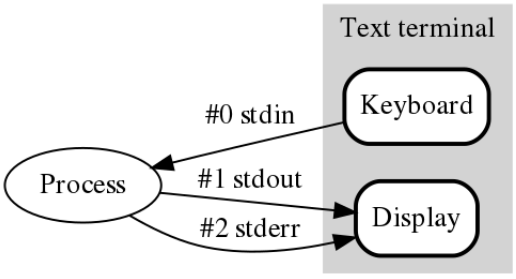
En programmation, l'utilité principale des flux réside dans les entrées/sorties. On distingue trois flux standards :

— le flux d'entrée standard `stdin`

— le flux de sortie standard `stdout`

— le flux de sortie d'erreurs standard `stderr`

Le fichier d'en-tête `<stdio.h>` déclare ces identificateurs comme étant de type pointeurs sur `FILE`.



Ces 3 descripteurs de fichiers sont initialisés au lancement du processus, et sont libérés à sa destruction. Il n'est donc pas nécessaire de les ouvrir et de les fermer comme on pourrait le faire avec des fichiers classiques.
Le flux d'entrée standard est tout ce qui est envoyé en entrée au programme sous la forme d'informations écrites au clavier en général. Toutefois, l'entrée standard peut prendre une forme différente dans

le cadre d'une redirection.
Le flux de sortie standard et le flux de sortie d'erreurs standard sont par défaut associés à la console. Il est possible de les isoler l'un de l'autre, afin de différencier messages d'informations, envoyés à la sortie standard, et messages d'erreurs ou avertissements, envoyés à la sortie d'erreurs standards.

Redirection d'un flux : l'utilité concrète des trois flux est attribuée à l'utilisateur. En effet, celui-ci peut rediriger un ou plusieurs des trois flux vers un fichier.

- La redirection du flux d'entrée standard peut permettre de soumettre à un programme une batterie de tests sans avoir à retaper les mêmes entrées.
- La redirection du flux de sortie standard peut permettre d'enlever tous les messages inutiles que certaines applications affichent.
- La redirection de flux de sortie d'erreurs standard peut permettre à l'utilisateur d'isoler les erreurs, et ainsi de les traiter pertinemment.

Parmi les techniques de redirection :

- `1>` redirige la sortie standard vers un fichier passé en paramètre à la suite du symbole
- `2>` redirige la sortie d'erreurs standards vers le fichier
- `<` redirige l'entrée standard vers un fichier

```c
#include <stdio.h>

int main(void) {
   printf("hello world\n");
   return 0;
}
```

```
$ gcc hello.c -o hello.out
$ ./hello.out 1> hello.txt
$ cat hello.txt
hello world !
```

2.2 La variable globale `errno`

Pour signaler une erreur, les fonctions renvoient une valeur spéciale, indiquée dans leur documentation. Celle-ci est généralement **-1** (sauf pour quelques exceptions). La valeur d'erreur alerte l'appelant de la survenance d'une erreur, mais elle ne fournit pas la description de ce qui s'est produit. La variable globale `errno` est alors utilisée pour en trouver la cause. Cette variable `errno` est donc le point de départ de la gestion des erreurs standard offerte par la bibliothèque standard de C.

```c
#include <errno.h>

extern int errno;
```

Sa valeur est valable uniquement juste après l'utilisation de la fonction que l'on veut tester. En effet, si on utilise une autre fonction entre le retour que l'on veut tester et l'exploitation de la variable de `errno`, la valeur de `errno` peut être modifiée entre temps. Elle contient donc une valeur correspondant au code de la dernière erreur s'étant produite.

Interprétation du contenu de `errno`

À chaque valeur d'erreur possible de `errno` correspond une constante du préprocesseur, dont la description est disponible dans le manuel (`man errno`).

La bibliothèque standard ne définit que trois codes d'erreur : ***EDOM*** (passage en paramètre en dehors du domaine attendu), ***ERANGE*** (résultat trop grand ou trop petit) et ***EILSEQ*** (erreur de transcodage). Cependant, les systèmes d'exploitation communs et actuels proposent souvent certaines extensions au contenu de `errno`. La norme ***POSIX*** définit par exemple une trentaine de constantes numériques supplémentaires.

Il est donc difficile d'associer directement le contenu de `errno` à des codes d'erreur, car la portabilité risque de nous faire défaut. C'est pourquoi la bibliothèque standard de C met à disposition deux fonctions qui interprètent le contenu de `errno`, évitant ainsi de passer par un code non standard, ou bien considérablement allongé.

La fonction `strerror` : associe au code d'erreur passé en paramètre une description de celui-ci.

```c
#include <string.h>

extern char* strerror(int errnum);
```

La fonction `perror` : plus utilisée et plus simple d'usage, associe à la valeur courante de `errno` sa description, l'affichant sur la sortie d'erreurs standard `stderr`. Il est également possible de placer un préfixe `s` devant cette description, que l'on pourra passer en paramètre.

```c
#include <stdio.h>

extern void perror(const char* s);
```

```c
if (fork() == -1) {
   perror("fork");
}
//Affiche "fork : Description de l'erreur"
```

3 Entrées / Sorties

Les entrées / sorties traitent des données qu'on peut lire ou écrire à partir de fichiers. Il existe 2 niveaux de gestion des entrées / sorties et fichiers :

— Entrées / sorties effectuées immédiatement (avec des appels-systèmes)

— Entrées / sorties où les données sont mises en mémoire temporaire (***buffer*** *ou mémoire tampon, zone de mémoire virtuelle utilisée pour stocker temporairement les données, notamment entre 2 processus*) (avec des fonctions standard)

3.1 Entrées / Sorties standard

Toutes les fonctions décrites ici sont déclarées dans la bibliothèque `stdio.h`.

Ouvrir et fermer un fichier :

```
FILE* fopen(const char* nomDuFichier, const char* modeOuverture);
// renvoie NULL si l'ouverture échoue

int fclose(FILE* pointeurSurFichier);
// renvoie 0 si la fermeture réussit, EOF sinon
```

Écrire dans un fichier :

```
int fputc(int caractere, FILE* pointeurSurFichier);
// écrit un seul caractère à la fois dans le fichier

char* fputs(const char* chaine, FILE* pointeurSurFichier);
// écrit une chaîne dans le fichier

int fprintf(FILE* pointeurDeFichier, const char *format, ...);
// écrit une chaîne formatée dans le fichier, fonctionnement quasi-identique
   à printf

// retournent EOF si écriture échoue
```

Lire dans un fichier :

```
int fgetc(FILE* pointeurDeFichier);
// lit un caractère, et avance la tête de lecture
// retourne le caractère lu ou EOF sinon

char* fgets(char* chaine, int nbreDeCaracteresALire, FILE*
   pointeurSurFichier);
// lit au maximum une ligne et s'arrête au premier \lstinline!\n!
// <nbreDeCaracteresALire> : pour s'arrêter le lire avant la fin de ligne,
   sert à définir une taille max
// retourne NULL si elle ne peut rien lire

int fscanf(FILE* pointeurDeFichier, const char *format, ...);
// écrit une chaîne formatée
```

Se déplacer dans un fichier : la tête de lecture est une sorte de curseur virtuel dans un fichier qui indique la position de lecture / écriture actuelle.

```
long ftell(FILE* pointeurSurFichier);
// indique la position actuelle dans le fichier

int fseek(FILE* pointeurSurFichier, long deplacement, int origine);
// déplace le curseur de <deplacement> caractères à partir de la position
   <origine>
```

```
// <deplacement> peut être positif (en avant), 0 ou négatif (en arrière)
// <origine> peut prendre les constantes SEEK_SET (début), SEEK_CUR
    (actuelle) ou SEEK_END (fin)

void rewind(FILE* pointeurSurFichier);
// retour au début
```

3.2 Entrées / Sorties système

Les fonctions d'entrée / sortie système sont des appels-systèmes communiquant directement avec le noyau. Les fonctions d'entrée / sortie standard font eux-même appel à ces fonctions système. Ces fonctions système se trouvent dans les bibliothèques suivantes :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

Ouvrir et fermer un fichier :

```
int open(const char* pathname, int flags, mode_t mode);
```

- **retour :** un "file descriptor" `fd` >= 0, prend la plus petite valeur disponible (non ouvert) ou **-1** si l'ouverture échoue, auquel cas errno contient le code d'erreur
- **`pathname` :** chemin du fichier à ouvrir
- **`flags` :** mode d'accès, contient une ou plusieurs valeurs dont obligatoirement une parmi O_RDONLY, O_WRONLY ou O_RDWR
- **`mode` :** indique les permissions à utiliser si un nouveau fichier est créé, doit être fourni lorsque O_CREAT est spécifié dans `flags` sinon ignoré

```
int close(int fd);
```

- **retour :** 0 ou -1 en cas d'échec
- **`fd` :** descripteur du fichier à fermer

Lire et écrire dans un fichier :

```
ssize_t read(int fd, void* buf, size_t count);
```

- **retour :** -1 en cas d'échec, et la position de la tête de lecture est indéfinie, sinon, renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre
- **`fd` :** descripteur du fichier à lire
- **`buf` :** pointe sur un tampon (buffer) où sont stockées les données lues
- **`count` :** nombre d'octets à lire, résultat indéfini si `count` est supérieur à `SSIZE_MAX`

```
ssize_t write(int fd, const void* buf, size_t count);
```

— **retour :** -1 en cas d'échec, sinon, renvoie le nombre d'octets écrits et avance la tête de lecture de ce nombre

— **`fd` :** descripteur du fichier dans lequel on va écrire

— **`buf` :** pointe sur un tampon (buffer) où sont stockées les données à écrire

— **`count` :** nombre d'octets à écrire

Se déplacer dans un fichier :

```
off_t lseek(int fd, off_t offset, int whence);
```

— **retour :** -1 en cas d'échec, sinon, renvoie le nouvel emplacement, mesuré en octets depuis le début du fichier

— **`offset` :** nombre d'octets de déplacement

— **`whence` :** SEEK_SET : la tête est placée à `offset` octets depuis le début du fichier, SEEK_CUR : la tête est avancée de `offset` octets, SEEK_END : la tête est placée à la fin du fichier plus `offset` octets

Exemples :

On peut réécrire une version personnalisée des fonctions standard d'entrée / sortie en utilisant des appels-systèmes.

```
int copie (char* fic1, char* fic2)
{
   int fd1 = open(fic1, O_RDONLY);
   int fd2 = open(fic2, O_WRONLY);

   int nb;
   char buf[1024];

   // tq nb != 0 et != -1
   while ( (nb = read(fd1, buf, 1024)) > 0 )
      if ( write(fd2, buf, nb) != nb )
         return -1;

   return 1;
}
```

```
int myGetChar (int fd)
{
   int retour;
   unsigned char c;

   if ( read(fd, &c, 1) == 1 )
      retour = c;
   else
      retour = EOF;

   return retour;
}
```

```c
int myGetCharBuffer (int fd)
{
   // on effectue un read une fois que tous les MAX caractères sont lus, et
   on mémorise la tête de lecture pour lire un caractère

   // variables static qui ne se réinitialisent pas à chaque appel de la
   fonction
   static unsigned char buf[MAX];
   static int nbCar = 0;   // nombre de caractères dans buf
   static char* p;   // tête de lecture

   int retour;

   // on fait un read si le buffer est vide
   if ( nbCarac == 0 ) {
      nbCarac = read(fd, buf, MAX);
      p = buf;
   }

   // si le buffer contient des caractères, on en lit le premier
   if ( nbCarac > 0)
      retour = *p;   // valeur à la tête de lecture
   else
      retour = EOF;  // plus rien à lire

   nbCarac--;
   p++;  // on avance la tête de lecture

   return retour;
}
```

4 Processus

5 Threads

Conclusion