Programmation système

Steve Alabi - Taha Bendjeddou - Younes Benyamna - Malek Zemni

M2 SeCReTS

22/11/2018

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

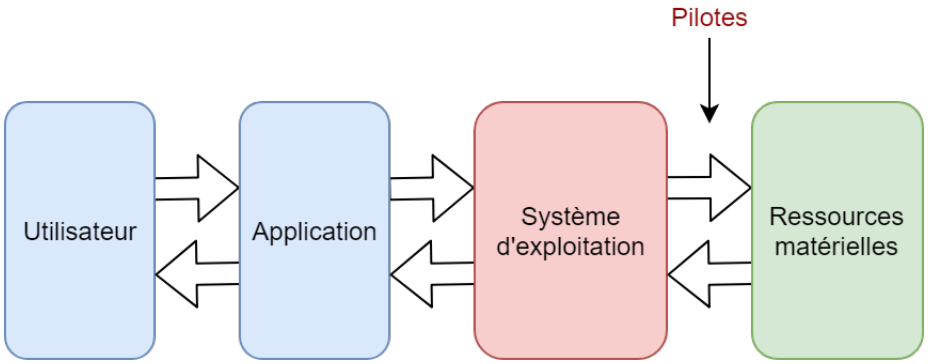
Table des matières

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.

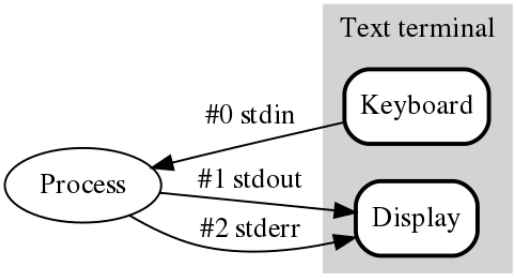
En programmation, l'utilité principale des flux réside dans les entrées/sorties. On distingue trois flux standards :

— 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 Les processus

4.1 Définitions

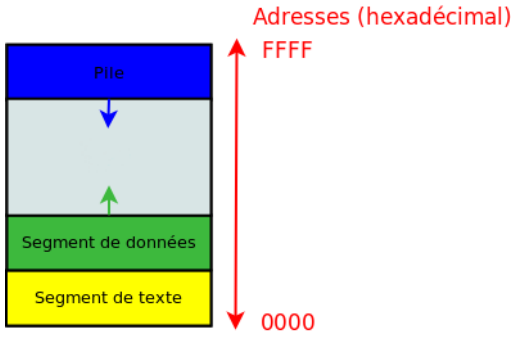
Un processus est programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire. En effet, au cours de son exécution, les instructions d'un programme modifient les valeurs des registres (compteur ordinal, registre d'état...) ainsi que le contenu de la pile. Un programme est une suite d'instructions (notion statique), tandis qu'un processus, c'est l'image du contenu des registres et de la mémoire centrale (notion dynamique).

Chaque processus possède un ***espace d'adressage***, c'est-à-dire un ensemble d'adresses mémoires dans lesquelles il peut lire et écrire. Cet espace est divisé en trois parties :

— le segment de texte (le code du programme)

— le segment de données (les variables)

— la pile



L'espace vide au milieu permet à la pile de s'étendre dessus de manière automatique et au segment de données de faire une extension lors d'une allocation dynamique.

Lorsqu'un processus est lancé, le système doit gérer la mémoire et l'allocation du processeur lui étant accordée. Il fait appel à l'***ordonnanceur***.
Un système d'exploitation est multitâche ***préemptif*** lorsqu'il peut arrêter à tout moment n'importe quelle application pour passer à la suivante (Windows XP, Windows 7 et GNU/Linux). Il garde donc le contrôle et se réserve le droit de fermer l'application.
Un système d'exploitation est multitâche ***coopératif*** quand il permet à plusieurs applications de fonctionner et d'occuper la mémoire, et leur laissant gérer cette occupation (Windows 95, 98 et Millénium).
Les systèmes basés sur UNIX sont tous des systèmes préemptifs.

4.2 Gestion des processus sous UNIX

Dans les systèmes basés sur UNIX particulièrement, les processus jouent un rôle très important. Le concept de processus a été mis au point dès les débuts de ce système : il a ainsi participé à sa gloire et à sa célébrité. Une des particularités de la gestion des processus sous UNIX consiste à séparer la création d'un processus et l'exécution d'une image binaire. Bien que la plupart du temps ces deux tâches sont exécutées ensemble, cette division a permis de nouvelles libertés quant à la gestion des tâches. Par exemple, cela permet d'avoir plusieurs processus pour un même programme.

Autrement dit, sous les autres systèmes d'exploitation (mis à part quelques exceptions), un processus est l'équivalent d'un nouveau programme, alors que sous UNIX ce n'est pas forcément le cas.
Ce principe, peu utilisé dans les autres systèmes, a survécu de nos jours. Alors que la plupart des systèmes d'exploitation offrent un seul appel-système pour exécuter un nouveau programme, UNIX en possède deux : `fork` et `exec`.

La commande `ps` permet d'afficher ses propres processus en cours d'exécution. Pour afficher tous les processus en cours d'exécution, on utilise l'option `aux` (a : processus de tous les utilisateurs, u : affichage détaillé, x : démons).

PID :

Chaque processus peut être identifié par son numéro de processus, ou PID (***Process IDentifier***). Un numéro de PID est unique dans le système : il est impossible que deux processus aient un même

PID au même moment.
Lorsque l'on crée un processus, on utilise une fonction qui permet de dupliquer le processus appelant. On distingue alors les deux processus par leur PID. Le processus appelant est alors nommé **processus père** et le nouveau processus **processus fils**. Quant on s'occupe du processus fils, le PID du processus père est noté **PPID** (***Parent PID***).

Par défaut, le noyau attribue un PID avec une valeur inférieure à 32768. Le 32768ème processus créé reçoit la plus petite valeur de PID libéré par un processus mort entre-temps. Cette valeur maximale peut être changée par l'administrateur en modifiant la valeur du fichier `/proc/sys/kernel/pid_max`. De plus, les PID sont attribués de façon linéaire. Par exemple, si 17 est le PID le plus élevé affecté, un processus créé à cet instant aura comme PID 18. Le noyau réutilise les PID de processus n'existant plus uniquement quand la valeur de `pid_max` est atteinte.

Organisation des processus :

Les processus sont organisés en ***hiérarchie***. Chaque processus doit être lancé par un autre (processus père et processus fils). La racine de cette hiérarchie est le programme initial.
En effet, le processus inactif du système (***System idle process*** processus que le noyau exécute tant qu'il n'y a pas d'autres processus en cours d'exécution) a le PID 0. C'est celui-ci qui lance le premier processus que le noyau exécute, le programme initial. Généralement, sous les systèmes basés sous UNIX, le programme initial se nomme `init`, et il a le PID 1.

Si l'utilisateur indique au noyau le programme initial à exécuter, celui-ci tente alors de le faire avec quatre exécutables, dans l'ordre suivant : `/sbin/init`, `/etc/init` puis `/bin/init`. Le premier de ces processus qui existe est exécuté en tant que programme initial. Si les quatre programmes n'ont pas pu être exécutés, le système s'arrête. Après son chargement, le programme initial gère le reste du démarrage : initialisation du système, lancement d'un programme de connexion... Il se charge également de lancer les démons. Un démon (***daemon***) est un processus qui est constamment en activité et fournit des services au système.

États d'un processus :

Un processus peut avoir plusieurs états :

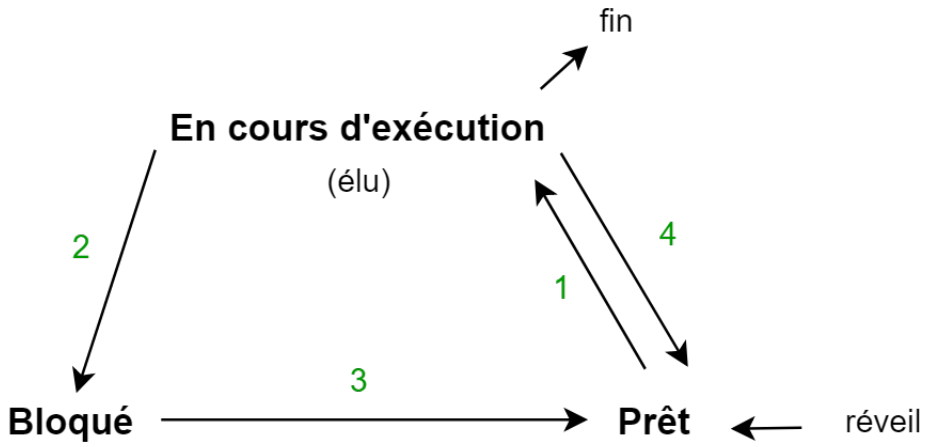
— exécution (R pour running) : le processus est en cours d'exécution

— sommeil (S pour sleeping) : dans un multitâche coopératif, quand il rend la main, ou dans un multitâche préemptif, quand il est interrompu au bout d'un quantum de temps

— arrêt (T pour stopped) : le processus a été temporairement arrêté par un signal, il ne s'exécute plus et ne réagira qu'à un signal de redémarrage

— zombie (Z pour ... zombie) : le processus fils (zombie) s'est terminé avant le père

Sous UNIX, un processus peut évoluer dans deux modes différents : le mode noyau et le mode utilisateur. Généralement, un processus utilisateur entre dans le mode noyau quand il effectue un appel-système.



— 1 : élection, l'ordonnanceur choisit ce processus

— 2 : blocage, processus bloqué en attente d'une donnée

— 3 : déblocage, donnée devient disponible

— 4 : l'ordonnanceur choisit un autre processus

Implémentation d'un processus :

Pour implémenter les processus, le système d'exploitation utilise un tableau de structure, appelé **table des processus**. Cette table comprend une entrée par processus, allouée dynamiquement, correspondant au processus associé à ce programme : c'est le bloc de contrôle du processus (***Process Control Block***, PCB). Ce bloc contient, entres autres, les informations suivantes :

— le PID, le PPID, l'UID (identifiant de l'utilisateur) et le GID (identifiant du groupe, chaque utilisateur du système appartient à un ou plusieurs groupes) du processus

— l'état du processus

— les fichiers ouverts par le processus

— le répertoire courant du processus

— le terminal attaché au processus

— les signaux reçus par le processus

— le contexte processeur et mémoire du processus (l'état des registres et des données mémoires du processus)

Grâce à ces informations stockées dans la table des processus, un processus bloqué pourra redémarrer ultérieurement avec les mêmes caractéristiques.

4.3 Manipulation des processus en C

4.3.1 Création d'un processus :

Pour créer un nouveau processus à partir d'un programme, on utilise la fonction système `fork`.

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

pid_t fork(void);
```

Le processus d'origine est le processus père et le nouveau processus créé est le processus fils, qui possède un nouveau PID. Les processus deux ont le même code source, mais la valeur retournée par `fork` nous permet de savoir si l'on est dans le processus père ou dans le processus fils. Ceci permet de faire deux choses différentes dans le processus père et le processus fils.
Lors de l'exécution de l'appel-système `fork`, le noyau effectue les opérations suivantes :

- alloue un bloc de contrôle dans la table des processus
- copie les informations contenues dans le bloc de contrôle du père dans celui du fils sauf les identificateurs (PID, PPID...)
- alloue un PID au processus fils
- associe au processus fils un segment de texte dans son espace d'adressage. Le segment de données et la pile ne lui seront attribués uniquement lorsque celui-ci tentera de les modifier. Cette technique, nommée copy on write, permet de réduire le temps de création du processus
- l'état du processus est mis à l'état exécution

La fonction `fork` retourne :

- -1 en cas d'erreur
- 0 si on est dans le processus fils
- le PID du fils si on est dans le processus père. Cela permet ainsi au père de connaître le PID de son fils

Autres fonctions :

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

pid_t getpid(void);
// retourne le PID du processus appelant

pid_t getppid(void);
// retourne le PPID du processus appelant
```

4.3.2 Terminaison d'un processus :

Un programme peut se terminer normalement de deux façons différentes. La plus simple consiste à laisser le processus finir le `main` avec l'instruction **`return`** suivie du code de retour du programme. Une autre façon est de terminer le programme grâce à la fonction `exit()` depuis n'importe quelle fonction.

```c
#include <stdlib.h>

void exit(status);
```

Un programme peut aussi se terminer de façon anormale, en cas d'erreur par exemple. Pour cela, on peut utiliser les fonctions `abort(void)` ou `assert(int condition)` par exemple.

4.3.3 Synchronisation entre processus :

Un processus **zombie** est un processus qui s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus. Lorsque le processus fils se termine avant le processus père (c'est à dire que le père n'a pas encore lu le code de retour du fils), le processus fils devient un zombie.

Pour permettre à un processus fils zombie de disparaître complètement, on utilise la fonction `wait()`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int* status);
```

Lorsque l'on appelle cette fonction, celle-ci bloque le processus à partir duquel elle a été appelée (père) jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID de ce dernier. En cas d'erreur, la fonction renvoie la valeur -1.
Le paramètre `status` correspond au code de retour du processus (ce code de retour est généralement indiqué avec la fonction exit).
Il faut veiller à mettre autant de `wait` dans le père qu'il a de processus fils.

Sans la fonction `wait`, le processus père s'arrête sans avoir lu le code de retour de son fils qui va devenir (ou rester s'il est déjà terminé) zombie. La seule manière d'éliminer ce processus zombies est de causer la mort du processus père (si ce n'est pas déjà fait). Les processus fils sont alors automatiquement rattachés au processus au PID 1, généralement `init`, qui se charge à la place du père original d'appeler `wait` sur ces derniers. Si ce n'est pas le cas, cela signifie que `init` est défaillant (ou que le processus 1 n'est pas `init`, mais un autre programme n'ayant pas été prévu pour ça) et le seul moyen de se débarrasser des zombies, dans ce cas, est le redémarrage du système.

La fonction `waitpid` : permet de suspendre l'exécution d'un processus père jusqu'à ce qu'un de ses fils, dont on doit passer le PID en paramètre, se termine.

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

pid_t waitpid(pid_t pid, int* status, int options);
\\waitpid(-1, status, 0) correspond à wait
```

- si pid $>$ 0, le processus père est suspendu jusqu'à la fin d'un processus fils dont le PID est égal à la valeur pid
- si pid $=$ 0, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils appartenant à son groupe
- si pid $=$ -1, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils
- si pid $<$ -1, le processus père est suspendu jusqu'à la mort de n'importe lequel de ses fils dont le GID est égal
- `status` a le même rôle qu'avec `wait`

— `options` permet de préciser le comportement de `waitpid` : WNOHANG (ne pas bloquer si aucun fils ne s'est terminé), WUNTRACED (recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue) ou 0.

Exemple :

La variable `quiSuisJe` va être modifiée pour le processus fils seulement. La position de `stdout` va avancer de 15 caractères pour le `printf` du père et 15 autres caractères pour le `printf` du fils pour valoir 30 au final pour les deux processus.
Les variables du processus père et celles du fils sont totalement distinctes. Par contre, leur descripteurs de fichiers sont les mêmes. Donc, si l'un des deux processus modifie son pointeur de position dans un fichier, ça se répercutera également chez l'autre. Cela ne vaut que pour les descripteurs de fichiers hérités durant le `fork`, c'est-à-dire, si le père ou le fils ouvre d'autres fichiers après le `fork`, ces descripteurs ne seront pas partagés entre eux deux. De même, si le fils ferme un descripteur de fichier hérité du père, le descripteur de fichier du père ne sera par contre pas fermé (même chose dans le sens inverse).

On ne peut pas savoir quel processus va s'exécuter en premier. Pour la fonction `wait`, il est préférable l'utiliser ainsi :

```
1 if ( wait(NULL) == -1 )
2    perror("wait");
```

5 Les threads

5.1 Definitions

Un **thread** ou **fil (d'exécution)** ou **tâche** (aussi processus léger) représente, comme un processus, l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un thread de contrôle unique, le thread principal. Du point de vue programmation, ce dernier exécute le main.
En général, le système réserve un processus à chaque application, sauf quelques exceptions. Beaucoup de programmes exécutent plusieurs activités en parallèle, du moins en pseudo-parallélisme. Comme à l'échelle des processus, certaines de ces activités peuvent se bloquer, et ainsi réserver ce blocage à un seul thread séquentiel, permettant par conséquent de ne pas stopper toute l'application.
Le principal avantage des threads par rapport aux processus, c'est la facilité et la rapidité de leur création. En effet, tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Cela évite donc l'allocation de tous ces espaces lors de la création, et il est à noter que, sur de nombreux systèmes, la création d'un thread est environ cent fois plus rapide que celle d'un processus.
Au-delà de la création, la superposition de l'exécution des activités dans une même application permet une importante accélération quant au fonctionnement de cette dernière.

6 Manipulation des thread en C

Compilation : toutes les fonctions relatives aux threads sont incluses dans le fichier d'en-tête `<pthread.h>` et dans la bibliothèque `libpthread.a`. Il faut donc utiliser l'option `-lpthread` à la compilation.

6.1 Création et suppression de threads

Créer un thread :

Pour créer un thread, il faut déjà déclarer une variable le représentant. Celle-ci sera de type `pthread_t\lstinline`. Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

```
#include <pthread.h>

int pthread_create(pthread_t* thread, pthread_attr_t * attr, void*
    (*start_routine) (void*), void* arg);
```

- retour : 0 si la création a été réussie ou une autre valeur si il y a eu une erreur
- `thread` : pointeur vers l'identifiant du thread à créer
- `attr` : attributs du thread, possible de mettre le thread en état joignable (NULL par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...)
- `(*start_routine) (void*)` : pointeur vers la fonction à exécuter dans le thread, de la forme `void* fonction(void* arg)` et contiendra le code à exécuter par le thread
- `arg` : l'argument à passer au thread

Supprimer un thread :

```
#include <pthread.h>

void pthread_exit(void* ret);
```

Prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

Attendre la fin d'un thread :

Le thread principal n'attendra pas que le thread créé termine de s'exécuter. On la fonction `pthread_join` pour cela.

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
d{itemize}
e prend en premier paramètre l'identifiant du thread et son second
    paramètre, un pointeur, permet de récupérer la valeur retournée par la
    fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de
    \lstinline!pthread_exit!).

bsection*{Exemple :}
```

```
gin{lstlisting}
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void* thread_1(void *arg)
{
   printf("Nous sommes dans le thread.\n");

   /* Pour enlever le warning */
   (void) arg;
   pthread_exit(NULL);
}

int main(void)
{
   pthread_t thread1;

   printf("Avant la création du thread.\n");

   if (pthread_create(&thread1, NULL, thread_1, NULL))
   {
      perror("pthread_create");
      return EXIT_FAILURE;
   }

   // on attend que le thread s'exécute
   if (pthread_join(thread1, NULL))
   {
      perror("pthread_join");
      return EXIT_FAILURE;
   }

   printf("Après la création du thread.\n");

   return EXIT_SUCCESS;
}

// Affiche uniformément :
// Avant la création du thread.
// Nous sommes dans le thread.
// Après la création du thread.
```

6.2 Exclusions mutuelles - mutex

Avec les threads, toutes les variables sont partagées (mémoire partagée). Cela pose des problèmes quand deux threads cherchent à modifier deux variables en même temps. Les mutex est mécanisme de synchronisation, un des outils permettant l'exclusion mutuelle.

Un **mutex** en C est une variable de type `pthread_mutex_t`. Elle va nous servir de verrou, pour nous permettre de protéger des données. Ce verrou peut donc prendre deux états : disponible et verrouillé. Quand un thread a accès à une variable protégée par un mutex, on dit qu'il tient le mutex. Il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

Le mutex soit accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale.

Initialiser un mutex :

On initialise un mutex avec la valeur de la constante `PTHREAD_MUTEX_INITIALIZER`, déclarée dans `pthread.h`.

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Verrouiller un mutex :

L'étape suivante consiste à établir une **zone critique**, c'est-à-dire la zone où plusieurs threads risquent de modifier ou de lire une même variable en même temps. On verrouille donc le mutex pour éviter ce risque.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mut);
```

Déverrouiller un mutex :

À la fin de la zone critique, il suffit de déverrouiller le mutex.

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mut);
```

Détruire un mutex :

Une fois le travail du mutex terminé, on peut le détruire.

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mut);
```

Les conditions :

Lorsqu'un thread doit patienter jusqu'à ce qu'un événement survienne dans un autre thread, on utilise les conditions.
Quand un thread est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre thread.
Comme avec les mutex, on déclare la condition en variable globale :

```c
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un mutex :

```c
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t
    *nomMutex);
```

Pour réveiller un thread en attente d'une condition, on utilise la fonction :

```c
int pthread_cond_signal(pthread_cond_t *nomCondition);
```

Exemple :

Créez un code qui crée deux threads : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de la
    condition */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du mutex */

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
   pthread_t monThreadCompteur;
   pthread_t monThreadAlarme;

   pthread_create (&monThreadCompteur, NULL, threadCompteur, (void*)NULL);
   pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL); /*
    Création des threads */

   pthread_join (monThreadCompteur, NULL);
   pthread_join (monThreadAlarme, NULL); /* Attente de la fin des threads */

   return 0;
}

void* threadCompteur (void* arg)
{
   int compteur = 0, nombre = 0;

   srand(time(NULL));

   while(1) /* Boucle infinie */
   {
      nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
      compteur += nombre; /* On ajoute ce nombre à la variable compteur */

```

```
36          printf("\n%d", compteur);
37
38          if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
39          {
40             pthread_mutex_lock (&mutex); /* On verrouille le mutex */
41             pthread_cond_signal (&condition); /* On délivre le signal :
    condition remplie */
42             pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */
43
44             compteur = 0; /* On remet la variable compteur à 0 */
45          }
46
47          sleep (1); /* On laisse 1 seconde de repos */
48       }
49
50       pthread_exit(NULL); /* Fin du thread */
51    }
52
53    void* threadAlarme (void* arg)
54    {
55       while(1) /* Boucle infinie */
56       {
57          pthread_mutex_lock(&mutex); /* On verrouille le mutex */
58          pthread_cond_wait (&condition, &mutex); /* On attend que la condition
    soit remplie */
59          printf("\nLE COMPTEUR A DÉPASSÉ 20.");
60          pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
61       }
62
63       pthread_exit(NULL); /* Fin du thread */
64    }
```