Chapitre 1: Introduction – Systèmes d'exploitation

Christophe Morvan

Université Paris-est, Marne-la-Vallée

8 septembre 2015

Préambule

Ce cours est inspiré du CS 140 de Stanford (donné par D. Mazière $\rightarrow$ 2011)
Il se repose également sur Systèmes d'exploitation de A. Tanenbaum

Préambule

Ce cours est inspiré du CS 140 de Stanford (donné par D. Mazière $\rightarrow$ 2011)
Il se repose également sur Systèmes d'exploitation de A. Tanenbaum

Les cours tendent principalement vers la théorie et les techniques de *conception* de systèmes d'exploitation

Préambule

Ce cours est inspiré du CS 140 de Stanford (donné par D. Mazière → 2011)
Il se repose également sur Systèmes d'exploitation de A. Tanenbaum

Les cours tendent principalement vers la théorie et les techniques de *conception* de systèmes d'exploitation

Les TD/TP couvrent principalement *l'utilisation* des appels systèmes donc essentiellement la programmation système.
Deux ingrédients complémentaires.

Cours de système

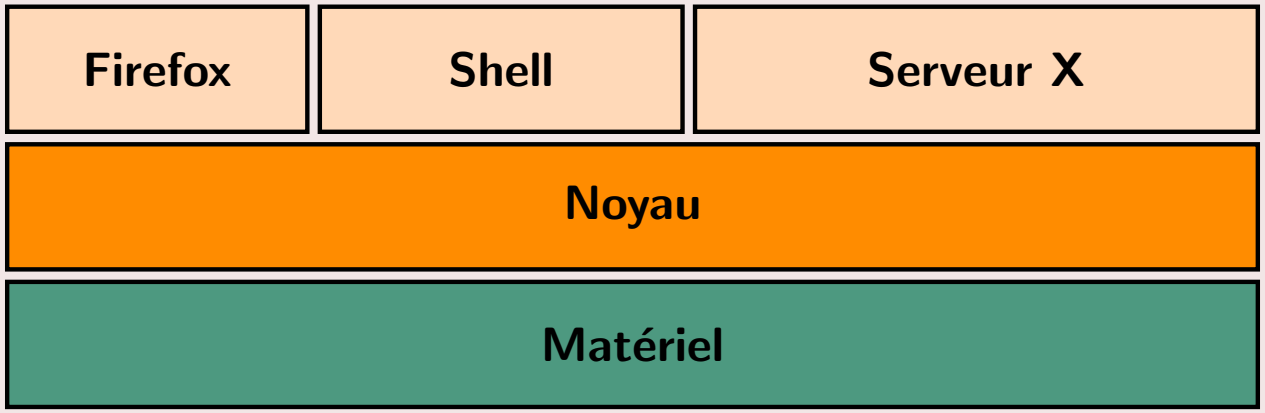
- **Threads & Processus**
- **Ordonnancement**
- **Disques et systèmes de fichiers**
- **Concurrence & Synchronisation**
- **Mémoire**
- **Entrées/Sorties**
- **Résau (sockets)**
- **Note : La plupart du temps le cours s'appuiera sur Unix**
  - La plupart des OS actuels et futurs sont fortement influencés par Unix
  - Windows est une exception qu'on ignorera pour l'essentiel

Objectifs

- **Introduire les principes définissant les OS**
  - Difficile d'interagir avec un ordinateur sans passer par l'OS
  - Comprendre le fonctionnement de l'OS permet d'être un meilleur programmeur
- **Comprendre les mécanismes essentiels des systèmes**
  - Gestion de cache, la concurrence, gestion mémoire, I/O, protection
- **Être confronté à des problèmes et des solutions mises en œuvre dans des systèmes réels**

Qu'est-ce qu'un système d'exploitation (OS) ?

Couche entre les applications et le matériel



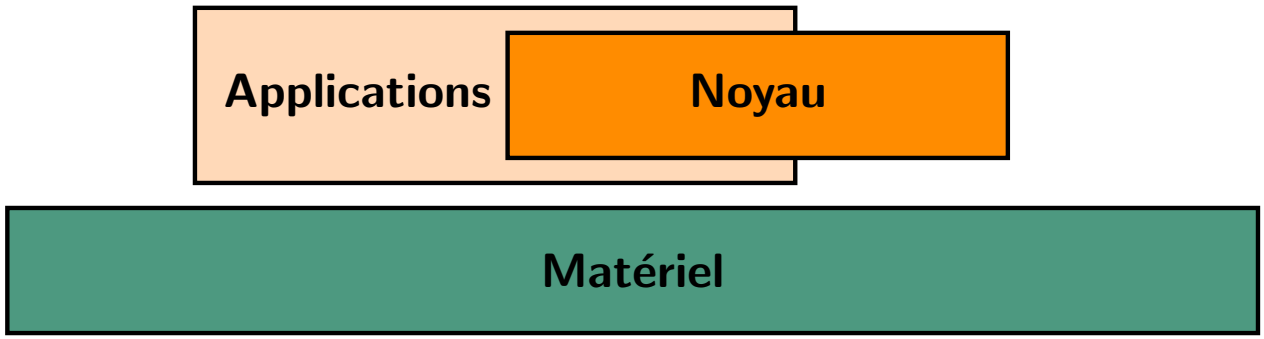
- **Facilite la programmation**
- **[La plupart du temps] Fournit des abstractions pour la programmation**
  - Gère et dissimule les détails du matériel
  - Interdit aux applications les accès de bas niveau sur le matériel.
- **[Souvent] Définit des mécanismes de protection**
  - Interdit à un utilisateur/processus d'en pénaliser un autre

Pourquoi étudier les systèmes ?

- **Les OS forment un domaine en évolution**
  - La plupart des gens utilisent peu d'OS
  - Il est difficile de faire changer d'OS les utilisateurs
  - Il est difficile pour un nouvel OS de sortir du lot
- **Les serveurs haute performance sont confronté à des problèmes de système**
- **La gestion des ressources est un problème de système**
  - Batterie, performance radio, etc.
- **La sécurité est un problème système**
  - Impossible d'avoir de la sécurité sans des fondations sécurisés
- **De plus en plus d'objets intelligents ont besoin d'OS**
- **Les navigateurs sont confrontés à des problèmes d'OS**

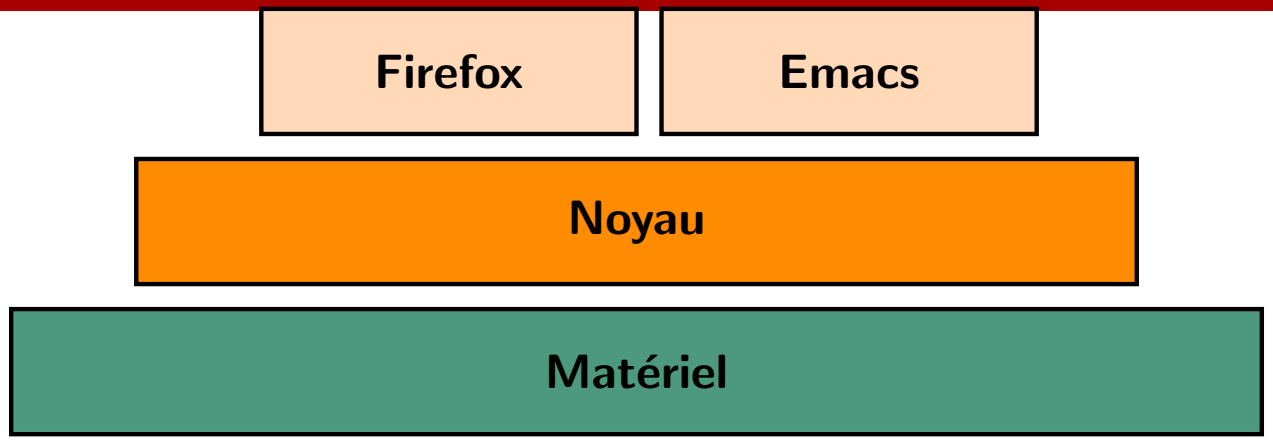
Systèmes primitifs

- **Simple bibliothèque de services standards [pas de protection]**



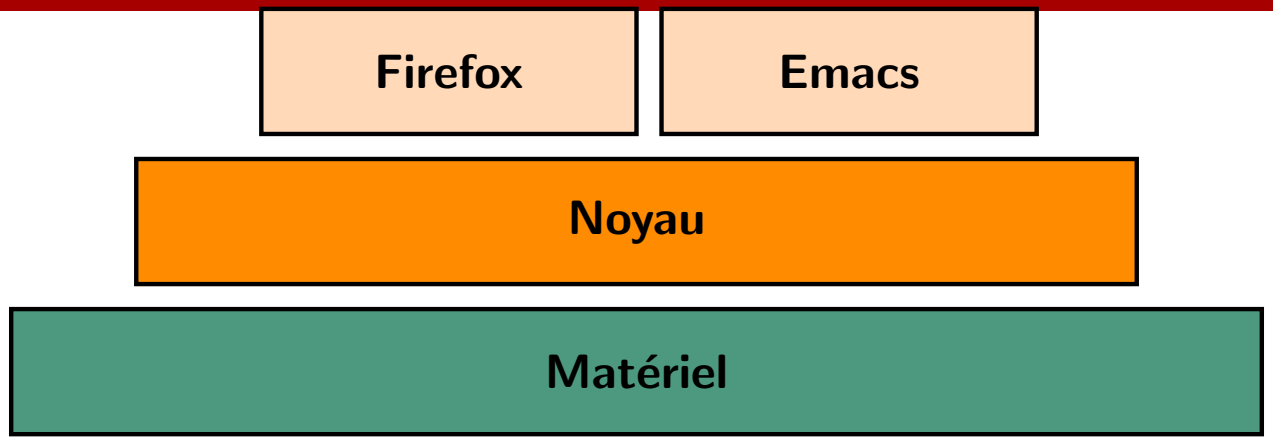
  - Interface standard au dessus de pilotes de périphériques
- **Simpliste**
  - Le système ne fait tourner qu'un programme à la fois
  - Pas d'utilisateur ou de programme malveillant (!)
- **Problème : faible utilisation**
  - . . . du matériel (le processeur est inutilisé lorsqu'il attend un disque)
  - . . . de l'utilisateur (qui doit attendre la fin de chaque programme de façon séquentielle)

Multi-tâches



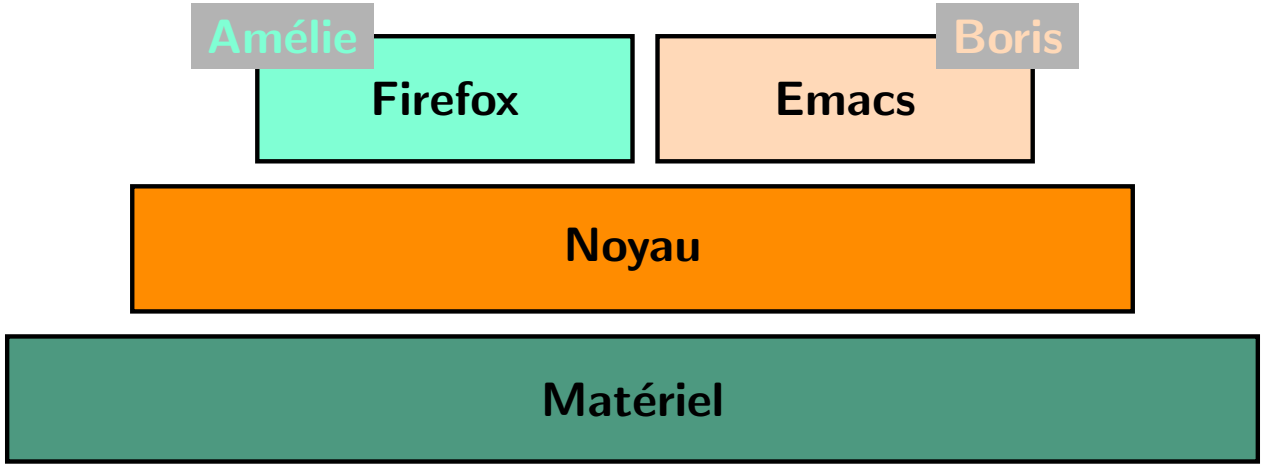
- **Idée : faire tourner plusieurs processus en parallèle**
  - Dès qu'un processus est bloqué (en attente du disque, du réseau, de l'utilisateur, etc.) faire tourner un autre processus
- **Problème : que peut faire un processus malveillant ?**

Multi-tâches



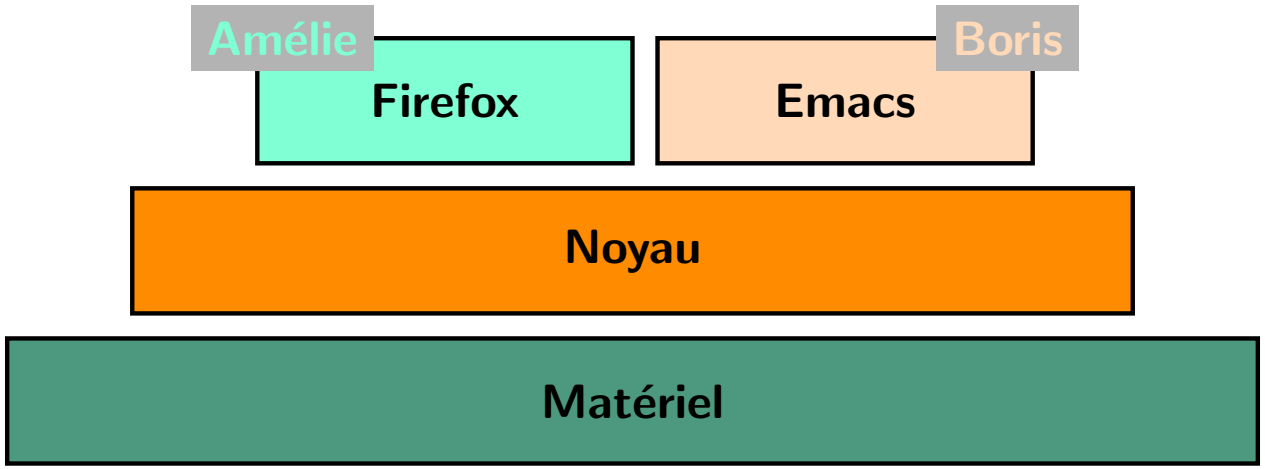
- **Idée : faire tourner plusieurs processus en parallèle**
  - Dès qu'un processus est bloqué (en attente du disque, du réseau, de l'utilisateur, etc.) faire tourner un autre processus
- **Problème : que peut faire un processus malveillant ?**
  - Entrer dans une boucle infinie et ne jamais libérer le processeur
  - Écrire dans les données d'autres programmes pour provoquer des défaillances
- **Les OS apportent des réponses à ces problèmes**
  - *Préemption* – libère le processeur d'une boucle infinie
  - *Protection mémoire* – protège la mémoire d'un processus vis-à-vis des autres

Systèmes multi-utilisateurs



- **Beaucoup d'OS mettent en place un système de protection vis-à-vis des utilisateurs malveillants**
- **Idée clé : N utilisateurs $\not\Rightarrow$ système N fois plus lent**
  - Les utilisateurs demandent proc, memoire, etc. par a-coups
  - Donner les ressources aux utilisateurs qui en ont le plus besoin
- **Problèmes ?**

Systèmes multi-utilisateurs

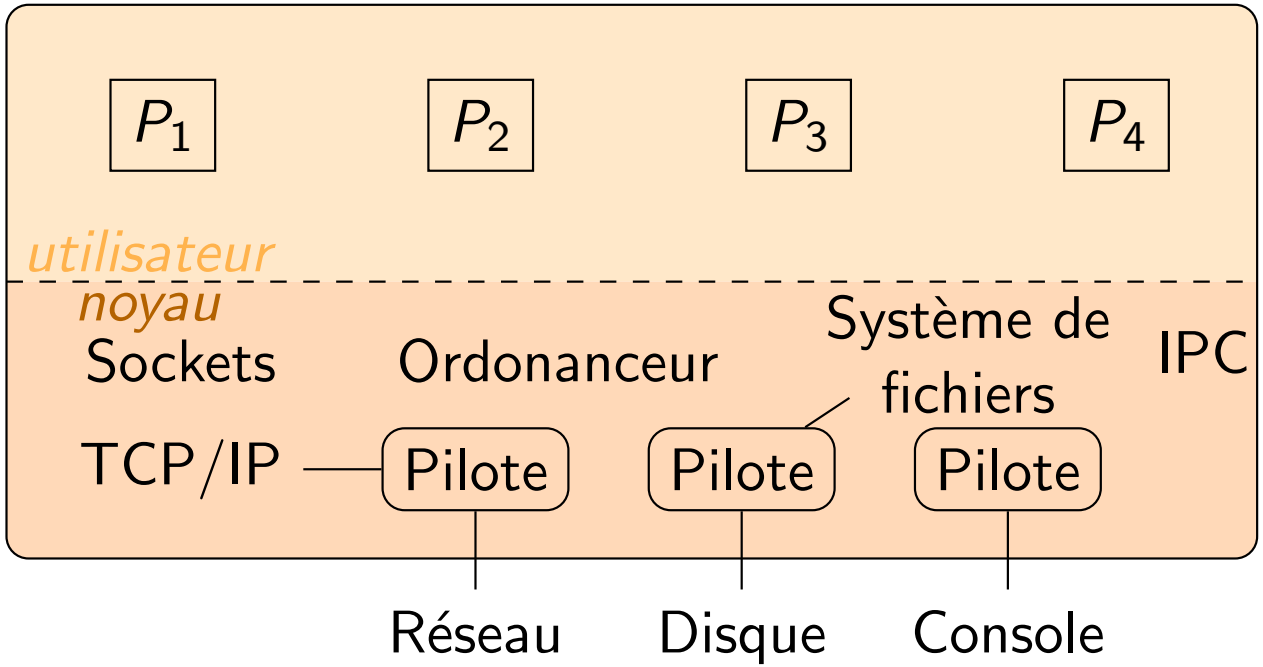


- **Beaucoup d'OS mettent en place un système de protection vis-à-vis des utilisateurs malveillants**
- **Idée clé : N utilisateurs $\not\Rightarrow$ système N fois plus lent**
  - Les utilisateurs demandent proc, memoire, etc. par a-coups
  - Donner les ressources aux utilisateurs qui en ont le plus besoin
- **Problèmes ?**
  - Les utilisateurs sont gourmands, utilisent trop de ressources. (définir des règles)
  - Le système peut utiliser plus de ressources que réellement disponible (virtualisation)
  - Effet d'effondrement dans certaines situations

La protection

- **Mécanisme qui permet d'isoler les éléments malicieux**
- **Préemption :**
  - Donner un ressource à une application la récupérer lorsque c'est nécessaire
- **Interposition :**
  - Intercaler l'OS entre les applications et le matériel
  - Conserver un catalogue d'autorisation pour chaque application (une table)
  - Pour chaque accès vérifier l'autorisation
- **Pour le processeur 2 modes : privilégié/non-privilégié**
  - Applications : mode non-privilégié (*user mode*)
  - OS : mode privilégié (*supervisor mode*)
  - Les opérations liées à la protection faite en mode privilégié

Organisation typique d'un OS



- **La plupart des soft sont des processus utilisateur**
- **Le noyau (kernel) de l'OS fonctionne en mode privilégié**
  - Crée et détruit les processus
  - Fournit l'accès au matériel
    (En 2009 56 % du code du noyau Linux)

Rappel de langage C – 1

Organisation de fichiers C

Deux types de fichiers :

- En-tête : `fichier.h`
  Ce fichier contient les signatures de fonctions du fichier " `.c`"
- Implémentation : `fichier.c`
  Contient le code des différentes fonctions. Un ou plusieurs fichiers d'en-tête peuvent être inclus à l'aide de la directive `#include`
  Peut également contenir des directives `#define` qui permette de définir des *macros*.

Rappel de langage C – 1

Organisation de fichiers C

Deux types de fichiers :

- En-tête : `fichier.h`
  Ce fichier contient les signatures de fonctions du fichier " `.c`"
- Implémentation : `fichier.c`
  Contient le code des différentes fonctions. Un ou plusieurs fichiers d'en-tête peuvent être inclus à l'aide de la directive `#include`
  Peut également contenir des directives `#define` qui permette de définir des *macros*.

Compilation d'un programme en C

Schéma général :

```
gcc -o executable fichier.c
./executable
```

Rappel de langage C – 2

Compilation séparée en C

Compilation de la "bibliothèque" :

```
gcc -c -o bibli.o biblio.c
gcc -o executable fichier.c bibli.o
./executable
```

Rappel de langage C – 2

Compilation séparée en C

Compilation de la "bibliothèque" :

```
gcc -c -o bibli.o biblio.c
gcc -o executable fichier.c bibli.o
./executable
```

Pointeurs

Rappel de la syntaxe :

```
int * myPointer ; // pointeur sur entier
myPointer = & myVar; // Adresse de myVar
```

Allocation de mémoire :

```
myPointer = (int*) malloc(n* (sizeof int));
// Espace pour "n" entiers
```

Bibliothèque standard du C

stdio.h

```
int printf(const char *format, ...)
int scanf(const char *format, ...)
int getchar(void)
int putchar(int char)
```

Bibliothèque standard du C

stdio.h

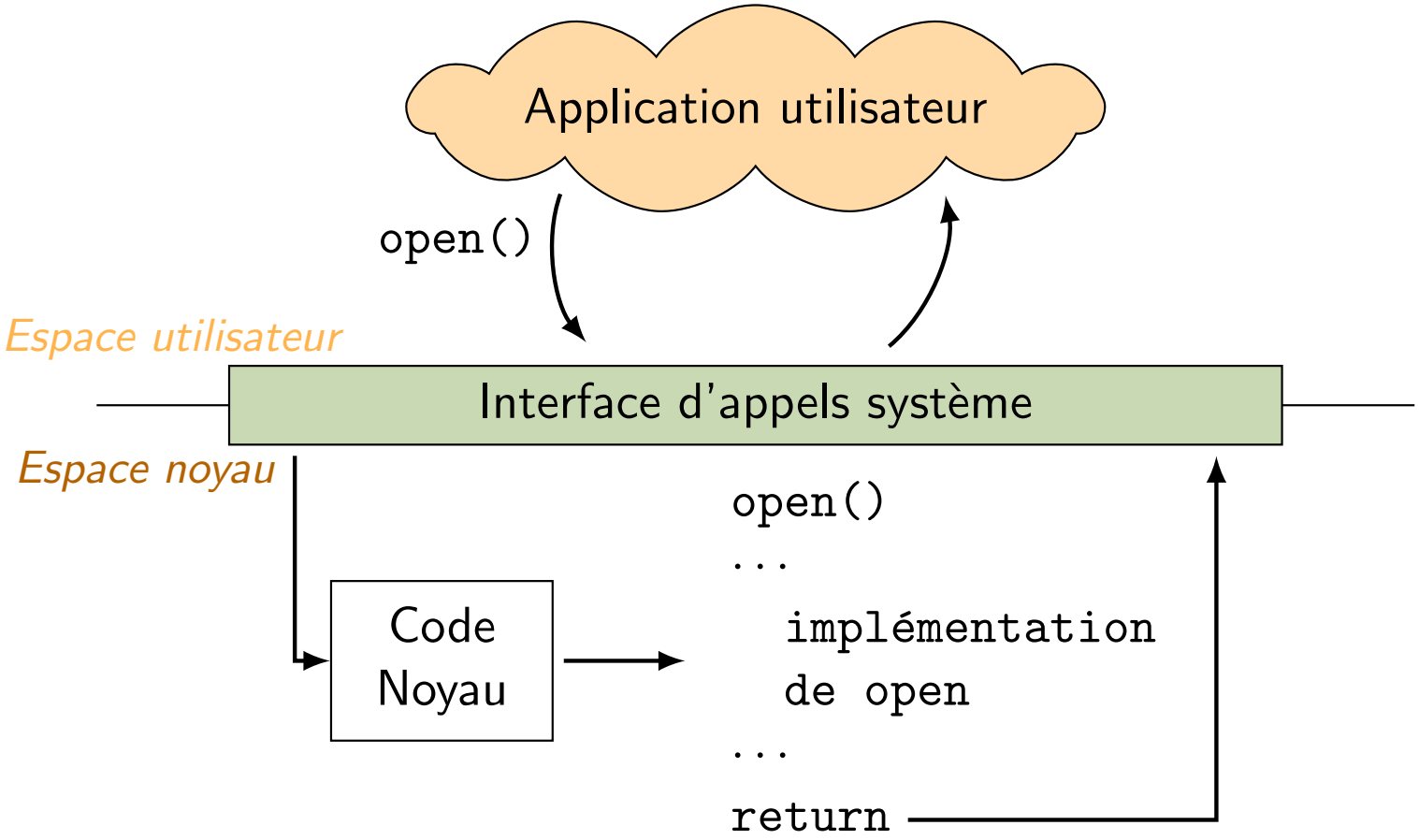
```
int printf(const char *format, ...)
int scanf(const char *format, ...)
int getchar(void)
int putchar(int char)
```

string.h

```
char *strncat(char *dest, const char *src, size_t n)
int strncmp(const char *str1, const char *str2, size_t n)
char *strncpy(char *dest, const char *src, size_t n)
```

Les fonctions de `string.h`, ci-dessus, intègrent une limite `n` et permettent une meilleure sureté de fonctionnement

Appels système – 1

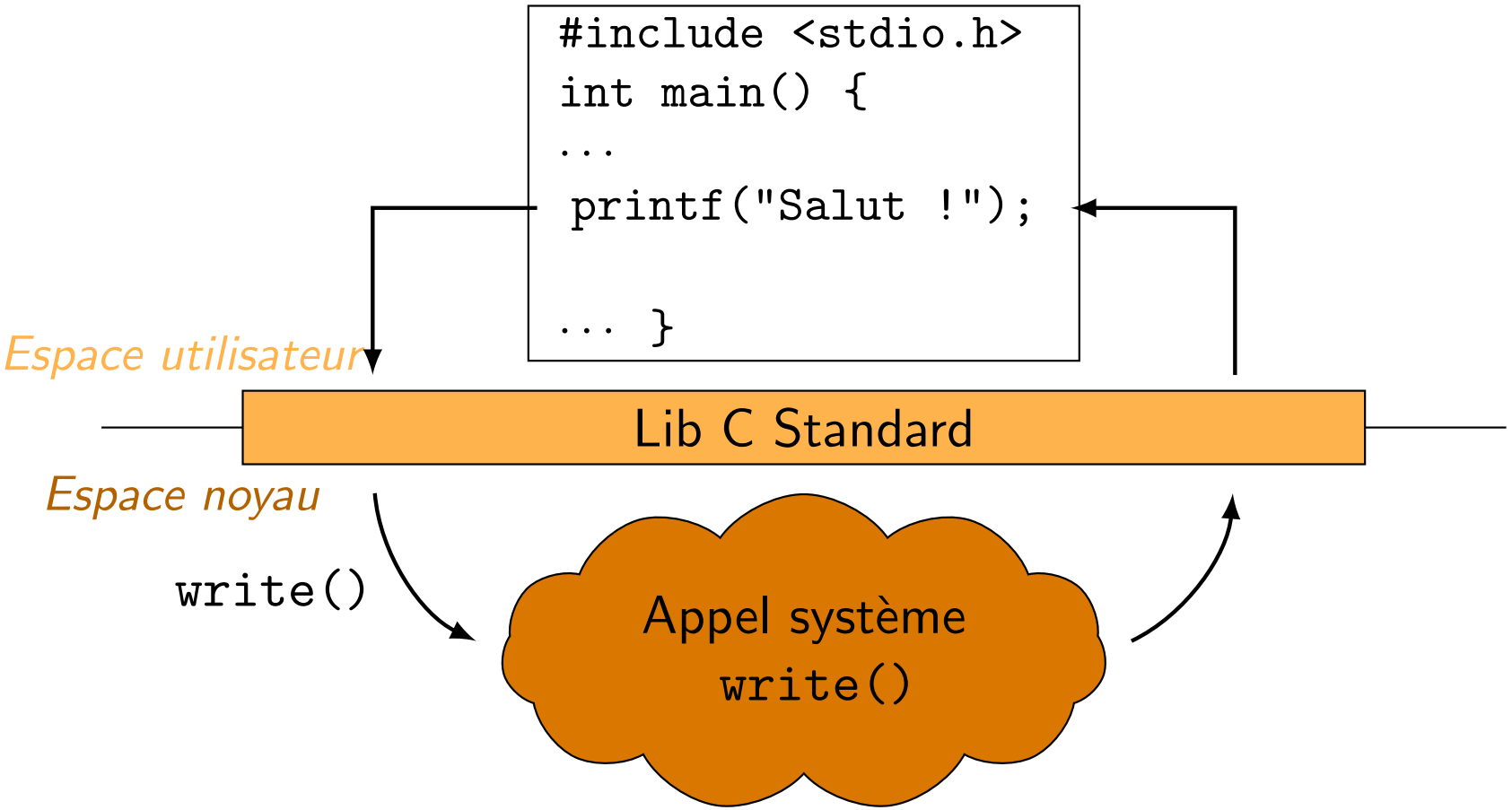


- **Les app. invoquent le noyau par des appels système**
  - Instructions spéciales qui donnent la main au noyau
  - . . . qui possède le code pour l'exécuter

Appels système – 2

- **Objectif : réaliser ce que les app. ne peuvent faire en mode non-privilégié**
  - Comme un appel de bibliothèque, mais avec passage en mode privilégié
- **Le noyau offre une interface bien définie d'appels système**
  - L'application assemble les arguments de l'appel puis les passe au noyau
  - Le noyau effectue l'opération et délivre le résultat
- **De nombreuses fonctions de haut niveau utilisent des appels systèmes**
  - `printf`, `scanf`, `gets`...
- **Exemple : norme POSIX**
  - `open`, `close`, `read`, `write`, `...`

Exemple d'appel système



- **La bibliothèque standard est implémentée avec des appels systèmes**
  - *printf* – (libc) même privilèges que l'application
  - appelle *write* – mode noyau → octets sur un port série

Appels système POSIX

Exemple : l'appel open

Permet d'ouvrir des fichiers/périphériques par leur nom

Les entrées/sorties passent par des fichies ouverts

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

Fournit comme résultat un descripteur (entier) utilisé pour accéder au fichier

`flags` :

- `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- `O_CREAT` : création éventuelle du fichier
- `O_EXCL` : (avec `O_CREAT`) provoque l'échec si le fichier existe déjà
- `O_TRUNC` : permet d'écrire dans un fichier (en le vidant)
- `O_APPEND` : écrit à la fin d'un fichie

`mode` : avec `O_CREAT`, définit les droits d'accès (`S_IRWXU`, `S_IWUSR`, ...)

Appels système POSIX

Exemple : l'appel open

Permet d'ouvrir des fichiers/périphériques par leur nom

Les entrées/sorties passent par des fichies ouverts

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

Fournit comme résultat un descripteur (entier) utilisé pour accéder au fichier

`flags` :

- `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- `O_CREAT` : création éventuelle du fichier
- `O_EXCL` : (avec `O_CREAT`) provoque l'échec si le fichier existe déjà
- `O_TRUNC` : permet d'écrire dans un fichier (en le vidant)
- `O_APPEND` : écrit à la fin d'un fichie

`mode` : avec `O_CREAT`, définit les droits d'accès (`S_IRWXU`, `S_IWUSR`, ...)

Attention : il s'agit de droits *maxima* : `umask` limite, éventuellement, ces droits.

Codes d'erreur

- **Que se passe-t-il en cas d'échec de `open` ? Fournit -1 (mauvais descripteur)**
- **La plupart des appels systèmes fournissent -1 en cas d'échec**
  - Code d'erreur spécifique dans la variable globale int `errno`
- `#include <sys/errno.h>` $\rightarrow$ **valeurs possibles**
  - 2 = `ENOENT` "Aucun fichier ou dossier de ce nom"
  - 13 = `EACCES` "Accès interdit"
- **La fonction `perror` affiche des messages lisibles**
  - `perror ("Mon texte");`
    $\rightarrow$ "`Mon texte No such file or directory`"

Quelques opérations sur les descripteurs de fichier

- `int read (int fd, void *buf, int nbytes);`
  - Fournit comme résultat le nombre de octets lu
  - Fournit 0 à la fin du fichier, ou -1 si erreur
- `int write (int fd, void *buf, int nbytes);`
  - Fournit comme résultat le nombre de octets écrits
  - -1 en cas d'erreur
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence` :
    - `SEEK_SET` Se place à `pos` octets dans le fichier
    - `SEEK_CUR` Se place à `pos` octets après la position actuelle
    - `SEEK_END` Se place à `posoctets` après la fin du fichier
  - Fournit comme résultat la position courante décalage mesuré en octets depuis le début du fichier
- `int close (int fd);`

Valeurs de descripteurs

- **Les descripteurs de fichiers sont hérités par les processus**
  - Lorsqu'un processus donne naissance à un nouveau processus, il hérite des descripteurs
- **Les descripteurs 0, 1, et 2 ont un sens particulier**
  - 0 – "entrée standard" (`stdin` en C ANSI)
  - 1 – "sortie standard" (`stdout`, `printfen` C ANSI)
  - 2 – "erreur standard" (`stderr`, `perroren` C ANSI)
  - Les trois sont attachés au terminal
- **Exemple :** `type.c`
  - affiche le contenu d'un fichier sur `stdout`

type.c

Programme exemple

```
void diplayfile (char *filename)
{
  int fd, nread;
  char buf[1024];
  fd = open (filename, O_RDONLY);
  if (fd == -1)
    {
      perror (filename);
      return;
    }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}
```

Documentation sur les appels systèmes

L'essentiel de la doc est accessible dans les pages manuel sous linux
En section 2
Précise les fichiers à inclure (très variables)

Exemple

```
$ man 2 open
NAME
open, creat - open and possibly create a file or device
SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Contextes système différents

- **En général un système peut être dans plusieurs contextes**
- **Niveau utilisateur – fait tourner une application**
- **Niveau noyau**
  - Exécutant du code noyau pour un processus (par exemple exécutant un appel système)
  - Traitant une exception (défaut de mémoire, exception numérique...)
  - Exécutant un processus noyau (ex : serveur de fichier réseau)
- **Code noyau non-associé à un processus**
  - Interruptions d'horloges
  - Interruptions de périphériques
- **Code de changement de contexte**

Changement de contexte

- **Utilisateur $\rightarrow$ contexte processus noyau : appel système, défaut de page**
- **Utilisateur/contexte processus $\rightarrow$ gestionnaire d'interruption : matériel**
- **Contexte processus $\rightarrow$ utilisateur/changement de contexte : return**
- **Contexte processus $\rightarrow$ changement de contexte : sleep**
- **Changement de contexte $\rightarrow$ contexte utilisateur/processus**

Préemption processeur

Mécanisme de protection évitant de monopoliser le processeur

- **Ex. le noyau programme des minuteurs d'interruption toutes les 10 ms**
- **De façon plus générale**
  - Le noyau programme des interruptions
  - Récupère la main à intervalle régulier
  - Donne l'accès au processeur à celui qui en a besoin
  - Nécessaire d'être en mode superviseur
  - Impossible pour le code utilisateur de détourner le gestionnaire d'interruption
- **Résultat : Impossible de garder le processeur avec une boucle infinie**
  - Au pire on obtient $1/N$ du temps processeur lorsqu'il y a N processus "gourmands"

La protection n'est pas la sécurité

- **Comment peut-on monopoliser le processeur ?**

La protection n'est pas la sécurité

- **Comment peut-on monopoliser le processeur ?**
- **Utiliser de nombreux processus**
- **Jusqu'à une période récente ce type de code pouvait planter n'importe quel OS :**

```
int main() { while(1) fork(); }
```

  - Crée des processus jusqu'à saturer la table du système hôte
- **Autres techniques : saturer la mémoire**
- **Résolu par la combinaison de plusieurs approches**
  - Technique : Limiter le nombre de processus par utilisateur
  - Sociale : Rebooter et invectiver les utilisateurs indélicats
  - Sociale : Mettre en place des lois (idée discutable)

Translation d'adresses

- **Protège la mémoire d'un processus**
- **Définitions**
  - *Espace d'adressage* : ensemble de la mémoire qu'un prog. peut adresser
  - *Adresse virtuelle* : adresses dans l'espace d'adressage du processus
  - *Adresse physique* : adresse dans la mémoire physique
  - *Translation* : association d'une adresse virtuelle à une adresse réelle
- **la translation est faite pour chaque lecture/écriture**
  - Les processeurs modernes ont du matériel pour faire ça
- **Principe : si tu ne peux pas l'adresser, tu ne peux pas y toucher**

  - Assure que les adresses physiques d'un processus n'a aucune intersection avec celles d'un autre

Plus de protection mémoire

- **Les processeurs permettent des espaces d'adressage virtuel réservé au noyau**
  - en générale le noyau accède tout les espaces d'adressages par exemple pour répondre aux appels système
  - Mais les applications ne doivent pas modifier la mémoire noyau
- **Les processeurs permettent au noyau de contrôler l'accès à la mémoire virtuelle**
  - Intercepte et arrête des programmes mal écrits qui font des accès fantaisistes
  - Mémoire virtuelle supérieure à la mémoire réelle (ex. ne charge une page sur le disque que lorsqu'elle est réellement accédée)
- **Les processeurs permettent de désigner des espaces en lecture seule**
  - Ex. partager de la mémoire entre processus
  - Autres optimisations noyaux

Allocation de ressources & performance

Le fonctionnement multi-tâches autorise une meilleure utilisation des ressources

Exemple

- Un processus qui effectue un téléchargement passe l'essentiel de son temps à attendre le réseau
- Il est possible de jouer tout en poursuivant le téléchargement
- Bien meilleure utilisation du processeur que le téléchargement seul

Le changement de contexte induit de la complexité

Supposons qu'un disque soit 1000 fois plus lent que la mémoire

- 1 Go de ram
- 2 Processus souhaite fonctionner en utilisant 1 Go
- Possible d'échanger les processus en passant par le disque
- Beaucoup plus rapide de fonctionner *séquentiellement*

Propriétés intéressantes à exploiter

- **Biais**
  - 80% du temps utilisé par 20% du code
  - 10% de la mémoire contient 90% des références
  - C'est l'astuce derrière le cache : placer 10% dans la mémoire rapide, 90% dans la lente, $\Rightarrow$ très grande mémoire rapide
- **Le passé prédit le futur**
  - Quel est la meilleure donnée du cache à remplacer ?
  - Si passé = futur, alors l'entrée la moins récemment accédée
- **Il y a un conflit entre l'équité et la performance**
  - Meilleure performance (pas de défaut de cache...) $\rightarrow$ faire tourner le même processus
  - Mais l'équité impose la préemption qui libère le processeur et le donne à un autre processus