IF 219 – Programmation Systèmes

Laurent Réveillère

Université de Bordeaux

Laurent.Reveillere@u-bordeaux.fr

http://www.reveillere.fr/INPIF219/

Programmation système

- Objectifs
  - Principes généraux des systèmes Unix
  - Point de vue du programmeur
    - Communication avec le système
    - Étudier les couches les plus basses d'un logiciel
    - Initiation à la programmation système
- Organisation
  - Enseignements intégrés sur machine
  - Supports de cours disponibles en ligne (à la fin)

Bienvenue en système !

- Comprendre et savoir programmer des ordinateurs

Pourquoi apprendre le système ?

- Par curiosité
  - Mieux comprendre ce qui se passe sous le capot
- Parce que cela peut être utile
  - Partage de ressources entre plusieurs applications
  - Besoin de parallélisme
  - Besoin de synchronisation
  - Besoin de performance
  - …

Bienvenue en informatique !

- Comprendre et savoir programmer des logiciels

Rôle d'un système d'exploitation

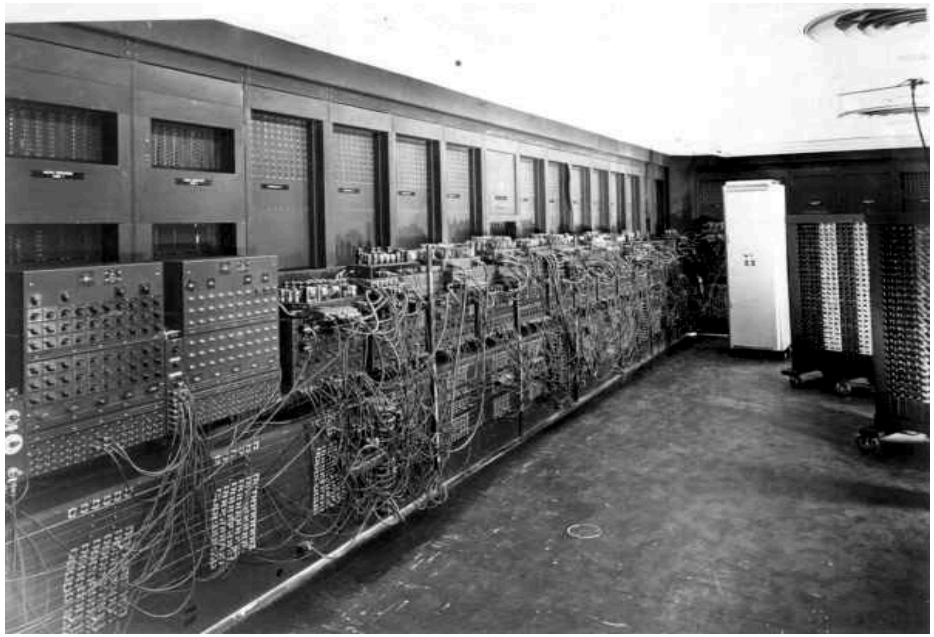
- Simplifier la vie du concepteur de logiciels
  - Masquer l'hétérogénéité du matériel
  - Masquer les détails de mise en œuvre de bas niveau
  - Offrir des abstractions pertinentes et de haut niveau pour développer des logiciels

I. Qu’est ce qu’un ordinateur ?

IF 219
Programmation Système

Définition d'un ordinateur

- Machine électronique capable d'exécuter des instructions effectuant des opérations sur des nombres



1946 : ENIAC

- *calculateur à tubes*
- *30 tonnes, 72m²*
- *357 mult/s*
- *35 div/s*

En panne la moitié du temps

Définition d'un ordinateur

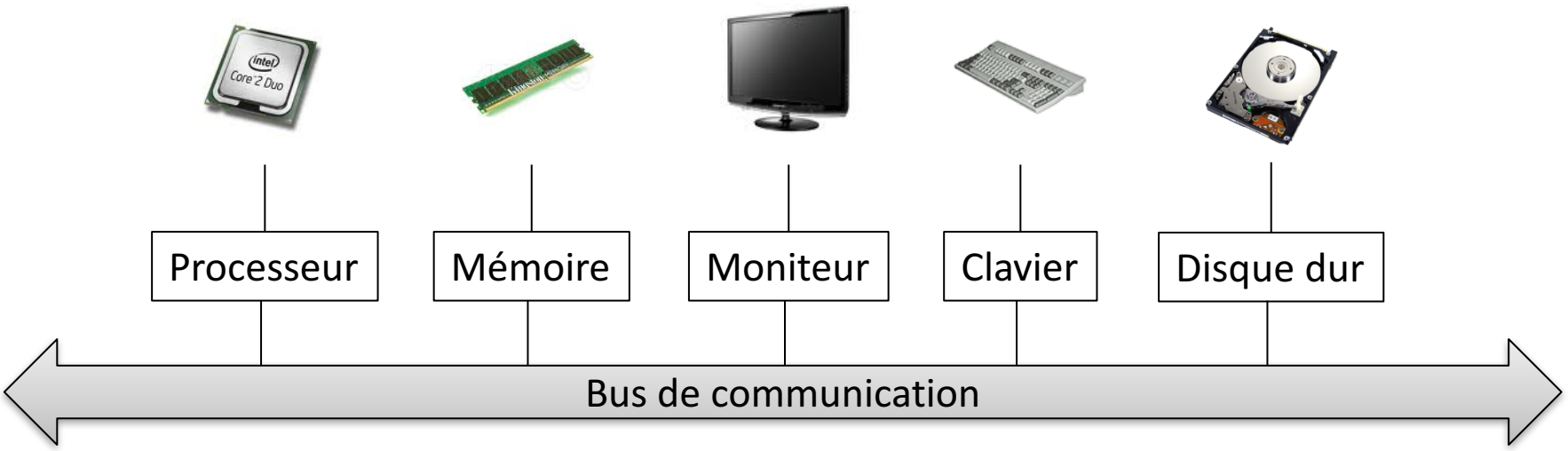
- Machine électronique capable d'exécuter des instructions effectuant des opérations sur des nombres

Janv 1948 : SSEC (premier ordinateur chez IBM) avec une capacité mémoire de 150 nombres

Définition d'un ordinateur

- Machine électronique capable d'exécuter des instructions effectuant des opérations sur des nombres

Structure matérielle d'un ordinateur



RAPPEL

Schéma de haut niveau d'un ordinateur

- **Processeur** : unité capable d'effectuer des calculs
- **Mémoire vive** : matériel stockant des données directement accessibles par le processeur
  - Accès rapide, données perdues en cas de coupure électrique.
- **Périphériques** : matériel fournissant ou stockant des données secondaires
  - Réseau, disque dur, souris, clavier, carte graphique, carte son…
- **Bus de communication** : bus interconnectant le processeur, la mémoire vive et les périphériques

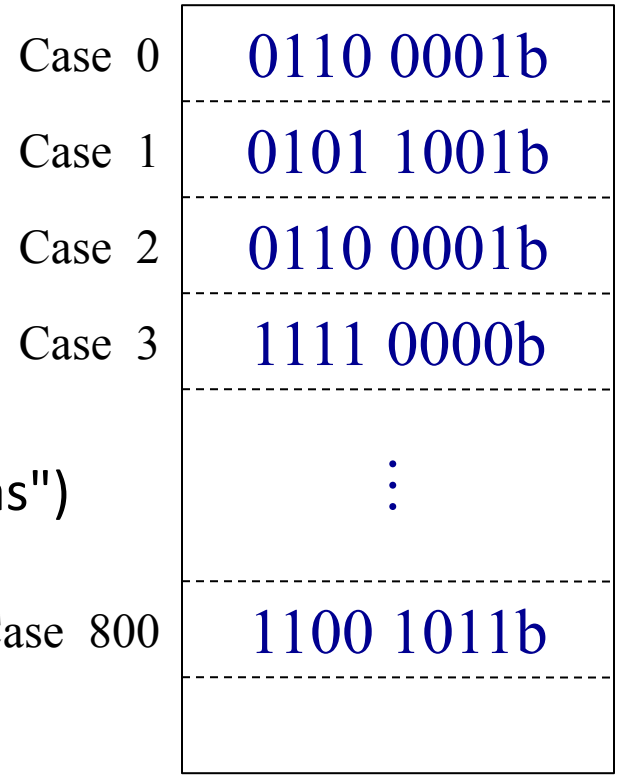
Qu'est ce que la mémoire vive ?

- Mémoire vive : ensemble de cases numérotées contenant des octets

- Une case contient un octet (*byte* en anglais) = regroupe 8 bits

- Bit : valeur valant 0 ou 1
  - 0 : bit non chargé ("courant ne passe pas")
  - 1 : bit chargé ("courant passe")

- Un octet permet de représenter $2^8 = 256$ valeurs



Représentation des nombres

- Notation décimale : un chiffre peut prendre 10 valeurs de 0 à 9

$$276 = 2*10^2 + 7*10^1 + 6*10^0$$

- Notation binaire : un chiffre peut prendre 2 valeurs de 0 à 1

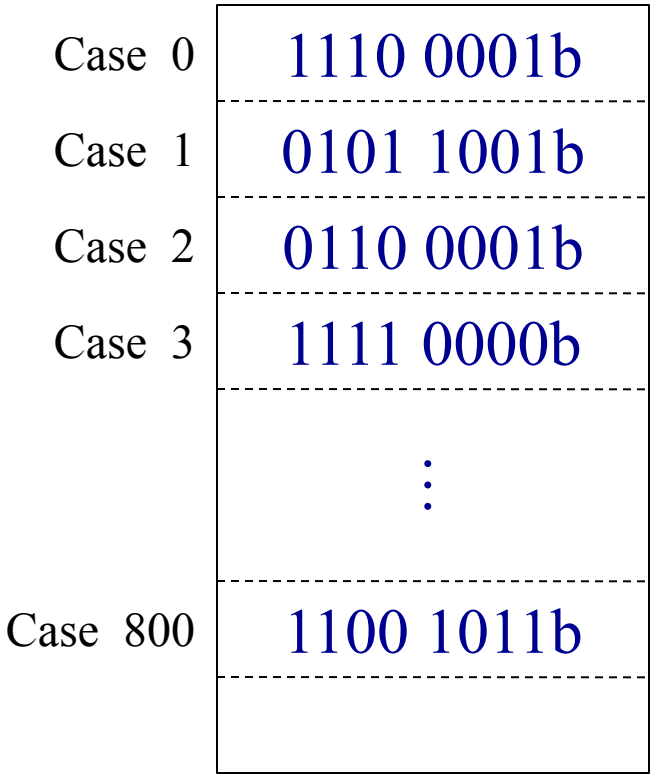
$$1101b = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 13$$

- Notation hexadécimale : un chiffre peut prendre 16 valeurs de 0 à f

$$0x276 = 2*16^2 + 7*16^1 + 6*16^0 = 630$$
$$0xb6 = 11*16^1 + 6*16^0 = 182$$

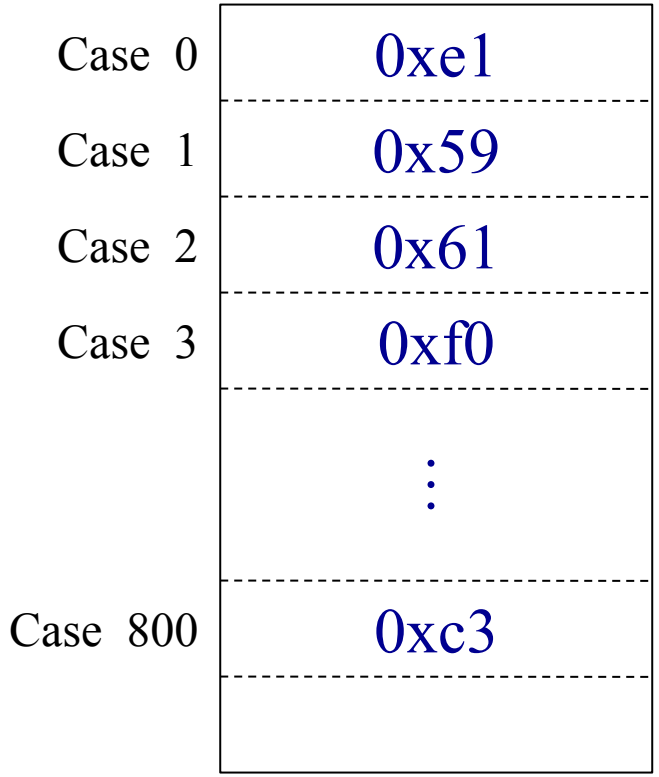
Hexadécimal en informatique

- Avec 4 bits, on encode 16 valeurs, soit 1 chiffre hexadécimal



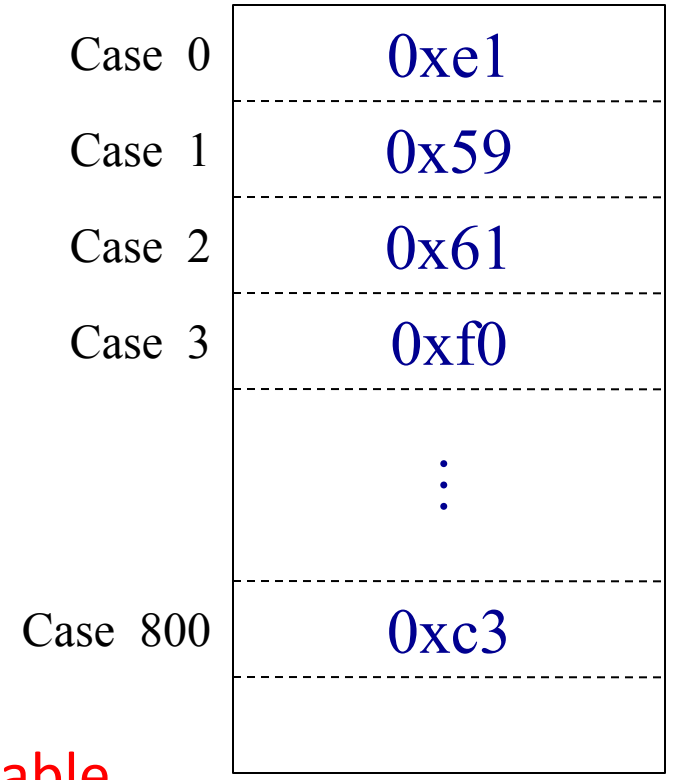
Hexadécimal en informatique

- Avec 4 bits, on encode 16 valeurs, soit 1 chiffre hexadécimal
- L'hexadécimal est donc plus concis pour représenter les valeurs des octets
- Un octet est représenté par 2 chiffres hexadécimaux



Que représentent les octets ?

- Une série d'octets peut représenter :
  - Un entier naturel
  - Un entier relatif
  - Une suite de caractères
  - Une valeur de vérité (vrai/ faux)
  - Un nombre flottant
  - Un nombre complexe
  - Une instruction machine
  - Ou tout autre ensemble énumérable



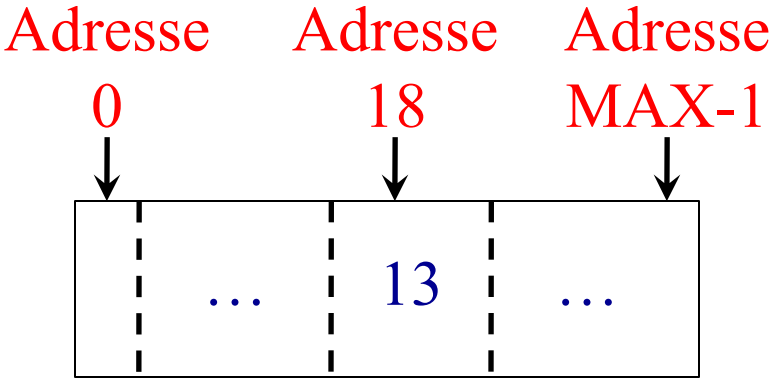
Fonctionnement d'un processeur

- Un processeur exécute des instructions qui peuvent
  - Effectuer des calculs
  - Accéder à la mémoire
  - Accéder aux autres périphériques
  - Sélectionner l'instruction suivante à exécuter (saut)

- Le processeur identifie une instruction par un numéro
  (Par exemple : 1 = additionne, 2 = soustrait, etc.)

Fonctionnement d'un ordinateur

- Mémoire : tableau d'octets, indexé à partir de 0



- Processeur : possède des variables internes appelées registres
  - PC (*Program Counter*) : adresse de l'instruction suivante

    Si PC == 18, alors l'instruction suivante à exécuter est l'instruction 13
  - Autres : registres sur lesquels le processeur effectue les calculs

Fonctionnement d'un ordinateur

- À chaque cycle d'horloge, le processeur :
  - Charge l'instruction à l'adresse PC à partir de la mémoire
  - Place le PC sur l'instruction qui suit
  - Sélectionne le circuit à activer en fonction du numéro d'instruction
  - Exécute l'instruction

- Quelques exemples d'instructions
  - 0x10 0x4000 ⇒ charge l'octet à l'adresse 0x4000 dans le registre nommé R0 (lit une variable)
  - 0x12 0x89 ⇒ ajoute 0x89 à PC (saut)
  - 0x14 0x20 ⇒ ajoute 0x20 au registre R0 (calcul)
  - 0x17 0x70 0x12 ⇒ envoie 0x70 au périphérique 0x12

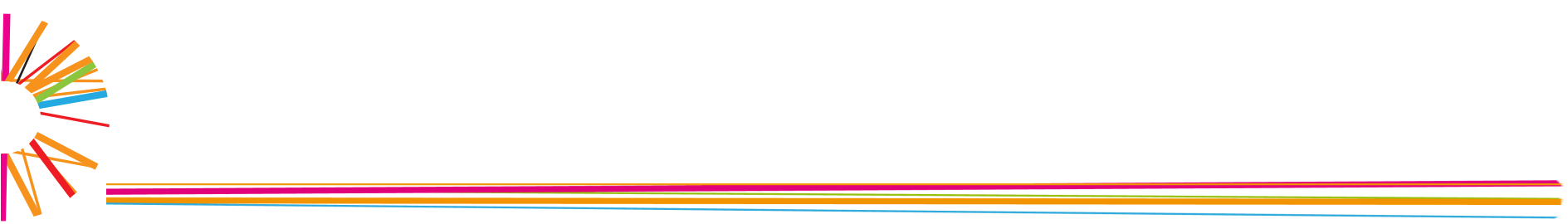
Fonctionnement d’un ordinateur

Et c’est tout!

Un ordinateur ne sait rien faire de mieux que des calculs

Ce qu'il faut retenir

- Une machine est constituée d'un processeur, d'une mémoire vive et de périphériques, le tout interconnecté par un bus
- Un processeur exécute de façon séquentielle des instructions qui se trouvent en mémoire
- Chaque instruction est identifiée par un numéro, elle peut
  - Effectuer une opération sur des variables internes (registres)
  - Lire ou écrire en mémoire ses registres
  - Accéder à un périphérique
  - Modifier la prochaine instruction à effectuer (saut)



II. Logiciels et programmes

Ordinateur vu par l'utilisateur

- L'utilisateur installe des **logiciels**

  Microsoft office, Chrome, Mario

- Logiciel = ensemble de fichiers
  - Fichiers ressources : images, vidéos, musiques…
  - Fichiers programmes : fichier contenant des données et des instructions destinées à être exécutées par un ordinateur

- *In fine*, l'utilisateur lance l'exécution de **programmes**

  Excel, Word, Chrome, Mario

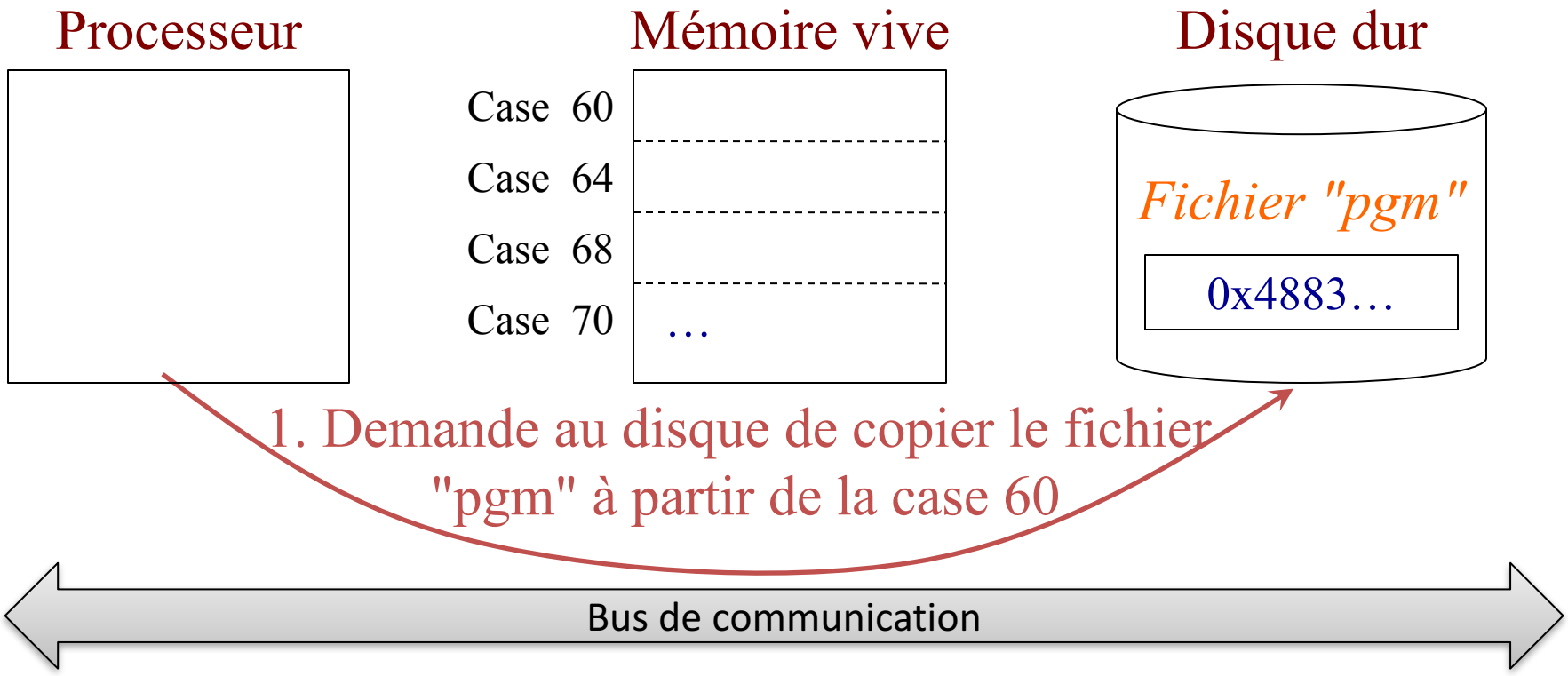
Qu'est ce qu'un programme ?

- **Programme binaire** =
  Ensemble d'instructions exécutables par le processeur + des données manipulées par ces instructions

- **Programme source** =
  Ensemble d'opérations abstraites décrivant les actions à effectuer + des données manipulées par ces opérations

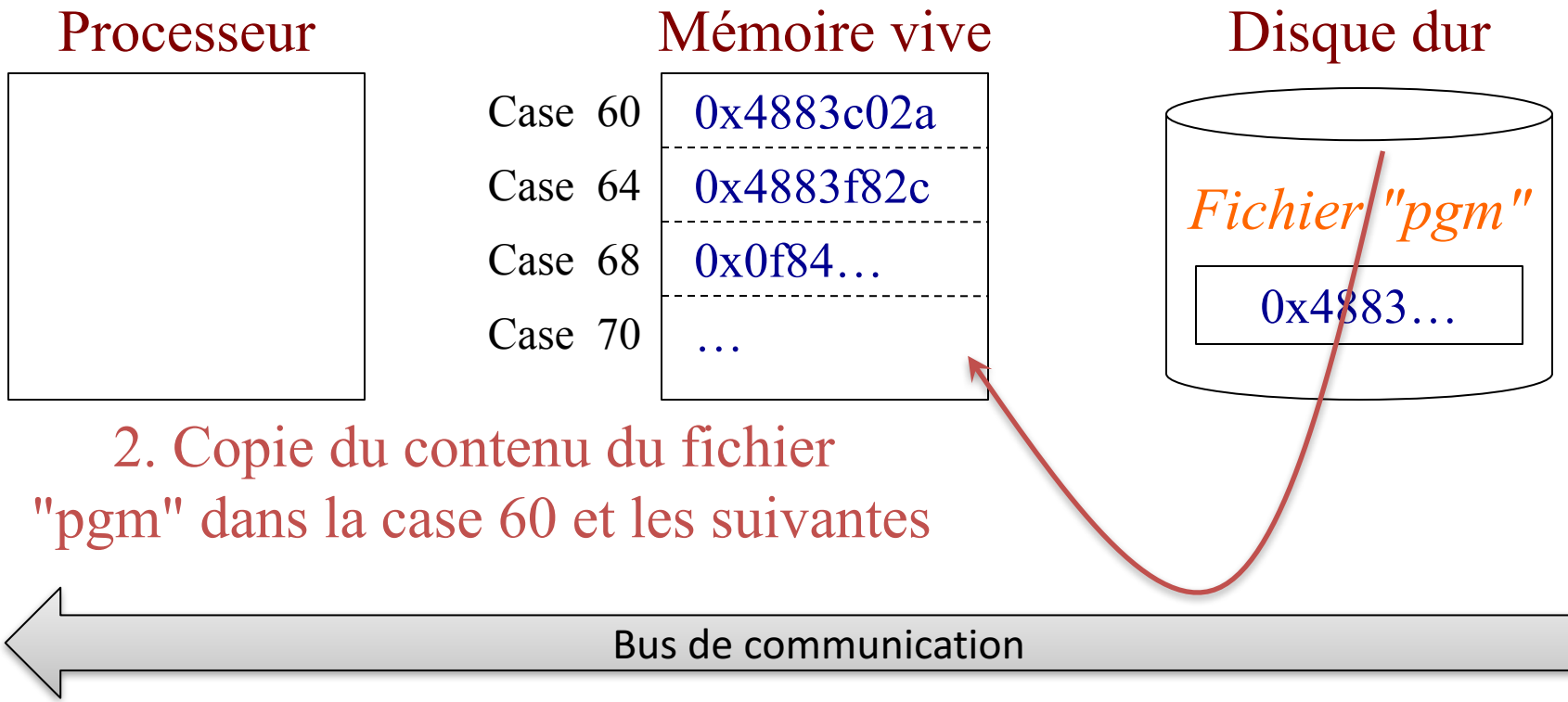
Exécution d'un programme binaire

- Un binaire doit être chargé en mémoire pour être exécuté (typiquement à partir du disque dur)



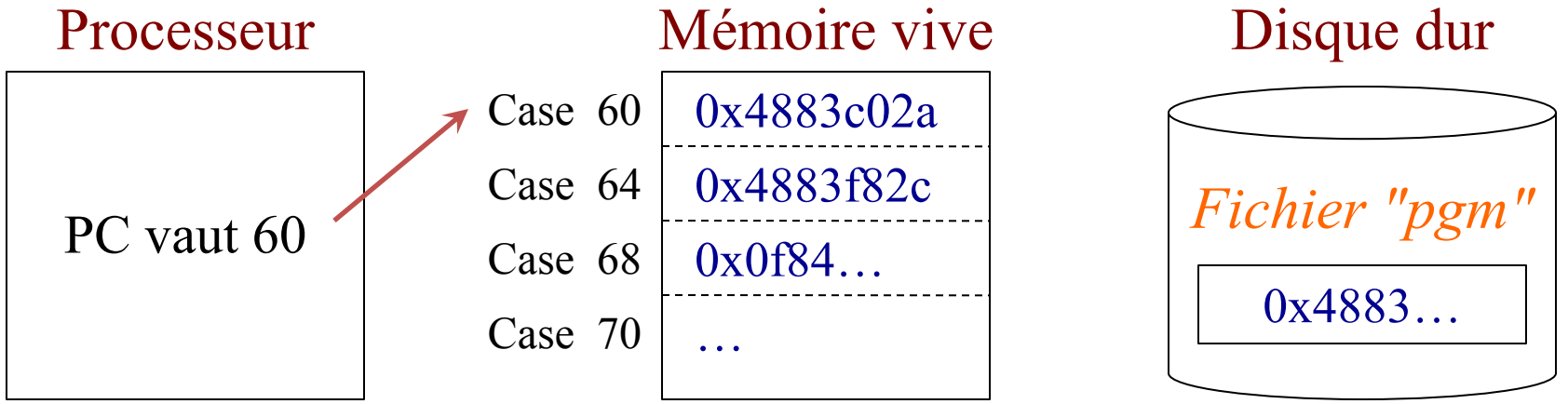
Exécution d'un programme binaire

- Un binaire doit être chargé en mémoire pour être exécuté



Exécution d'un programme binaire

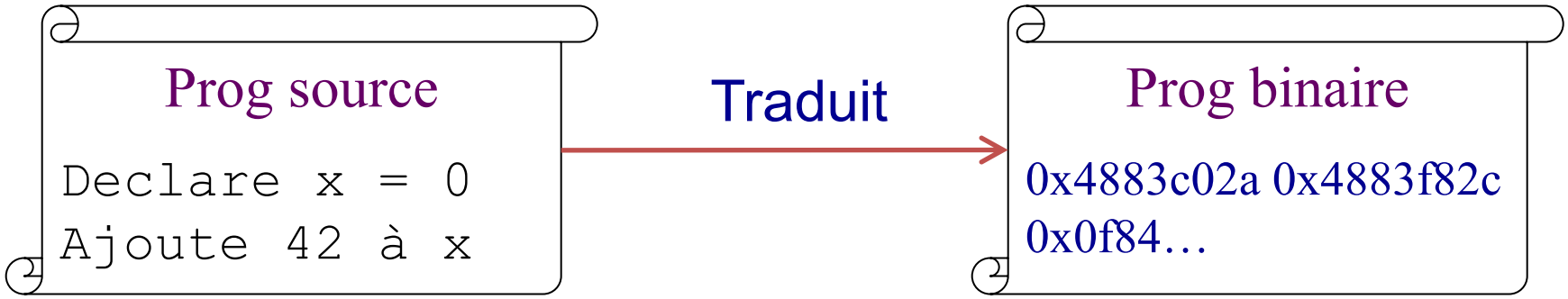
- Exécution d'un programme à partir de la mémoire centrale



3. L'exécution du programme peut alors démarrer

Exécution d'un programme source

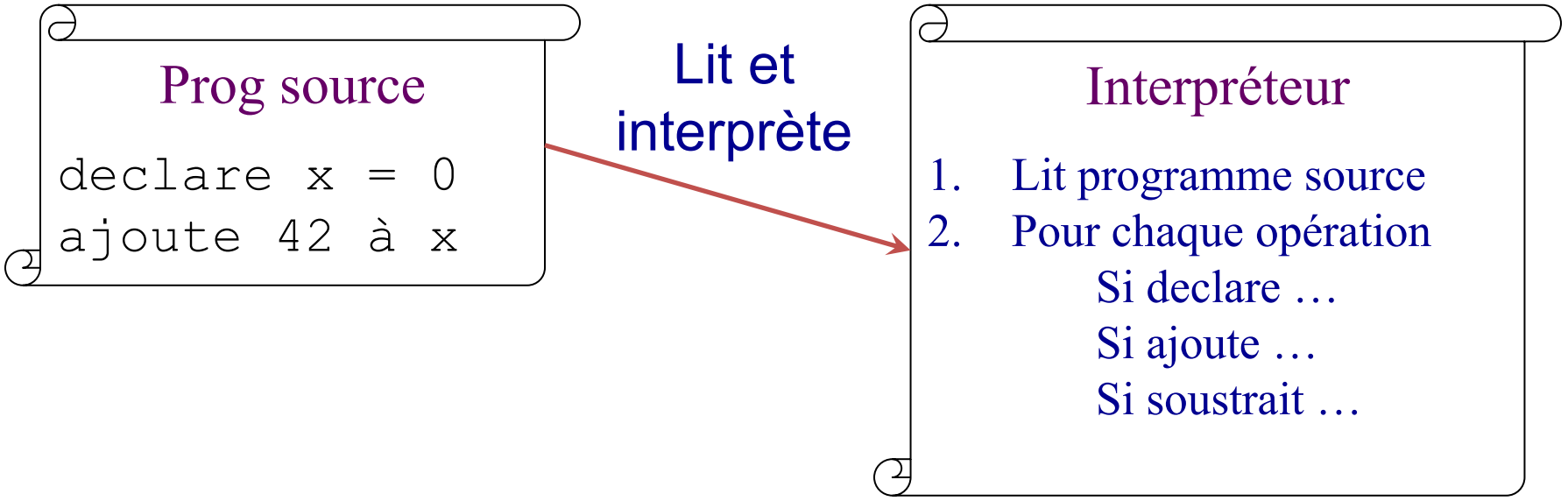
- Solution 1 : après une traduction vers un programme binaire



En informatique le traducteur s'appelle un compilateur

Exécution d'un programme source

- Solution 2 : en le faisant interpréter par un autre programme (appelé interpréteur)



Quelques exemples de programmes

- Word, Excel ou Chrome sont des programmes binaires
- En général, dans un logiciel de jeux
  - Le jeu lui-même est un programme binaire
  - Capable d'interpréter les *mods* qui, eux, sont directement des programmes sources *(mod = extension du jeu)*
- Les applications Android sont
  - Interprétées avant Android KitKat (version 4.4)
  - Compilées dès qu'elles sont installées depuis Android KitKat
- Les pages Web dynamiques sont interprétées

Processus et système

Du programme au processus

- Un **processus** est un programme en cours d'exécution
  - Contient bien sûr les opérations du programme
  - Mais aussi son état **à un instant donné**
    - Données en mémoire manipulées par le programme
    - Valeurs des registres du processeur
    - État des périphériques (fichiers ouverts, connexions réseaux…)

Gestion des processus

- Le **système d'exploitation** est un logiciel particulier qui gère les processus

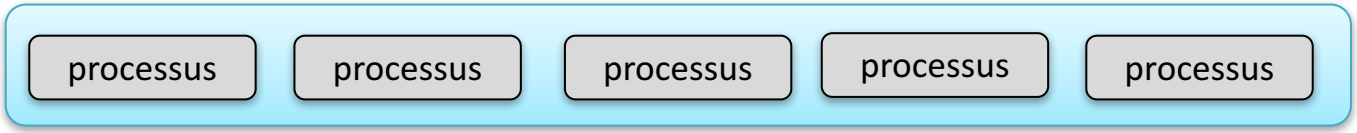
  (Le système est le seul programme qu'on n'appelle pas processus quand il s'exécute)

- Rôle du système d'exploitation
  - Démarrer des processus

    (en chargeant le programme binaire ou l'interpréteur adéquat)
  - Arrêter des processus
  - Offrir une vision de haut niveau du matériel aux processus
  - Offrir des mécanismes de communication inter-processus (*IPC*)

Architecture globale à l'exécution

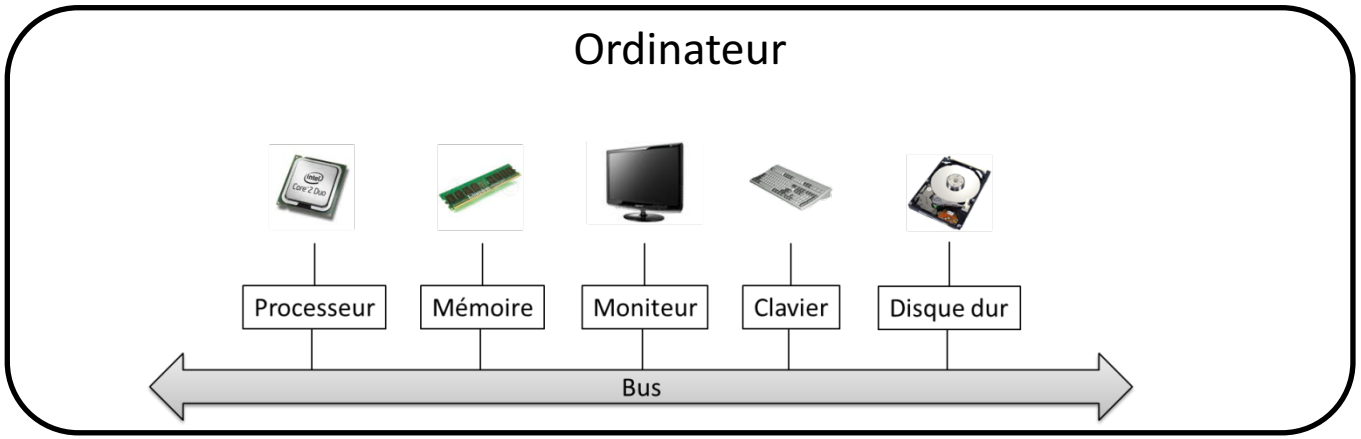
Utilisateurs



Processus

Système d'exploitaion

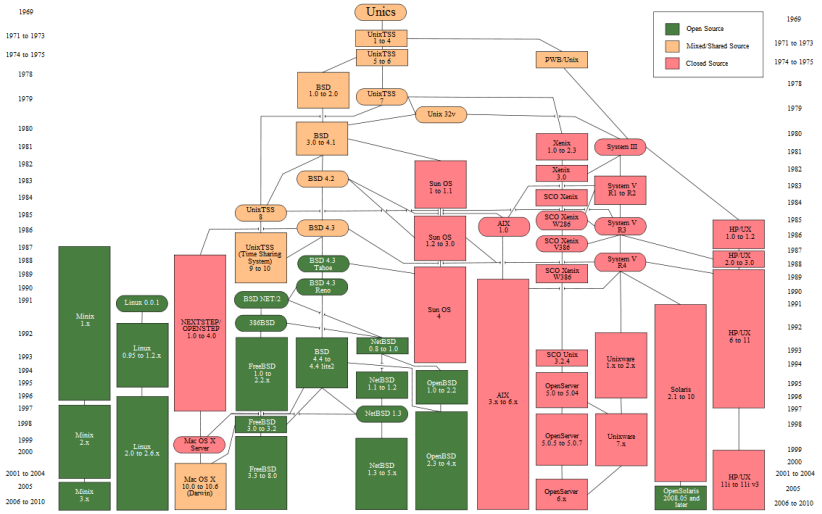
Abstractions
- Processus
- Matériel



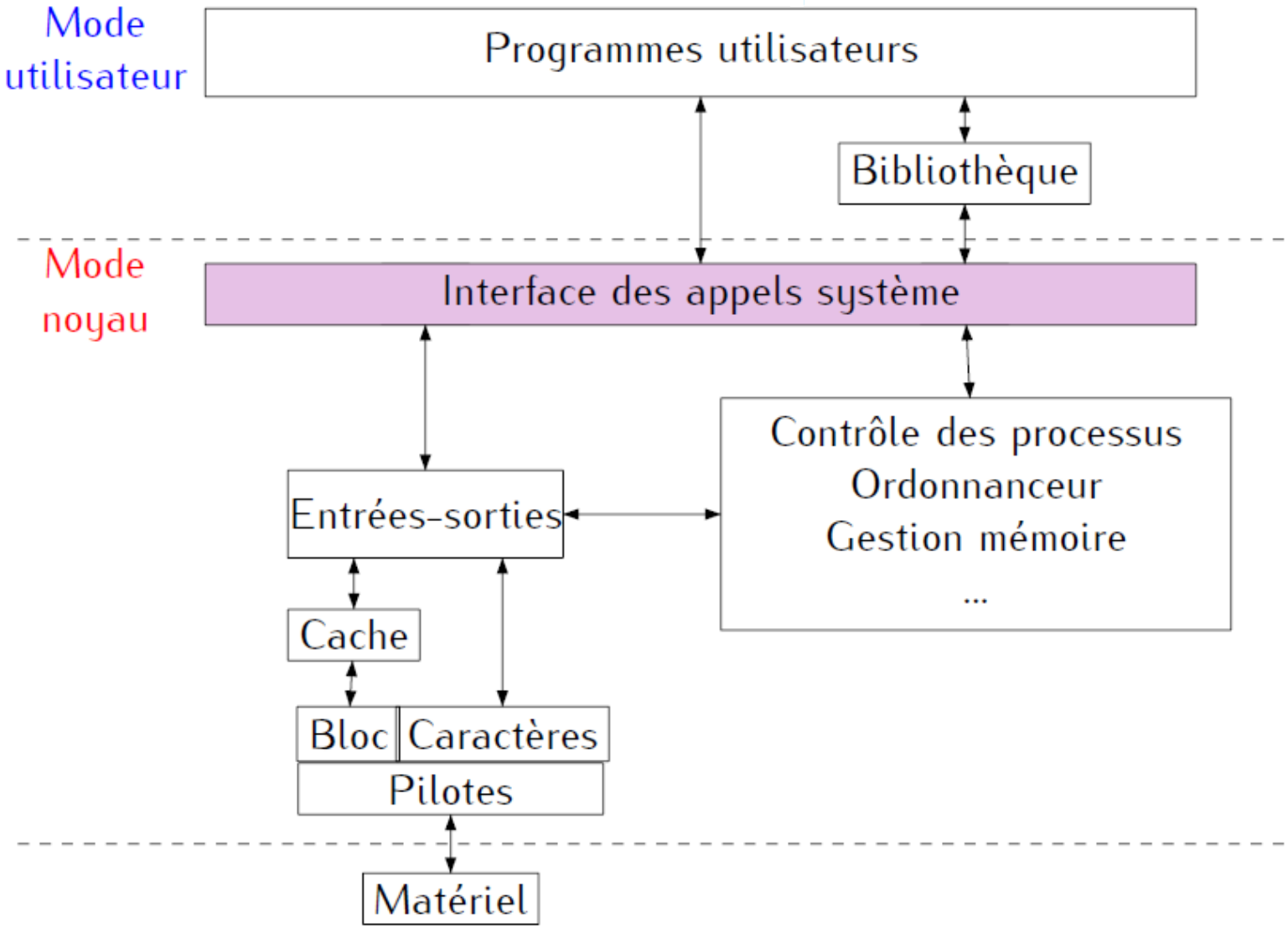
Composants
- Processeur
- Disque

Naissance des premiers systèmes UNIX

- 1969 : première version d'UNIX en assembleur
- 1970 : le nom UNIX est créé
- 1971 : invention du langage de programmation C pour réécrire UNIX dans un langage de haut niveau
- 1991 : première version de Linux



Architecture simplifié



Modes d'exécution

- Micro-architectures modernes
  - Implémentent nativement des mécanismes matériels permettant de distinguer entre plusieurs modes d'exécution

- Modes de fonctionnement
  - mode utilisateur (ou *user*, ou protégé)
    - » Accès au matériel impossible, accès interdit à certaines zones mémoires et certaines instructions
  - mode noyau (ou *kernel*, ou moniteur, ou privilégié)
    - » Accès à tour l'espace d'adressage et à toutes les instructions

Mode d'exécution et appels systèmes

- Programmes d'application
  - S'exécutent en mode non privilégié
  - Ne peuvent accéder au matériel sans passer par les routines de contrôle d'accès au système
- Appels systèmes
  - Permettent de basculer en mode privilégié
  - Accès à l'ensemble des ressources de la machine
  - Passage strictement contrôlée
  - Notion de « *traps* » et d'interruptions

Interruptions

- Événement qui, une fois reçu par le processeur, conduit à l'exécution d'une routine de traitement adaptée
  - Exécution du programme en cours suspendue pour exécuter la routine de traitement
  - Analogue à un appel de fonction, mais de façon asynchrone

Interruptions (2)

- Il existe plusieurs types d'interruptions identifiées par leur numéro
  - Numéro d'IRQ (« *Interrupt ReQuest* »)
- Permet d'effectuer un pré-tri en appelant la routine de traitement ad-hoc selon le périphérique qui a émis l'interruption
  - Configuration manuelle sur les anciens systèmes (jumpers)
  - Auto-configuration en mode « PnP »

Interruptions (3)

- Différents types d'interruptions
  - Asynchrones : interruptions matérielles reçues par le processeur par activation de certaines de ses lignes de contrôle
    - Gestion des périphérique
  - Synchrones : interruptions générées par le processeur lui-même
    - Par exécution d'une instruction spécifique (« trap »)
      - Exemple : instruction INT sur architecture x86
      - Sert à mettre en œuvre les appels système
    - Sur erreur logicielle (erreur d'accès mémoire, calcul …)
      - Set à mettre en œuvre les exceptions

Exécution d'une interruption

- Sauvegarde dans la pile l'adresse de la prochaine instruction à exécuter dans le cadre de déroulement normal
- Utilisation du numéro de l'interruption pour indexer une table contenant les adresses des différentes routines de traitement (vecteur d'interruptions)
  - » Accessible en lecture seule en mode non privilégié
- Déroutement à cette adresse
  - ➢ Passage en mode privilégié si le processeur en dispose

Exécution d'une interruption (2)

- Systèmes récents mettent en place une pile dédiée pour traiter les appels
- Avantages
  - Éviter de mettre en danger le système si la pile utilisateur est prête à déborder
  - Éviter qu'un pilote détruise la pile d'un processus en cours d'appel système
  - Éviter que l'utilisateur puisse récupérer des information sensibles en analysant sa pile au retour de l'appel

Appels système

- Arguments empilés dans la pile utilisateur
  - Tout comme un appel de fonction classique
- Mise en œuvre d'un « trap »
  - Appel de « INT/syscall » au lieu d'un simple « call »
  - Ou autre chose en fonction du système
- Mise en œuvre d'un appel système (très) coûteuse
  - Plusieurs centaines de cycles
  - Pas toujours optimal en termes de performance!
- Documentation
  - Section 2 du manuel ou section 3p pour les appels POSIX

Norme POSIX (IEEE 1003)

- POSIX : Portable Operating System Interface
- Norme de l' IEEE spécifiant les services principaux d'un OS
- Définit les comportements d'une collection de fonctions permettant l'accès au système
- Peut-être implémentée
  - soit par un appel système,
  - soit par un appel de bibliothèque
- Plusieurs mises à jours

Limites

- Plusieurs constantes (limites) sont définies par l'implémentation.
  - Par ex. la longueur maximale d'un nom de login, le nombre maximal de fichiers qu'un processus peu ouvrir simultanément
- POSIX essaie de fournir des façons portables de tester ces constantes.
- Les limites peuvent être détectées
  - à la compilation fichier en-tête limits.h, ou
  - à l'exécution : sysconf, pathconf.

Appels système et erreurs

- Important de tester si les appels système réussissent
- Pour les fonctions renvoyant un entier, l'échec d'un appel système se matérialise par une valeur retour de −1
- La variable extern int errno est affectée par les appels système ayant échoué. Sa valeur, non nulle, détermine le type de l'erreur.

  !!! Un appel système réussi laisse errno a sa valeur précédente
- La fonction void perror (const char *s) ; affiche un diagnostic dépendant de la valeur de errno
- Le programme strace permet de lister les appels systèmes d'une commande

Entrées/sorties

Rappel sur les fichiers

- Distinction sous Unix entre
  - Contenu d'un fichier (data)
  - informations sur le fichier (metadata): inode
- Information d'un inode
  - Type de fichier
  - Nombre de liens durs (hard links) partageant l'inode
  - Taille du fichier en octets
  - Identifiant du périphérique
  - Identifiant de l'utilisateur (UID) propriétaire, de son groupe (GID)
  - Date de création, modification
  - Droits d'accès

Entrée/sorties

- Lectures/écritures d'un fichier
- Provoque une interruption
  - Mise en attente du processus demandeur
- Système utilise un mécanisme de cache
  - Réalisation des écriture dans une mémoire tampon en RAM
  - Synchronisation périodique du cache avec le périphérique
  - Transparent pour l'utilisateur
  - … sauf que parfois au retour d'un écriture les données ne sont pas écrites (encore) sur le périphérique

Descripteurs

- Avant de réaliser une E/S, un processus doit acquérir les droits en mode
  - lecture, ou
  - écriture, ou
  - lecture-écriture.
- Il le fait en demandant une ouverture du fichier, dans un des 3 modes.
- En cas de succès, le système fournit au programmeur un identifiant entier, appelé descripteur permettant les accès ultérieurs.

Descripteurs (2)

- Un descripteur obtenu par un processus lui est propre.
- Il donne au processus le droit d'accès à une entrée dans une « table des fichiers ouverts » (TFO).
- Un même fichier peut être ouvert plusieurs fois (dans des modes éventuellement différents) par un ou plusieurs processus.
- 3 descripteurs spéciaux :
  - STDIN_FILENO, entrée standard,
  - STDOUT_FILENO, sortie standard,
  - STDERR_FILENO, sortie erreur standard.

Ouverture d'un fichier

```
#include <sys/stat.h> // pour le mode d'ouverture.
#include <fcntl.h>
int open( const char *path , int flag , ... /* mode */ );
```

- 1er argument : chemin d'accès au fichier

Ouverture d'un fichier

```
#include <sys/stat.h> // pour le mode d'ouverture.
#include <fcntl.h>
int open( const char *path , int flag , ... /* mode */ );
```

- 1er argument : chemin d'accès au fichier
- 2ème argument : mode d'ouverture. OU bit à bit contenant
  - l'une des constantes **O_RDONLY, O_WRONLY, O_RDWR**.
  - des options supplémentaires parmi
  - **O_APPEND** les écritures se feront en fin de fichier,
  - **O_CREAT** création du fichier s'il n'existe pas,
  - **O_TRUNC** tronque le fichier s'il existe,
  - **O_EXCL** échec si création demandée et le fichier existe,
  - **O_SYNC** attente que l'écriture soit réalisée physiquement.
  - + . . .
- 3ème argument optionnel : mode de création (modifié par l'umask)

Ouverture : structures mises en jeu

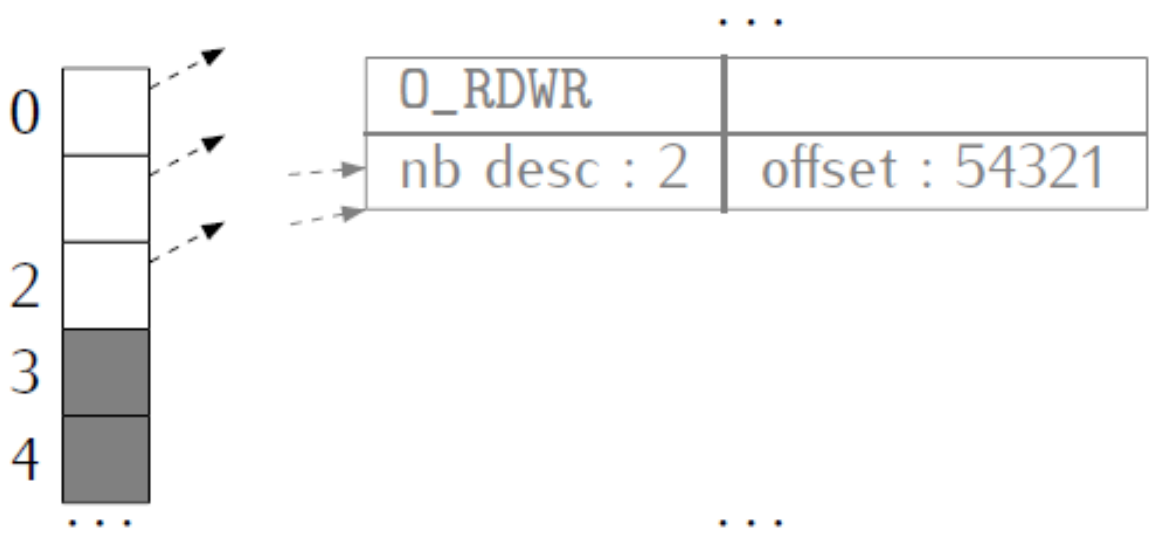
- En cas de succès, open renvoie un descripteur entre 0 et **OPEN_MAX**.
  - Identifie l'ouverture du fichier par le processus.
- Permet au noyau d'accéder à une entrée de la TFO précisant
  - le mode d'ouverture du fichier,
  - le nombre de descripteurs pointant sur l'entrée.
  - le décalage du curseur de lecture/écriture par rapport au début (offset),
  - un pointeur vers un i-noeud en mémoire.

Ouverture : structures mises en jeu

Table descripteurs d'un processus

TFO, globale, dans le système

Table v-noeuds (globale)

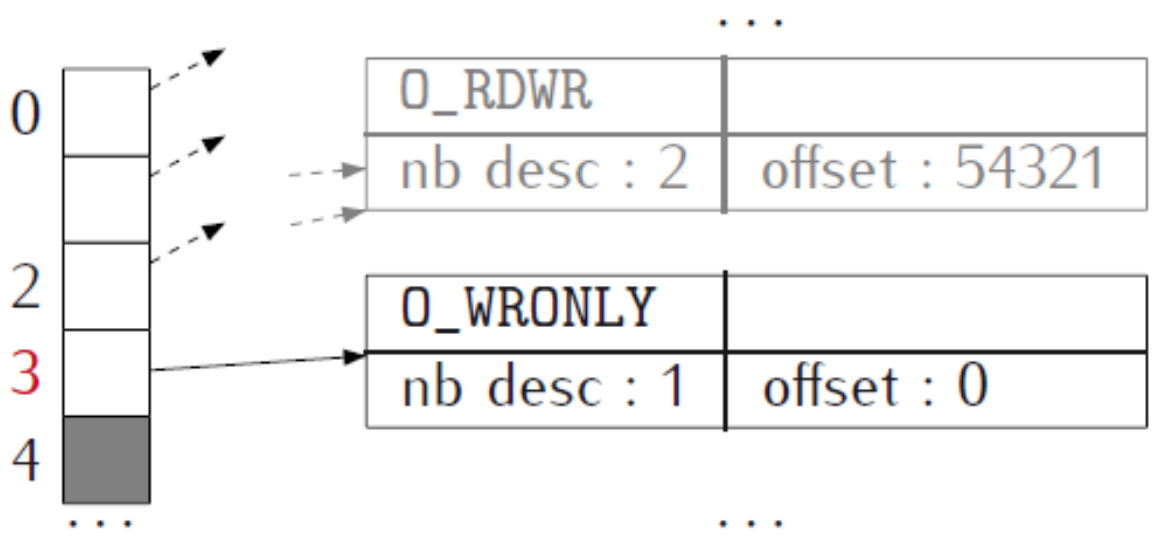


Ouverture : structures mises en jeu

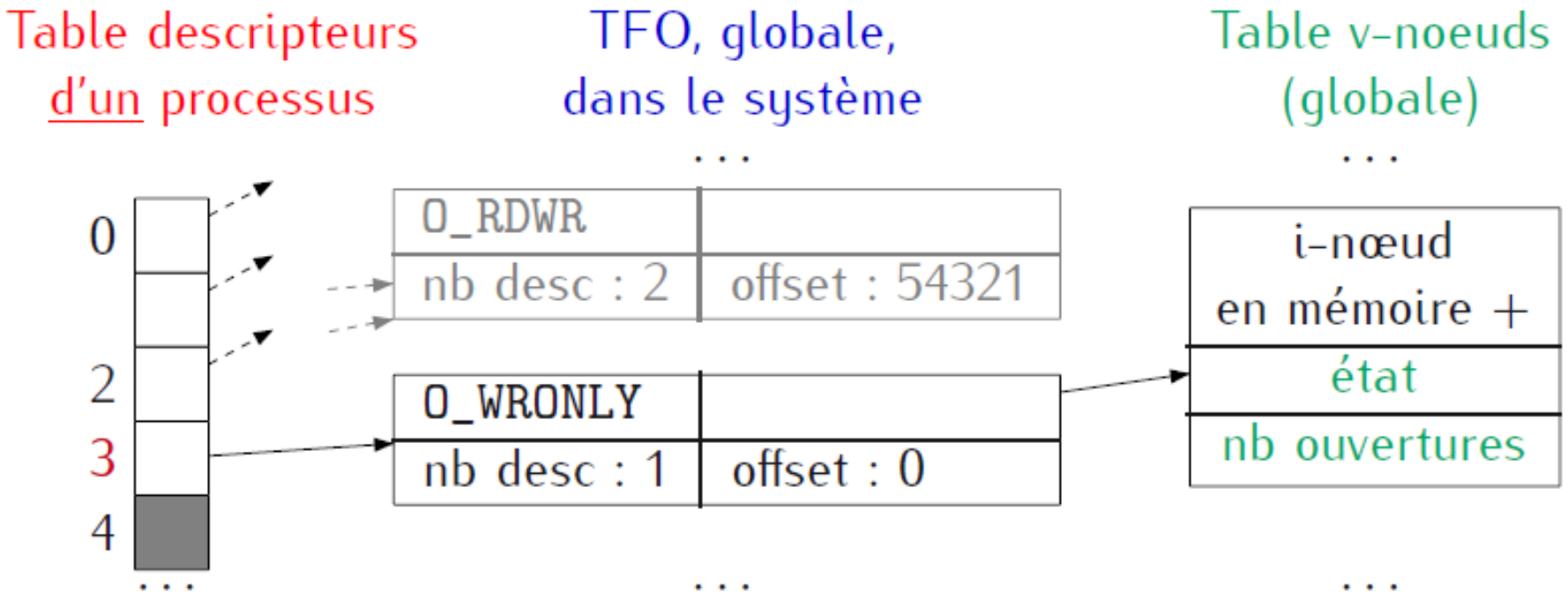
Table descripteurs d'un processus

TFO, globale, dans le système

Table v-noeuds (globale)



Ouverture : structures mises en jeu



Ouverture d'un fichier par **open**

1. Allocation d'un descripteur de fichier.
2. Allocation d'une entrée dans la table des fichiers ouverts (TFO).
3. Allocation éventuelle d'une entrée dans la table des v-noeuds, avec récupération du i-noeud sur disque s'il n'est pas dans le cache
4. Vérifications de droits d'accès au fichier selon le mode demandé.
5. Rejet éventuel d'opérations illégales (ouverture d'un répertoire, ou ouverture en écriture d'un exécutable ouvert en cours d'exécution).
6. Création éventuelle du fichier s'il n'existe pas et **O_CREAT** est demandé
7. Si le fichier régulier existe et peut être ouvert en mode écriture ou lecture-écriture, et si **O_TRUNC** est demandé, le fichier est tronqué libération éventuelle des blocs.
8. Initialisation du fichier ouvert (offset, pointeur sur v-noeud, mode d'ouverture et compteur de références).

Atomicité

- Pour un processus P1, ouvrir un fichier en mode **O_CREAT | O_EXCL** n'est pas équivalent à
  1. tester s'il existe
  2. si oui, ne rien faire. Sinon, l'ouvrir avec le mode O_CREAT,
- Dans le second cas, scénario suivant possible :
  - P1 constate que le fichier n'existe pas.
  - P2 crée le fichier et y écrit.
  - P1, qui a déjà fait le test, ouvre le fichier bien qu'il existe.
- Mode **O_CREAT | O_EXCL** : [test et création] atomique : garantie de non interruption entre test et création.

Fermeture d'un fichier

- Lorsqu'un processus a fini d'utiliser un fichier, il utilise

```
#include <unistd.h>
int close ( int descr );
```

- L'appel désalloue
  - le descripteur,
  - éventuellement l'entrée de la TFO si le compte de descripteurs pointant sur l'entrée passe à 0,
  - éventuellement l'i-noeud en mémoire si le nombre d'ouvertures passe à 0,
  - éventuellement les blocs du fichier si le nombre de liens et le nombre d'ouvertures est 0.
- Un processus qui se termine ferme implicitement tous ses descripteurs.
- Il est parfois crucial de ne pas oublier close().

Tête (ou curseur) de lecture/écriture

- Pour lire ou écrire dans un fichier, tout se passe comme si on avait une « tête (ou curseur) de lecture-écriture ».
- La tête sera déplacée automatiquement à chaque lecture ou écriture (tout se passe comme sur une bande magnétique)

Déplacement du curseur

- Le curseur de lecture et/ou écriture est associé à un fichier ouvert.

  !!! Un même fichier peut être ouvert plusieurs fois, avec des curseurs à des places différentes.
- La position du curseur est mémorisée par un entier > 0 dans l'entrée de la TFO.
- À l'ouverture, il est positionné à :
  - la taille du fichier en mode O_APPEND ;
  - 0 sinon.
- Sur les fichiers ordinaires, on peut demander un déplacement du curseur.
- Ce n'est pas possible sur tout fichier (ex. terminal).

Déplacement du curseur

```
#include <unistd.h>
off_t lseek (int descr, off_t offset, int whence);
```

- Si whence vaut
  - **SEEK_SET**, déplacement 2ème argument > 0 relatif au début de fichier
  - **SEEK_CUR**, déplacement 2ème argument relatif à la position courante
  - **SEEK_END**, déplacement 2ème argument relatif à la fin de fichier
- L'appel retourne la nouvelle position
- On peut positionner le curseur après la fin de fichier

Lecture

```
#include <unistd.h>
ssize_t read (int descr, void *buf, size_t nbytes);
```

- Lecture via le descripteur **descr** de **nbytes** octets, à placer dans **buf**
  1. Lecture à partir de la position (offset) correspondante dans la TFO
  2. Avancement du curseur du nombre d'octets lus
- Lecture atomique : ce qui est lu n'est pas entrelacé avec une autre E/S
- Retourne le nombre d'octets effectivement lus
- **read** renvoie donc moins que **nbytes** si la fin de fichier est atteinte
- Comportement sur tubes/sockets : vu plus tard

Écriture

```
ssize_t write(int desc, const void *buf, size_t
nbytes);
```

- Demande l'écriture des **nbytes** premiers octets de **buf** via **descr**
  - à partir de la position courante
  - avec écrasement des anciennes données (n'insère pas) ou
  - extension du fichier si la fin est rencontrée
- Retourne le nombre d'octets effectivement écrits
- Un retour inférieur à nb peut se produire

- Remarque : on peut écrire des données autres que chaînes (codages d'entiers, structures, etc.). Attention au codage (big/little endian, alignement)

Taille I/O et efficacité

- Si on doit faire plusieurs lectures ou écritures, la taille du buffer influe notablement sur les performances
- Les écritures en mode synchrone sont beaucoup plus lentes

Atomicité

- Ouvrir un fichier en mode **O_APPEND** n'est pas équivalent à
  - l'ouvrir sans le mode **O_APPEND**
  - avant chaque écriture, déplacer le curseur en fin de fichier

Atomicité

- Ouvrir un fichier en mode **O_APPEND** n'est pas équivalent à
  - l'ouvrir sans le mode **O_APPEND**
  - avant chaque écriture, déplacer le curseur en fin de fichier
- Dans le second cas, le scénario suivant est possible :
  - Fichier ouvert en mode **O_WRONLY** par 2 processus P1, P2
  - P1 déplace le curseur en fin de fichier pour y écrire
  - P2 idem
  - P2 écrit en fin de fichier
  - P1 écrit, mais pas en fin de fichier ! (écrase l'écriture de P2)
  - P1 et P2 pensent avoir écrit en fin de fichier. Ce n'est pas le cas pour P1
- En mode O_APPEND, [déplacement du curseur + et écriture] atomique : garantie de non interruption entre déplacement du curseur et écriture

Duplications et redirections

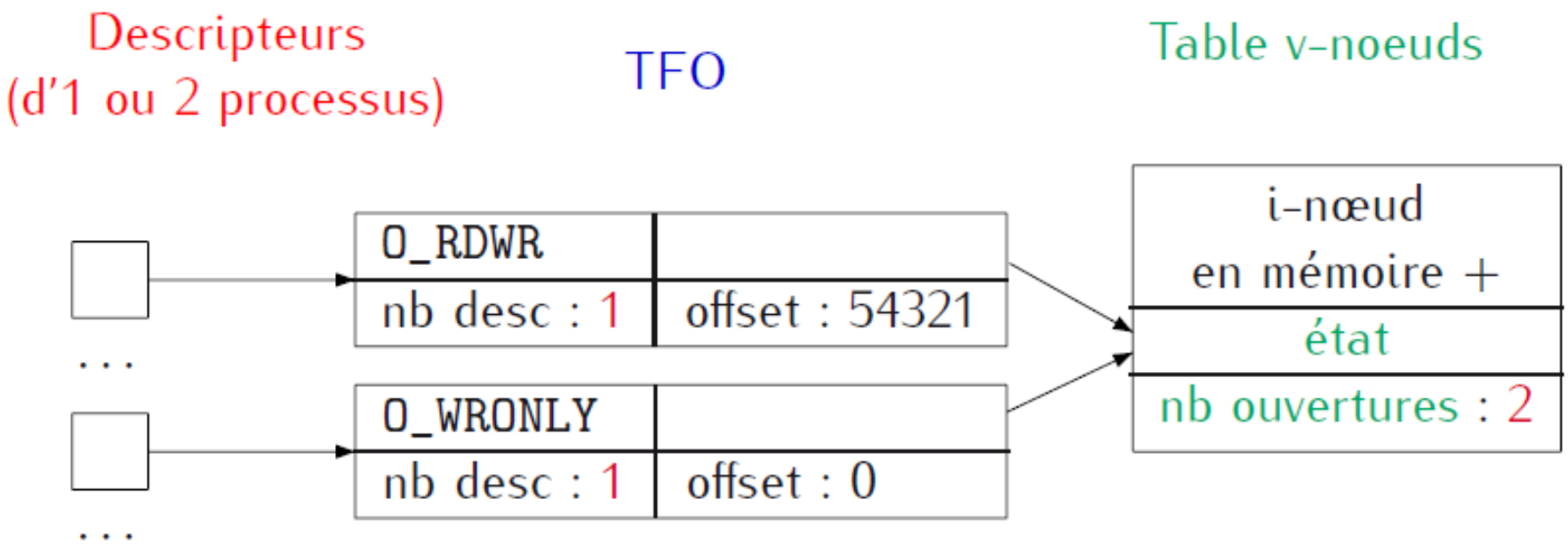
- L'objectif de la TFO est de créer une indirection, pour permettre à plusieurs processus de partager le même curseur
- Plusieurs moyens de créer ce partage.
  - Quand un processus en crée un autre, il lui « transmet » ses descripteurs
  - On peut demander explicitement de dupliquer un descripteur par dup2
  - . . .

```
int dup2( int old_descr , int new_descr);
```

- **old_descr** doit correspondre à un descripteur ouvert
- La fonction
  1. ferme le descripteur **new_descr** s'il est ouvert,
  2. le fait pointer vers la même entrée de la TFO que **old_descr**.
  3. retourne la valeur de **new_descr** (−1 si erreur).
- Les appels **dup** et **fcntl** permettent aussi de dupliquer

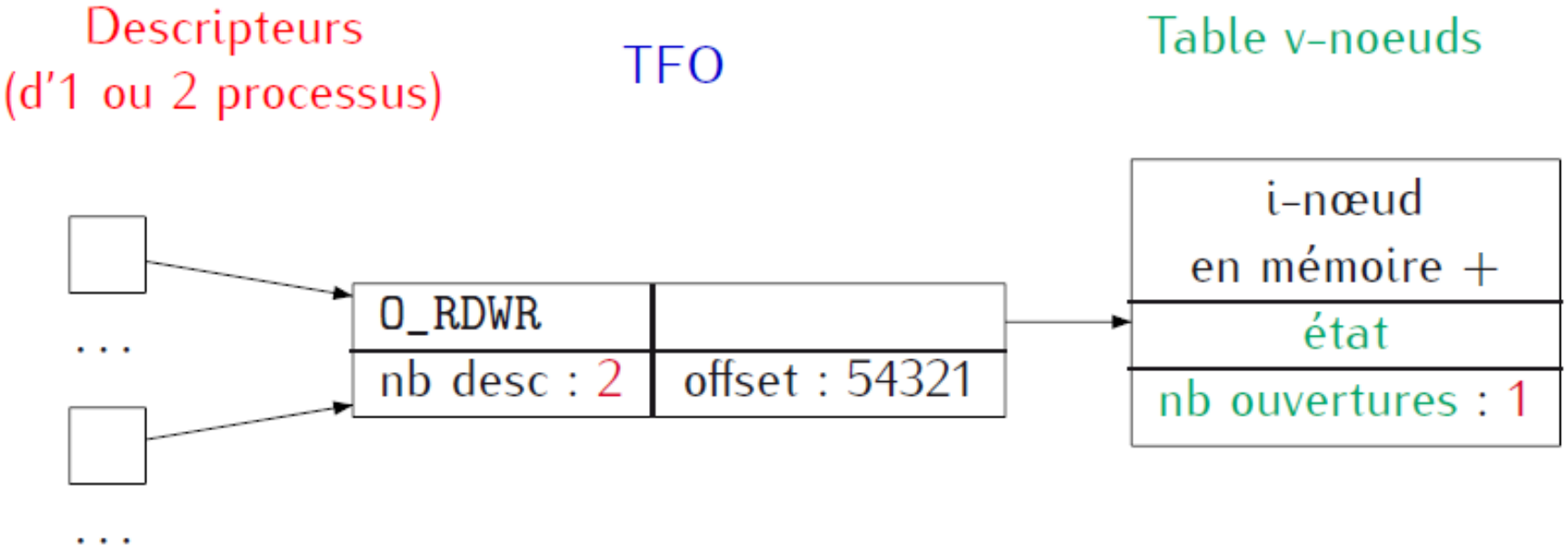
Duplications et redirections

- Deux ouvertures indépendantes du même fichier



Duplications et redirections

- Une ouverture et une duplication



Manipulation de l'entrée TFO

- L'appel fcntl permet de manipuler les attributs associés à un descripteur ou à une entrée de la TFO

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int descr, int commande, ...);
```

- Le second argument donne le type d'opération à réaliser :
  - Récupération/changement du mode d'ouverture : **F_GETFL/F_SETFL**
  - Manipulation d'un attribut du descripteur lui-même (vu plus tard)
  - Duplication à la dup
  - Pose de verrous

Entrées-sorties sur répertoires

- On ne peut pas utiliser open() et read() sur les répertoires. À la place

```
#include <dirent.h>
DIR * opendir (const char * dirname);
struct dirent * readdir (DIR * dirp );
void rewinddir( DIR * dirp );
```

- La structure **dirent** contient un champ **d_name** : le nom de l'entrée dans le répertoire
- Curseur déplacé à l'entrée suivante après une lecture réussie
- Remise du curseur au début du répertoire par la fonction **rewindir**

Bufferisation des E/S

- Les entrées-sorties représentent la majeure partie des appels système
- Les appels système individuels coûtent cher
- Il faut les « factoriser »
  - Un même appel système doit regrouper plusieurs entrées/sorties demandées par l'utilisateur
  - Cette factorisation doit avoir lieu dans l'espace de l'utilisateur, avant l'appel système proprement dit

Bufferisation des E/S (2)

- À chaque descripteur de « haut niveau » doit correspondre une zone de stockage temporaire (tampon ou « buffer »)
- Les données en écriture sont stockées dans la zone avant leur utilisation
  - En lecture : lecture d'un bloc entier, puis consommation caractère par caractère
  - En écriture : accumulation des caractères puis écriture d'un bloc entier

Bufferisation des E/S (3)

- Descriteurs de « haut niveau » ➔ structure FILE
  - Pointeur de flot / flux
  - Voir « /usr/include/stdio.h »
  - Contiennent le descripteur de fichier
  - Contiennent des pointeurs vers
    - Le début des tampons en lecture et en écritures
    - La position courante qui leur correspond
    - La taille des données qu'ils contiennent

Bufferisation des E/S (4)

- Utilisation de fonctions spécifiques
  - Fopen(), fread(), fseek(), fprintf(), fscanf(), etc.
- Création d'un flot à partir d'un descripteur
  - fdopen()
- Des flots standard correspondent aux descripteurs standards de bas niveau
  - stdin : descripteur 0 (STDIN_FILENO)
  - stdout : descripteur 1 (STDOUT_FILENO)
  - stderr : descripteur 2 (STDERR_FILENO)

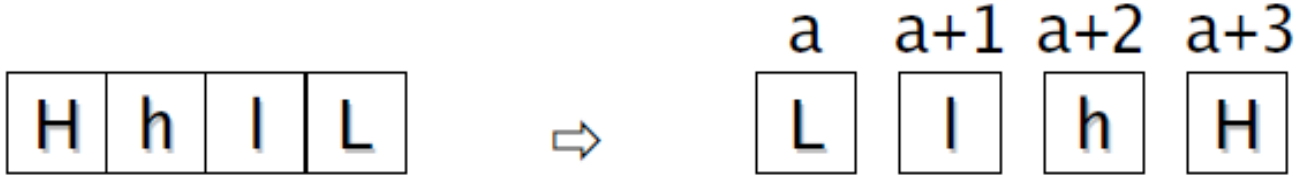
Types de bufferisation

- Bufferisation bloc : flot de sortie correspond à un fichier
- Bufferisation ligne : lorsque le flot de sortie correspond à un terminal
  - Vidange déclenchée par le caractère \n
- Demande explicite de vidange
  - fflush()
- Quand vidanger
  - En pratique, le plus tard possible

Mot machine

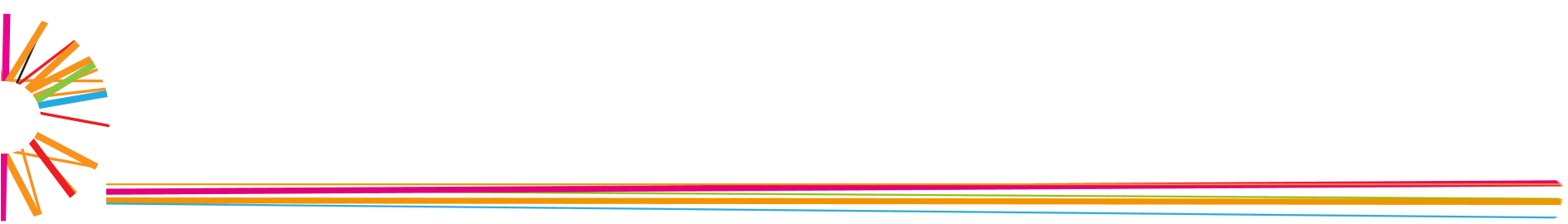
- Stocker un mot machine en mémoire
  - Savoir dans quel ordre stocker les octets du mots dans les cases successives de la mémoire
  - Octet de poids fort en premier (« big endian »)

  - Octet de poids faible en premier (« little endian »)



Mot machine

- Lecture/écriture de données sous forme binaire dans un fichier
  - Même problème que précédemment
- Fonctions normalisées de conversion
  - Htons(), ntohs(), htonl(), ntonl()
  - « Host to Network » et « Network to Host »
- C'est le « big endian » qui a gagné sur Internet



Gestion de processus

Notion de processus

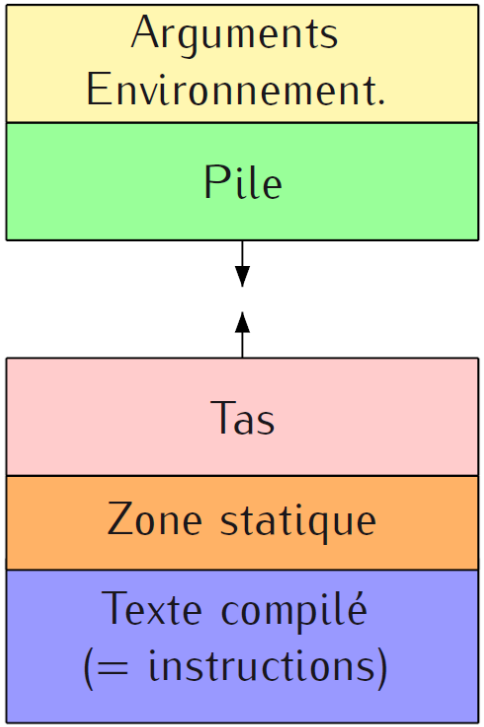
- Processus = programme en cours d'exécution
  - Un espace d'adressage virtuel + contexte d'exécution (mémoire, état des descripteurs, etc.)
- Caractéristiques statiques
  - PID : Process Identifier (identifie le processus)
  - PPID : Parent Processus Identifier (identifie le parent)
  - Utilisateur propriétaire
  - Droits d'accès aux ressources (fichiers, etc.)
- Caractéristiques dynamiques
  - Priorité, environnement d'exécution, etc.
  - Quantité de ressources consommées (temps CPU, etc.)

Espace d’adressage

- Chaque processus a un espace d’adressage privé
  - Garantit le cloisonnement entre processus (sécurité et protection)
  - Abstraction de la mémoire (mémoire virtuelle)
  - Cache le noyau et les autres processus !
- Chaque processus ne voit que son espace d’adressage

Organisation mémoire

- Plusieurs zones
  - taille fixée à la compilation : texte, statique
  - taille évoluant au cours de l'exécution : pile, tas



Création de nouveaux processus

- Sous Unix,, un processus ne peutt êttrre crréé..... que par un autre processus !!
  - Problème de l'oeuf et de la poule, résolu au démarrage du système par la création « à la main » du « processus 0 »

Création de nouveaux processus

- Chaque processus est identifié par un numéro entier unique, son PID (*Processus ID*)
  - Deux processus ne peuvent pas avoir le même numéro
  - `pid_t getpid ( void );`

- Chaque processus a un unique parent qui l'a créé
  - Idenifiant : PPID (*Parent Processus ID*)
  - Un parent peut créer de multiples fils au cours de sa vie
  - `pid_t getppid( void );`

Cycle de vie

- Deux primitives élémentaires
  - Fork() : un processus demande à se duplique
  - Exec() : un processus demande à remplacer ses segments par ceux d'un executable present dans le système de fichier

Duplication : fork

- Appelé par le processus lui-même
- En cas de succès, le système crée un nouveau processus « fils », clone du processus « père »
  - Copie de la mémoire du père : segments, pile, …
    - Mais pas de partage en écriture
    - Copie paresseuse et partage en lecture
  - Copie des descripteurs des ressources systèmes
    - Descripteurs de fichiers ouverts, etc.

Fonctionnement de fork

- Un seul processus appele “fork()” et deux processus en reviennnet!
  - Comment les distinguer?
- Valeur de retour de fork
  - 0 : je suis le fils
  - >0 : je suis le père et la valeur est le PID du fils
    - Aucun autre moyen pour le père de connaître le PID du fils
  - -1 : erreur

Fonctionnement de fork

- Qui revient en premier ?
  - Impossible à savoir
  - Responsabilité de l'ordonanceur de decider
  - Ne doit pas être une question pertinente
    - Sinon mettre en place des mécanismes de synchronisation

Exemple

```
#include <stdio.h>
#include <unistd.h>
main ()
{
  pid_t  pid;

  pid = fork ();
  if (pid == 0)
    printf ("Le fils parle\n");
  else if (pid > 0)
    printf ("Le pere parle\n");
  return (0);
}
```

- À tester avec “ `while true; do ./fork` ”

Généalogie

- Liens de parenté des processus forment un arbre généalogique
  - Visible avec la commande « pstree »
- Père et fils diffèrent sur leurs :
  - PID, PPID
  - Statistiques temporelles, verrous, signaux en attente, …
  - Commande « ls -l »

Terminaison des processus

- Terminsaion normale
  - Appel de « exit() » ou « _exit() » à tout moment
  - Appel de « return » dans la fonction main
- Fonction « exit() »
  - Appelle les fonctions enregistrées (callbacks) au moyen de la fonction « atexit() »
  - Vidange des tampons des flots ouverts
  - Continue comme _exit()

Terminsaison des processus (2)

- Appel système _exit
  - Ferme les descripteurs ouverts
  - Transfère au processus 1 la parenté des fils du processus
  - Envoie un signal SIGCHILD au processus père du processus
  - Termine le processus

- Un processus peut se terminer anormalement de manière asynchrone sur réception d'un signal
  - Déclenche l'appel à _exit()

Modèle fork/join

- Être notifié du décès d'un de ses fils

```
pid_t wait (int *status)
```

  - Impossible de choisir le fils que l'on souhaite attendre
  - Appel bloquant jusqu'à ce qu'un des fils ait terminé son exécution
  - Macros disponibles pour manipuler la valeur status
    - WIFEXITED(status) TRUE si le fils s'est terminé correctement
    - WEXITSTATUS(status) 8 bits de poids faible de l'entier renvoyé par le processus fils lors de son appel à exit, _exit ou return
    - WIFSIGNALED(status) TRUE si term. du fils suite à réception sig.
    - WTERMSIG(status) num. du signal qui a causé la term. du fils

Variante de wait

- Choix du procesuss fils

```
pid_t waitpid (pid_t pid, int *status, int options);
```

- Paramètres
  - 1 PID du processus fils avec lequel se synchronizer
  - 2 Informations sur le fils (possiblement NULL)
  - 3 Option `WNOHANG` pour appel non bloquant
- Équivalent de wait en mode non bloquant
  - `waitpid (-1, &status, WNOHANG);`

Synchronisation

- Synchronisation entre le père et le fils
  - Caractère bloquant de wait
  - Le père s'endort tant qu'un de ses fils n'a pas terminé
  - Exemple des interpréteur de commande
    - Sans le « & » l'interpréteur de commande attend la fin de son fils avant de reprendre la main

Processus zombies

- Décès d'un processus ➔ pas directement supprimé de la table des processus du système
  - Permettre à son père de récupérer son code de retour
    - Utilisation de son PID qui ne doit pas être réalouer
- Si pas de wait ➔ état zombie
  - N'utilise plus de ressources (mémoire) mais une entrée dans la table des processus
  - Table finit ➔ nombre max. de processus manipulables
  - Nombre max. de processus qu'un père peut créer
  - Le processus 1 sert de fossoyeur de tous les processus dont il hérite

Exemple de zombies

zombie.c

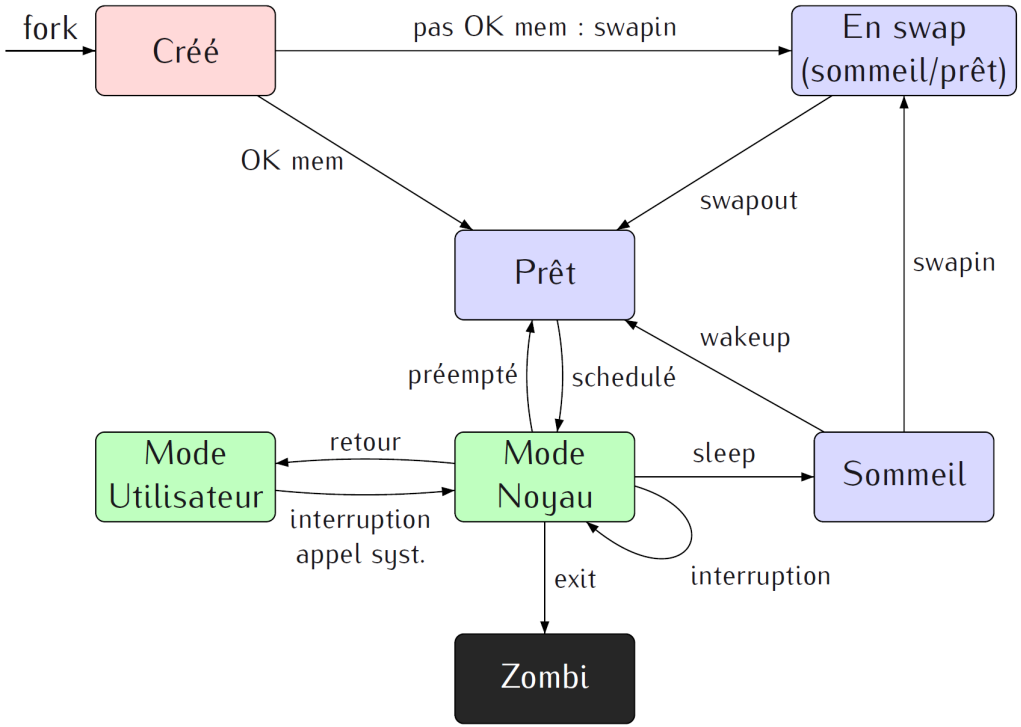
```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
main ()
{
  pid_t     pid;
  int       i;

  for (i = 0; i < 60; i ++) {
    if ((pid = fork ()) <= 0) /* Crée des fils          */
      return (pid);           /* Qui terminent de suite */
  }
  sleep (15);
  do {
    wait (NULL);
    printf ("RIP !\n");
  } while (errno != ECHILD);

  return (0);
}
```

États des processus

- Les processus, une fois créés, prennent successivement de nombreux états :
  - Actif (user/kernel), attente du processeur, attente d'une resource, zombie, etc.



Changement d'état

- Mettre en sommeil volontairement un processus

```
unsigned int sleep (unsigned int seconds);
```

  - Mise en œuvre d'attente passive (opp. attente active)

- Rendre la ressource processeur volontairement

```
int sched_yield(void);
```

  - Ne choisit pas le prochain processus qui aura la ressource

Recouvrement

- Remplace le contenu d'un processus par un nouveau programme exécutable
- Appel système « execve »

```
int execve(const char *path,
           char *const argv[],
           char *const arge[]);
```

- Paramètres
  - 1: Nom du fichier exécutable à exécuter
  - 2: Arguments du programme
  - 3: Variables d'environement (couples key=value)
- Code de retour: -1 si problème (cf. errno)

Fonctionnement de execve

- Tableau d'arguments
  - 1ère chaîne : nom du programme
  - Dernier chaîne : NULL (permet de calculer argc)

- Valeur de retour
  - Seulement en cas d'échec
  - En cas de succès, exécution d'un autre code, celui du nouveau programme!!!!
  - Le code après un exec ne sera jamais exécuté en cas de succès!!!

Variantes

- Ensemble de six fonctions
  - execv, execl, execlp, execle, execvp, execvpe
  - Toutes construites au dessus de execve
- Recherche du programme
  - Défaut : répertoire de travail
  - Versions « vp » : recherche dans le path du processus
- Construction du vecteur d'arguments (cl, clp, cle)
  - Arguments sous forme de liste
  - NULL pour terminer la liste

```
execl("./toto", "toto", NULL);
```

Mise en œuvre du recouvrement

- Espace d'adressage complètement recouvert
  - Mais pas une « remise à zéro » du processus
- Comportement du processus dépendant de ce qui a été effectué avant le recouvrement
  - Au niveau des signaux
    - Les traitants des signaux sont réinitialisés après exec
    - Les masques de signaux sont conservés après l'appel à exec
    - …
  - Au niveau de l'espace d'adressage
    - Les mapping mémoire (mmap) ne sont pas conservés
    - Les sémaphores POSIX sont fermés
    - Les callback de fin de processus (atexit,on_exit) ne sont pas conservés)
    - …

Comportement de exec

- Descripteurs de fichiers (important)
  - Fermés suite à un appel à exec **uniquement** si l'option FD_CLOSEXEC a été positionnée
    - À la création du descripteur
    - Plus tard en utilisant la fonction *fcntl*
  - Dans le cas contraire
    - Descripteurs de fichiers restent utilisables dans le nouveau code
- PPID est conservé
- Valeur de nice est conservé