

Systèmes d'exploitation et Programmation système

Moufida Maimour <moufida.maimour@telecomnancy.eu>

Supports de cours, TD et TP basés sur ceux de
© 2014 Martin Quinson <martin.quinson@irisa.fr>
© 2022 Lucas Nussbaum <lucas.nussbaum@loria.fr>

Telecom Nancy – 2^{ième} année

2024-2025



Organisation pratique du module

- ▶ 5 cours, 3 TD, 3 TP
- ▶ Examen (début décembre)
 - ▶ Documents interdits sauf un A4 recto/verso **manuscrit**

Remerciements


- ▶ Sacha Krakowiack pour son cours et Bryant et O'Hallaron pour leur livre
- ▶ Les (autres) emprunts sont indiqués dans le corps du document

Aspects techniques

- ▶ Document \LaTeX (classe `latex-beamer`), compilé avec `latexmk`
- ▶ Schémas : Beaucoup de `xfig`, un peu de `inkscape`

Document diffusé selon les termes de la licence



 Licence Creative Commons version 4.0 France (ou ultérieure)

 Attribution ;  Partage dans les mêmes conditions

<http://creativecommons.org/licenses/by-sa/4.0/fr/>

Pourquoi des cours de système ?

- ▶ Quatre concepts fondamentaux de l'Informatique (G. Dowek) : Information, **Machine**, Algorithme, Langage

Métiers (à la sortie de TELECOM Nancy) :

- ▶ **IT Operations / Administration système et réseaux**
→ Assurer le maintien en conditions opérationnelles d'une infrastructure (suivi des incidents, montées de version, etc.)
- ▶ Mouvement vers le modèle **DevOps** (\approx Google Site Reliability Engineers)
 - ▶ Suppression des silos *software development vs operations*
 - ▶ Infrastructure as Code : cloud, *pet vs cattle*
 - ▶ Itérations rapides, tests automatiques, déploiement automatiques et continus

Compétences nécessaires : développement logiciel, compréhension profonde des systèmes (combinaison très recherchée sur le marché du travail)

- ▶ **Systèmes embarqués / enfouis** : domotique, automobile, *appliances*
- ▶ **Sécurité informatique** (souvent lié à des aspects système/réseau)
- ▶ Même comme pur développeur, savoir ce qui se passe sous le capot est utile !

Objectif du module

Utiliser efficacement le système d'exploitation

Comment les utiliser efficacement ? Comment les concevoir ?

→ Performances, sécurité, résilience, efficacité énergétique

Contenu et objectifs du module

- ▶ Grandes lignes du **fonctionnement d'un système d'exploitation** (OS)
Focus sur UNIX (et Linux) par défaut, mais généralisations
- ▶ **Concepts clés des OS** : processus, fichier, édition de liens, synchronisation
- ▶ **Utilisation des interfaces système** : programmation pratique, interface POSIX
- ▶ **Programmation système** (et non programmation interne *du* système)
Plutôt du point de vue de l'utilisateur (conception d'OS en RSA)

Motivations

- ▶ OS = **systèmes complexes** les plus courants ; Concepts et abstractions claires
- ▶ Impossible de faire un **programme efficace** sans comprendre l'OS
- ▶ Comprendre ceci aide à **comprendre les abstractions supérieures**

Prérequis : Pratique du langage C et du shell UNIX

Bibliographie succincte (pour cette partie)

Livres

- ▶ Bryant, O'Hallaron : **Computer Systems, A Programmer's Perspective**.
- ▶ A. Silberschatz, P. B. Galvin and G. Gagne : **Operating Systems Concepts** www.os-book.com (2018).¹

Autres cours disponibles sur Internet

- ▶ **Introduction aux systèmes et aux réseaux (S. Krakowiak, Grenoble)**
Source de nombreux transparents présentés ici.
<http://sardes.inrialpes.fr/~krakowia/Enseignement/L3/SR-L3.html/>
- ▶ **Programmation des systèmes (Ph. Marquet, Lille)**
<http://www.lifl.fr/~marquet/cnl/pds/>
- ▶ **Operating Systems and System Programming (B. Pfaff, Stanford)**
<http://cs140.stanford.edu/>

Sites d'information

- ▶ <http://systeme.developpez.com/cours/>
Index de cours et tutoriels sur les systèmes

1. Source de nombreuses figures de ce document

Plan du module :

Systèmes d'exploitation et programmation système

1 Introduction

Système d'exploitation : interface du matériel et gestionnaire des ressources.

2 Processus

Processus et programme ; Utilisation des processus UNIX et Réalisation ; Signaux.

3 Fichiers et entrées/sorties

Fichiers et systèmes de fichiers ; Utilisation ; Réalisation.

4 Exécution des programmes

Schémas d'exécution : interprétation (shell) et compilation (liaison et bibliothèques)

5 Synchronisation entre processus

Problèmes classiques (compétition, interblocage, famine) ; Schémas classiques.

6 Programmation concurrente

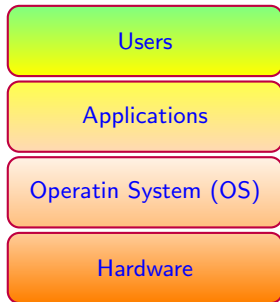
Qu'est ce qu'un thread ; Modèles d'implémentation ; POSIX threads.

Premier chapitre

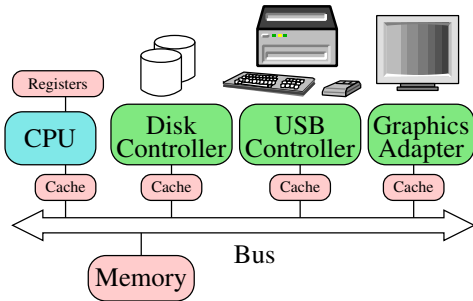
Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Organisation en couches et notion d'interfaces
- Protection des ressources
- Évolutions récentes
- Conclusion

Structure générale d'un système informatique



OS : logiciel qui permet l'exploitation du matériel (interface inférieure) offrant des services (interface supérieure unifiée) aux applications



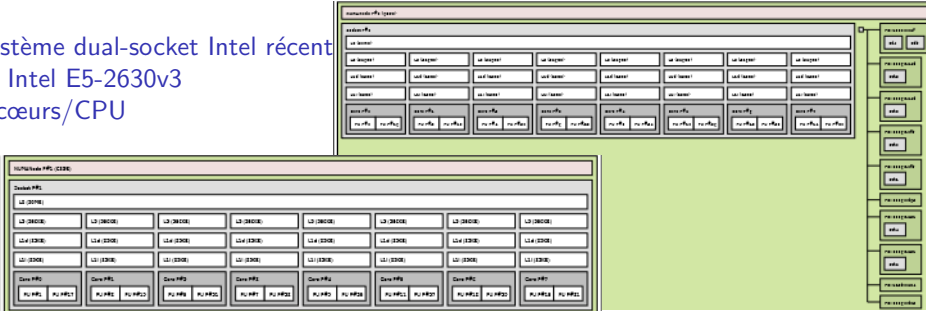
- ▶ CPU (Central Processing Unit)
- ▶ Structures de stockage
- ▶ Périphériques d'E/S
- ▶ Communication via un **bus**

Les architectures et infrastructures modernes sont complexes

- ▶ Wikipedia : 1145 serveurs, 30 900 CPUs
- ▶ OVH : 250 000 serveurs

Systeme informatique : le matériel

Système dual-socket Intel récent
2x Intel E5-2630v3
8 cœurs/CPU

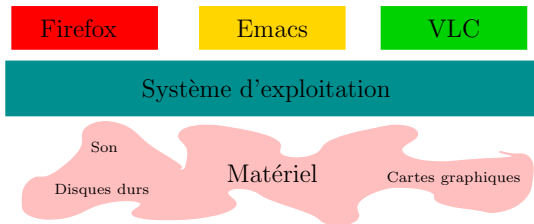


- ▶ Hiérarchie de caches entre les processeurs et la mémoire
 - ▶ Partagés (L3) ou non (L1, L2) entre cœurs physiques
- ▶ Plusieurs files d'attente pour le même cœur physique (*Hyperthreading*)
- ▶ Mémoire séparée en deux, chaque moitié reliée à un processeur différent
 - ▶ Architecture NUMA : Non-Uniform Memory Access
- ▶ Périphériques PCI reliés à un seul des deux processeurs
- ▶ Pour l'exploiter pleinement, il faut en comprendre les détails

Rôle de l'OS : **intermédiaire matériel** ↔ **applications**

Deux fonctions complémentaires

- ▶ **Adaptation d'interface** : offre une interface plus pratique que le matériel
 - ▶ Dissimule les détails de mise en œuvre (abstraction)
 - ▶ Dissimule les limitations physiques (taille mémoire) et le partage des ressources
- ▶ **Gestion des ressources** (mémoire, processeur, disque, réseau, affichage, etc.)
 - ▶ Alloue les ressources aux applications qui le demandent
 - ▶ Partage les ressources entre les applications
 - ▶ Protège les applications les unes des autres ; empêche l'usage abusif de ressources



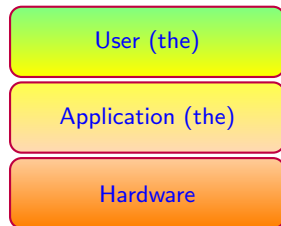
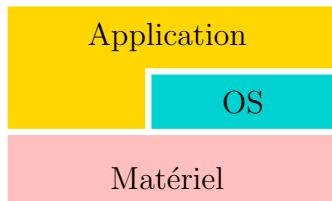
Fonctions du système d'exploitation

	Organe physique	Entité logique
▶ Gestion de l'activité		
▶ Déroulement de l'exécution (processus)		
▶ Événements matériels	Processeur	Processus
▶ Gestion de l'information		
▶ Accès dynamique (exécution, mémoire)	Mémoire principale	Mémoire virtuelle
▶ Conservation permanente	Disque	Fichier
▶ Partage		
▶ Gestion des communications		
▶ Interface avec utilisateur	Écran	
▶ Impression et périphériques	clavier, souris	Fenêtre
▶ Réseau	Imprimante	
	Réseau	
▶ Protection		
▶ de l'information		
▶ des ressources	Tous	Toutes

Évolution des systèmes d'exploitation (1/2)

Une machine, un utilisateur, une application

- ▶ À l'origine, pas d'OS :
 - ▶ Premières machines : besoin d'un opérateur humain²
 - ▶ Aujourd'hui : systèmes embarqués (voitures, ascenseurs)
- ▶ Entre temps : MS-DOS / mode non protégé



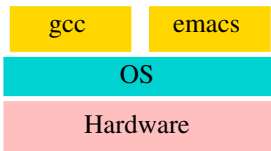
Systèmes monoprogrammés (monojob)

- ▶ **Problème** : usage inefficace des ressources matérielles : pas de recouvrement des E/S
- ▶ **Solution** : permettre la coexistence de plusieurs programmes dans le système afin de maximiser l'utilisation du matériel

Évolution des systèmes d'exploitation (2/2)

Systèmes multiprogrammés (multijob)

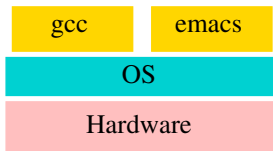
- ▶ Plusieurs programmes peuvent être chargés en mémoire en même temps.
Si un processus se bloque (E/S), un autre peut être exécuté
- ▶ Problèmes liés à la **concurrency** entre processus sont à résoudre : Si un processus fait une boucle infinie, écrit dans la mémoire d'un autre processus ?
→ **Fonction de protection** à assurer par l'OS



Évolution des systèmes d'exploitation (2/2)

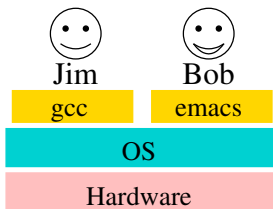
Systèmes multiprogrammés (multijob)

- ▶ Plusieurs programmes peuvent être chargés en mémoire en même temps.
Si un processus se bloque (E/S), un autre peut être exécuté
- ▶ Problèmes liés à la **concurrence** entre processus sont à résoudre : Si un processus fait une boucle infinie, écrit dans la mémoire d'un autre processus ?
→ **Fonction de protection** à assurer par l'OS



Systèmes multi-utilisateur (multiuser)

- ▶ apparus avec l'arrivée des terminaux (70's) :
mutualisation des ressources (1 machine/user : cher)
- ▶ Plusieurs utilisateurs sont autorisés à lancer leurs programmes sur la même machine.
- ▶ Problèmes liés à la **gestion** des utilisateurs sont à résoudre : utilisateur trop gourmand, mal intentionné, ...
→ **Fonction de protection** (OS)



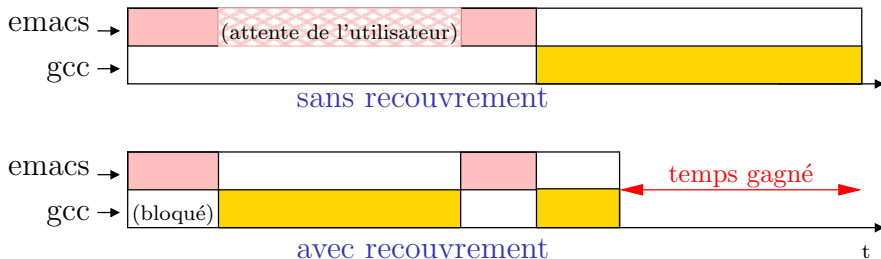
Le recouvrement des calculs et des communications

► L'ordinateur a des activités différentes

La décomposition est une réponse classique à la complexité

► Les communications bloquent les processus

(communication au sens large : réseau, disque ; utilisateur, autre programme)

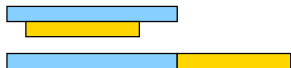


► Parallélisme sur les machines multi-processeurs

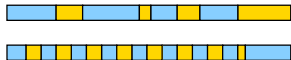
Parallélisme et pseudo-parallélisme

Que faire quand deux processus sont prêts à s'exécuter ?

- ▶ Si deux processeurs, tout va bien.
- ▶ Sinon, FIFO ? Mauvaise interactivité !



- ▶ Pseudo-parallélisme = chacun son tour
- ▶ Autre exécution pseudo-parallèle



Le pseudo-parallélisme

- ▶ fonctionne grâce aux interruptions matérielles régulières rendant contrôle à OS
- ▶ permet également de recouvrir calcul et communications

On y reviendra.

Premier chapitre

Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Organisation en couches et notion d'interfaces
- Protection des ressources
- Évolutions récentes
- Conclusion

UNIX, POSIX et les autres

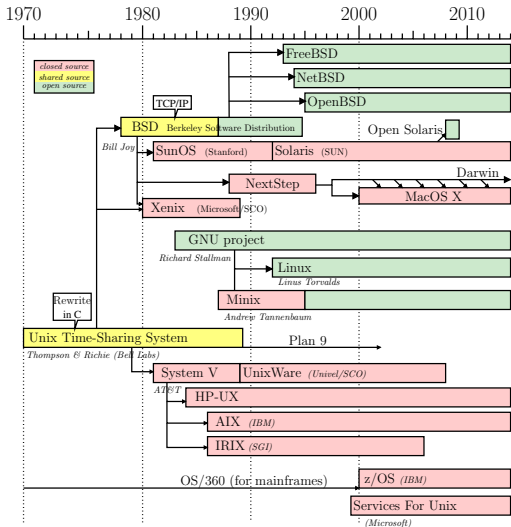
Qu'est ce qu'UNIX ? Pourquoi étudier UNIX ?

- ▶ Famille de systèmes d'exploitation, nombreuses implémentations
- ▶ Impact historique primordial, souvent copié
- ▶ Concepts de base simple, interface *historique* simple
- ▶ Des versions sont *libres*, tradition de code ouvert

Et pourquoi pas Windows ?

- ▶ En temps limité, il faut bien choisir ; je connais mieux les systèmes UNIX
- ▶ Même s'il y a de grosses différences, les concepts sont comparables
- ▶ D'autres cours à TELECOM Nancy utilisent Windows occasion d'étudier ce système ; volum horaire des cours systèmes est réduit
- ▶ Système plus *transparent* que Windows : plus facile de comprendre ce qui se passe sous le capot

Historique



Youtube Video - Histoire d'UNIX

1965 MULTICS : projet de système ambitieux (Bell Labs)

1969 Bell Labs se retire de MULTICS, Début de UNICS

1970 Unix projet Bell Labs officiel

1973 Réécriture en C
Distribution source aux universités
Commercialisation par AT&T

80-90 Unix War : BSD vs. System V

90-10 Effort de normalisation :

Posix1(88) processus, signaux, tubes

Posix1.b(93) semaphores, shared memory

Posix1.c (95) threads

Posix2 (92) interpréteur

Posix :2001, Posix :2004, Posix :2008

Mises à jour

Premier chapitre

Introduction

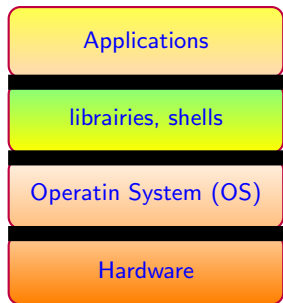
- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Organisation en couches et notion d'interfaces
- Protection des ressources
- Évolutions récentes
- Conclusion

Organisation en couches

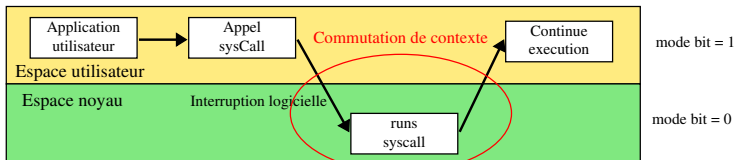
- ▶ facilite la conception et permet la protection du matériel
- ▶ 2 modes d'exécution minimum fournis par le matériel :
 - ▶ mode utilisateur : (**user**)
 - ▶ mode noyau (**kernel**)

user mode

kernel mode

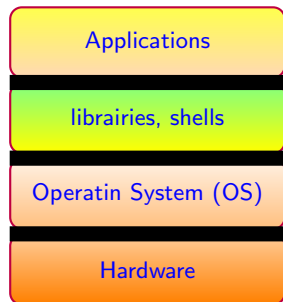


- ▶ Le mode noyau permet la réalisation des services privilégié (accès au matériel - périphériques (E/S)) - **Appels systèmes** ou syscalls



Notion d'interfaces

- ▶ Un service (d'une couche à une autre) est caractérisé par son **interface** = l'ensemble des fonctions accessibles aux utilisateurs du service
- ▶ Chaque fonction est définie par la description de :
 - ▶ son mode d'utilisation : le format (**syntaxe**)
 - ▶ son effet : la spécification (**sémantique**)
- ▶ Les descriptions doivent être précises, complètes (y compris les cas d'erreur), non ambiguës.



Principe : **séparation entre interface et réalisation**

- ▶ Facilite la portabilité
 - ▶ Transport d'une application utilisant le service sur une autre réalisation
 - ▶ Passage d'un utilisateur sur une autre réalisation du système
- ▶ Les réalisations sont interchangeables facilement

Dans POSIX

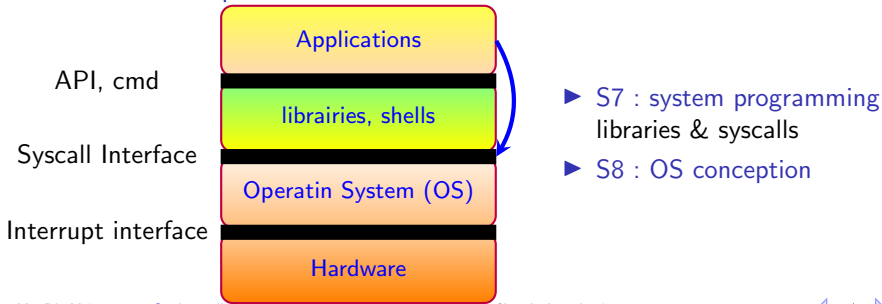
- ▶ C'est l'interface qui est normalisée, pas l'implémentation

Interfaces d'un OS type UNIX

Interface supérieure fournit des services aux processus utilisateurs

- ▶ Interface de programmation
 - ▶ Bibliothèques de fonctions C - **API** :
 - ▶ Les **appels systèmes** (fonctions spéciales)
- ▶ Interface de commande ou interface de l'utilisateur composée d'un ensemble de **commandes** :
 - ▶ textuelles - exemple : `rm *.o`
 - ▶ graphiques - exemple : déplacer l'icône d'un fichier

Interface inférieure composée d'un ensemble de sous-programmes invoqués pour le traitement des **interruptions matérielles**



Exemple d'usage des interfaces d'UNIX

Copier un fichier dans un autre

► Appels systèmes

read et write offrent un contrôle fin sur les opérations d'E/S

```
while((nb_read = read(src, buf, sizeof(buf)))>0){  
    nb_written = 0;  
    while (nb_written < nb_read) {  
        tmp = write(dest, buf + nb_written, nb_read - nb_written);  
        if (tmp == -1) break; /* erreur */  
        nb_written += tmp;  
    }  
}
```

► Fonctions de librairie

fread et fwrite gèrent automatiquement les buffers et positions dans les fichiers

```
while((nb_read = fread(buf, 1, sizeof(buf), src))>0){  
    if (fwrite(buf, 1, nb_read, dest) != nb_read) {  
        break; /* erreur */  
    }  
}
```

► Interface de commande shell : \$ cp src dest

Documentation

`man 1 <nom>` : documentation des commandes (par défaut)

`man 2 <nom>` : documentation des appels système

`man 3 <nom>` : documentation des fonctions

Premier chapitre

Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Organisation en couches et notion d'interfaces
- Protection des ressources
- Évolutions récentes
- Conclusion

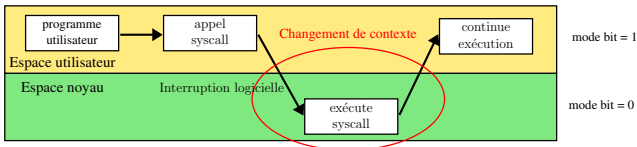
Protection des ressources (rapide introduction – cf. RSA)

Méthodes d'isolation des programmes (et utilisateurs) dangereux

- ▶ **Prémption** : ne donner aux applications que ce qu'on peut leur reprendre
- ▶ **Interposition** : pas d'accès direct, vérification de la validité de chaque accès
- ▶ **Mode privilégié** : certaines instructions machines réservées à l'OS

Exemples de ressources protégées

- ▶ **Processeur** : préemption
 - ▶ Interruptions (matérielles) à intervalles réguliers → contrôle à l'OS
 - ▶ (l'OS choisit le programme devant s'exécuter ensuite)
- ▶ **Mémoire** : interposition (validité de tout accès mémoire vérifiée au préalable)
- ▶ **Exécution** : mode privilégié (espace noyau) ou normal (espace utilisateur)
 - ▶ Le CPU bloque certaines instructions assembleur (E/S) en mode utilisateur



Limites de la protection

Les systèmes réels ont des failles

Les systèmes ne protègent pas de tout

- ▶ `while true ; do mkdir toto; cd toto; done` (en shell)
- ▶ `while(1) { fork(); }` (en C)
- ▶ `while(1) { char *a=malloc(512); *a='1'; }` (en C)

Réponse classique de l'OS : gel (voire pire)

On suppose que les utilisateurs ne sont pas mal intentionnés (erreur?)

Unix was not designed to stop people from doing stupid things, because that would also stop them from doing clever things.

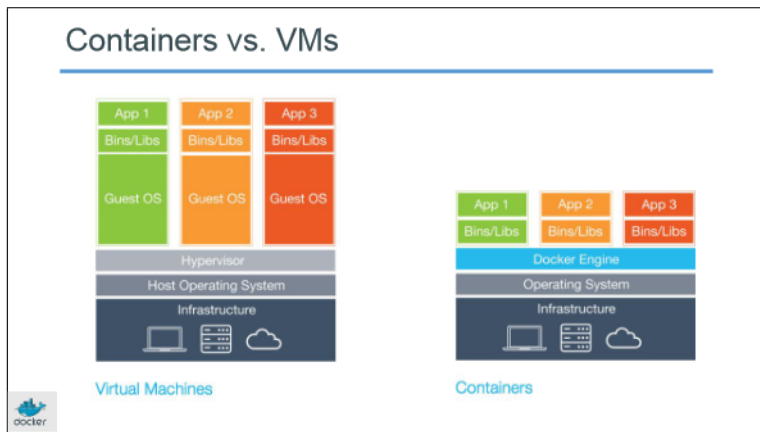
– Doug Gwyn

Deux types de solutions

Technique : mise en place de quotas

Sociale : “éduquer” les utilisateurs trop gourmands

Évolution récente : virtualisation et conteneurs



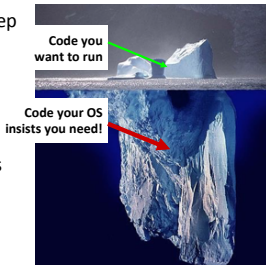
- ▶ **Virtualisation** : un OS hôte (l'hyperviseur) permet d'héberger plusieurs systèmes *invités* en virtualisant le matériel \leadsto Xen, KVM, VMWare ESXi, etc.
- ▶ **Conteneurs** : l'OS hôte permet de distinguer plusieurs espaces utilisateurs isolés (mais qui partagent le même espace noyau) \leadsto Docker, LXC

Évolution récente : unikernels

- ▶ Noyau minimaliste, spécialisé (un seul service), même modes d'exécution et même espace mémoire (rapidité),
- ▶ Un ensemble minimal de bibliothèques qui correspondent aux services OS requis par une application sont compilées avec l'application et ses configurations pour créer des images fixes.
- ▶ Une image fonctionne directement sur un hyperviseur ou sur du matériel (sans OS).

Microservices: Tip of the Iceberg

- The horrors of the deep
 - Microservices rely on millions of lines of unnecessary, unsafe code
 - Attack surface
- So very much systems code



2

Résumé du premier chapitre

- ▶ Le système d'exploitation
 - ▶ Un intermédiaire entre les applications et le matériel - organisation en couches
- ▶ Rôles et fonctions du système d'exploitation
 - ▶ Offrir une **interface du matériel** unifiée et plus adaptée
 - ▶ Assurer la **gestion** et la **protection des ressources**
- ▶ Les différentes interfaces d'un système d'exploitation
 - ▶ Interface de programmation : API (bibliothèques C & appels systèmes)
 - ▶ Interface de commande (shell, environnement graphique)
- ▶ Techniques classiques de protection des ressources
 - ▶ Préemption
 - ▶ Interposition
 - ▶ Mode d'exécution (protégé ou non)

Yet another video !

Deuxième chapitre

Processus

- Introduction
- Processus UNIX
 - Contexte, environnement
 - Création des processus dans Unix
 - Quelques interactions entre processus dans Unix
 - Réalisation des processus : Ordonnancement & Mémoire virtuelle
- Communication par signaux
 - Principe et utilité
 - Terminaux, sessions et groupe en Unix
 - Exemples d'utilisation des signaux
- Conclusion

Qu'est ce qu'un processus ?

Un Processus est l'entité dynamique qui représente l'exécution un programme sur un processeur

► Programme :

Code + data (**passif**)

```
int i;  
int main() {  
    printf("Salut\n");  
}
```

► Processus :

Programme **en cours d'exécution**

Stack	
Heap	
Data	int i;
Code	main()

- 2 personnes qui exécutent le même programme sur la même machine :
 - 2 processus différents.

Relations entre processus

Compétition

- ▶ Plusieurs processus veulent accéder à une ressource exclusive (*i.e.* ne pouvant être utilisée que par un seul à la fois) :
 - ▶ Processeur (cas du pseudo-parallélisme)
 - ▶ Imprimante, carte son
- ▶ Une **solution possible** parmi d'autres :
FIFO : premier arrivé, premier servi (les suivants **attendent** leur tour)

Coopération

- ▶ Plusieurs processus collaborent à une tâche commune
- ▶ Souvent, ils doivent se **synchroniser** :
 - ▶ p1 produit un fichier, p2 imprime le fichier
 - ▶ p1 met à jour un fichier, p2 consulte le fichier
- ▶ La synchronisation se ramène à :
p2 doit **attendre** que p1 ait franchi un certain point de son exécution

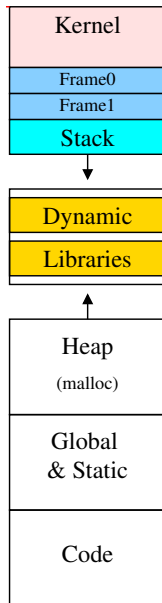
Deuxième chapitre

Processus

- Introduction
- Processus UNIX
 - Contexte, environnement
 - Création des processus dans Unix
 - Quelques interactions entre processus dans Unix
 - Réalisation des processus : Ordonnancement & Mémoire virtuelle
- Communication par signaux
 - Principe et utilité
 - Terminaux, sessions et groupe en Unix
 - Exemples d'utilisation des signaux
- Conclusion

Processus UNIX

- ▶ Processus = exécution d'un programme
 - ▶ Commande (shell)
 - ▶ Application
- ▶ **Le contexte d'un processus** est l'ensemble des données qui le caractérise permettant de reprendre son exécution si interrompu.
 - ▶ Contexte utilisateur : code, data & stack (user)
 - ▶ Contexte matériel : l'ensemble des registres du processeur
 - ▶ Contexte système : les structures de données du système pour la réalisation du processus.
- ▶ Les processus sont **identifiés** par leur **pid**
 - ▶ Commande ps : liste des processus
 - ▶ Commande top : montre l'activité du processeur
 - ▶ Primitive getpid() : renvoie le pid du processus courant



Environnement d'un processus

- ▶ Ensemble de variables accessibles par le processus (sorte de configuration)
- ▶ Principaux avantages :
 - ▶ L'utilisateur n'a pas à redéfinir son contexte pour chaque programme
Nom de l'utilisateur, de la machine, terminal par défaut, ...
 - ▶ Permet de configurer certains éléments
Chemin de recherche des programmes (PATH), shell utilisé, ...
- ▶ Certaines sont prédéfinies par le système (et modifiables par l'utilisateur)
- ▶ L'utilisateur peut créer ses propres variables d'environnement
- ▶ Interface (dépend du shell) :

Commande tcsh	Commande bash	Action
setenv	printenv	affiche toutes les variables définies
setenv VAR <valeur>	export VAR=<valeur>	attribue la valeur à la variable
echo \$VAR	echo \$VAR	affiche le contenu de la variable

- ▶ Exemple : `export DISPLAY=blaise.loria.fr:0.0` définit le terminal utilisé
- ▶ L'interface de programmation sera vue en TD/TP

Vie et mort des processus

Tout processus a un début et une fin

- ▶ **Début** : création par un autre processus
 - ▶ `init` est le processus originel : `pid=1` (`launchd` sous mac)
Créé par le noyau au démarrage, il lance les autres processus système
- ▶ **Fin**
 - ▶ Auto-destruction (à la fin du programme) (par `exit`)
 - ▶ Destruction par un autre processus (par `kill`)
 - ▶ Destruction par l'OS (en cas de violation de protection et autres)
 - ▶ Certains processus ne se terminent pas avant l'arrêt de la machine
 - ▶ Nommés «démons» (`disk and execution monitor` → `daemon`)
 - ▶ Réalisent des fonctions du système (`login utilisateurs`, `impression`, `serveur web`)

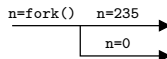
Création de processus dans UNIX

- ▶ Dans le langage de commande :
 - ▶ Chaque commande est exécutée dans un processus séparé
 - ▶ On peut créer des processus en (pseudo-)parallèle :
`$ prog1 & prog2 & # crée deux processus pour exécuter prog1 et prog2`
`$ prog1 & prog1 & # lance deux instances de prog1`
- ▶ Par l'API : clonage avec l'appel système `fork` (cf. transparent suivant)

Création des processus dans Unix (1/3)

Appel système `pid_t fork()`

- ▶ Effet : **clone** le processus appelant
- ▶ Le processus créé (**fil**s) est une copie conforme du processus créateur (**pèr**e)
Copies conformes comme une bactérie qui se coupe en deux
- ▶ Ils se reconnaissent par la valeur de retour de `fork()` :
 - ▶ Pour le père : le pid du fils (ou `-1` si erreur)
 - ▶ Pour le fils : `0`



Exemple :

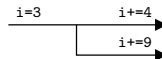
```
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
} else {
    printf("je suis le fils, mon PID est %d; mon père est %d\n",
        getpid(), getppid());
    /* en général exec(), (exécution d'un nouveau programme) */
}
```

Création des processus dans Unix (2/3)

Duplication du processus père \Rightarrow duplication de l'espace d'adressage

```
int i=3;
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    i += 4;
} else {
    printf("je suis le fils, mon PID est %d; mon père est %d\n",
        getpid(), getppid());
    i += 9;
}
printf("pour %d, i = %d\n", getpid(), i);
```

je suis le fils, mon PID est 10271; mon père est 10270
pour 10271, i = 12
je suis le père, mon PID est 10270
pour 10270, i = 7

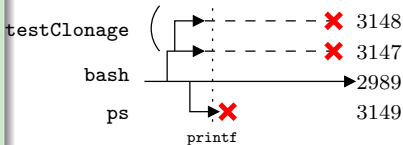


Création des processus dans Unix (3/3)

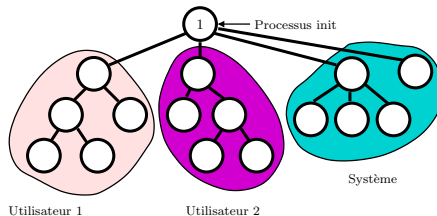
testClonage.c

```
int main() {
    if (fork() != 0) {
        printf("je suis le père, mon PID est %d\n", getpid());
        sleep(10) /* blocage pendant 10 secondes */;
        exit(0);
    } else {
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(10) /* blocage pendant 10 secondes */;
        exit(0);
    }
}
```

```
$ gcc -o testClonage testClonage.c
$ ./testClonage & ps
je suis le fils, mon PID est 3148
je suis le père, mon PID est 3147
[2] 3147
  PID TTY          TIME CMD
2989 pts/0    00:00:00 bash
3147 pts/0    00:00:00 testClonage
3148 pts/0    00:00:00 testClonage
3149 pts/0    00:00:00 ps
$
```



Hiérarchie de processus Unix

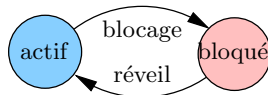


► Quelques appels systèmes utiles :

- `getpid()` : obtenir le numéro du processus
- `getppid()` : obtenir le numéro du père
- `getuid()` : obtenir le numéro d'utilisateur (auquel appartient le processus)

Blocage de processus

- Définition d'un nouvel état de processus : **bloqué** (exécution suspendue; réveil explicite par un autre processus ou par le système)



Fonction **sleep(n)**

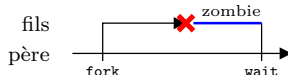
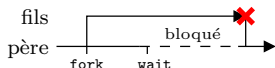
- Bloque le processus courant pour n secondes
- `unsigned int sleep(unsigned int seconds);`
- `usleep()` et `nanosleep()` offrent meilleures résolutions (micro, nanoseconde) mais interfaces plus compliquées et pas portables

somnole : affiche à chaque seconde le temps restant à dormir

```
void somnole(unsigned int secondes) {
    int i;
    for (i=0; i<secondes; i++) {
        printf("Déjà dormi %d secondes sur %d\n", i, secondes);
        sleep(1);
    }
}
```

Synchronisation entre un processus père et ses fils

- ▶ Fin d'un processus : `exit`(etat)
etat est un code de fin (convention : 0 si ok, code d'erreur sinon – cf. `errno`)
- ▶ Le père attend la fin de l'un des fils : `pid_t wait(int *ptr_etat)`
retour : pid du fils qui a terminé; code de fin stocké dans `ptr_etat`
- ▶ Attendre la fin du fils `pid` :
`pid_t waitpid(pid_t pid, int *ptr_etat, int options)`
- ▶ Processus zombie : terminé, mais le père n'a pas appelé `wait()`.
Il ne peut plus s'exécuter, mais consomme encore des ressources. À éviter.

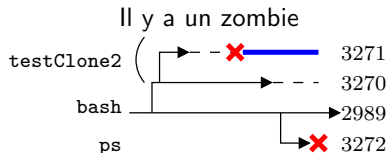


Exemple de synchronisation entre père et fils

testClone2.c

```
int main() {
    if (fork() != 0) {
        printf("je suis le père, mon PID est %d\n", getpid());
        while (1) ; /* boucle sans fin sans attendre le fils */
    } else {
        printf("je suis le fils, mon PID est %d\n", getpid());
        sleep(2) /* blocage pendant 2 secondes */
        printf("fin du fils\n");
        exit(0);
    } }
}
```

```
$ gcc -o testClone2 testClone2.c
$ ./testClone2
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
->l'utilisateur tape <ctrl-Z> (suspendre)
Suspended
$ ps
  PID TTY          TIME CMD
 2989 pts/0    00:00:00 bash
 3270 pts/0    00:00:03 testClone2
 3271 pts/0    00:00:00 testClone2 <defunct>
 3272 pts/0    00:00:00 ps
$
```



Autre exemple de synchronisation père fils

testClone3.c

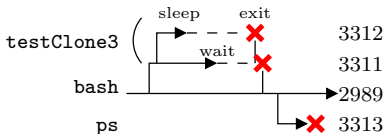
```
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    if (fork() != 0) {
        int statut; pid_t fils;
        printf("Le père (%d) attend.\n", getpid());
        fils = wait(&statut);
        if (WIFEXITED(statut)) {
            printf("%d : fils %d terminé (code %d)\n",
                getpid(), fils, WEXITSTATUS(statut));
        }
    };
    exit(0);
}
```

```
$ ./testClone3
je suis le fils, PID=3312
Le père (3311) attend
fin du fils
3311: fils 3312 terminé (code 1)
$ ps
  PID TTY          TIME CMD
 2989 pts/0    00:00:00 bash
 3313 pts/0    00:00:00 ps
$
```

(suite de testClone3.c)

```
    } else { /* correspond au if (fork() != 0) */
        printf("je suis le fils, PID=%d\n",
            getpid());
        sleep(2) /* blocage pendant 2 secondes */
        printf("fin du fils\n");
        exit(1);
    } }
```

Il n'y a pas de zombie



Exécution d'un programme spécifié sous UNIX

Appels systèmes **exec**

- ▶ Pour faire exécuter un nouveau programme par un processus
- ▶ Souvent utilisé immédiatement après la création d'un processus :
fork+exec = lancement d'un programme dans un nouveau processus
- ▶ **Effet** : remplace la mémoire virtuelle du processus par le programme
- ▶ Plusieurs variantes existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement)
- ▶ C'est aussi une primitive du langage de commande (même effet)

Exemple :

```
main() {  
    if (fork() == 0) {  
  
        execl("/bin/ls", "ls", "-a", 0); /* le fils exécute : /bin/ls -a .. */  
  
    } else {  
        wait(NULL); /* le père attend la fin du fils */  
    }  
    exit(0);  
}
```

Exécution d'un programme spécifié sous UNIX

Appels systèmes **exec**

- ▶ Pour faire exécuter un nouveau programme par un processus
- ▶ Souvent utilisé immédiatement après la création d'un processus :
fork+exec = lancement d'un programme dans un nouveau processus
- ▶ **Effet** : remplace la mémoire virtuelle du processus par le programme
- ▶ Plusieurs variantes existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement)
- ▶ C'est aussi une primitive du langage de commande (même effet)

Exemple :

```
main() {
    if (fork() == 0) {

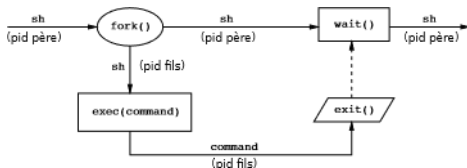
        code=execl("/bin/ls", "ls", "-a", 0); /* le fils exécute : /bin/ls -a .. */
        if (code != 0) { ... } /* Problème dans l'appel système; cf. valeur de errno */
    } else {
        wait(NULL); /* le père attend la fin du fils */
    }
    exit(0);
}
```

L'exemple du shell

Exécution d'une commande en premier plan

\$ commande

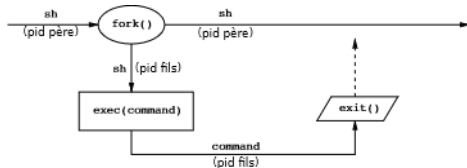
- 4 syscalls : fork, exec, exit, wait



Exécution d'une commande en tâche de fond

\$ commande &

- Le shell ne fait pas wait()
- Il n'est plus bloqué



D'après Philippe Marquet, CC-BY-NC-SA.

Résumé du début du deuxième chapitre

- ▶ Utilité des processus
 - ▶ Simplicité (séparation entre les activités)
 - ▶ Sécurité (séparation entre les activités)
 - ▶ Efficacité (*quand l'un est bloqué, on passe à autre chose*)
- ▶ Interface UNIX
 - ▶ Création : `fork()`
 - ▶ `résultat=0` → je suis le fils
 - ▶ `résultat>0` → je suis le père (`résultat=pid` du fils)
 - ▶ `résultat<0` → erreur
 - ▶ Attendre un fils : deux manières
 - ▶ `wait()` : n'importe quel fils
 - ▶ `waitpid()` : un fils en particulier
 - ▶ Processus zombie : un fils terminé dont le père n'a pas fait `wait()`
 - ▶ Appel système `exec()` : remplace l'image du process actuel par le programme spécifié

Réalisation des processus

Processus = mémoire virtuelle + flot d'exécution

L'OS fournit ces deux ressources en allouant les ressources physiques

Objectif maintenant :

En savoir assez sur le fonctionnement de l'OS pour utiliser les processus

- ▶ À propos de **mémoire**
 - ▶ Organisation interne de la mémoire virtuelle d'un processus Unix
- ▶ À propos de **processeur**
 - ▶ Pseudo-parallélisme : allocation successive aux processus par tranches de temps



Objectifs repoussés à plus tard :

- ▶ Les autres ressources : disque (chapitre 3), réseau (seconde moitié du module)
- ▶ Détails de conception *sous le capot* (module RSA)

Allocation du processeur aux processus

Pseudo-parallélisme

Principe

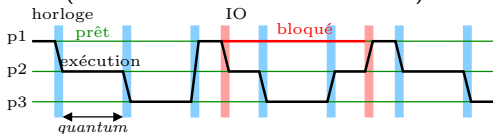
- ▶ Allocation successive aux processus par tranches de temps fixées (multiplexage du processeur par préemption)

Avantages

- ▶ Partage équitable du processeur entre processus (gestion + protection)
- ▶ Recouvrement calcul et communications (ou interactions)

Fonctionnement

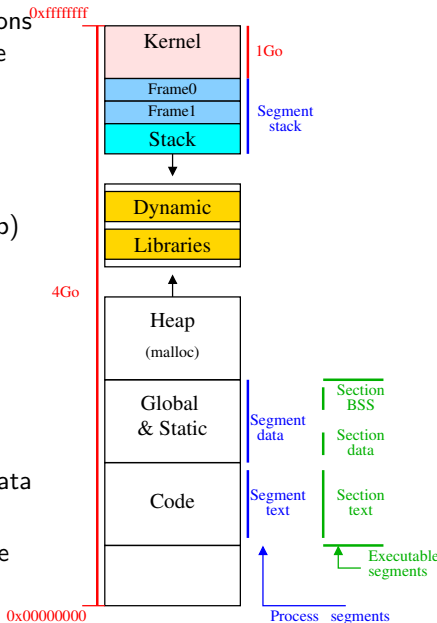
- ▶ Interruptions matérielles (top d'horloge, I/O, ...) rendent le contrôle à l'OS
- ▶ L'OS **ordonnance** les processus (choisit le prochain à bénéficier de la ressource)
- ▶ Il réalise la **commutation de processus** pour passer le contrôle à l'heureux élu
- ▶ (Comment ? Vous verrez en RSA !)



- ▶ **quantum** : $\approx 10\text{ms}$
(\approx millions d'instructions à 1Ghz)
- ▶ **temps de commutation** : $\approx 0,5\text{ms}$
- ▶ **yield()** rend la main volontairement

Structure de la mémoire virtuelle d'un processus

- ▶ Pile (stack) : contient les cadres de fonctions (*frame*) : arguments des fonctions, adresse de retour, variables locales (non statique)
- ▶ Variables globales et statiques
 - ▶ .data, si initialisées
 - ▶ .bss, sinon
- ▶ Variables dynamique (malloc) → tas (heap)
- ▶ Le tas et la pile n'ont pas de taille fixe, augmentent de taille dans 2 sens opposés
- ▶ Bibliothèques dynamiques : code chargé dynamiquement
- ▶ Noyau : infos sur le processus - contexte système : pid, autorisations, fichiers ouverts, ...
- ▶ L'exécutable contient les sections .text, .data et .bss
- ▶ **exec** lit l'exécutable et initialise la mémoire correspondante



Sécurité : attaques par corruption de la mémoire

```
void fonction(char * chaine) {  
    char buffer[128];  
    strcpy(buffer, chaine);  
}
```

- ▶ Famille de failles très répandue (*buffer overflow*)
 - ▶ Récemment : ransomware WannaCry (mai 2017)
- ▶ Principe général : faire exécuter du code quelconque à un processus
- ▶ Si chaine fait plus de 128 caractères, alors strcpy écrasera l'adresse de retour de la fonction
- ▶ Si chaine contient du code machine, le processus exécutera ce code machine
- ▶ Plus de détails : <http://mdeloisson.free.fr/downloads/memattacks.pdf>
- ▶ Mécanismes de défense :
 - ▶ **W xor X** : interdire qu'une zone mémoire soit à la fois inscriptible et exécutable
 - ▶ **canaris** pour détecter les dépassements de pile
 - ▶ **ASLR (Address Space Layout Randomization)** : les adresses des fonctions et de la pile sont plus difficiles à prévoir
 - ▶ En assembleur/C/C++, utiliser des fonctions permettant de traiter des données de manière sécurisée (`strcpy` \leadsto `strncpy`)
 - ▶ Utiliser des langages de plus haut niveau
 - ▶ Utiliser des outils de test automatiques (*fuzzing*)

Deuxième chapitre

Processus

- Introduction
- Processus UNIX
 - Contexte, environnement
 - Création des processus dans Unix
 - Quelques interactions entre processus dans Unix
 - Réalisation des processus : Ordonnancement & Mémoire virtuelle
- Communication par signaux
 - Principe et utilité
 - Terminaux, sessions et groupe en Unix
 - Exemples d'utilisation des signaux
- Conclusion

Aspect central de la programmation système

Moyens de communication entre processus sous Unix

- ▶ Signaux : suite de cette séance
- ▶ Fichiers et tubes (*pipes*, FIFOs) : [partie 3](#).
- ▶ Files de messages : pas étudié dans ce module
- ▶ Mémoire partagée et sémaphores : [partie 5](#).
- ▶ Sockets (dans les réseaux, mais aussi en local) : RSA - Réseaux

Définition : événement asynchrone

- ▶ Émis par l'OS ou un processus
- ▶ Destiné à un (ou plusieurs) processus
 - ▶ en shell, `kill <pid>` tue pid (plus de détails plus tard)

Intérêts et limites

- ▶ Simplifient le contrôle d'un ensemble de processus (comme le shell)
- ▶ Pratiques pour traiter des événements liés au temps
- ▶ Mécanisme de bas niveau à manipuler avec précaution (risque de perte de signaux en particulier)

Définition : événement asynchrone

- ▶ Émis par l'OS ou un processus
- ▶ Destiné à un (ou plusieurs) processus
 - ▶ en shell, `kill <pid>` tue pid (plus de détails plus tard)

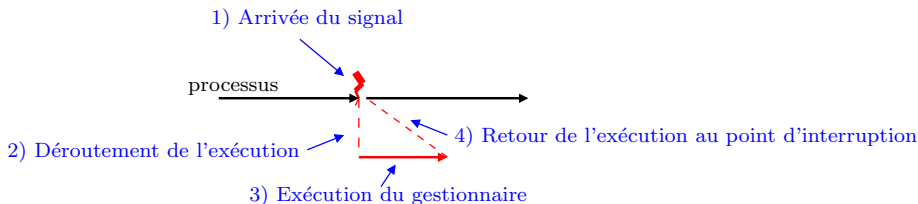
Intérêts et limites

- ▶ Simplifient le contrôle d'un ensemble de processus (comme le shell)
- ▶ Pratiques pour traiter des événements liés au temps
- ▶ Mécanisme de bas niveau à manipuler avec précaution (risque de perte de signaux en particulier)

Comparaison avec les interruptions matérielles :

- ▶ Analogie : la réception déclenche l'exécution d'un gestionnaire (*handler*)
- ▶ Différences : interruption reçue par processeur ; signal reçu par processus
Certains signaux traduisent la réception d'une interruption (on y revient)

Fonctionnement des signaux



Remarques (on va détailler)

- ▶ On ne peut évidemment *signaler* que ses propres processus (même uid)
- ▶ Différents signaux, identifiés par un nom symbolique (et un entier)
- ▶ Gestionnaire par défaut pour chacun
- ▶ Gestionnaire vide \Rightarrow ignoré
- ▶ On peut changer le gestionnaire (sauf exceptions)
- ▶ On peut bloquer un signal : mise en attente, délivré qu'après déblocage
- ▶ Limites aux traitements possibles dans le gestionnaire (ex : pas de `signal()`)

Quelques exemples de signaux

Nom symbolique	Cause/signification	Par défaut
SIGINT	frappe du caractère <CTRL-C>	terminaison
SIGTSTP	frappe du caractère <CTRL-Z>	suspension
SIGSTOP	blocage d'un processus (*)	suspension
SIGCONT	continuation d'un processus stoppé	reprise
SIGTERM	demande de terminaison	terminaison
SIGKILL	terminaison immédiate (*)	terminaison
SIGSEGV	erreur de segmentation (violation de protection mémoire)	terminaison + <i>core dump</i>
SIGALRM	top d'horloge (réglée avec <code>alarm</code>)	terminaison
SIGCHLD	terminaison d'un fils	ignoré
SIGUSR1	pas utilisés par le système	terminaison
SIGUSR2	(disponibles pour l'utilisateur)	terminaison

► KILL et STOP : ni bloquables ni ignorables ; gestionnaire non modifiable.

► Valeurs numériques associées (ex : SIGKILL=9), mais pas portable

► Voir `man 7 signal` pour d'autres signaux (section 7 du `man` : conventions)

core dump : copie image mémoire sur disque (premières mémoires : toriques → core ; dump=vider)

États d'un signal

Signal **pendant** (*pending*)

- ▶ Arrivé au destinataire, mais pas encore traité

Signal **traité**

- ▶ Le gestionnaire a commencé (et peut-être même fini)

Pendant, mais pas traité ? Est-ce possible ?

- ▶ Il est **bloqué**, càd retardé : il sera délivré lorsque débloqué
- ▶ Lors de l'exécution du gestionnaire d'un signal, ce signal est bloqué

Attention : au plus un signal pendant de chaque type

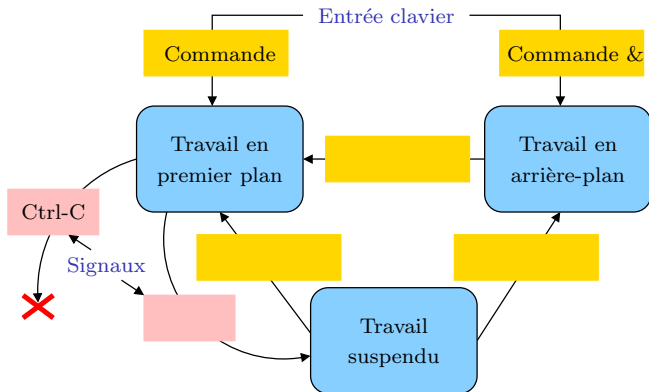
- ▶ L'information est codée sur un seul bit
- ▶ S'il arrive un autre signal du même type, le second est perdu

Deuxième chapitre

Processus

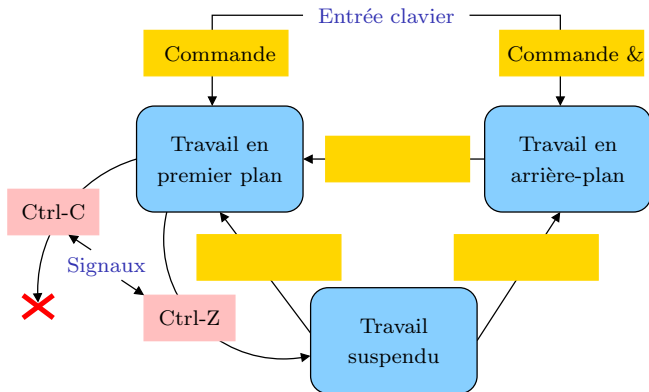
- Introduction
- Processus UNIX
 - Contexte, environnement
 - Création des processus dans Unix
 - Quelques interactions entre processus dans Unix
 - Réalisation des processus : Ordonnancement & Mémoire virtuelle
- Communication par signaux
 - Principe et utilité
 - Terminaux, sessions et groupe en Unix
 - Exemples d'utilisation des signaux
- Conclusion

États d'un travail



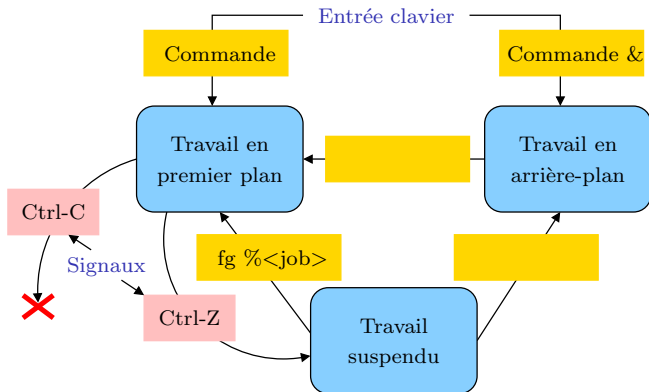
- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

États d'un travail



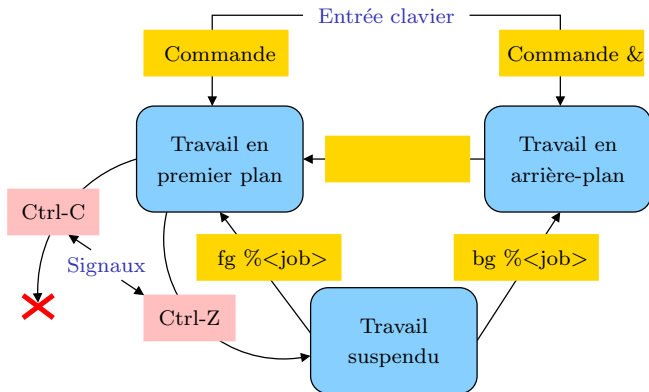
- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

États d'un travail



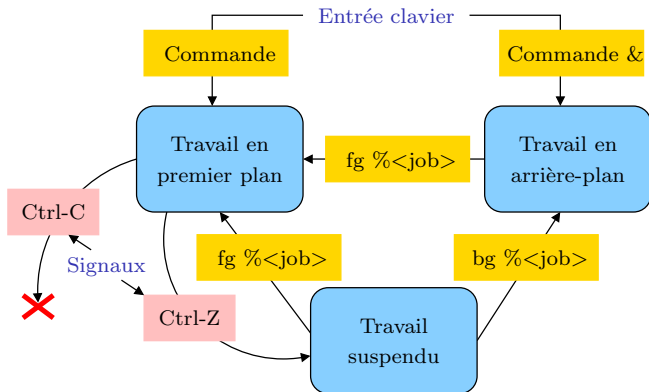
- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

États d'un travail



- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

États d'un travail



- ▶ **Travail** (*job*) = (groupe de) processus lancé par une commande au shell
- ▶ Seul le travail en premier plan peut recevoir des signaux du clavier
- ▶ Les autres sont manipulés par des commandes

Terminaux, sessions et groupes en Unix

- ▶ Concept important pour comprendre les signaux sous Unix (détaillé plus tard)
- ▶ Une **session** est associée à un **terminal**, donc au login d'un utilisateur par shell
Le processus de ce shell est le **leader de la session**.
- ▶ Plusieurs groupes de processus par session. On dit plusieurs **travaux** (*jobs*)
- ▶ Au plus un travail interactif (**avant-plan**, *foreground*)
Interagissent avec l'utilisateur via le terminal, seuls à pouvoir lire le terminal
- ▶ Plusieurs travaux en **arrière plan** (*background*)
Lancés avec & ; Exécution en travail de fond
- ▶ Signaux SIGINT (frappe de <CTRL-C>) et SIGTSTP (frappe de <CTRL-Z>) sont passés au groupe interactif et non aux groupes d'arrière-plan

Exemple avec les sessions et groupes Unix

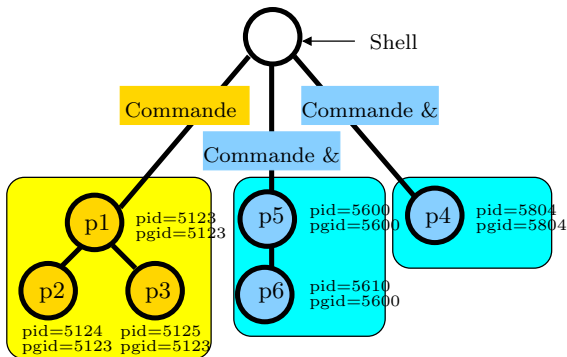
loop.c

```
int main() {  
    printf("processus %d, groupe %d\n", getpid(), getpgrp());  
    while(1) ;  
}
```

```
$ loop & loop & ps  
processus 10468, groupe 10468  
[1] 10468  
processus 10469, groupe 10469  
[2] 10469  
    PID TTY          TIME CMD  
  5691 pts/0    00:00:00 bash  
 10468 pts/0    00:00:00 loop  
 10469 pts/0    00:00:00 loop  
 10470 pts/0    00:00:00 ps  
$ fg %1  
loop  
[frappe de control-Z]  
Suspended  
$ jobs  
[1] + Suspended          loop  
[2] - Running             loop
```

```
$ bg %1  
[1] loop &  
$ fg %2  
loop  
[frappe de control-C]  
$ ps  
    PID TTY          TIME CMD  
  5691 pts/0    00:00:00 bash  
 10468 pts/0    00:02:53 loop  
 10474 pts/0    00:00:00 ps  
$ [frappe de control-C]  
$ ps  
    PID TTY          TIME CMD  
  5691 pts/0    00:00:00 bash  
 10468 pts/0    00:02:57 loop  
 10475 pts/0    00:00:00 ps  
$
```

Exemple de travaux



- ▶ Signaux CTRL-C et CTRL-Z adressés à **tous** les processus du groupe jaune
- ▶ Commandes shell fg, bg et stop pour travaux bleus

Deuxième chapitre

Processus

- Introduction
- Processus UNIX
 - Contexte, environnement
 - Création des processus dans Unix
 - Quelques interactions entre processus dans Unix
 - Réalisation des processus : Ordonnancement & Mémoire virtuelle
- Communication par signaux
 - Principe et utilité
 - Terminaux, sessions et groupe en Unix
 - Exemples d'utilisation des signaux
- Conclusion

Envoyer un signal à un autre processus

Interfaces

- ▶ Langage de commande :
- ▶ Appel système :

```
kill -NOM victime
```

```
#include <signal.h>

int kill(pid_t victime, int sig);
```

Sémantique : à qui est envoyé le signal ?

- ▶ Si $victime > 0$, au processus tel que $pid = victime$
- ▶ Si $victime = 0$, à tous les processus du même groupe (pgid) que l'émetteur
- ▶ Si $victime = -1$:
 - ▶ Si super-utilisateur, à tous les processus sauf système et émetteur
 - ▶ Si non, à tous les processus dont l'utilisateur est propriétaire
- ▶ Si $victime < -1$, aux processus tels que $pgid = |victime|$ (tout le groupe)

Redéfinir le gestionnaire associé à un signal (POSIX)

Structure à utiliser pour décrire un gestionnaire

```
struct sigaction {  
    void (*sa_handler)(int);           /* gestionnaire, interface simple */  
    void (* sa_sigaction) (int, siginfo_t *, void *); /* gestionnaire, interface complète */  
    sigset_t sa_mask;                  /* signaux à bloquer pendant le traitement */  
    int sa_flags;                       /* options */  
};
```

- ▶ **Gestionnaires particuliers** : SIG_DFL : action par défaut ; SIG_IGN : ignorer signal
- ▶ Deux types de gestionnaires
 - ▶ sa_handler() connaît le numéro du signal
 - ▶ sa_sigaction() a plus d'infos

Primitive à utiliser pour installer un nouveau gestionnaire

```
#include <signal.h>  
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

- ▶ Voir man sigaction pour les détails

Autre interface existante : ANSI C

- ▶ Peut-être un peu plus simple, mais bien moins puissante et pas thread-safe

Exemple 1 : traitement d'une interruption du clavier

- ▶ Par défaut, Ctrl-C tue le processus ; pour survivre : il suffit de redéfinir le gestionnaire de SIGINT

test-int.c

```
#include <signal.h>
void handler(int sig) {                               /* nouveau gestionnaire */
    printf("signal SIGINT reçu !\n");
    exit(0);
}
int main() {
    struct sigaction nvt,old;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = &handler;
    sigaction(SIGINT, &nvt, &old);                    /*installe le gestionnaire*/
    pause ();                                           /* se bloquer en attente d'un signal */
    printf("Ceci n'est jamais affiché.\n");
}
```

```
$ ./test-int
[frappe de CTRL-C]
signal SIGINT reçu !
$
```

Exercice : modifier ce programme pour qu'il continue après un CTRL-C

Note : il doit rester interruptible, *i.e.* on doit pouvoir le tuer d'un CTRL-C de plus

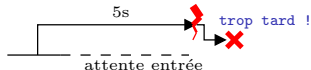
Exemple 2 : temporisation

`unsigned int alarm(unsigned int nb_sec)`

- ▶ `nb_sec > 0` : demande l'envoi de SIGALRM après environ `nb_sec` secondes
- ▶ `nb_sec = 0` : annulation de toute demande précédente
- ▶ Retour : nombre de secondes restantes sur l'alarme précédente
- ▶ Attention, `sleep()` réalisé avec `alarm()` \Rightarrow mélange dangereux

```
#include <signal.h>
void handler(int sig) {
    printf("trop tard !\n");
    exit(1);
}
int main() {
    struct sigaction nvt,old;
    int reponse,restant;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = handler;
    sigaction(SIGALRM, &nvt, &old);
```

```
    printf("Entrez un nombre avant 5 sec:");
    alarm(5);
    scanf("%d", &reponse);
    restant = alarm(0);
    printf("bien reçu (en %d secondes)\n",
        5 - restant);
    exit (0);
}
```



Exemple 3 : synchronisation père-fils

- ▶ Fin ou suspension d'un processus \Rightarrow SIGCHLD automatique à son père
- ▶ **Traitement par défaut** : ignorer ce signal
- ▶ **Application** : wait() pour éviter les zombies mangeurs de ressources
C'est ce que fait le processus init (celui dont le pid est 1)

```
#include <signal.h>
void handler(int sig) {                               /* nouveau gestionnaire */
    pid_t pid;
    int statut;
    pid = waitpid(-1, &statut, 0); /* attend un fils quelconque */
    return;
}
int main() {
    struct sigaction nvt,old;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = &handler;
    sigaction(SIGCHLD, &nvt, &old); /* installe le gestionnaire */
    ... <création d'un certain nombre de fils> ...
    exit (0);
}
```

Résumé de la fin du deuxième chapitre

- ▶ Quelques définitions
 - ▶ pgid : groupe de processus ; un par processus lancé par shell (et tous ses fils)
 - ▶ Travail d'avant-plan, d'arrière-plan : lancé avec ou sans &
- ▶ Communication par signaux
 - ▶ Envoyer un signal $\langle NOM \rangle$ à $\langle victime \rangle$:
 - ▶ Langage commande : `kill -NOM victime`
 - ▶ API : `kill(pid_t victime, int sig)`
 - $victime > 0$: numéro du pid visé
 - $victime = 0$: tous les process de ce groupe
 - $victime = -1$: tous les process accessibles
 - $victime < -1$: tous les process du groupe $abs(victime)$
 - ▶ Astuce : envoyer le signal 9 à tous les processus : `kill -9 -1`
 - ▶ Changer le gestionnaire d'un signal : `sigaction` (en POSIX)
 - ▶ Exemples de signaux
 - ▶ SIGINT, SIGSTOP : Interaction avec le travail de premier plan (ctrl-c, ctrl-z)
 - ▶ SIGTERM, SIGKILL : demande de terminaison, fin brutale
 - ▶ SIGALARM : temporisation
 - ▶ SIGCHLD : relations père-fils

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
 - Introduction
 - Désignation des fichiers
- Interface d'utilisation des fichiers
 - Interface en langage de commande
 - Interface de programmation
 - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
 - Implantation des fichiers
 - Manipulation des répertoires
 - Protection des fichiers
- Conclusion

Définitions

- ▶ **Fichier** : un ensemble d'informations groupées pour conservation et utilisation
- ▶ **Système de gestion de fichiers (SGF)** : partie de l'OS conservant les fichiers et permettant d'y accéder

Fonctions d'un SGF

- ▶ Conservation permanente des fichiers (*ie*, après la fin des processus, sur disque)
- ▶ Organisation logique et désignation des fichiers
- ▶ Partage et protection des fichiers
- ▶ Réalisation des fonctions d'accès aux fichiers

Les fichiers jouent un rôle central dans Unix

- ▶ Support des données
- ▶ Support des programmes exécutables
- ▶ Communication avec l'utilisateur : fichiers de config, stdin, stdout
- ▶ Communication entre processus : sockets, tubes, etc.
- ▶ Interface du noyau : /proc
- ▶ Interface avec le matériel : périphériques dans /dev

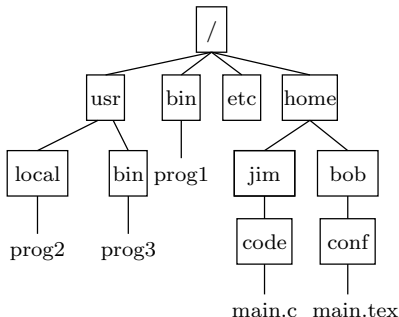
Désignation des fichiers

Désignation symbolique (nommage) : Organisation hiérarchique

- ▶ Noeuds intermédiaires : répertoires (*directory* – ce sont aussi des fichiers)
- ▶ Noeuds terminaux : fichiers simples
- ▶ Nom absolu d'un fichier : le chemin d'accès depuis la racine

Exemples de chemins absolus :

```
/
/bin
/usr/local/bin/prog
/home/bob/conf/main.tex
/home/jim/code/main.c
```



Raccourcis pour simplifier la désignation

Noms relatifs au répertoire courant

- ▶ Depuis `/home/bob`, `conf/main.tex` = `/home/bob/conf/main.tex`

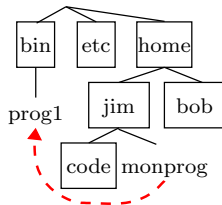
Abréviations

- ▶ **Répertoire père** : depuis `/home/bob`, `../jim/code` = `/home/jim/code`
- ▶ **Répertoire courant** : depuis `/bin`, `./prog1` = `/bin/prog1`
- ▶ Depuis n'importe où, `~bob/` = `/home/bob/` et `~/` = `/home/<moi>/`

Liens symboliques

Depuis `/home/jim`

- ▶ Création du lien : `ln -s cible nom_du_lien`
Exemple : `ln -s /bin/prog1 monprog`
- ▶ `/home/jim/prog1` désigne `/bin/prog1`
- ▶ Si la cible est supprimée, le lien devient invalide



Liens durs : `ln cible nom`

- ▶ Comme un lien symbolique, mais copies indifférenciables (ok après `rm cible`)
- ▶ Interdit pour les répertoires (cohérence de l'arbre)

Règles de recherche des exécutables

- ▶ Taper le chemin complet des exécutable (/bin/ls) est lourd
- ▶ \Rightarrow on tape le nom sans le chemin et le shell cherche
- ▶ Variable environnement PATH : liste de répertoires à examiner successivement
`/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/usr/bin:/usr/bin/X11`
- ▶ Commande `which` indique quelle version est utilisée

Exercice : Comment exécuter un script nommé `gcc` dans le répertoire courant ?

- ▶ Solution 1 : `export PATH=".: $PATH" ; gcc <bla>`
- ▶ Solution 2 : `./gcc <bla>`

Utilisations courantes des fichiers

- ▶ Unix : fichiers = suite d'octets sans structure (interprétée par utilisateur)
- ▶ Windows : différencie fichiers textes (où `\n` est modifié) des fichiers binaires

Programmes exécutables

- ▶ Commandes du système ou programmes créés par un utilisateur
- ▶ **Exemple** : `gcc -o test test.c ; ./test`
Produit programme exécutable dans fichier `test` ; exécute le programme `test`
- ▶ **Question** : pourquoi `./test` ? (car `test` est un binaire classique : `if test $n -le 0;`)

Fichiers de données

- ▶ Documents, images, programmes sources, etc.
- ▶ **Convention** : le suffixe indique la nature du contenu
Exemples : `.c` (programme C), `.o` (binaire translatable, cf. plus loin), `.h` (entête C), `.gif` (un format d'images), `.pdf` (Portable Document Format), etc.
Remarque : ne pas faire une confiance aveugle à l'extension (cf. `man file`)

Fichiers temporaires servant pour la communication

- ▶ Ne pas oublier de les supprimer après usage
- ▶ On peut aussi utiliser des tubes (cf. plus loin)

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
 - Introduction
 - Désignation des fichiers
- Interface d'utilisation des fichiers
 - Interface en langage de commande
 - Interface de programmation
 - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
 - Implantation des fichiers
 - Manipulation des répertoires
 - Protection des fichiers
- Conclusion

Utilisation des fichiers dans le langage de commande

Créer un fichier

- ▶ Souvent créés par applications (éditeur, compilateur, etc), pas par shell
- ▶ On peut néanmoins créer explicitement un fichier (cf plus loin, avec flots)

Quelques commandes utiles

créer un répertoire `mkdir <nom du répertoire>` (initialement vide)

détruire un fichier `rm <nom du fichier>` (option `-i` : interactif)

détruire un répertoire `rmdir <rep>` s'il est vide; `rm -r <rep>` sinon

chemin absolu du répertoire courant `pwd` (*print working directory*)

contenu du répertoire courant `ls` (*list* – `ls -l` plus complet)

Expansion des noms de fichiers (*globbing*)

- ▶ `*` désigne n'importe quelle chaîne de caractères :
 - `rm *.o` : détruit tous les fichiers dont le nom finit par `.o`
 - `ls *.c` : donne la liste de tous les fichiers dont le nom finit par `.c`
 - `*.[co] [A-Z]` : fichiers dont le nom termine par 'c' ou 'o' puis une majuscule

Interface de programmation POSIX des fichiers

Interface système

- ▶ Dans l'interface de programmation, un fichier est représenté par **descripteur**
Les descripteurs sont de (petits) entiers (indice dans des tables du noyau)
- ▶ Il faut **ouvrir** un fichier avant usage :

```
fd = open("/home/toto/fich", O_RDONLY, 0);
```

 - ▶ Ici, ouverture en lecture seule (écriture interdite)
 - ▶ Autres modes : O_RDWR, O_WRONLY
 - ▶ fd stocke le descripteur alloué par le système (ou -1 si erreur)
 - ▶ Fichier créé s'il n'existe pas (cf. aussi **creat**(2))
 - ▶ man 2 open pour plus de détails
- ▶ Il faut **fermer** les fichiers après usage :

```
close (fd)
```

 - ▶ Descripteur fd plus utilisable ; le système peut réallouer ce numéro

Interface standard

- ▶ Il existe une autre interface, plus simple (on y revient)

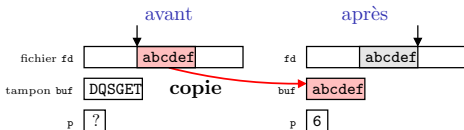
Interface de programmation POSIX : `read()`

L'objet fichier

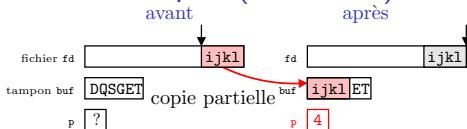
- ▶ Chaque fichier a un **pointeur courant** : tête de lecture/écriture logique
- ▶ Les lectures et écritures le déplacent implicitement
- ▶ On peut le déplacer explicitement (avec **`lseek()`** – voir plus loin)

Lecture normale (`read()`)

```
p=read(fd, buf, 6)
```



S'il n'y a pas assez de données : lecture tronquée (*short read*)



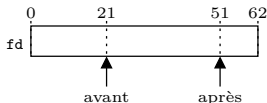
Interface de programmation POSIX : lseek()

Objectif

- Permet de déplacer le pointeur de fichier explicitement

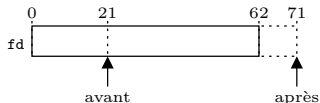
Exemples

`lseek(fd, 30, SEEK_CUR)`



+30 octets depuis position courante

`lseek(fd, 71, SEEK_SET)`



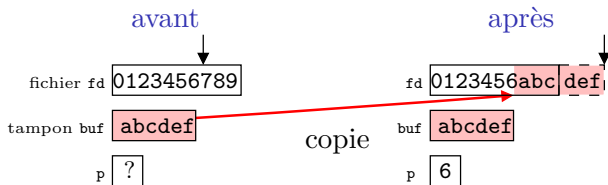
place le pointeur à la position 71

- Le pointeur peut être placé après la fin du fichier

Interface de programmation POSIX : write()

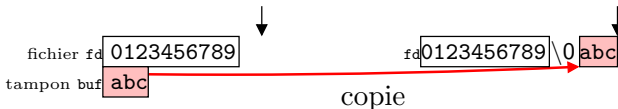
- Écriture dans les conditions normales :

```
p=write(fd, buf, 6)
```



Le fichier a été rallongé

- Si un `lseek()` a déplacé le pointeur après la fin, remplissage avec des zéros (qui ne sont pas immédiatement alloués par le SGF)



- Possibilité d'écriture tronquée (*short-write*) si le disque est plein, ou si descripteur est un tube ou une socket (cf. plus loin)

Différentes interfaces d'usage des fichiers

Interface POSIX système

- ▶ Appels systèmes `open`, `read`, `write`, `lseek` et `close`.
- ▶ Utilisation délicate : lecture/écriture tronquée, traitement d'erreur, *etc.*
- ▶ Meilleures performances après réglages précis

Différentes interfaces d'usage des fichiers

Interface POSIX système

- ▶ Appels systèmes `open`, `read`, `write`, `lseek` et `close`.
- ▶ Utilisation délicate : lecture/écriture tronquée, traitement d'erreur, etc.
- ▶ Meilleures performances après réglages précis

Bibliothèque C standard

- ▶ Fonctions `fopen`, `fscanf`, `fprintf` et `fclose`.
- ▶ Plus haut niveau (utilisation des formats d'affichage)
- ▶ Meilleures performances sans effort (POSIX : perfs au tournevis)
- ▶ Un syscall ≈ 1000 instructions \Rightarrow écritures dans un tampon pour les grouper

```
#include <stdio.h>
```

```
FILE *fd = fopen("mon_fichier","w");  
if (fd != NULL) {  
    fprintf(fd,"Résultat %d\n", entier);  
    fprintf(fd,"un autre %f\n", reel);  
    fclose(fd);  
}
```

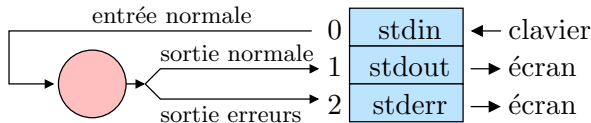
```
#include <stdio.h>
```

```
FILE *fd = fopen("mon_fichier","r");  
if (fd != NULL) {  
    char toto[50];  
    fscanf(fd,"%s%d",toto, &entier);  
    fscanf(fd,"%s%s%f",toto,toto, &reel);  
    fclose(fd);  
}
```

Fichiers et flots d'entrée sortie

Sous Unix, tout est un fichier

- ▶ Les périphériques sont représentés par des fichiers dans /dev
- ▶ Tout processus est créé avec des flots d'entrée/sortie standard :
 - ▶ `stdin` : entrée standard (connecté au terminal)
 - ▶ `stdout` : sortie standard (connecté à l'écran)
 - ▶ `stderr` : sortie d'erreur (connecté à l'écran)
- ▶ Ces flots sont associés aux descripteurs 0, 1 et 2
- ▶ Ils peuvent être fermés ou redirigés vers des «vrais» fichiers



Flots et interface de commande

Rediriger les flots en langage de commande : < et >

```
cat fich # écrit le contenu de fich sur la sortie standard (l'affiche à l'écran)
cat fich > fich1 # copie fich dans fich1 (qui est créé s'il n'existe pas)
./prog1 < entrée # exécute prog en écrivant le contenu de entrée sur son entrée standard
```

Flots et interface de commande

Rediriger les flots en langage de commande : **<** et **>**

```
cat fich # écrit le contenu de fich sur la sortie standard (l'affiche à l'écran)
cat fich > fich1 # copie fich dans fich1 (qui est créé s'il n'existe pas)
./prog1 < entrée # exécute prog en écrivant le contenu de entrée sur son entrée standard
```

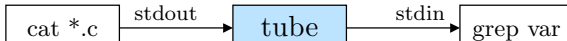
Tubes (pipes) : moyen de communication inter-processus

► Branche la sortie standard d'un processus sur l'entrée standard d'un autre

► Exemple : `cat *.c | grep var`

► crée deux processus : `cat *.c` et `grep var`

► connecte la sortie du premier à l'entrée du second à travers un tube



Exercice : que fait la commande suivante ?

```
cat f1 f2 f3 | grep toto | wc -l > resultat
```

Interface de programmation des tubes

Appel système `pipe()`

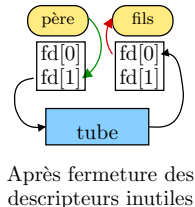
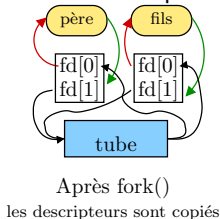
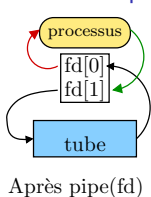
- ▶ Crée un tube : deux descripteurs (l'entrée et la sortie du tube)

```
int fd[2];  
pipe(fd);
```

- ▶ Ce qui est écrit sur l'entrée (`fd[1]`) est disponible sur la sortie (`fd[0]`) par défaut, cela permet de se parler à soi-même

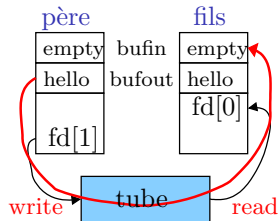
Mnémotechnique : On lit sur 0 (`stdin`), et on écrit sur 1 (`stdout`)

- ▶ Application classique : communication père-fils



Programmation d'un tube père/fils

```
#define BUFSIZE 10
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[ ] = "hello";
    pid_t childpid;
    int fd[2];
    pipe(fd);
    if (fork() > 0) { /* père */
        close(fd[0]);
        write(fd[1], bufout, strlen(bufout)+1);
    } else { /* fils */
        close(fd[1]);
        read(fd[0], bufin, BUFSIZE);
    }
    printf("[%d]: bufin = %.s, bufout = %s\n",
           getpid(), BUFSIZE, bufin, bufout);
    return 0;
}
```



```
$ ./parentwritepipe
[29196]: bufin=empty, bufout=hello
[29197]: bufin=hello, bufout=hello
$
```

Exercice : modifier le programme pour tenir compte des lectures/écritures tronquées

```
int todo = strlen(bufout)+1, done;
char *p=bufout;
while (todo) {
    done = write(fd[1],p,todo);
    todo -= done; p += done; }
```

```
int done=0; bufin[0]='a';
while (bufin[done]) {
    done += read(fd[0],
                bufin+done,
                BUFSIZE-done); }
```


Capacité d'un tube

`write(fd, &BUFF, TAILLE)` : écriture d'au plus **TAILLE** caractères

- ▶ S'il n'y a plus de lecteur :
 - ▶ Écriture inutile : on ne peut pas ajouter de lecteurs après la création du tube
 - ▶ Signal SIGPIPE envoyé à l'écrivain (mortel par défaut)
- ▶ Sinon si le tube n'est pas plein : Écriture atomique
- ▶ Sinon : Écrivain bloqué tant que tous les caractères n'ont pas été écrits (tant qu'un lecteur n'a pas consommé certains caractères)

`read(fd, &BUFF, TAILLE)` : lecture d'au plus **TAILLE** caractères

- ▶ Si le tube contient n caractères : Lecture de $\min(n, TAILLE)$ caractères.
- ▶ Sinon s'il n'y a plus d'écrivains : Fin de fichier ; `read` renvoie 0.
- ▶ Sinon : Lecteur bloqué jusqu'à ce que le tube ne soit plus vide (jusqu'à ce qu'un écrivain ait produit suffisamment de caractères)

Tubes nommés (FIFOs)

- ▶ **Problème des tubes** : seulement entre descendants car passage par clonage
- ▶ **Solution** : **tubes nommés** (même principe, mais nom de fichier symbolique)

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(char *nom, mode_t mode);
```

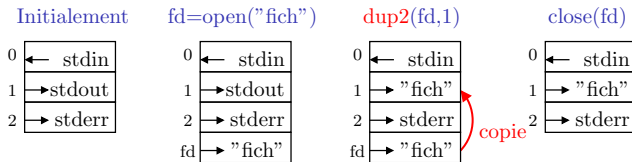
- ▶ renvoie 0 si OK, -1 si erreur
 - ▶ mode est numérique (on y revient)
- ▶ Après création, un processus l'ouvre en lecture, l'autre en écriture
- ▶ Il faut connecter les deux extrémités avant usage (bloquant sinon)
- ▶ la *commande* `mkfifo(1)` existe aussi

Copie de descripteurs : dup() et dup2()

ls > fich

- ▶ Cette commande inscrit dans `fich` ce que `ls` aurait écrit.
- ▶ Comment ça marche ?

On utilise les appels systèmes `dup()` et `dup2()` pour copier un descripteur :

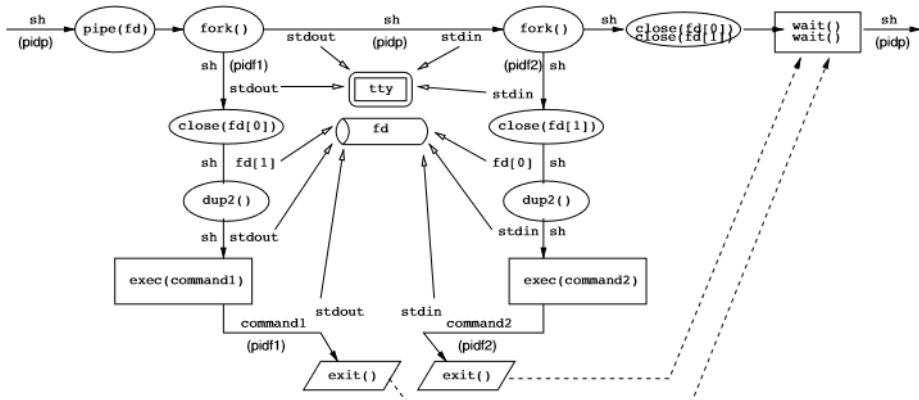


- ▶ Appel système `dup(fd)` duplique `fd` dans le premier descripteur disponible
- ▶ Appel système `dup2(fd1, fd2)` recopie le descripteur `fd1` dans `fd2`
on dit "dup to", duplicate to somewhere

L'exemple du shell

Création d'un tube entre deux commandes

```
$ commande1 | commande2
```



(c) Philippe Marquet, CC-BY-NC-SA.

- (Un peu touffu, mais pas si compliqué à la réflexion :)
- Si on oublie des `close()`, les lecteurs ne s'arrêtent pas (reste des écrivains)

Projection mémoire

Motivation

- ▶ Adresser le fichier dans l'espace d'adressage virtuel
Pratique pour sérialiser des données complexes de mémoire vers disque
- ▶ Le fichier n'est pas lu/écrit au chargement, mais à la demande

Réalisation

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

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);  
int munmap(void *addr, size_t length);
```

- ▶ **addr** : où le mettre, quand on sait. NULL très souvent
- ▶ **prot** : protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
- ▶ **flags** : visibilité (MAP_SHARED entre processus; MAP_PRIVATE copy-on-write, etc)
- ▶ **fd, offset** : Quel fichier, quelle partie projeter.
- ▶ Retourne l'adresse (virtuelle) du début du block

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
 - Introduction
 - Désignation des fichiers
- Interface d'utilisation des fichiers
 - Interface en langage de commande
 - Interface de programmation
 - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
 - Implantation des fichiers
 - Manipulation des répertoires
 - Protection des fichiers
- Conclusion

Représentation physique et logique d'un fichier

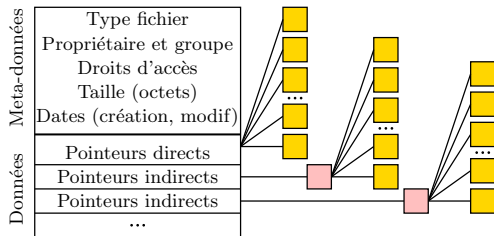
Représentation physique d'un fichier

- ▶ Le disque est découpé en *clusters* (taille classique : 4ko)
- ▶ Les fichiers sont écrits dans un ou plusieurs *clusters*
- ▶ Un seul fichier par *cluster*

Structure de données pour la gestion interne des fichiers

- ▶ Chaque fichier sur disque est décrit par un **inode**

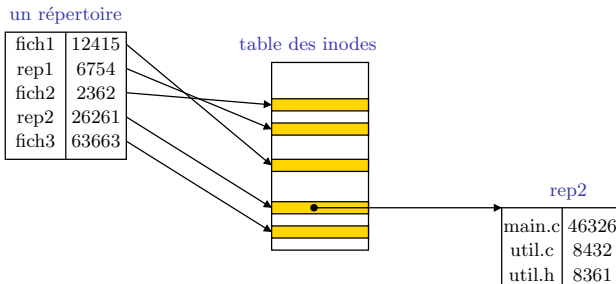
Structure d'un inode :



- ▶ Pointeurs et tables d'indirection contiennent adresses de *clusters*
- ▶ ≈ 10 blocs adressés directement \Rightarrow accès efficace petits fichiers (les plus nombreux)
- ▶ **stat(2)** : accès à ces info

Noms symboliques et inodes

- ▶ Les inodes sont rangés dans une table au début de la partition
 - ▶ On peut accéder aux fichiers en donnant leur inode (cf. `ls -li` pour le trouver)
 - ▶ Les humains préfèrent les noms symboliques aux numéros d'identification
- ⇒ notion de répertoire, sous forme d'arborescence (chainage avec racine)
- ▶ Les répertoires sont stockés sous forme de fichiers «normaux»
 - ▶ Chaque entrée d'un répertoire associe nom symbolique ↔ numéro d'inode



Manipulation de répertoires

Appels systèmes `opendir`, `readdir` et `closedir`

- ▶ Équivalent de `open`, `read` et `close` pour les répertoires

EXEMPLE : Implémentation de `ls -i .`

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main() {
    DIR *dirp;
    struct dirent *dp;

    dirp = opendir(".");
    while ((dp = readdir(dirp)) != NULL) {
        printf ("%s\t%d\n", dp->d_name, dp->d_ino);
    }
    closedir(dirp);
}
```

- ▶ On pourrait compléter cette implémentation avec `stat(2)`

Protection des fichiers : généralités

Définition (générale) de la sécurité

- ▶ **confidentialité** : informations accessibles aux seuls usagers autorisés
- ▶ **intégrité** : pas de modifications indésirées
- ▶ **contrôle d'accès** : qui a le droit de faire quoi
- ▶ **authentification** : garantie qu'un usager est bien celui qu'il prétend être

Comment assurer la sécurité

- ▶ Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- ▶ Mise en place de **mécanismes de protection** pour faire respecter ces règles

Règles d'éthique

- ▶ Protéger ses informations confidentielles (comme les projets et TP notés !)
- ▶ Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- ▶ Règles de bon usage avant tout :
La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers sous Unix

Sécurité des fichiers dans Unix

- ▶ Trois types d'opérations sur les fichiers : lire (r), écrire (w), exécuter (x)
- ▶ Trois classes d'utilisateurs vis à vis d'un fichier : propriétaire du fichier ; membres de son groupe ; les autres

rwX	rwX	rwX
propriétaire	groupe	autres

Granularité plus fine avec les Access Control List (cf `getfacl(1)`)

- ▶ Liste d'utilisateurs
- ▶ Pour les répertoires, r = ls, w = créer des éléments et x = cd.
- ▶ `ls -l` pour consulter les droits ; `chmod` pour les modifier

Mécanisme de délégation : `setuid`, `setgid`

- ▶ **Problème** : programme dont l'exécution nécessite des droits que n'ont pas les usagers potentiels (**exemple** : gestionnaire d'impression, d'affichage, ping)
- ▶ **Solution** (**setuid** ou **setgid**) : ce programme s'exécute toujours sous l'identité du propriétaire du fichier ; identité utilisateur *effective* momentanément modifiée. **Exemple.** `/usr/bin/passwd`

Résumé du troisième chapitre

- ▶ Désignation des fichiers
 - ▶ Chemin relatif vs. chemin absolu
 - ▶ \$PATH
- ▶ Utilisation des fichiers
 - ▶ Interface de bas niveau : open, read, write, close
Problèmes : I/O tronquée, perfs par manque de tampon
 - ▶ Interface standard : fopen, fprintf, fscanf, fclose
 - ▶ Trois descripteurs par défaut : stdin (0), stdout (1), stderr (2)
 - ▶ Rediriger les flots en shell : <, > et |
 - ▶ Tubes, tubes nommés et redirection en C : pipe(), mkfifo(), dup() et dup2()
- ▶ Réalisation et manipulation des fichiers et répertoires
 - ▶ Notion d'inode
 - ▶ Manipulation des répertoires : opendir, readdir, closedir
- ▶ Quelques notions et rappels de protection des fichiers