

# Introduction aux systèmes d'exploitation (IF218)

Basé sur les cours de David Mazières (Stanford)  
et Arnaud Legrand (IMAG)

Ludovic Courtès<sup>1</sup>

ENSEIRB-MATMÉCA, septembre–octobre 2022

---

1. ludovic.courtes@inria.fr

## Outline

### Intro

séance1

Présentation du cours

Qu'est-ce qu'un SE ?

### Objectifs d'un système d'exploitation

séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

### Fils d'exécution et processus

séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau

séance2

### Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

Plan	
Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

Plan	
Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Objectifs du cours

- ▶ introduction aux systèmes d'exploitation
  - ▶ on en utilise tous les jours
  - ▶ comprendre le fonctionnement d'un SE aide à programmer
- ▶ couvrir des concepts importants
  - ▶ concurrence, gestion mémoire, entrées/sorties, protection
  - ▶ notions de conception d'un SE : architecture, compromis
- ▶ un sujet vaste et passionnant !

## Organisation du cours

- ▶ **6 séances de 2h** (27, 29 sept. puis 6, 7, 10 et 13 octobre)
  - ▶ introduction aux systèmes d'exploitation
  - ▶ fils d'exécutions (*threads*) et processus
  - ▶ gestion de la mémoire virtuelle
- ▶ **examen de cours** (13 octobre 2022)
- ▶ en parallèle, cours/TP/TD de **programmation système** (G. Mercier)
  - ▶ interface de programmation du système d'exploitation (POSIX)

## Plan

### Intro

séance1

Présentation du cours

Qu'est-ce qu'un SE ?

### Objectifs d'un système d'exploitation

séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

### Fils d'exécution et processus

séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau

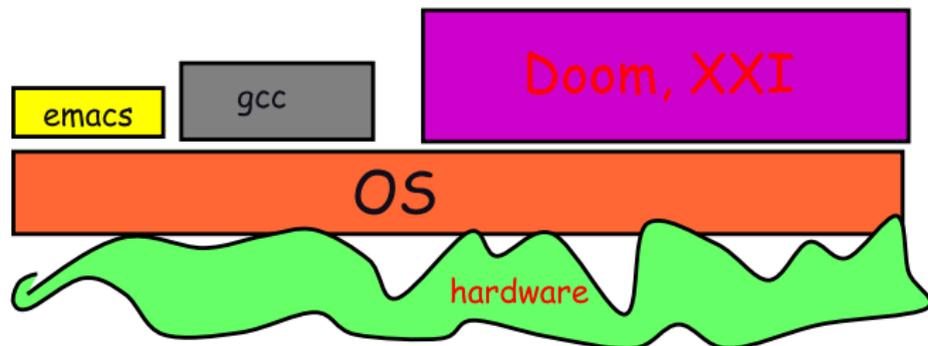
séance2

### Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

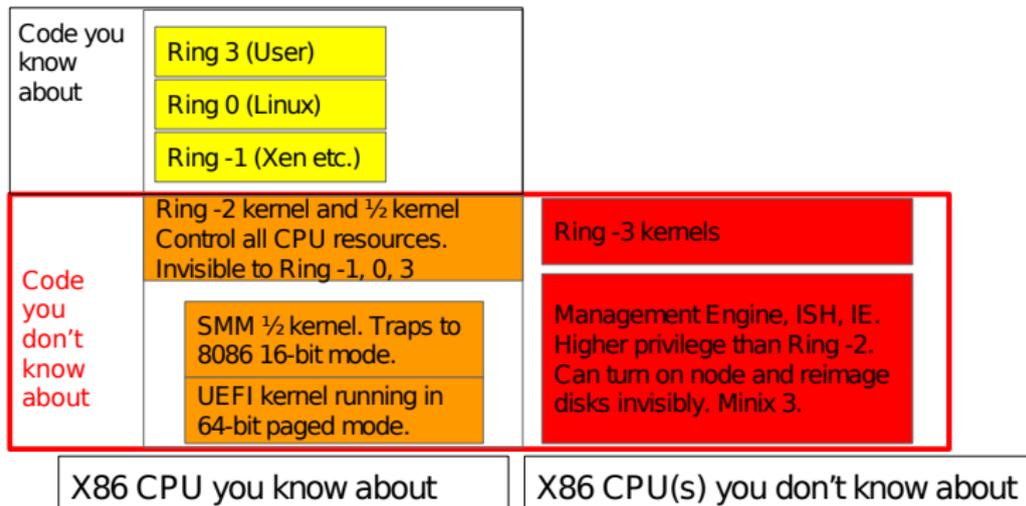
## Couche entre matériel et applications



- ▶ rend le matériel utilisable
- ▶ fournit des abstractions aux applications
  - ▶ cache les détails du matériel
  - ▶ accède au matériel avec des moyens bas niveau
- ▶ protège les applications/utilisateurs les uns des autres

## Couche entre matériel et applications ?

Machines Intel (de *Replace your exploit-ridden firmware with a Linux kernel*)



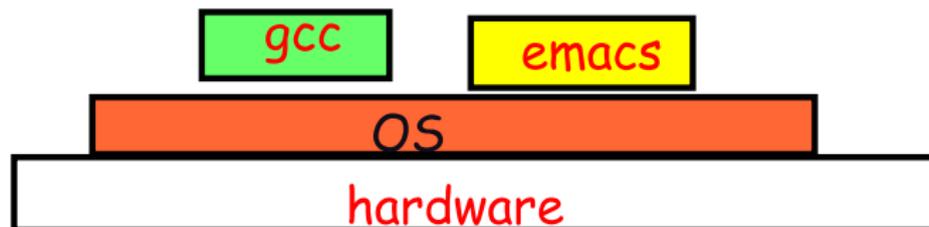
## Pourquoi s'intéresser aux systèmes d'exploitation ?

- ▶ un problème réglé ?
  - ▶ **oui** : quelques SE mûrs utilisés par tous
  - ▶ **non** : questions ouvertes
    - ▶ programmes malintentionnés
    - ▶ extensibilité
    - ▶ adaptation à d'autres applications ou "machines"
- ▶ le SE est une fondation
  - ▶ pour la gestion des ressources
  - ▶ pour la sûreté de fonctionnement (sécurité, tolérance aux fautes, etc.)
- ▶ les problèmes de SE se retrouvent ailleurs
  - ▶ supports de langages de programmation
  - ▶ navigateurs web ( ? )

## Système d'exploitation primitif

- ▶ limité à l'abstraction du matériel, pas de protection
- ▶ hypothèses simplificatrices
  - ▶ un seul programme s'exécute à chaque instant
  - ▶ les programmes n'ont pas de bugs, sont bien intentionnés
- ▶ pauvre utilisation des ressources matérielles
- ▶ exemple : MS DOS, systèmes critiques dédiés

## Multi-tâche (« multi-programmation »)

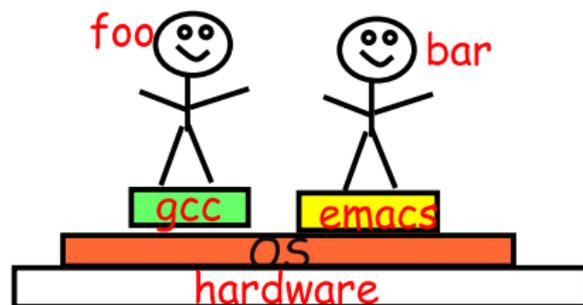


- ▶ plusieurs programmes s'exécutent en même temps
- ▶ quand un programme attend (le disque, le réseau, etc.), un autre tourne

## Et si un programme se comporte mal ?

- ▶ exemples
  - ▶ entre dans une boucle infinie
  - ▶ accède à des endroits de la mémoire « où il ne fallait pas »
  - ▶ essaye de « faire tomber » les autres programmes
- ▶ solution : **protection, isolation**
  - ▶ **préemption** : reprend le temps CPU ou la mémoire physique allouée
  - ▶ **protection mémoire** : mémoire des autres programme inaccessibles
  - ▶ **moindre autorité** : un programme ne peut accéder qu'aux ressources explicitement données (idéalement. . .)

## Multi-utilisateur



- ▶ utilisateurs mutuellement suspicieux
  - ▶ solution : **protection, isolation**, comme pour les applications
- ▶ utilisation efficace des ressources
  - ▶ ordonnancement des ressources entre utilisateurs
  - ▶ donner les ressources à ceux qui en ont vraiment besoin
- ▶ ce qui peut mal se passer
  - ▶ utilisateurs trop gloutons (besoin d'une politique)
  - ▶ demande mémoire trop grande (besoin de virtualisation)

## Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

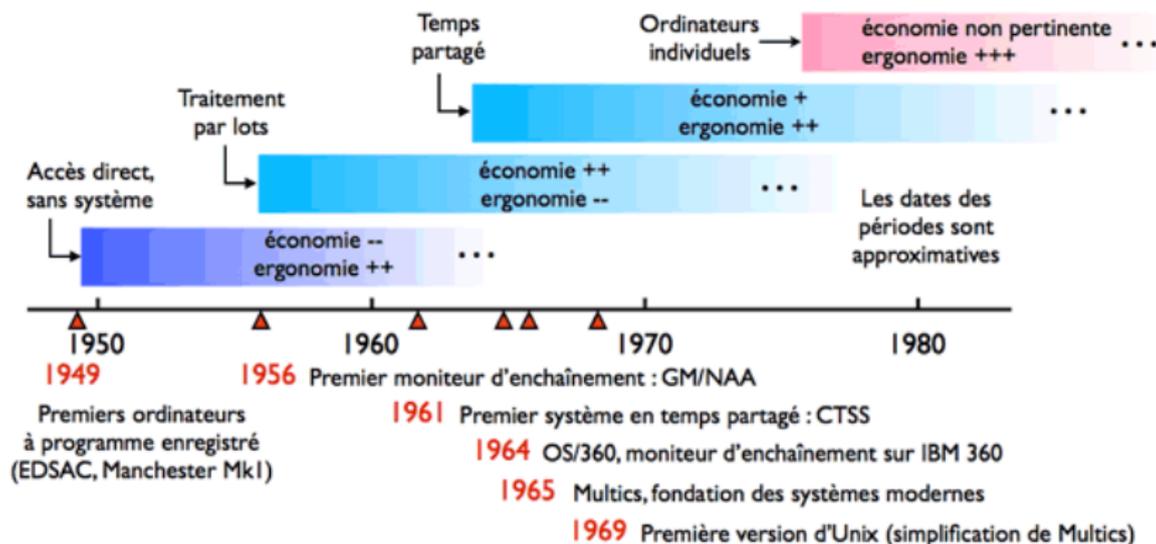
Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination



## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

**Abstraction**

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Services fournis par un système d'exploitation

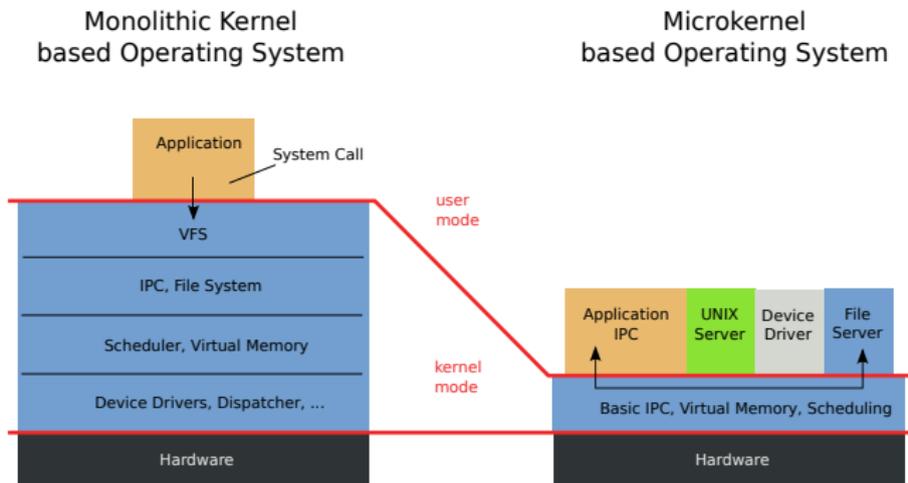
- ▶ gestion de la mémoire virtuelle
- ▶ fils d'exécution (*threads*) et processus
- ▶ stockage à long terme (système de fichiers)
- ▶ protocoles réseau (TCP, IP, etc.)
- ▶ interface de haut niveau au matériel
  - ▶ pilotes de disque dur, de carte son, de carte réseau, de périphériques USB, etc.
- ▶ ...

## Protection

- ▶ seul le **noyau du SE** peut accéder directement au matériel
  - ▶ tourne en mode CPU **privilegié**
  - ▶ protège des applications malintentionnés
  - ▶ protège des applications buggées
  - ▶ **espace noyau** (*kernel space*)
- ▶ les applications sont **non privilégiées**
  - ▶ font appel au noyau pour accéder au matériel
  - ▶ **espace utilisateur** (*user space*)
- ▶ **ordonnancement des ressources** matérielles, préemption
  - ▶ donne une ressource, la reprend si besoin

# Structure typique d'un SE

à base de noyau monolithique vs. micro-noyau



- ▶ GNU/Linux, Android, variantes BSD, Solaris, Windows, etc.

- ▶ GNU/Hurd, MINIX, QNX, INTEGRITY, Symbian, macOS

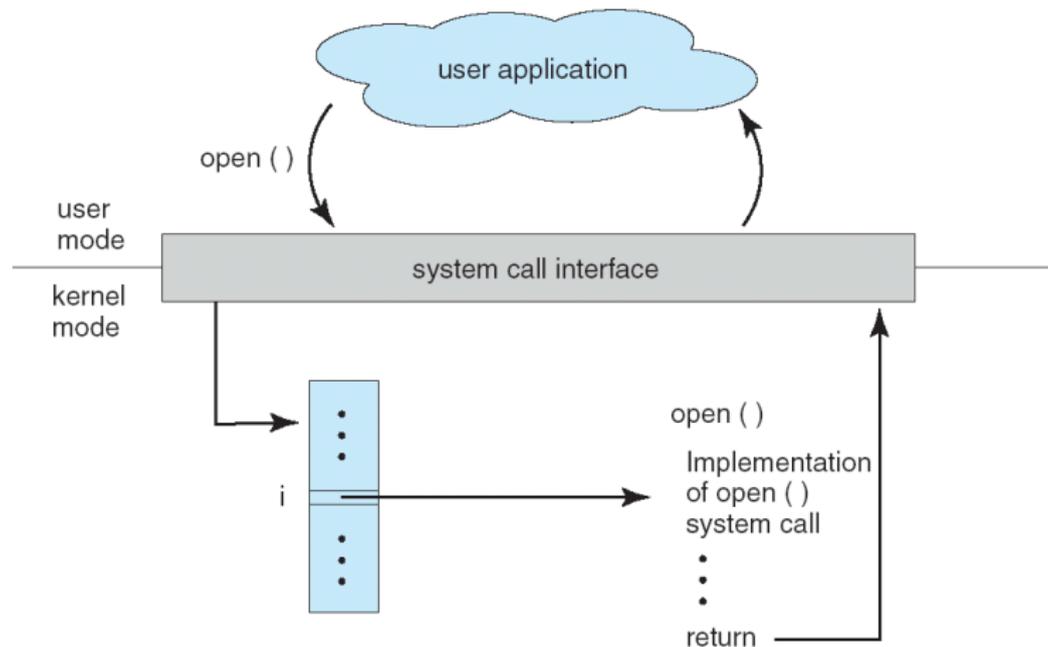
## Noyau monolithique vs. micro-noyau + serveurs

un débat classique...

- ▶ noyau monolithique
  - ▶ toutes les fonctionnalités sont intimement liées
  - ▶ vue globale de l'utilisation des ressources : facilite l'ordonnancement
  - ▶ séparation nette utilisateur/noyau
  - ▶ difficulté d'étendre les fonctionnalités (code noyau difficile à écrire ; nouvelle fonctionnalité = reboot)
- ▶ micro-noyau + serveurs
  - ▶ fonctionnalités séparées dans des « serveurs » (processus)
  - ▶ conséquences d'un bug isolées
  - ▶ extensible : les serveurs sont des « programmes normaux »
  - ▶ ordonnancement & imputabilité des ressources plus difficiles

# L'application parle au SE : « appels systèmes »

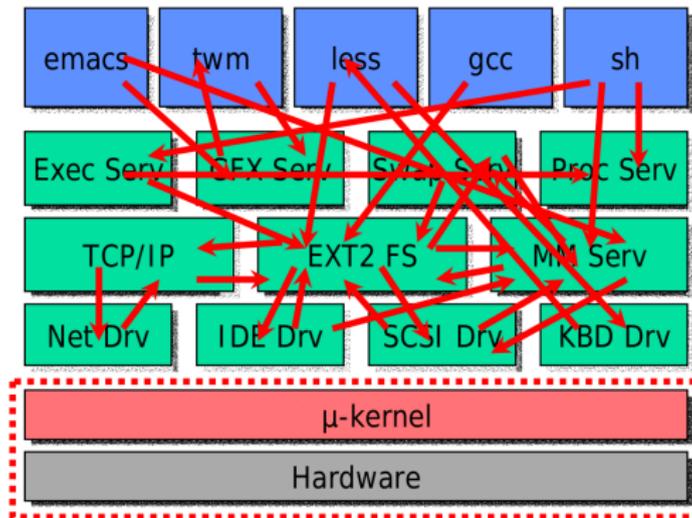
noyau monolithique



► services fournis par le noyau

# L'application parle au SE : comm. inter-processus

système multi-serveur à base de micro-noyau



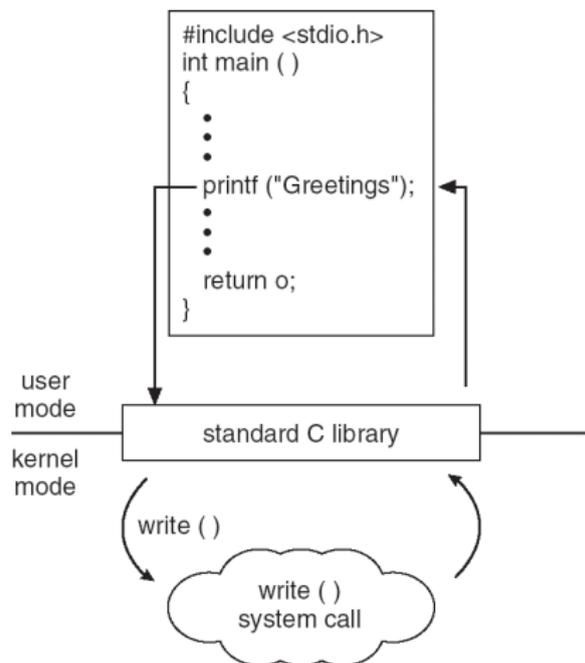
- ▶ services fournis par des "serveurs" en espace utilisateur
  - ▶ communication inter-processus (IPC) entre applications et serveurs

## Exemples d'appels système POSIX (Unix, Windows, etc.)

- ▶ `void *sbrk (intptr_t increment);`
  - ▶ ajoute `increment` octets de mémoire accessible au programme
- ▶ `int open (const char *chemin, int drapeaux);`
  - ▶ ouvre le fichier situé à `chemin` (ex. : `cours.pdf`)
  - ▶ renvoie un *descripteur de fichiers*
  - ▶ **ou** renvoie une erreur (entier négatif)
- ▶ `ssize_t write (int desc_fich, const void *tampon, size_t combien);`
  - ▶ écrit au plus `combien` octets depuis `tampon` dans `desc_fich`
  - ▶ retourne le nombre d'octets effectivement écrits
  - ▶ **ou** retourne une erreur (entier négatif)

## Appels systèmes, vu de l'application

- ▶ l'application utilise typiquement des fonctions de plus haut niveau
  - ▶ fonction `printf` vs. appel système `write`
  - ▶ fonction `malloc` vs. appel système `sbrk`



## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Ressources matérielles

- ▶ donner à chaque application l'impression de disposer seule de la ressource
  - ▶ chaque appli dispose de « tout le CPU pour elle »
  - ▶ chaque appli dispose de « toute la mémoire pour elle »
- ▶ garantir l'isolation entre applications par défaut
  - ▶ les calculs de l'appli A n'interfèrent pas sur ceux de l'appli B
  - ▶ les données de l'appli A ne sont pas visibles de B
- ▶ permettre le partage/l'échange
  - ▶ mémoire partagée entre deux applications

## Ressources logicielles

- ▶ donner à chaque appli l'accès à des ressources « logicielles »
  - ▶ stockage persistant, p. ex., fichier
  - ▶ connexion TCP/IP
- ▶ garantir l'isolation
  - ▶ l'appli A n'accède qu'aux fichiers auxquels elle a le droit (+/-)
  - ▶ la connexion SSH de A n'interfère pas avec la connexion HTTP de B
- ▶ permettre le partage/l'échange
  - ▶ fichier partagé entre deux applications
  - ▶ connexion ouverte passée d'une application à une autre

## Contextes systèmes

- ▶ contexte d'une application/processus utilisateur qui s'exécute
- ▶ contexte noyau d'une application
  - ▶ code exécuté dans le noyau à l'intention d'un processus (p. ex. un appel système)
- ▶ contexte noyau non associé à une application
  - ▶ gestion des interruptions matérielles
  - ▶ interruptions de la minuterie (*timer*)
  - ▶ tâches différées du noyau (« softirqs » et « tasklets » dans Linux)
- ▶ changement de contexte

## Transitions entre contextes

- ▶ utilisateur vers noyau
  - ▶ appel système
  - ▶ défaut de page (*page fault*)
  - ▶ interruption temporelle, matérielle

## Ordonnancement des ressources

- ▶ choix basé sur le comportement observé des applications
  - ▶ l'application n'est pas consultée
  - ▶ le noyau « devine » ce qu'essaye de faire l'application
- ▶ le SE a des heuristiques pour « bien ordonnancer » les ressources pour une « application typique »
  - ▶ gestion de la mémoire virtuelle constamment ajustée (Linux)
  - ▶ différents ordonnanceurs CPU, entrées/sorties (Linux)
- ▶ allocation de ressources transparente/opaque pour l'application
  - ▶ transparente : l'application ne se soucie de rien
  - ▶ opaque : l'application ne peut pas savoir ce que va chercher à faire le SE
  - ▶ parfois, interfaces pour désactiver/guider l'intelligence du SE (O\_DIRECT, mlock, madvise, pthread\_setaffinity)

## Aperçu



# Ordonnancement et préemption CPU

## Définitions

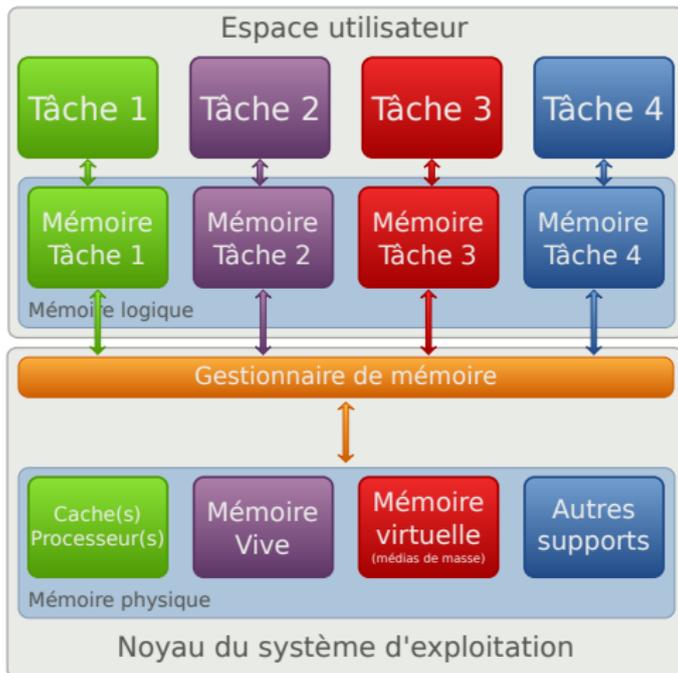
- ▶ **fil d'exécution** ou *thread*
  - ▶ contexte d'exécution de code utilisateur
  - ▶ « unité de traitement » ordonnancée par le noyau

## Ordonnancement et préemption CPU

- ▶ pour partager le CPU entre applications
  - ▶ chaque appli se voit donnée du temps de calcul
  - ▶ chaque appli a l'impression d'être seule sur le CPU
  - ▶ ex. : une appli ne monopolise pas le CPU avec une boucle infinie
- ▶ déroulement
  - ▶ interruption de la minuterie, p. ex., toutes les 10s
  - ▶ le noyau prend la main (*contexte noyau*)
  - ▶ installe le contexte utilisateur d'une appli
  - ▶ passe la main à cette appli

# Gestion de la mémoire

## Aperçu



# Ordonnancement mémoire

## Définitions

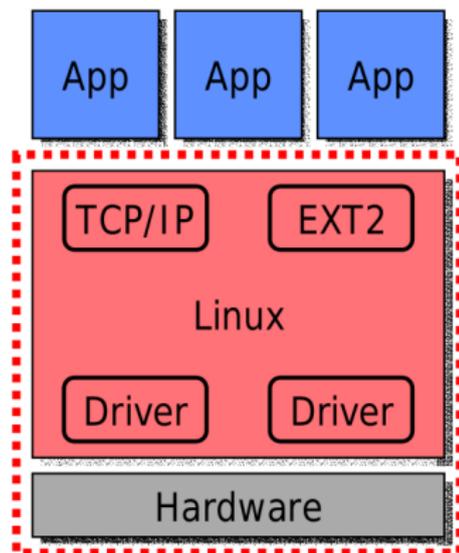
- ▶ **adresse physique** : adresse dans la mémoire vive (la « vraie » mémoire)
- ▶ **adresse virtuelle** : utilisées par un programme pour désigner « sa » mémoire
- ▶ **projection mémoire** (*mapping*) : établissement d'une correspondance entre un intervalle d'adresses virtuelles et « quelque chose » (ex. : mémoire physique)
- ▶ **traduction d'adresse** : traduction virtuelle/physique
  - ▶ fait par le matériel, la MMU (*memory management unit*)
- ▶ **espace d'adressage** : ensemble d'adresses de mémoire virtuelle qu'un programme peut désigner
- ▶ **processus** : *thread(s)* + espace d'adressage

## Ordonnancement et « préemption » mémoire

- ▶ pour partager la mémoire physique entre applications
  - ▶ chaque appli a accès a « de la mémoire »
  - ▶ **tous ses accès mémoires** sont contrôlés par le matériel/noyau
  - ▶ un programme ne peut désigner **que la mémoire visible dans son espace d'adressage**
  - ▶ le SE **ordonnance la mémoire physique** entre applis (envoie sur disque la mémoire inutilisée, alloue paresseusement, etc.)
- ▶ déroulement
  - ▶ le noyau crée un espace d'adressage pour chaque appli
  - ▶ l'appli demande au noyau la projection de mémoire physique dans son E.A.
  - ▶ le noyau maintient des tables de traduction d'adresses pour chaque E.A.

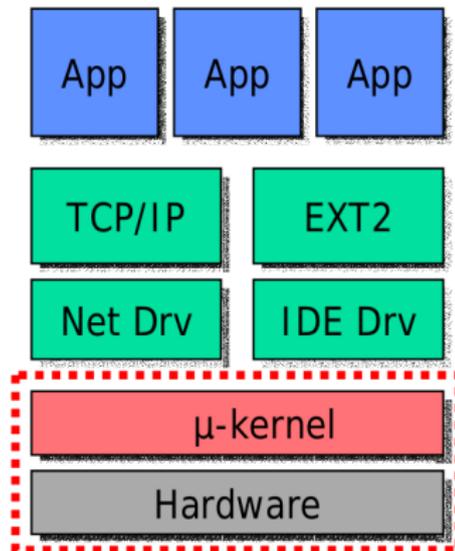
## Bénéfices de la protection mémoire

- ▶ Firefox ne peut pas lire le mot de passe dans la mémoire de Thunderbird (*sécurité-immunité*)
- ▶ un pointeur fou dans un programme n'affecte pas les autres (*tolérance aux fautes*)
- ▶ mais un pointeur fou dans le noyau...



## Bénéfices de la protection mémoire

- ▶ Firefox ne peut pas lire le mot de passe dans la mémoire de Thunderbird (*sécurité-immunité*)
- ▶ un pointeur fou dans un programme n'affecte pas les autres (*tolérance aux fautes*)
- ▶ mais un pointeur fou dans le noyau. . .



## Remarque : Protection mémoire par le langage

- ▶ isolation mémoire **au sein d'un même espace d'adressage**
  - ▶ avec un langage de prog. à gestion de la mémoire automatique (Lisp/Scheme, Python, Java, etc.)
  - ▶ applications toutes écrites dans ce langage (ou avec langage intermédiaire commun)
  - ▶ un seul espace d'adressage
- ▶ exemples de SE « basé langage »
  - ▶ Microsoft Singularity (C#), JNode (Java), Genera (Lisp), Inferno (Limbo)
- ▶ *Cette approche n'est pas étudiée dans ce cours.*

## Contrôle de la consommation de ressources

```
int main () {  
    /* crée des processus */  
    while (1)  
        fork ();  
}
```

▶ quotas (processus fils, taille du tas, etc.)

▶ *out-of-memory killer* (Linux)

▶ invoqué en cas de pénurie de mémoire

▶ termine la tâche estimée la plus "mauvaise"

```
int main () {  
    /* alloue de la mémoire */  
    while (1)  
        malloc (1234);  
}
```

▶ ?

```
# http://lwn.net/Articles/524301/  
while true; do mkdir x; cd x; done
```

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

**Objectifs d'un système d'exploitation** **séance1**

Historique

Abstraction

Démultiplexage des ressources & isolation

**Utilisation efficace des ressources**

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Propriétés utiles à exploiter

- ▶ biais
  - ▶ 80% du temps pris par 20% du code
  - ▶ 10% de la mémoire absorbe 90% des références
- ▶ prédictions basées sur l'historique (*localité temporelle*)
  - ▶ ex. : pré-charger la suite d'un fichier
  - ▶ ex. : évincer du cache les données pas accédées récemment

## Partage de code entre applications

- ▶ exemples
  - ▶ 99% des programmes utilisent la bibliothèque C (libc)
  - ▶ beaucoup utilisent la bibliothèque GTK+ (interfaces graphiques)
  - ▶ le code binaire de ces bibliothèques est le même pour tous (code indépendant de la position, *PIC*)
- ▶ idée : ne charger qu'une seule fois le code de ces bibliothèques
  - ▶ pris en charge par l'éditeur de liens dynamique, ld.so (espace utilisateur)
  - ▶ tire partie de la gestion de la mémoire virtuelle
- ▶ l'édition de liens dynamique est un sujet à part entière. . .

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Objectifs d'un système d'exploitation

- ▶ fournir une **abstraction** du matériel
- ▶ permettre le **partage sans interférence** des ressources matérielles/logicielles
- ▶ permettre la **coopération choisie** entre applications
- ▶ utiliser **efficacement** le matériel

# Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

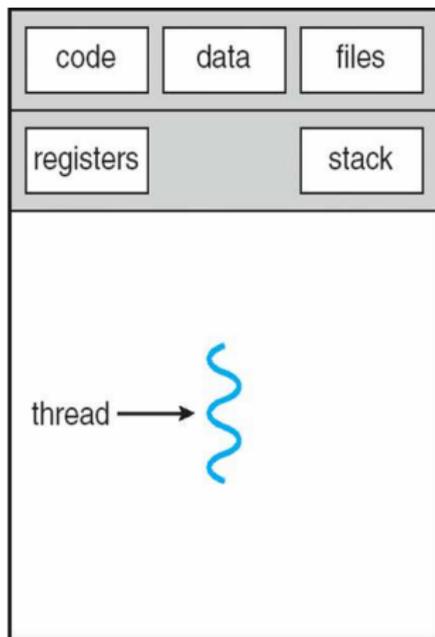
Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

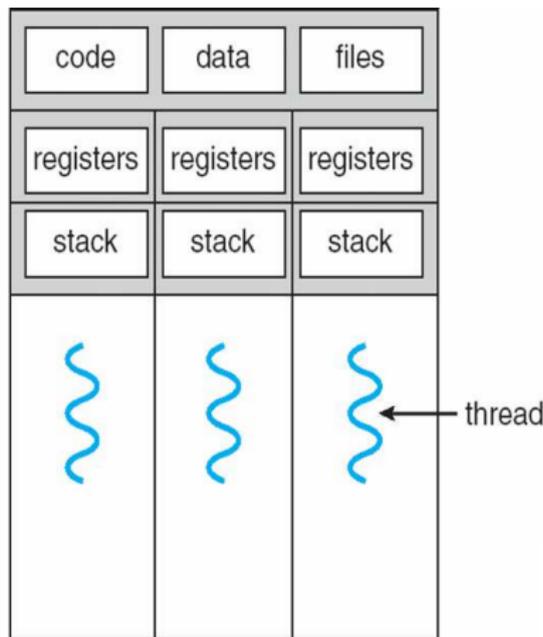
## Au quotidien

- ▶ **processus** = instance d'un programme qui s'exécute
- ▶ un SE moderne peut faire tourner plusieurs processus "simultanément"
- ▶ exemples de processus pouvant s'exécuter simultanément
  - ▶ `gcc foo.c` — compilateur qui compile `foo.c`
  - ▶ `gcc chbouib.c` — compilateur qui compile `chbouib.c`
  - ▶ `emacs` — éditeur de texte (et courrier électronique, etc.)
  - ▶ `firefox` — navigateur web
- ▶ non-exemples : mis en œuvre par un seul processus
  - ▶ plusieurs fenêtres Emacs
  - ▶ plusieurs fenêtres GIMP
- ▶ pourquoi des processus ?
  - ▶ meilleure utilisation du CPU
  - ▶ tâches concurrentes

## Du fil d'exécution au processus

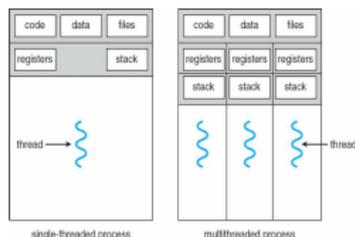


single-threaded process



multithreaded process

## Du fil d'exécution au processus

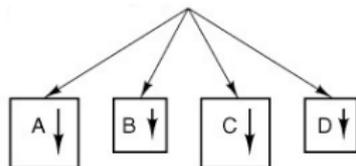


- ▶ **fil d'exécution** ou *thread*
  - ▶ contexte d'exécution de code utilisateur
  - ▶ « unité de traitement » ordonnancée par le noyau
- ▶ **processus = thread(s) + espace d'adressage**
  - ▶ protection mémoire entre processus
  - ▶ un processus peut contenir plusieurs *threads*
  - ▶ même environnement pour tous les threads d'un processus
    - ▶ même espace d'adressage (mémoire partagée)
    - ▶ mêmes descripteurs de fichiers

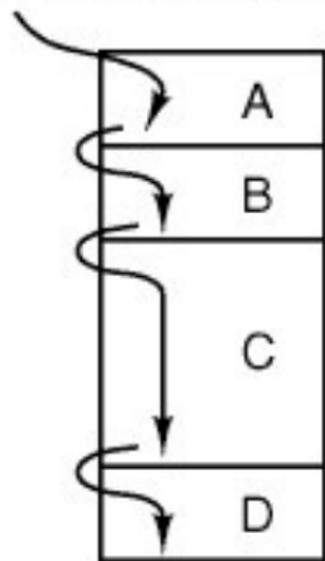
## Choisir entre fil d'exécution et processus

- ▶ protection mémoire
  - ▶ les processus sont protégés entre eux, pas les *threads*
  - ▶ pour un langage à gestion automatique de la mémoire, cet argument est peu pertinent. . .
- ▶ performance
  - ▶ création de *threads* moins coûteuse que création de *processus*
  - ▶ communication entre *threads* moins coûteuse, plus simple
  - ▶ changement de contexte entre *threads* moins coûteux
- ▶ passage à l'échelle
  - ▶ plusieurs *threads* pour un programme permettent d'utiliser plusieurs cœurs
  - ▶ ex. : parallélisation des algorithmes internes au programme

## Entrelacement de l'exécution des fils/processus



- ▶ conceptuellement : 4 programmes s'exécutent « en même temps »
  - ▶ 4 pointeurs d'instruction (ou *program counters*)
  - ▶ 4 piles, 4 jeux de registres, etc.
- ▶ en pratique : exécution entrelacée
  - ▶ le noyau/ordonnanceur reprend la main
  - ▶ installe la pile du fil/processus suivant
  - ▶ lui passe la main : **changement de contexte**

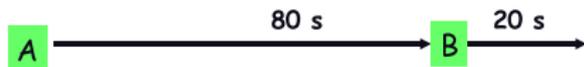


## Concurrence vs. parallélisme

- ▶ **concurrency**
  - ▶ plusieurs fils d'exécution *progressent* en même temps
  - ▶ éventuellement exécution entrelacée, chacun son tour
- ▶ **parallélisme**
  - ▶ plusieurs fils d'exécution *s'exécutent* en même temps
  - ▶ exemple : sur un processeur multi-cœur
- ▶ en pratique :
  - ▶ les CPU modernes ont plusieurs cœurs
  - ▶ donc « vrai » parallélisme *et* concurrence

## Avantages potentiels pour la performance

- ▶ améliore l'utilisation du CPU
  - ▶ recouvrement des attentes d'un processus par le calcul d'un autre
- ▶ réduit la latence
  - ▶ exécuter *A* puis *B* prend 100s



- ▶ exécuter *A* et *B* de manière concurrente finit plus tôt



- ▶ à supposer que *A* et *B* ne fassent pas que des calculs
- ▶ ralentissement engendré par les changements de contexte quasi-négligeable

## Interface POSIX pour manipuler les *threads*

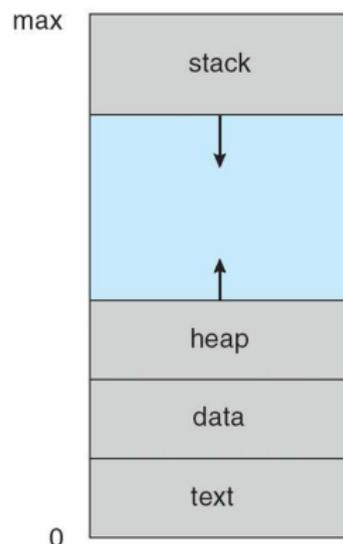
- ▶ `int pthread_create (pthread_t *fil, pthread_attr_t *attr, void *(*entrée) (void *), void *arg)`
  - ▶ crée un nouveau fil d'exécution dans le processus courant
  - ▶ le nouveau fil a une nouvelle pile, mais le **même environnement** que le reste du processus
  - ▶ retourne 0 et renvoie son identifiant dans `fil`
  - ▶ l'exécution commence par l'appel `entrée (arg)`
- ▶ `int pthread_join (pthread_t fil, void **valeur_retour)`
  - ▶ attend la terminaison de `fil`
  - ▶ retourne dans `*valeur_retour` sa valeur de retour (pointeur)
- ▶ `void pthread_exit (void *valeur_retour)`
  - ▶ termine le fil courant avec `valeur_retour`

## Exemple : Création et attente d'un fil d'exécution

```
#include <pthread.h>
static void *code_du_fil (void *arg) {
    static int result = 2;
    result += *(int *) arg;
    return &result;
}
int main (int argc, char *argv[]) {
    pthread_t fil; int x = 3; int *resultat;
    pthread_create (&fil, NULL, code_du_fil, &x);
    pthread_join (fil, &resultat);
    printf ("resultat : %i\n", *resultat); /* 5 */
    return *resultat;
}
```

## Ce que voit un processus

- ▶ chaque processus a sa vue de la machine
  - ▶ son propre espace d'adressage
    - ▶ `(char *) 0x1234` différent dans chaque proc.
  - ▶ ses propres descripteurs de fichiers (fichiers ouverts, etc.)
    - ▶ descripteur de fichier 7 différent dans chaque processus
  - ▶ son « CPU à lui » (via le multitâche préemptif)
- ▶ modèle de programmation simple
  - ▶ gcc ne se soucie pas que firefox tourne



## Interface Unix/POSIX pour manipuler les processus

- ▶ chaque processus identifié par un entier, le **PID**
- ▶ `pid_t fork (void)` : création de processus fils
  - ▶ crée un processus fils copie conforme du parent
  - ▶ le fils hérite des projections mémoire, mais *copy-on-write*
  - ▶ le fils hérite des descripteurs de fichier, etc.
  - ▶ le fils a un PID différent (`fork` retourne le PID du fils dans le parent, 0 dans le fils)
- ▶ `pid_t waitpid (pid_t pid, int *stat, int opt)` : attend la fin d'un processus
  - ▶ attend la fin du fils `pid`, ou de tous les fils si `pid = -1`
  - ▶ retourne dans `stat` le statut de terminaison
  - ▶ retourne 0 si succès, -1 si erreur
- ▶ `pid_t getpid (void)` : renvoie le PID de l'appelant
- ▶ `pid_t getppid (void)` : renvoie le PID du parent

## Terminaison de processus

- ▶ `void exit (int statut)` : termine le processus courant
  - ▶ termine le processus courant avec `statut`
  - ▶ par convention, `statut = 0` signifie « succès »
- ▶ `int kill (pid_t pid, int signal)` : termine ou envoie un signal à `pid`
  - ▶ si `signal = SIGKILL`, le noyau termine `pid` sans préavis
  - ▶ si `signal = SIGTERM`, le noyau notifie `pid` de sa fin prochaine
    - ▶ le processus peut p. ex. sauvegarder ses données puis `exit`
  - ▶ les autres signaux peuvent aussi être « attrapés » par le destinataire, comportement par défaut sinon

## Terminaison de processus

### folklore Unix/POSIX

- ▶ quand un processus se termine
  - ▶ les ressources du processus sont libérées par le système
  - ▶ le signal SIGCHLD est envoyé au parent
  - ▶ le parent **doit** l'attendre avec `waitpid` ou `wait`
    - ▶ sinon le fils devient *zombie* après sa terminaison
- ▶ quand un parent se termine avant son fils
  - ▶ le fils est adopté par le processus 1 (`init`)

## Exécution de programme

- ▶ `int execve (const char *chemin, char *argv[], char *env[])`
  - ▶ **remplace** l'image du processus courant par le programme à `chemin` (p. ex. `/bin/date`)
  - ▶ programme lancé avec les arguments `argv`
  - ▶ ... et les variables d'environnement `env`
  - ▶ les descripteurs de fichier ouverts sont conservés
- ▶ interfaces de plus haut niveau
  - ▶ `int execvp (const char *nom, char *argv[])` cherche `nom` dans `$PATH`, utilise l'environnement courant
  - ▶ `int execlp (const char *nom, char *argv, ...)` idem, mais arguments listés

## Exemple : lancer /bin/date

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (int argc, char *argv[]) {
    pid_t enfant;
    switch ((enfant = fork ())) {
        case -1: return 1; /* erreur */
        case 0: execlp ("date", "date program", NULL); /* enfant
        default: { int statut; /* parent */
            waitpid (enfant, &statut, 0);
            if (WIFEXITED (statut))
                printf ("OK: %i\n", WEXITSTATUS (statut)); }
    }
    return 0;
}
```

## fork + exec = posix\_spawn

cf. <https://www.microsoft.com/en-us/research/publication/a-fork-in-the-road/>

- ▶ `int posix_spawn (pid_t *pid, const char *fichier, ...)`
- ▶ `open` et `O_CLOEXEC`, `socket` et `SOCK_CLOEXEC`, etc.
- ▶ `pipe2`, `accept4`, etc.

## Interaction et coopération entre processus

- ▶ à travers l'environnement initial
  - ▶ mémoire, desc. de fichier ouverts, var. d'environnement, etc.
- ▶ à travers des fichiers
  - ▶ emacs écrit un fichier, gcc le lit et le compile
- ▶ par communication inter-processus
  - ▶ appel système pipe pour créer un canal de communication entre processus parent et fils
  - ▶ appels système socket, connect, etc. pour ouvrir une connexion
  - ▶ appel système kill pour envoyer un signal
- ▶ par transfert de descripteur de fichier (sendmsg)
- ▶ par mémoire partagée (shmctl, etc.)

## Communication inter-processus par tube

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

int main (int argc, char *argv[]) {
    int enfant, tube[2];
    pipe (tube); /* crée un tube */
    switch ((enfant = fork ())) {
        case -1: return 1;
        case 0: /* le fils écrit dans 1 */
            write (tube[1], "Salut !", 8);
            return 0;
        default: { /* le parent lit dans 0 */
            char buf[8]; int statut;
            read (tube[0], buf, sizeof (buf));
            printf ("message : %s\n", buf);
            waitpid (enfant, &statut, 0);
            return WEXITSTATUS (statut); }
    }
}
```

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

**Programmation concurrente**

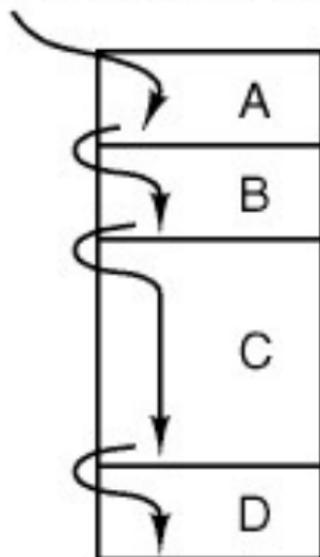
Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Concurrence, entrelacement et synchronisation



## Accès à l'état partagé

Quelle est la sortie de ce programme ?

```
#include <stdlib.h>
int count = 0; /* modifié par différents fils */
void *loop (void *ignored) {
    int i;
    for (i = 0 ; i < 100000 ; i++)
        count++;
    return NULL;
}
int main () {
    pthread_t tid;
    pthread_create (&tid, NULL, loop, NULL);
    loop (NULL);
    pthread_join (tid, NULL);
    printf("%d\n",count);
}
```

## Accès à l'état partagé

Quelle est la sortie de ce programme ?

```
#include <stdlib.h>
int count = 0; /* modifié par différents fils */
void *loop (void *ignored) {
    int i;
    for (i = 0 ; i < 100000 ; i++)
        count++;
    return NULL;
}
int main () {
    pthread_t tid;
    pthread_create (&tid, NULL, loop, NULL);
    loop (NULL);
    pthread_join (tid, NULL);
    printf("%d\n",count);
}
```

► une valeur entre 100000 et 2e6

► count++ peut devenir :

```
reg1 <- count
reg1 <- reg1 + 1
count <- reg1
```

## Accès à l'état partagé

Est-ce que les 2 sections critiques vont être appelées ?

```
int flag1 = 0, flag2 = 0;
void *p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) { section_critique_1 (); }
}
void *p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) { section_critique_2 (); }
}
int main () {
    pthread_t tid1;
    pthread_create (&tid1, NULL, p1, NULL);
    p2 ();
    pthread_join (tid1, NULL);
}
```

## Accès à l'état partagé

Est-ce que les 2 sections critiques vont être appelées ?

```
int flag1 = 0, flag2 = 0;
void *p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) { section_critique_1 (); }
}
void *p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) { section_critique_2 (); }
}
int main () {
    pthread_t tid1;
    pthread_create (&tid1, NULL, p1, NULL);
    p2 ();
    pthread_join (tid1, NULL);
}
```

► on ne sait pas !

## Besoin de synchronisation

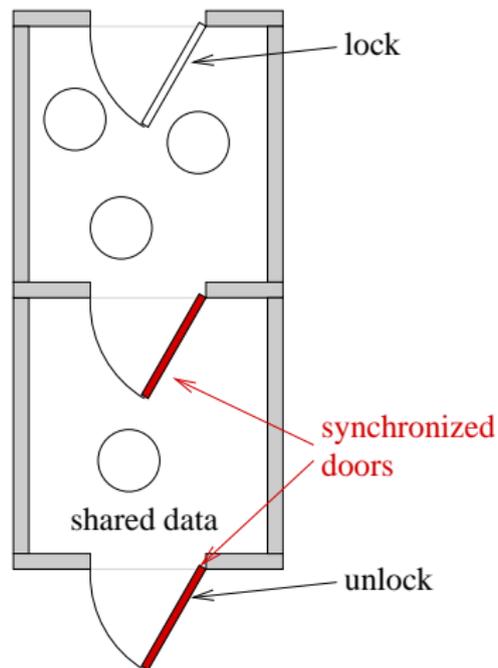
- ▶ ordre des instructions produites par le compilateur ?
- ▶ entrelacement/ordonnancement des *threads* imprévisible
- ▶ non-atomicité des opérations
  - ▶ même `count++` peut correspondre à plusieurs instructions
- ▶ cohérence cache/mémoire sur systèmes SMP ou multi-cœurs ?
  - ▶ sur x86, préfixe `lock` pour verrouiller le bus mémoire
  - ▶ mais très coûteux, et pas généré par le compilateur C

# Exclusion mutuelle

## les *mutexes*

- ▶ fourni par les bibliothèques de *threads* (ex. : pthread)
  - ▶ `void pthread_mutex_init (pthread_mutex_t *)`
  - ▶ `int pthread_mutex_lock (pthread_mutex_t *m)` bloque jusqu'à acquisition de `m`
  - ▶ `int pthread_mutex_trylock (pthread_mutex_t *m)` retourne 0 si `m` acquise
  - ▶ `int pthread_mutex_unlock (pthread_mutex_t *m)`
- ▶ toute donnée globale partagée doit être protégée par une **mutex** !

## Analogie des mutexes avec les verrous



## Exemple du compteur avec mutex

```
pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
int count = 0; /* modifié par différents fils */

void *loop (void *ignored) {
    for (int i = 0 ; i < 100000 ; i++) {
        pthread_mutex_lock (&count_lock);
        count++;
        pthread_mutex_unlock (&count_lock);
    }
    return NULL;
}

int main () {
    pthread_t tid;
    pthread_create (&tid, NULL, loop, NULL);
    loop (NULL);
    pthread_join (tid, NULL);
    printf ("%d\n", count);
}
```

## Exemple producteur/consommateur

```
mutex_t mutex = MUTEX_INITIALIZER;
```

```
void producer (void *ignored) {  
    for (;;) {  
        /* produce an item and put in  
         nextProduced */  
  
        while (count == BUFFER_SIZE) {  
  
            /* Do nothing */  
  
        }  
  
        buffer[in] = nextProduced;  
        in = (in + 1) % BUFFER_SIZE;  
        count++;  
  
    }  
}
```

```
void consumer (void *ignored) {  
    for (;;) {  
  
        while (count == 0) {  
  
            /* Do nothing */  
  
        }  
  
        nextConsumed = buffer[out];  
        out = (out + 1) % BUFFER_SIZE;  
        count--;  
  
        /* consume the item in  
         nextConsumed */  
  
    }  
}
```

## Exemple producteur/consommateur

```
mutex_t mutex = MUTEX_INITIALIZER;
```

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in
           nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {

            thread_yield ();

        }

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {

            thread_yield ();

        }

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        /* consume the item in
           nextConsumed */
    }
}
```

## Exemple producteur/consommateur

```
mutex_t mutex = MUTEX_INITIALIZER;
```

```
void producer (void *ignored) {  
    for (;;) {  
        /* produce an item and put in  
         nextProduced */  
  
        mutex_lock (&mutex);  
        while (count == BUFFER_SIZE) {  
            mutex_unlock (&mutex);  
            thread_yield ();  
            mutex_lock (&mutex);  
        }  
  
        buffer[in] = nextProduced;  
        in = (in + 1) % BUFFER_SIZE;  
        count++;  
        mutex_unlock (&mutex);  
    }  
}
```

```
void consumer (void *ignored) {  
    for (;;) {  
        mutex_lock (&mutex);  
        while (count == 0) {  
            mutex_unlock (&mutex);  
            thread_yield ();  
            mutex_lock (&mutex);  
        }  
  
        nextConsumed = buffer[out];  
        out = (out + 1) % BUFFER_SIZE;  
        count--;  
        mutex_unlock (&mutex);  
  
        /* consume the item in  
         nextConsumed */  
    }  
}
```

## Exemple producteur/consommateur

### problèmes et solution

- ▶ attente active dans l'application : mauvaise idée
  - ▶ le `while` consomme du CPU, même quand on ne peut pas progresser
- ▶ mieux : **informer l'ordonnanceur** de quand on peut travailler
- ▶ utilisation de **variables de condition**
- ▶ `cond_init (cond_t *c, ...)`
- ▶ `cond_wait (cond_t *c, mutex_t *m)`
  - ▶ atomiquement déverrouille `m` et dort jusqu'à `c` signalée
  - ▶ puis ré-acquiert `m` et poursuit
- ▶ `cond_signal (cond_t *c)`  
`cond_broadcast (cond_t *c)`
  - ▶ réveille un/tous les *threads* en attente
- ▶ **structure de synchronisation associée à une condition logique**

## Producteur/consommateur avec variables de condition

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;
```

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and
           put in nextProduced */

        mutex_lock (&mutex);
        if (count == BUFFER_SIZE)
            cond_wait (&nonfull, &mutex);

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        if (count == 0)
            cond_wait (&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        /* consume the item
           in nextConsumed */
    }
}
```

Et avec plusieurs lecteurs/écrivains ?

## Producteur/consommateur avec variables de condition

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;
```

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and
           put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
            cond_wait (&nonfull, &mutex);

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
            cond_wait (&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        /* consume the item
           in nextConsumed */
    }
}
```

Mettre un while autour du cond\_wait.

## Producteur/consommateur avec variables de condition

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;
```

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and
           put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
            cond_wait (&nonfull, &mutex);

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
            cond_wait (&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        /* consume the item
           in nextConsumed */
    }
}
```

**Attention** : pas de garantie premier arrivé premier servi

## Variables de condition

- ▶ pourquoi `cond_wait` doit à la fois relâcher la mutex et dormir ? pourquoi ne pas séparer ?

```
while (count == BUFFER_SIZE) {
    mutex_unlock (&m);
    cond_wait_hypothétique (&nonfull);
    mutex_lock (&m);
}
```

- ▶ parce qu'on pourrait avoir un **mauvais entrelacement**
  - ▶ p. ex. le producteur manquerait le signal :

```
/* producteur */                | /* consommateur */
while (count == BUFFER_SIZE) {  |
    mutex_unlock (&m);          |
    | mutex_lock (&m);          |
    | /* ... */                 |
    | count--;                  |
    | cond_signal_hypo (&nonfull); |
    cond_wait (&nonfull);       |
```

## Autres problèmes courants de synchronisation

### ▶ interblocage (*deadlock*)

- ▶  $P_0$  et  $P_1$  veulent accéder à  $R$  et  $S$  protégées par un verrou

$P_0$	$P_1$
lock(S)	lock(R)
lock(R)	lock(S)
...	...
unlock(R)	unlock(S)
unlock(S)	unlock(R)

### ▶ famine (*starvation*)

- ▶ un des *threads* n'acquiert jamais la ressource
- ▶ p. ex. à cause de l'ordre dans lequel on sert les *threads*

## Alternatives à la concurrence avec mémoire partagée

- ▶ la programmation parallèle n'implique pas *forcément* la mémoire partagée !
- ▶ passage de messages
  - ▶ MPI (*Message Passing Interface*) sur grappes de calcul
  - ▶ langage Erlang
- ▶ programmation fonctionnelle (Scheme, OCaml, Haskell, etc.)
  - ▶ passage de valeurs (constantes) entre fonctions
  - ▶ pas d'accès globaux à synchroniser

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Mise en œuvre des fils d'exécution et processus par le noyau

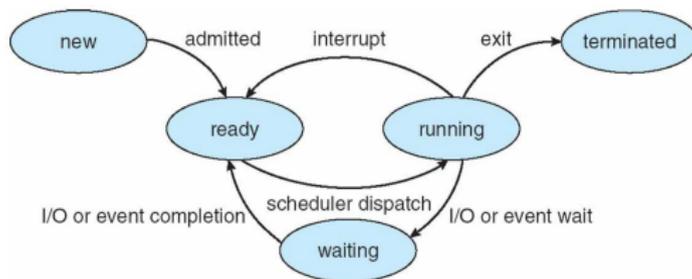
- ▶ le SE garde une structure pour chaque fil ou proc.
  - ▶ *Thread/Process Control Block* (TCB/PCB)
  - ▶ PCB appelé `task_struct` dans Linux
- ▶ contient l'état du fil/processus
  - ▶ en cours d'exécution, prêt, bloqué, etc.
- ▶ inclus l'info nécessaire pour le faire tourner
  - ▶ registres (dont pointeurs de pile et de programme)
  - ▶ pour un processus : projections mémoire, desc. de fichier
- ▶ d'autres informations
  - ▶ ID utilisateur/groupe, masque de signal, terminal de contrôle, stats de conso. de ressources, etc.

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

## fork et exec

- ▶ `fork` crée une copie du PCB
  - ▶ descripteurs de fichier hérités
  - ▶ fichiers ouverts à la fois par le parent et le fils (chacun doit les fermer)
  - ▶ pages mémoires partagées (text, rodata, etc.)
- ▶ `exec` **remplace** tout
  - ▶ registres (dont pointeur d'instruction), espace d'adressage
  - ▶ **mais** descripteurs de fichier conservés

## État des processus et fils



- ▶ fils et processus dans un état
  - ▶ *nouveau* ou *terminé* au début/fin
  - ▶ en cours d'exécution (*running*)
  - ▶ *prêt* – pourrait s'exécuter, mais CPU occupé
  - ▶ *en attente* – attend la complétion d'entrée/sortie, verrou libéré, etc.
- ▶ que processus/fil le noyau choisit-il d'exécuter ?
  - ▶ si 0 prêt, on attend ; si 1 prêt, on l'exécute
  - ▶ si  $> 1$  prêt, il faut faire un choix d'ordonnancement

## Ordonnancement de processus/fils

Quel processus/thread exécuter ?

- ▶ parcourir la table des processus pour en trouver un prêt ?
  - ▶ coûteux, effets douteux (ex. : triés par PID)
  - ▶ séparer les *prêts* et les autres
- ▶ FIFO ?
  - ▶ processus ajoutés en fin de file
  - ▶ retirés en tête de file
- ▶ priorité ?
  - ▶ donner plus de chance à CPU

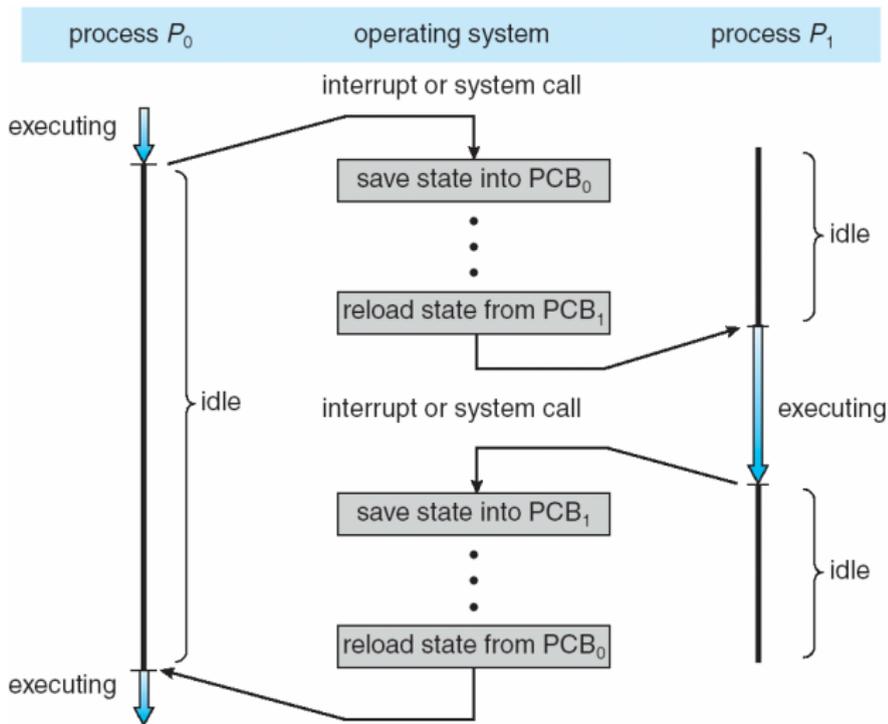
## Politique d'ordonnancement

- ▶ équilibre entre plusieurs objectifs
  - ▶ *équité* – éviter la famine de certains processus/fils
  - ▶ *priorité* – prendre en compte l'importance relative
  - ▶ *échéances* – doit faire  $x$  d'ici  $t$  (ex. : audio/vidéo)
  - ▶ *débit* – bonne performance globale
  - ▶ *réactivité* – minimiser le temps de réponse (ex. : GUI)
  - ▶ *efficacité* – minimiser le surcoût engendré par l'ordonnanceur
- ▶ **pas de politique universelle**
  - ▶ beaucoup de critères, conflictuels
- ▶ ordonnancement mémoire et CPU intimement liés
  - ▶ ex. : un processus avec mémoire et sans CPU ne peut pas progresser, donc la mémoire est gâchée

## Préemption

- ▶ processus/fil effectivement préempté quand le noyau prend la main
- ▶ un processus peut passer la main au noyau
  - ▶ appel système, défaut de page, instruction invalide, division flottante par zéro, etc.
  - ▶ peut s'endormir — p. ex. en initiant des E/S
  - ▶ peut réveiller d'autres — p. ex. `fork`, écriture dans un tube
- ▶ la minuterie donne la main au noyau périodiquement
  - ▶ si le processus courant a assez tourné, ordonnance un autre
- ▶ interruption matérielle
  - ▶ requête disque servie, paquet arrivé sur le réseau, etc.
  - ▶ peut rendre des processus *prêts*
  - ▶ peut mener à l'ordonnancement d'un autre proc. avec plus haute priorité
- ▶ **changement de contexte**

## Changement de contexte (*context switch*)



## Détails du changement de contexte

- ▶ dépendant de l'architecture :
  - ▶ sauvegarde du pointeur d'instruction + registres entiers
  - ▶ sauvegarde des registres virgule flottante
  - ▶ si changement de processus, changement de traductions d'adresses virtuelles
- ▶ coûts non négligeables
  - ▶ sauvegarde/restauration des registres flottants
    - ▶ optimisation : que si ils ont été utilisés
  - ▶ peut nécessiter de vider le TLB (traduction d'adresses matérielle)
    - ▶ optimisation : conserver les données du noyau dans le TLB
  - ▶ entraîne des défauts de cache

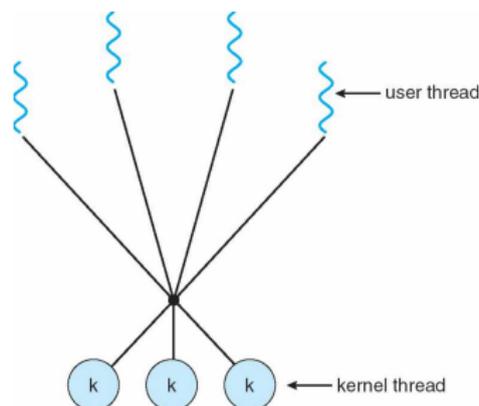
## Limitations des *threads* mis en œuvre par le noyau

- ▶ chaque opération sur les *threads* passe par le noyau
  - ▶ un appel système prend  $\sim 50\times$  plus de temps qu'un appel de fonction
- ▶ une implémentation des *threads* pour plaire à tous (`pthread`)
  - ▶ répond à beaucoup de besoins
  - ▶ mais chacun paye pour des fonctionnalités parfois superflues (priorités, etc.)
- ▶ demande mémoire lourde
  - ▶ allocation des structures de données du noyau à la création, etc.

## Mise en œuvre des *threads* en espace utilisateur

- ▶ utilise un seul *thread* noyau : **n : 1**
- ▶ exemple : bibliothèque GNU Pth, <http://gnu.org/software/pth>
- ▶ ordonnanceur en espace utilisateur
  - ▶ file de *threads* utilisateur prêts, en attente, etc.
  - ▶ TCB avec contexte d'exécution de chaque *thread*
  - ▶ éventuellement préemption avec minuterie (`setitimer`)
- ▶ remplace les appels bloquants : `read` vs. `pth_read`, etc.
  - ▶ si l'opération peut bloquer, ordonnance un autre *thread*
- ▶ **avantage** : création et changement de contextes légers
- ▶ **inconvénient** : pas de parallélisme (un seul cœur de CPU)
- ▶ cf. fonctions `getcontext` et `swapcontext` de la `libc`

## Mise en œuvre des *threads* en espace utilisateur ( $n : m$ )



- ▶  $n$  threads utilisateurs sur  $m$  threads noyau :  $n : m$
- ▶ exemple : bibliothèque Marcel, <http://runtime.bordeaux.inria.fr/marcel/>
- ▶ **avantages** : threads légers, parallélisme
- ▶ **inconvénients** : le noyau ne dévoile pas tout, implémentation compliquée

## Micro-noyaux et ordonnancement en espace utilisateur \

### micro-noyau L4 — <http://14ka.org>

- ▶ *thread* associé à sa création à un **thread d'ordonnancement** (ThreadControl)
- ▶ chaque *thread* a 3 attributs d'ordonnancement :
  - ▶ priorité, durée des intervalles (*timeslice*), temps total (*total quantum*)
- ▶ protocole de préemption
  - ▶ quand un *thread* est préempté ou passe explicitement la main (*yields*)
  - ▶ message de préemption envoyé au *thread* d'ordonnancement
  - ▶ le *thread* d'ordonnancement peut changer la priorité, le temps imparti au *thread*, etc. (Schedule)
- ▶ **avantage** : ordonnancement des ressources de l'appli par l'appli, meilleure connaissance des besoins

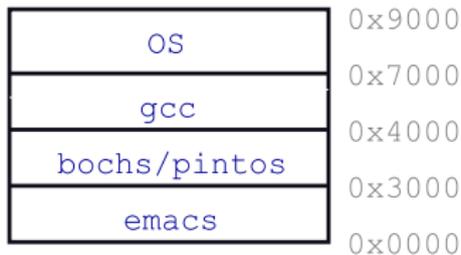
## Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## On veut que les processus puissent co-exister

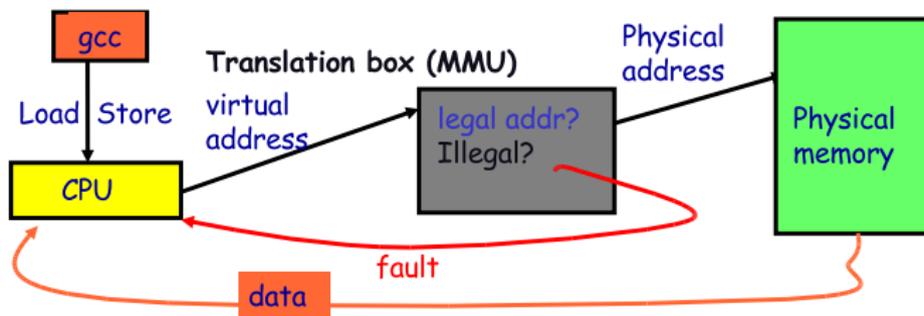


- ▶ imaginons le multi-tâche sur mémoire physique
  - ▶ qu'est-ce qui se passe si pintos a besoin de plus de mémoire ?
  - ▶ si emacs a besoin de plus de mémoire que disponible sur la machine ?
  - ▶ si pintos a un bug et écrit à 0x7100 ?
  - ▶ quand gcc a-t-il besoin de savoir qu'il tourne à 0x4000 ?
  - ▶ et si emacs n'utilise pas sa mémoire ?

## Problèmes avec le partage de la mémoire physique

- ▶ **protection**
  - ▶ un bug dans un processus peut corrompre un autre
  - ▶ empêcher que  $A$  puisse corrompre la mémoire de  $B$
  - ▶ empêcher que  $A$  puisse observer la mémoire de  $B$
- ▶ **transparence**
  - ▶ un processus ne devrait pas requérir des adresses spécifiques
  - ▶ les processus ont souvent besoin de larges zones contiguës (pile, grosses structures de données, etc.)
- ▶ **épuisement des ressources**
  - ▶ le programmeur suppose que la machine a « assez » de mémoire
  - ▶ somme des tailles des processus souvent supérieure à la mémoire physique

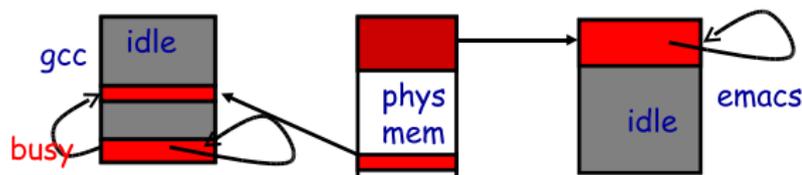
## Buts de la mémoire virtuelle



- ▶ donner à chaque programme **son** espace d'adressage
  - ▶ convertir chaque accès mémoire virtuelle en un accès mémoire physique
  - ▶ l'appli ne se soucie pas de quelle mémoire physique elle utilise
- ▶ garantir la protection
  - ▶ empêche une appli de toucher à la mémoire d'une autre
- ▶ permettre aux programmes de voir plus de mémoire qu'il n'en existe (*swap* sur disque)

## Avantages de la mémoire virtuelle

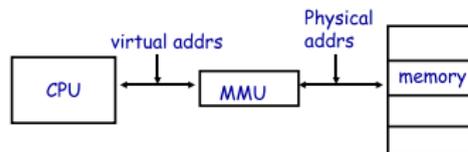
- ▶ peut **reloger un programme** pendant qu'il tourne
  - ▶ partiellement en mémoire physique, partiellement sur disque
- ▶ l'essentiel de la mémoire d'un processus est **inutilisée** (règle du 80/20)



- ▶ écrire les parties actuellement inutilisées sur disque
- ▶ laisse la mémoire physique libérée aux autres processus
- ▶ similaire au démultiplexage/virtualisation du CPU
  - ▶ quand un processus n'utilise pas le CPU, on le passe à un autre
  - ▶ quand un processus n'utilise pas une page mémoire, on la passe à un autre
- ▶ **défi** : une indirection en plus, pourrait être lent

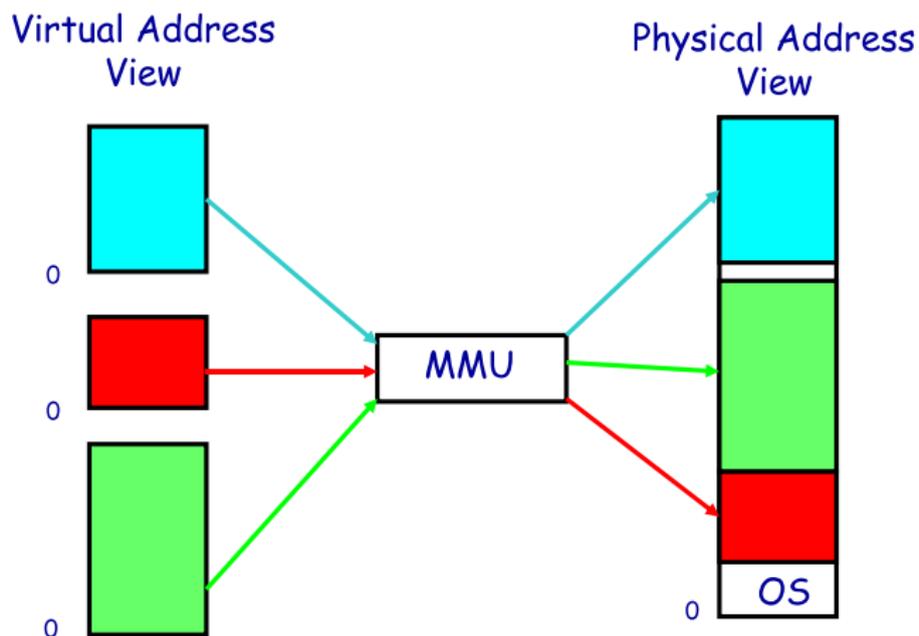
## Définitions

- ▶ les programmes chargent/écrivent à des **adresses virtuelles** (ou logiques)
- ▶ la mémoire vive est désignée avec des **adresses physiques** (ou réelles)
  - ▶ adresses physiques inaccessibles et inconnues de l'espace utilisateur
- ▶ unité **matérielle** de gestion de la mémoire (MMU, *memory management unit*)
  - ▶



- ▶ fait typiquement partie du CPU
  - ▶ accès avec des instructions privilégiées
- ▶ traduit les adresses virtuelles en adresses physiques
- ▶ donne une vue par processus appelée **espace d'adressage**

## Espace d'adressage



## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

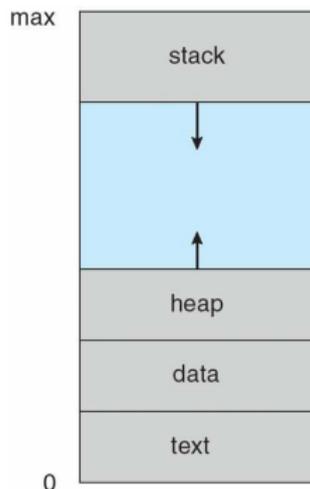
Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Ce que voit un processus

- ▶ chaque processus a son propre espace d'adressage
  - ▶ (char \*) 0x1234 différent dans chaque proc.
- ▶ zones mémoires typiques
  - ▶ **pile d'appel** (*stack*) : variables automatiquement allouées/déallouées (r/w)
  - ▶ **tas** (*heap*) : mém. allouée/déallouée dynamiquement, p. ex. malloc (r/w)
  - ▶ **données** (*data*) : variables globales initialisées (r/w)
  - ▶ **BSS** : variables globales non-initialisées, zéros (r/w)
  - ▶ **texte** (*text*) : code binaire (r/x)
- ▶ modèle de programmation simple

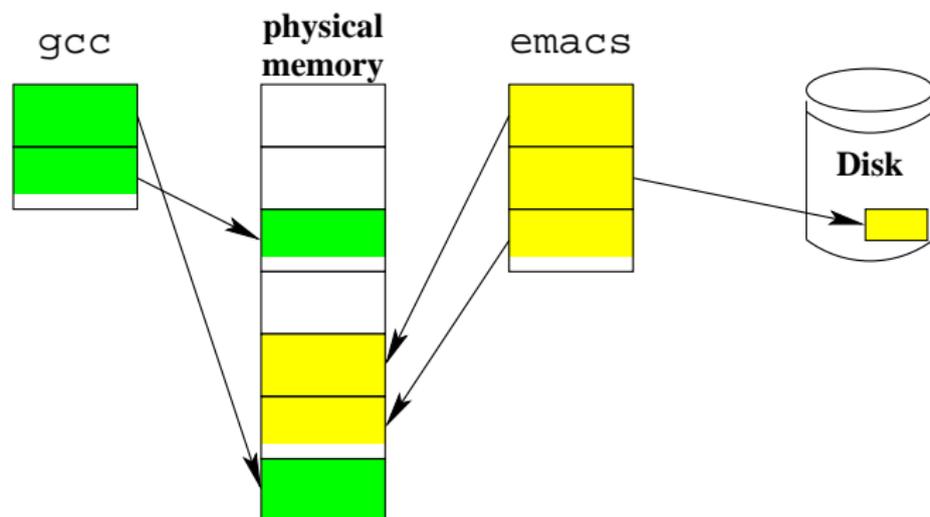




## Principe de la pagination

- ▶ mémoire divisées en **pages**
- ▶ correspondance **pages virtuelles** → **pages physiques**
  - ▶ projection virtuelle → physique propre à chaque processus
- ▶ permet au SE d'avoir le **contrôle sur certaines opérations**
  - ▶ pages en lecture seule : le SE est consulté si accès en écriture
  - ▶ accès invalides en lecture ou écriture : le SE est notifié
  - ▶ le SE peut changer la projection et faire reprendre l'exécution du processus
- ▶ autres fonctionnalités
  - ▶ le matériel peut maintenir des bits « accédé » ou « modifié » pour chaque page
  - ▶ avoir des droits « exécution », en plus de lecture/écriture
    - ▶ p. ex. : pile non exécutable pour éviter les attaques par débordement de tampon (*buffer overflow*)

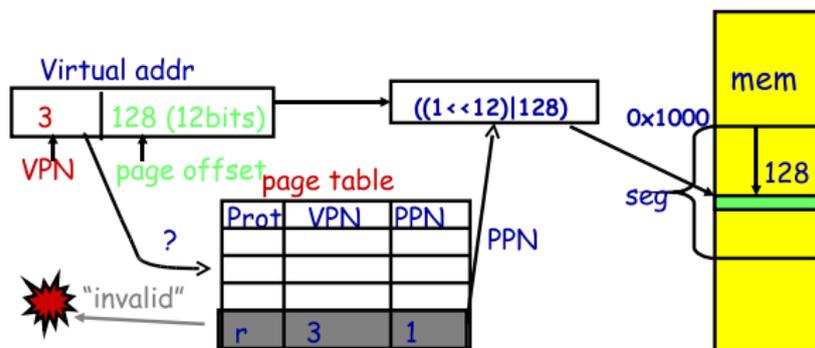
## Allocation des pages



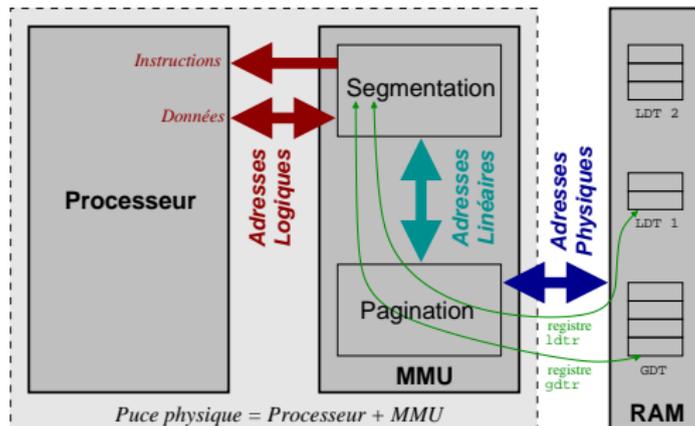
- ▶ n'importe quelle page physique allouée à n'importe quel processus
- ▶ les pages inutilisées peuvent être envoyées sur disque (*swap*)

## Structures de données pour la pagination

- ▶ les pages ont une taille fixe, p. ex. 4 Kio
  - ▶ 12 bits de poids faible : décalage dans la page (*page offset*)
  - ▶ bits de poids fort : numéro de page
- ▶ une **table de pages** associée à chaque processus
  - ▶ n° de page virtuelle (VPN) → n° de page physique (PPN)
  - ▶ inclue l'information concernant la protection (r/w/x), la validité, etc.
- ▶ à chaque accès : **traduction VPN → PPN + décalage**



## Gestion de la mémoire sur architecture Intel x86 (aka. IA32)



- ▶ **2 traductions** d'adresses successives
  - ▶ adresses virtuelles/logiques → linéaires → physiques
- ▶ le cœur du CPU ne perçoit pas les adresses physiques
  - ▶ traduction entièrement déléguée à la MMU

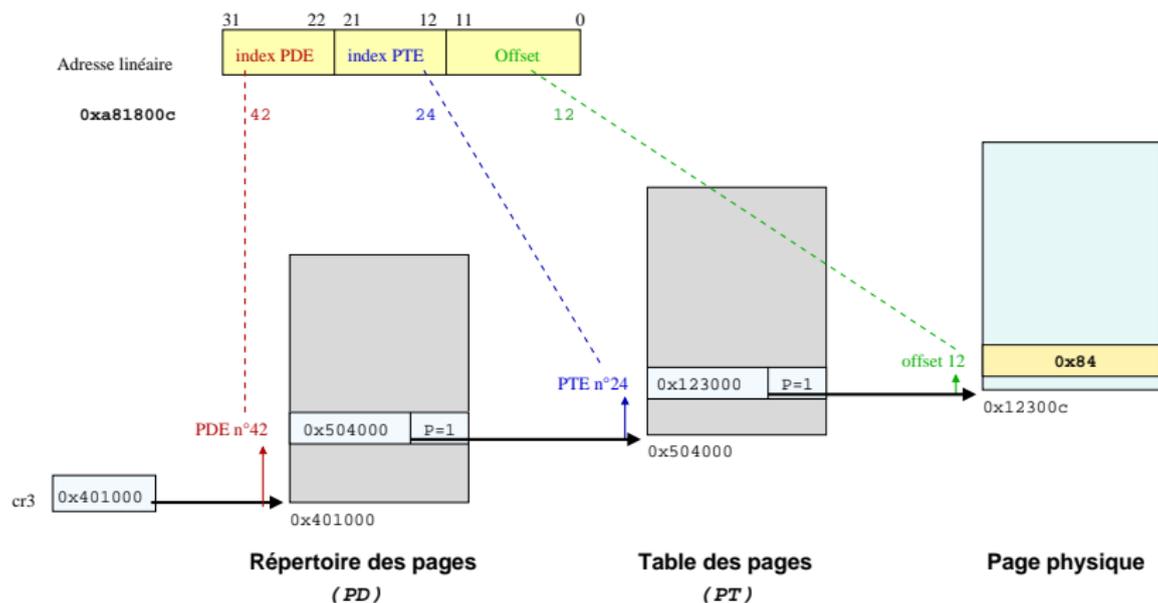
## Pagination sur architecture Intel x86 (aka. IA32) \

### traduction adresse linéaire → physique

- ▶ activée par des bits dans le registre de contrôle `%cr0`
  - ▶ privilégié : seul le noyau peut y toucher
- ▶ **pages de 4 Kio** (par défaut)
  - ▶ mots de 32 bits sur IA32 → 4 Gio de mémoire adressable
  - ▶ 12 bits pour le décalage, 20 bits pour le n° de page
- ▶ le registre `%cr3` pointe sur un **répertoire de pages**
  - ▶ 1 répertoire de pages par processus
  - ▶ 1024 entrées (*page directory entries*, PDE)
    - ▶ chaque entrée contient l'adr. physique d'une table des pages
    - ▶ indice de table sur 10 bits
- ▶ **table des pages** : 1024 entrées (*page table entries*, PTE)
  - ▶ chaque PTE contient l'adr. physique d'une page virtuelle
  - ▶ la table des pages couvre 4 Mio de mémoire virtuelle
  - ▶ indice de page sur 10 bits

# Pagination sur architecture Intel x86 (aka. IA32) \

## illustration



## Besoin d'un cache de traduction d'adresses

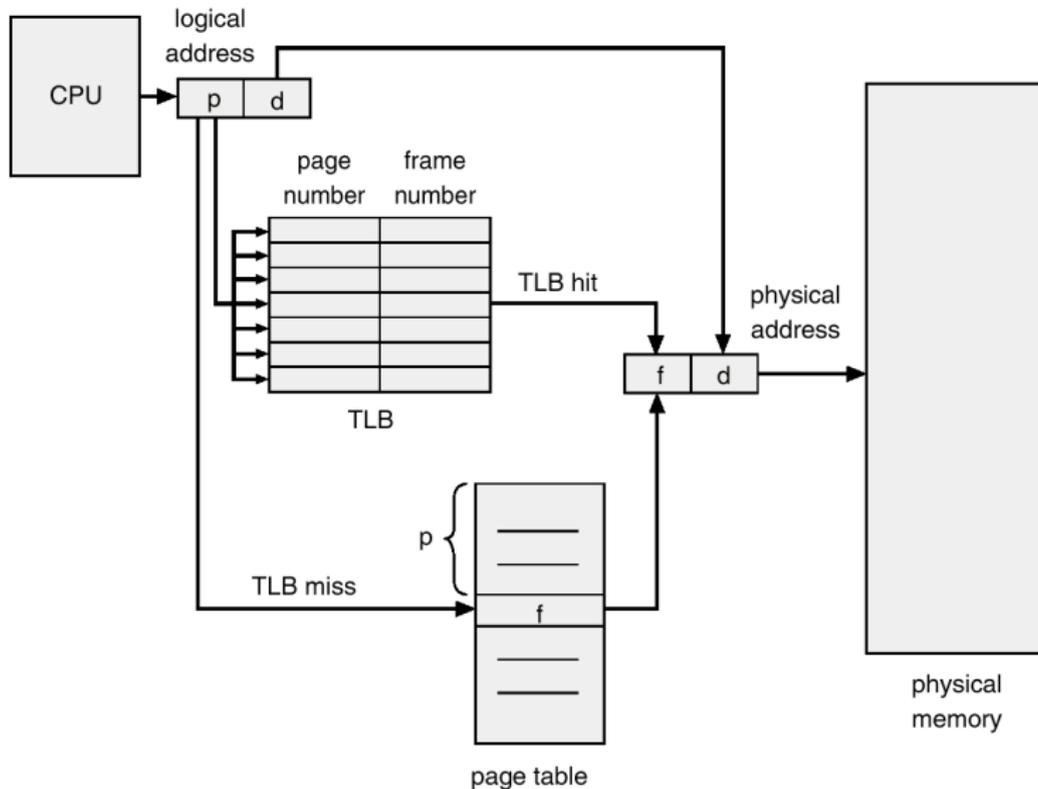
- ▶ la traduction adr. linéaire → physique implique **3 accès** en RAM
  1. répertoire des pages
  2. table des pages
  3. donnée demandée
- ▶ problème : **la mémoire vive est lente**

stockage →	registre	cache L1	cache L2	RAM	swap
nb cycles →	1	≈ 3	≈ 14	≈ 240	≈ 10 <sup>7</sup>
géographie →	Pessac	Talence	Cestas	Toulouse	Mars

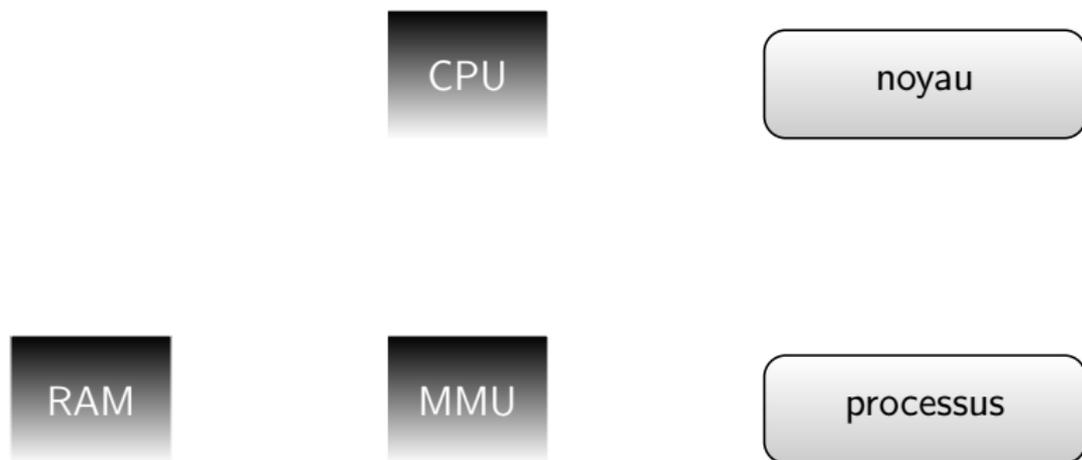
## Un cache de traduction d'adresses : le TLB

- ▶ la MMU dispose d'un cache : le **TLB** (*translation lookaside buffer*)
  - ▶ conserve les traductions les plus récentes
- ▶ le noyau est **responsable de garder le TLB valide**
  - ▶ chaque traduction peut être invalidée (instruction `invlpg`)
  - ▶ et lors d'un **changement de contexte**?
    - ▶ typiquement, **vide** le TLB (moins vrai sur x86-64 récent)

## Le TLB illustré

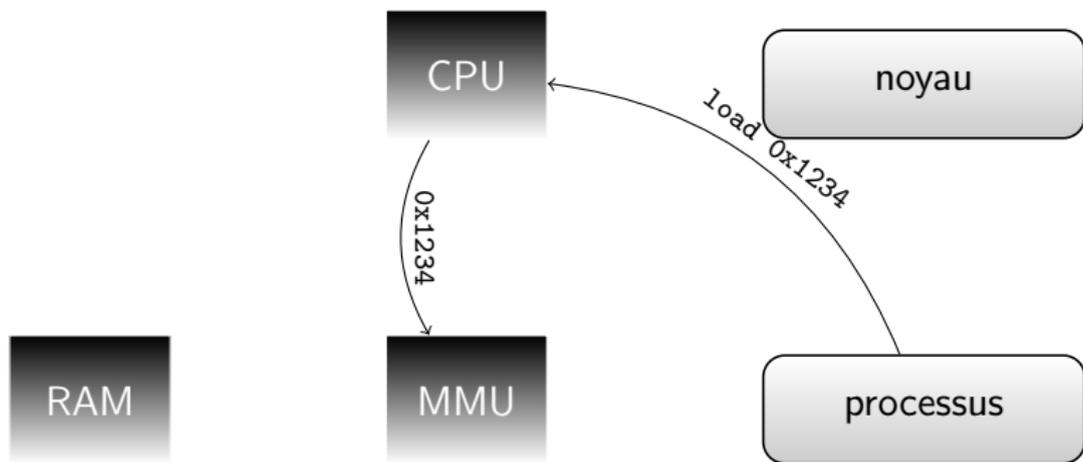


## Défauts de page : le film



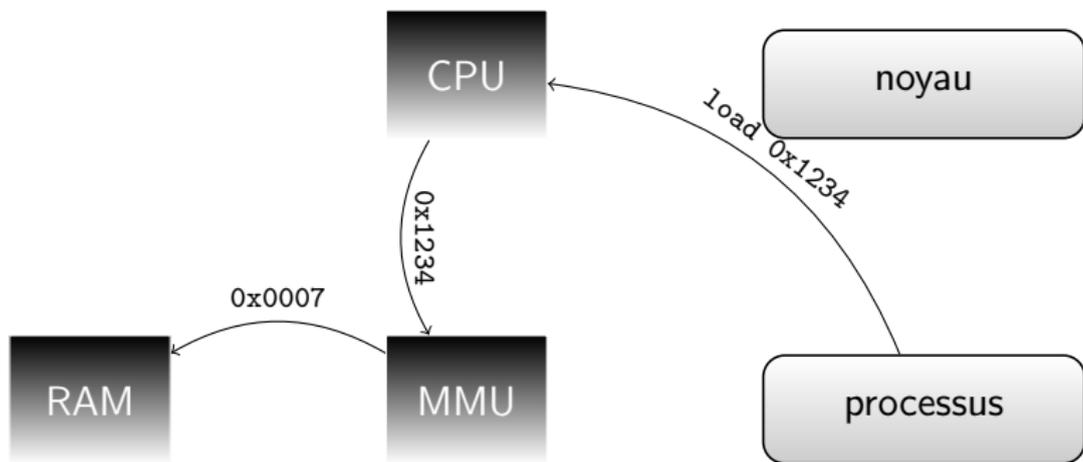
- ▶ accès mémoire virtuelle quand tout va bien : pas d'intervention du noyau

## Défauts de page : le film



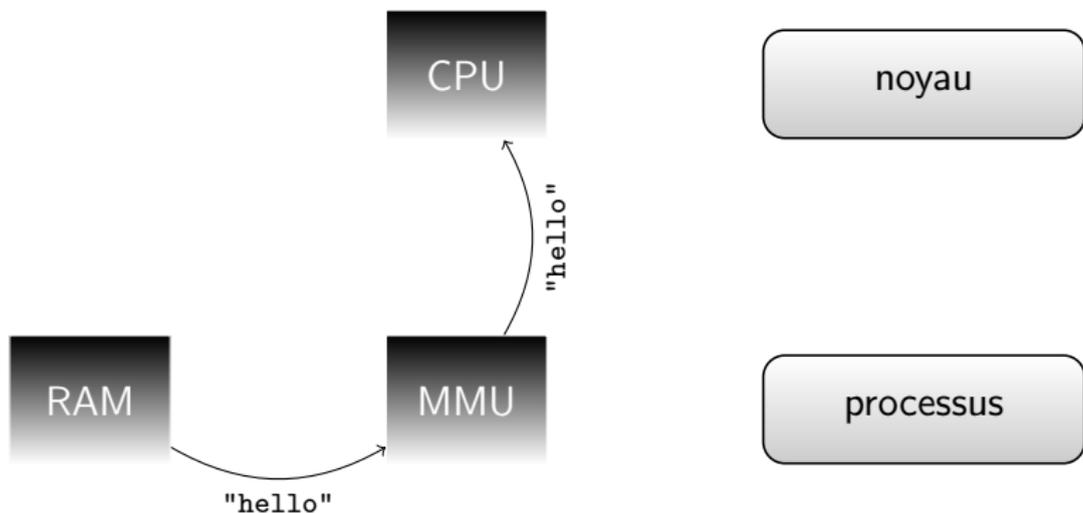
- ▶ accès mémoire virtuelle quand tout va bien : pas d'intervention du noyau

## Défauts de page : le film



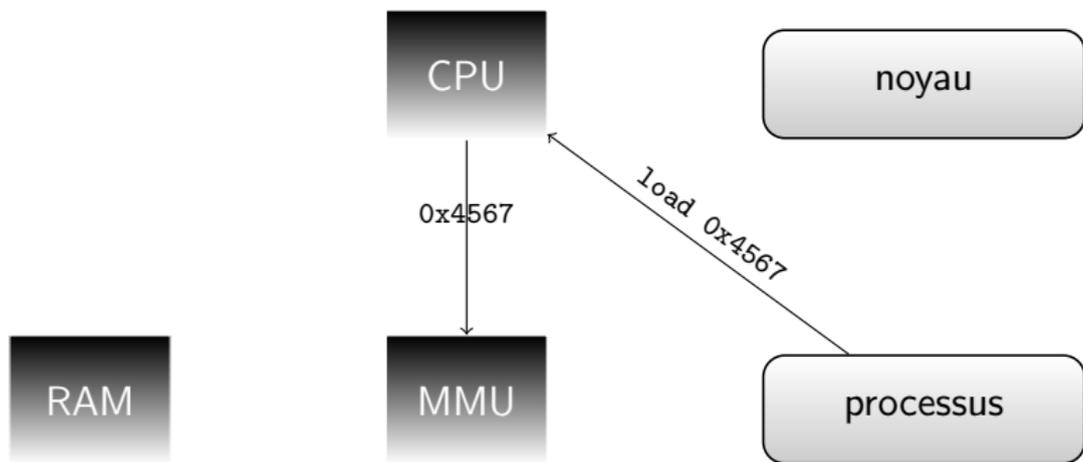
- ▶ accès mémoire virtuelle quand tout va bien : pas d'intervention du noyau

## Défauts de page : le film



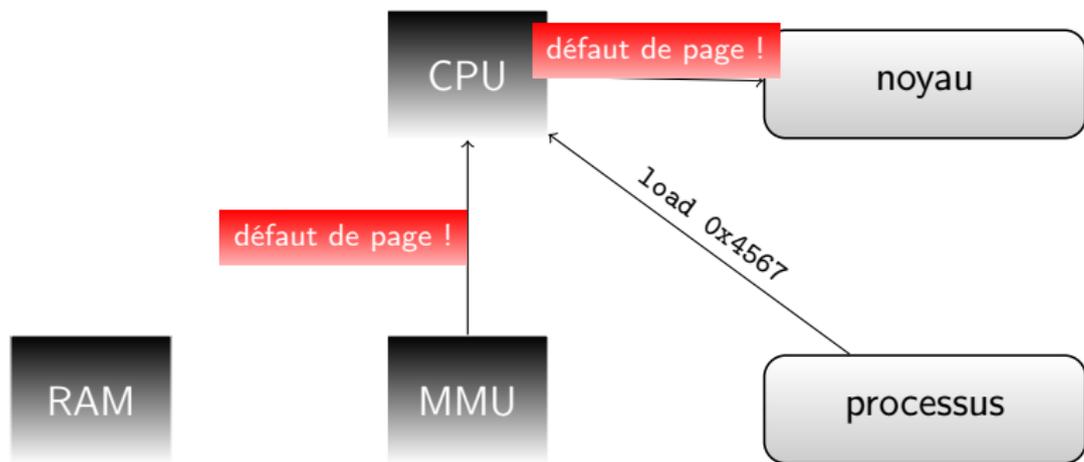
- ▶ accès mémoire virtuelle quand tout va bien : pas d'intervention du noyau

## Défauts de page : le film



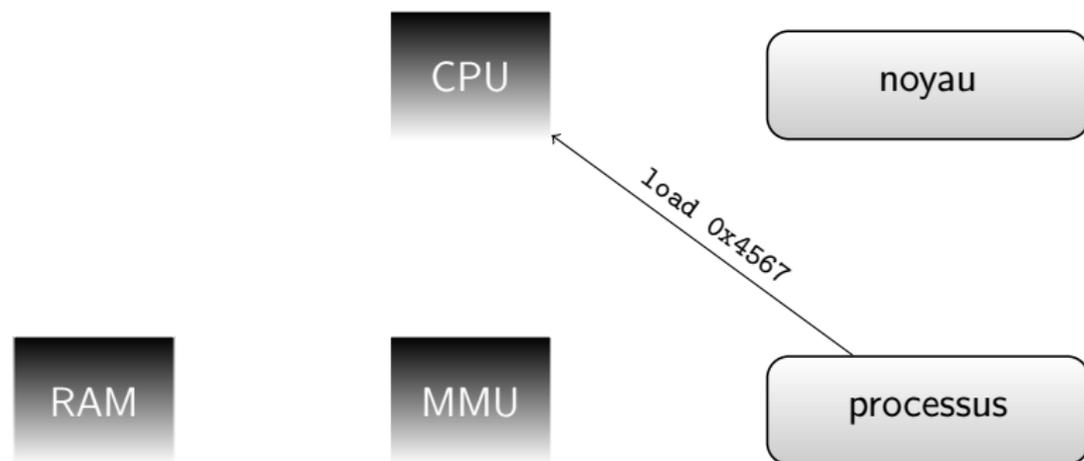
- ▶ accès à une adresse virtuelle pas encore projetée dans l'E.A. du processus, absente de la table des pages

## Défauts de page : le film



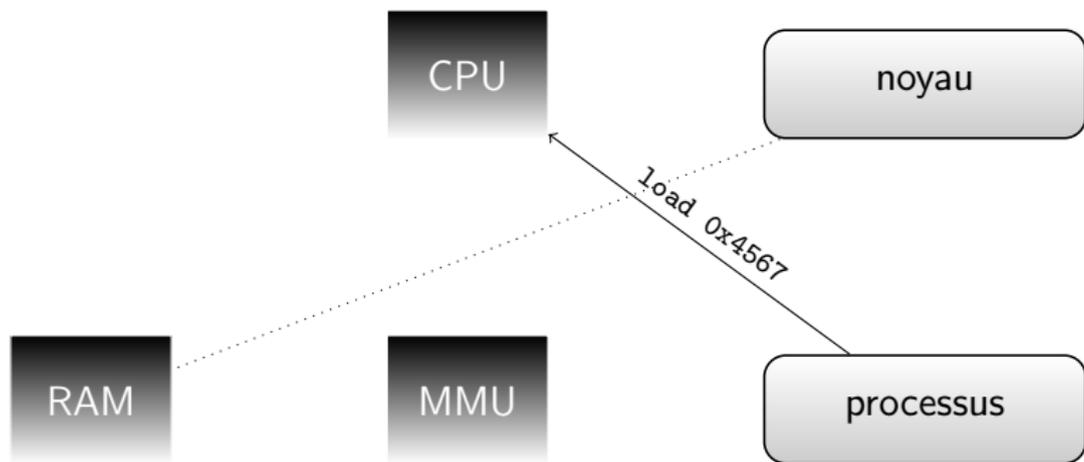
- ▶ la MMU lève une exception matérielle **défaut de page** (*page fault*), traitée par le noyau

## Défauts de page : le film



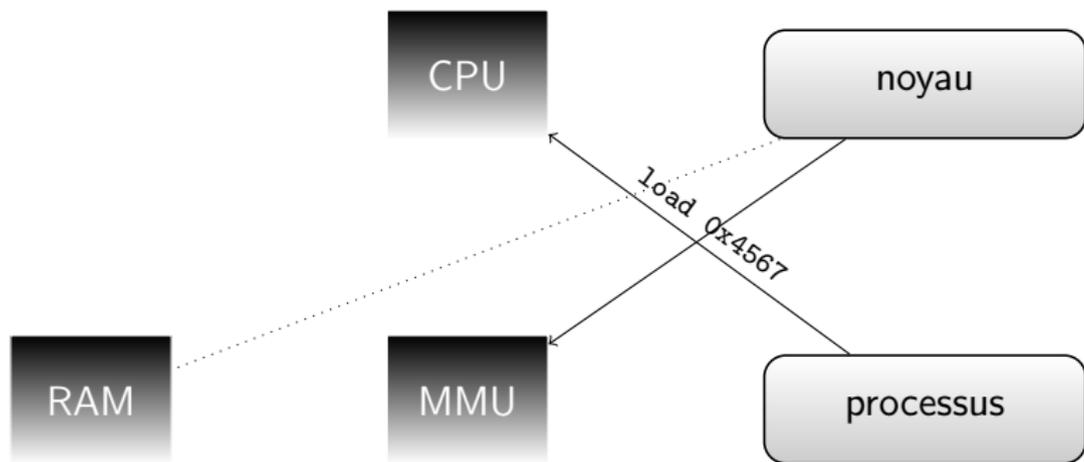
- ▶ le noyau cherche dans la table de pages de ce processus à quoi est associée la page virtuelle fautive

## Défauts de page : le film



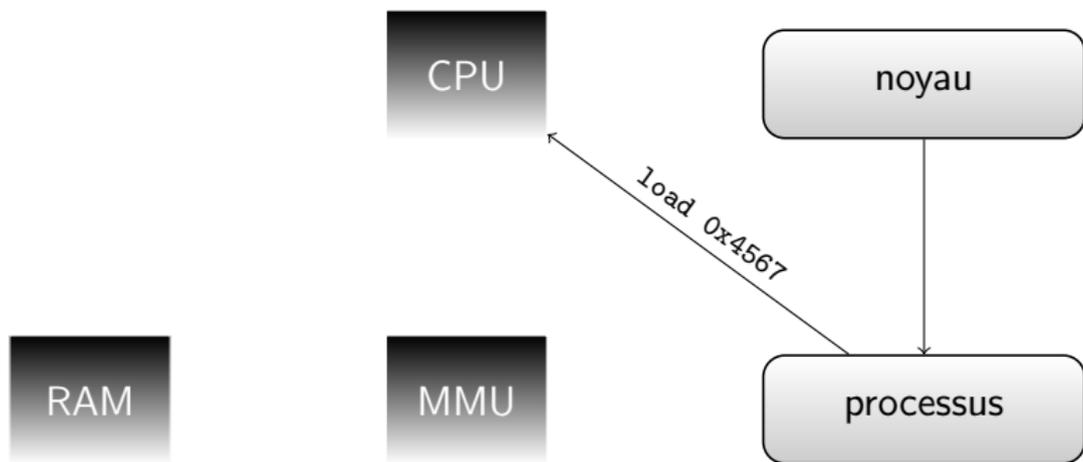
- ▶ si c'est une **adresse valide** : le noyau alloue une page physique...

## Défauts de page : le film



- ▶ ... projette la page physique dans l'E.A. du processus, met à jour la table des pages dans la MMU

## Défauts de page : le film



- ▶ ... puis repasse la main au processus, qui poursuit son exécution

## Défauts de page : le film

CPU

noyau

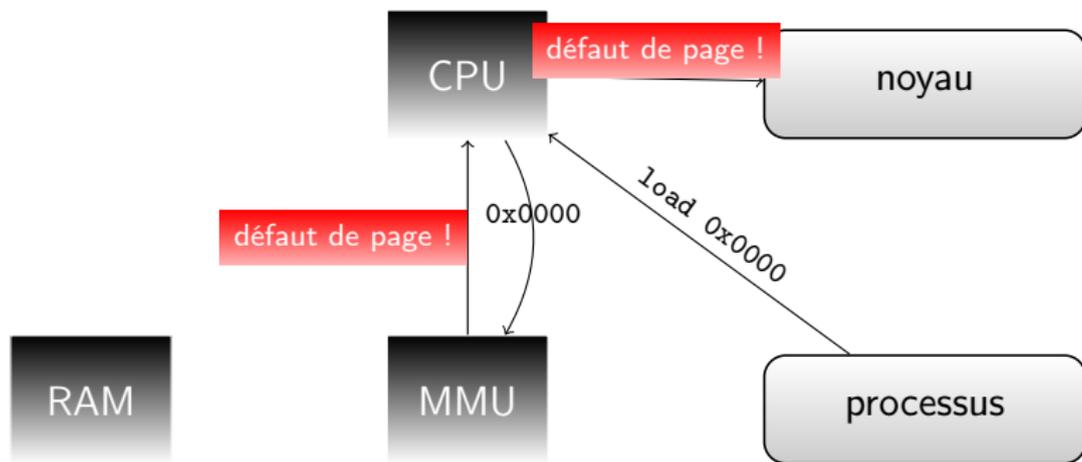
RAM

MMU

processus

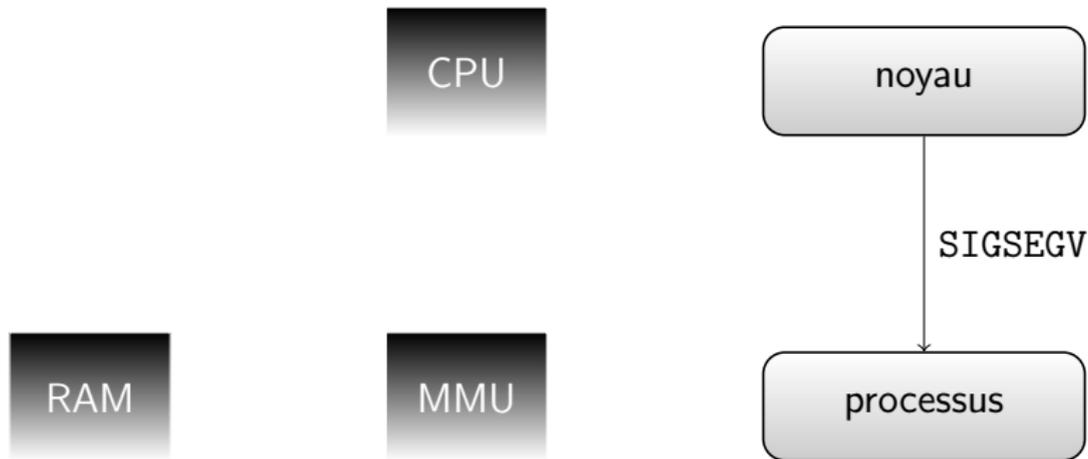
- ▶ ... puis repasse la main au processus, qui poursuit son exécution

## Défauts de page : le film



- ▶ accès à une **adresse virtuelle invalide**

## Défauts de page : le film

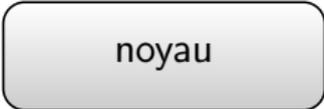


- ▶ le noyau constate l'accès invalide, envoie le signal SIGSEGV (*segmentation fault*) au processus...

## Défauts de page : le film



CPU



noyau



RAM



MMU

- ▶ ... et termine le processus (par défaut)

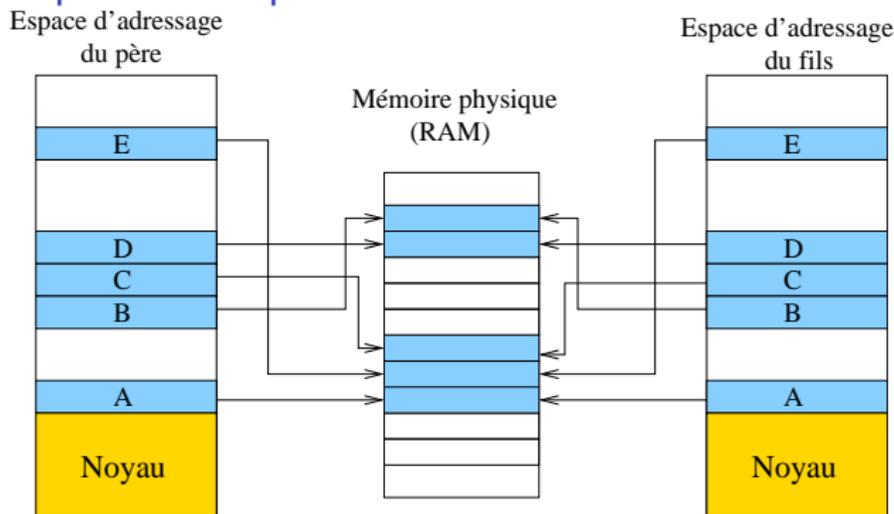
## La pagination au quotidien

- ▶ pagination **à la demande**, quand l'appli touche une page
  - ▶ agrandissement de la pile d'appel
  - ▶ allocation du BSS (données non initialisées)
  - ▶ chargement du text (code)
  - ▶ chargement des bibliothèques partagées (.so sous GNU/Linux, DLL sous Windows, etc.)
  - ▶ projection du contenu de fichiers en mémoires
- ▶ pagination **vers le disque** (*swap*)
  - ▶ donner l'impression d'avoir plus de mémoire qu'en réalité
- ▶ **copie sur écriture** (*copy-on-write*, COW)
  - ▶ copier  **paresseusement**  les pages physiques
  - ▶ utilisé par `fork`, `mmap`, etc.
- ▶ ...

## Exemple : la copie sur écriture et fork

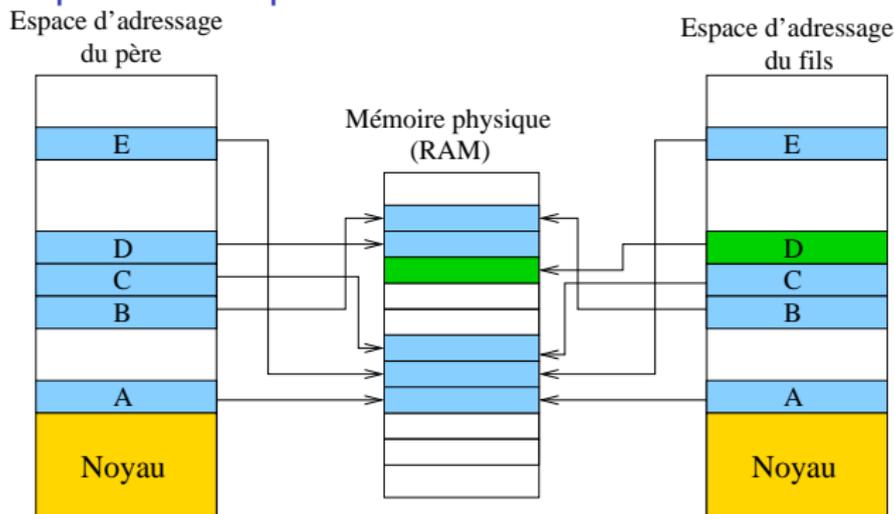
- ▶ fork crée un processus fils **copie conforme du père**
  - ▶ initialement : données du fils 100% identiques au père
  - ▶ puis divergence progressive
- ▶ optimisation : la **copie sur écriture** (*copy-on-write*)
  - ▶ au lieu de copier les pages physiques du père au fils, projeter les pages du père chez le fils **en lecture seule**
  - ▶ lorsque le fils écrit, **défaut de page** :
    - ▶ le SE **copie la page physique**
    - ▶ met à jour la projection dans le fils
    - ▶ redémarre le fils, qui utilise maintenant une copie

## Exemple : la copie sur écriture et fork



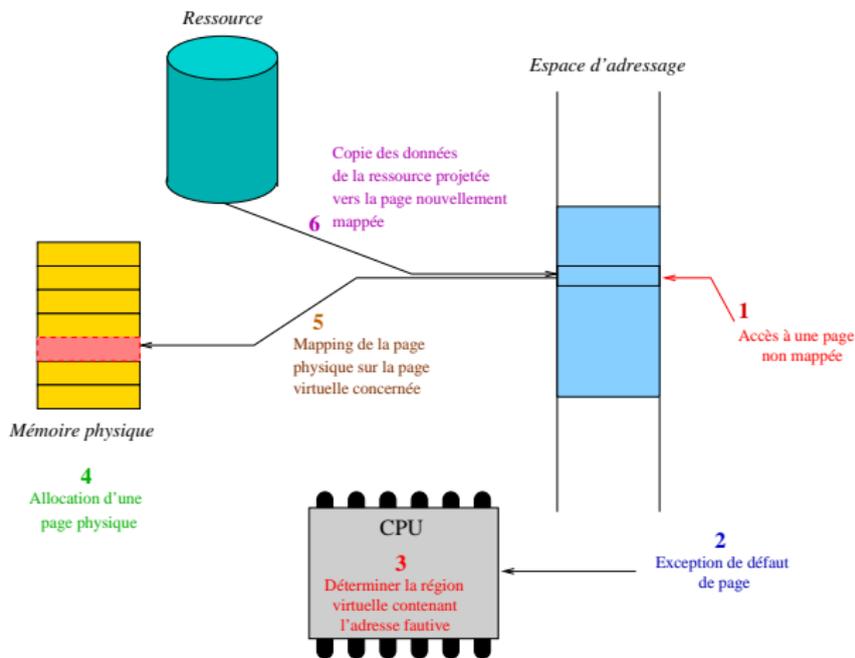
- ▶ juste après le fork : copies conformes, pages physiques partagées

## Exemple : la copie sur écriture et fork

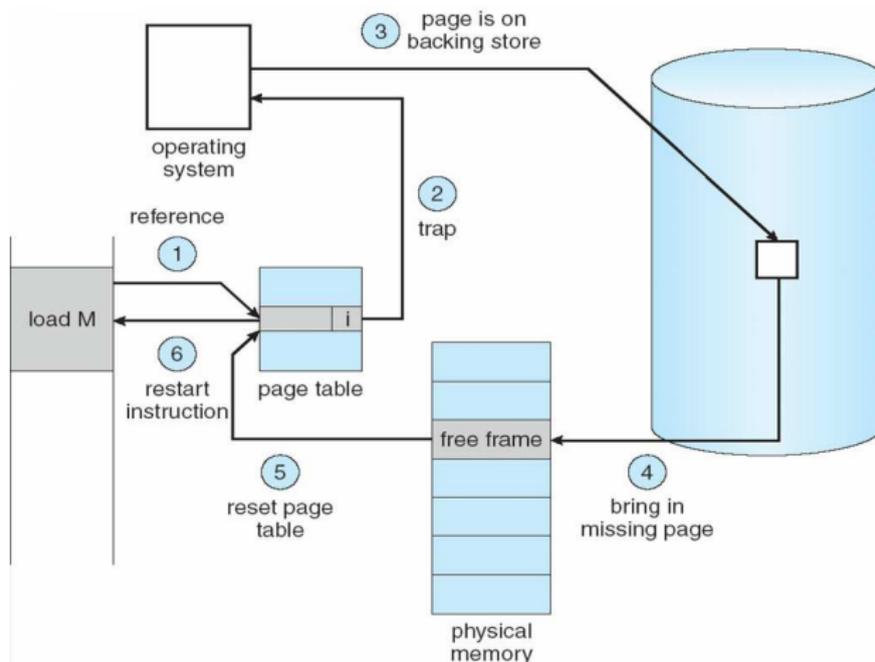


- ▶ plus tard : le fils a modifié la page *D* → **défait de page** → copie de *D* → **projection de la copie** dans le fils

## Exemple : projection d'un fichier en mémoire



- fichier projeté en mémoire, p. ex. avec `mmap` (voir plus loin)

Exemple : pagination sur disque (*swap*)

► la RAM est un « cache du disque dur »

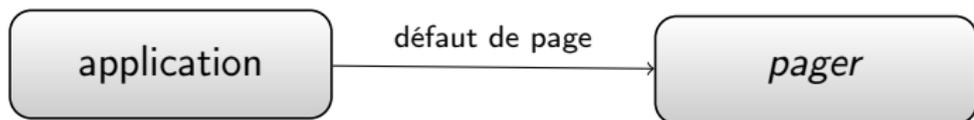
## Exemple : pagination sur disque (*swap*)

1. gcc a besoin d'une nouvelle page
2. le SE récupère une page inutilisée de emacs
  - ▶ si la page est **propre** (stockée sur disque à l'identique)
    - ▶ p. ex. le code binaire de emacs
    - ▶ peut toujours relire cette page depuis le fichier `/bin/emacs`
    - ▶ donc peut donner la page physique à gcc
  - ▶ si la page est **sale** (càd., pas d'exemplaire à jour sur disque)
    - ▶ doit écrire la page sur disque avant de la donner à gcc
3. marque la page virtuelle comme invalide dans emacs
  - ▶ au prochain accès à la page, emacs fera un *défaul de page*
  - ▶ alors le SE lira la page depuis le disque, la projettera dans l'E.A. de emacs, et poursuivra son exécution

## Pagination sur disque : comment choisir ?

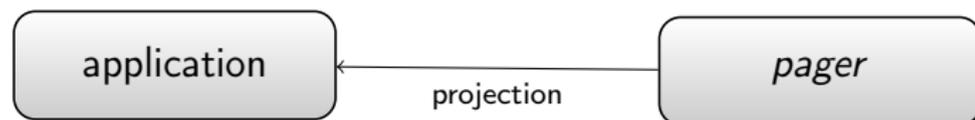
- ▶ quand la mémoire vive est pleine, il faut **évincer des pages**
  - ▶ càd. libérer des pages physiques en les envoyant sur disque
- ▶ politique d'**éviction des pages** : quelle page envoyer sur disque ?
  - ▶ typiquement implémentée dans le noyau (monolithique)
  - ▶ sans connaissance du sens des données contenues dans la page, sans informations du processus, etc.
  - ▶ càd. devinnette
- ▶ problème : envoyer sur disque peut **être une mauvaise idée**
  - ▶ ex. : un cache de rendu de fichier PDF
  - ▶ plus performant de détruire la page et de la reconstruire plus tard si besoin
  - ▶ cf. `madvise VOLATILE` and `vmpressure_fd`
- ▶ solution (?) : **coopérer avec l'application**
  - ▶ la notifier de la « pression mémoire »
  - ▶ lui permettre d'avoir sa politique d'éviction

## Auto-pagination des applications sur micro-noyau \ micro-noyau L4



- ▶ chaque processus a un *thread* dédié : le *paginateur* (*pager*)
  - ▶ défaut de page → message envoyé au *pager*
  - ▶ le *pager* envoie un message « projection » au *thread* fautif
  - ▶ le *thread* fautif reprend son exécution
- ▶ hiérarchie de *paginateurs*
  - ▶ le *paginateur* racine a toute la mémoire physique projetée
  - ▶ peut en déléguer aux processus fils

## Auto-pagination des applications sur micro-noyau \ micro-noyau L4



- ▶ chaque processus a un *thread* dédié : le *paginateur* (*pager*)
  - ▶ défaut de page → message envoyé au *pager*
  - ▶ le *pager* envoie un message « projection » au *thread* fautif
  - ▶ le *thread* fautif reprend son exécution
- ▶ hiérarchie de *paginateurs*
  - ▶ le *paginateur* racine a toute la mémoire physique projetée
  - ▶ peut en déléguer aux processus fils

## Interface POSIX pour manipuler les projections

- ▶ permet à un **processus utilisateur** de demander une projection ou d'en annuler une
- ▶ `void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - ▶ retourne un pointeur (adresse virtuelle) de la projection, peut-être `addr`
  - ▶ `prot` définit les droits : `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`
  - ▶ `flags` définit le type de projection
    - ▶ `MAP_ANONYMOUS` : mémoire physique (« anonyme »), `fd` ignoré
    - ▶ sinon projette le fichier pointé par `fd` (desc. de fichier)
    - ▶ `MAP_SHARED` → changements écrits dans le fichier
    - ▶ `MAP_PRIVATE` → changements privés au processus (COW)
- ▶ `int munmap(void *addr, size_t length)`

## Exemple d'utilisation de mmap

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

int main (int argc, char *argv[]) {
    int fd; char *projection;
    fd = open ("/dev/zero", O_RDONLY);
    projection = mmap (NULL, 123, PROT_READ,
        MAP_PRIVATE, fd, 0);
    printf ("%02x %02x %02x...\n",
        projection[0], projection[1],
        projection[2]); /* 00 00 00... */
    projection[0] = 1;      /* défaut de page -> SIGSEGV */
    return 0;
}
```

## Plan

Intro séance1

Présentation du cours

Qu'est-ce qu'un SE ?

Objectifs d'un système d'exploitation séance1

Historique

Abstraction

Démultiplexage des ressources & isolation

Utilisation efficace des ressources

Résumé

Fils d'exécution et processus séance1

Point de vue de l'application

Programmation concurrente

Point de vue du noyau séance2

Gestion de la mémoire virtuelle

Pourquoi a-t-on besoin de mémoire virtuelle ?

Pagination

## Quelle est la difficulté ?

- ▶ permettre d'allouer des blocs de **n'importe quelle taille**
- ▶ serait facile sans free



- ▶ on ne ferait que agrandir le tas (sbrk)
- ▶ problèmes de **fragmentation**
  - ▶ plein de trous peuvent apparaître

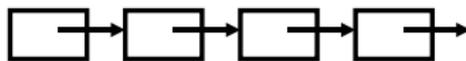


- ▶ mémoire libre mais requêtes non satisfaites !

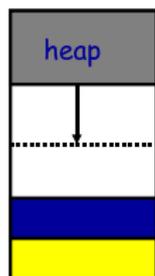
## Comment malloc s'y prend

- ▶ maintient une liste chaînée des zones libres

freelist



- ▶ au début, un seul élément : le tas
- ▶ puis un élément par intervalle libre
- ▶ si allocation  $\geq 4$  Kio  $\rightarrow$  mmap (anonyme)
- ▶ sinon
  - ▶ cherche un intervalle libre de taille suffisante
  - ▶ ou agrandit le tas (sbrk)



sbrk(4096)

```

/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
    void *p;
    if(!(p = sbrk(nbytes)))
        error("virtual memory exhausted");
    return p;
}

```

## Résumé

- ▶ la mémoire virtuelle, c'est bien
  - ▶ permet la répartition de la mémoire physique entre processus
  - ▶ permet l'isolation entre processus
- ▶ pagination gérée par le matériel et le système d'exploitation
  - ▶ unité matérielle de gestion de la mémoire virtuelle : la MMU
  - ▶ traduction d'adresses virtuelles → physiques par la MMU
  - ▶ guidées par le noyau (table des pages)
- ▶ bénéfiques
  - ▶ économie de la mémoire physique
  - ▶ allocation à la demande des pages physiques
  - ▶ permet le *swap*, la projection de fichiers, etc.

## Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Plan

Intro	séance1
Présentation du cours	
Qu'est-ce qu'un SE ?	
Objectifs d'un système d'exploitation	séance1
Historique	
Abstraction	
Démultiplexage des ressources & isolation	
Utilisation efficace des ressources	
Résumé	
Fils d'exécution et processus	séance1
Point de vue de l'application	
Programmation concurrente	
Point de vue du noyau	séance2
Gestion de la mémoire virtuelle	
Pourquoi a-t-on besoin de mémoire virtuelle ?	
Pagination	

## Matrice de contrôle d'accès \ Unix/POSIX, etc.

Objects

	File 1	File 2	File 3	...	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
...					
User m	read	write	read	write	read

Subjects

- ▶ des **subjects** accèdent à des **objects**
  - ▶ sujet = processus/utilisateur
  - ▶ object = fichier

## Utilisateurs Unix

- ▶ chaque utilisateur a un nom symbolique et un UID (entier)
  - ▶ exemples : bob → 101 ; root → 0
- ▶ mots de passe et autres infos stockés dans /etc
  - ▶ /etc/passwd contient les noms/UID
  - ▶ /etc/shadow idem + un condensé des mots de passe
    - ▶ condensé = résultat d'une fonction de hachage à sens unique

## Droits d'accès spécifiés par rôle/groupe

- ▶ chaque fichier Unix a un propriétaire et un groupe
  - ▶ droits *read*, *write*, *execute*
  - ▶ `chown -R ludo $HOME ; chgrp -R prof $HOME`
  - ▶ `chmod -R u+rw,g-w,o-rw $HOME`
- ▶ optionnellement une liste de contrôle d'accès (ACL)
- ▶ matrice de contrôle d'accès découpée par colonne :
  - ▶ à un objet est associé une série de paires sujets/permissions
- ▶ exemples
  - ▶ 

```
$ ls -l examen.lout  
-rw-r--r-- 1 ludo users ... examen.lout
```
  - ▶ 

```
$ ls -l /bin/sh  
-r-xr-xr-x 1 root root ... /bin/sh
```

## Droits d'accès Unix

- ▶ les répertoires ont des droits aussi
  - ▶ droit en écriture requis pour pouvoir ajouter/effacer un fichier
- ▶ Le super-utilisateur root (UID 0) a tous les privilèges
  - ▶ peut lire/écrire tous les fichiers, changer leur propriétaire, etc.
  - ▶ nécessaire pour l'administration (ajout d'utilisateurs, sauvegarde, etc.)

## Droits d'accès Unix hors système de fichiers

- ▶ certains périphériques vus comme des fichiers
  - ▶ `$ ls -l /dev/tty1`  
`crw---- 1 root root 4, 1 ... /dev/tty1`
- ▶ autres ressources traitées différemment :
  - ▶ changement du propriétaire d'un fichier
  - ▶ changement de l'utilisateur/groupe du processus courant
  - ▶ montage/démontage de systèmes de fichiers
  - ▶ création de nœuds de périphériques (comme `/dev/tty1`)
  - ▶ ports TCP/UDP < 1024
  - ▶ `reboot`, `halt`, etc.

## Programmes `setuid`

- ▶ certaines actions légitimes demandent davantage de privilège
  - ▶ ex. : comment changer un mot de passe ?
  - ▶ mots de passe stockés dans `/etc/shadow` propriété de root
- ▶ « solution » : programmes `setuid/setgid`
  - ▶ s'exécute avec les privilège du propriétaire/groupe du fichier
  - ▶ chaque processus a un UID/GID *réel* et *effectif*
    - ▶ *réel* est celui qui a lancé le programme
    - ▶ *effectif* est le propriétaire/groupe du fichier — utilisé dans les vérifications de droits d'accès
- ▶ vu comme `s` dans la sortie de `ls`
  - ▶ `ls -l /bin/passwd`  
`-r-s--x--x 1 root root ... /bin/passwd`
  - ▶ `passwd`, `su`, `ping`, `netstat`, etc.

## Problèmes des programmes `setuid`

- ▶ l'utilisateur contrôle l'environnement du programme
  - ▶ peut interférer sur son fonctionnement
  - ▶ changer `$PATH` si le programme utilise `exec1p` & co.
  - ▶ fermer le descripteur de fichier 2 pour pousser le prog. à écrire un message d'erreur dans un fichier protégé
  - ▶ ...

## Problèmes des droits Unix

- ▶ gestion des droits hétérogène
  - ▶ tout n'est pas vu comme un fichier
- ▶ un processus a plus de droits que nécessaire
  - ▶ tous les droits de l'utilisateur
  - ▶ accès à toute l'arborescence, au réseau, etc.
- ▶ noms de ressources symboliques découplés des droits afférents
  - ▶ on peut nommer un fichier même sans y avoir accès
  - ▶ solution : lier désignation et autorité (*capabilities*)
    - ▶ possibilité de désigner = possibilité d'accéder
    - ▶ matrice découpée par ligne
    - ▶ exemple : descripteurs de fichier ; pointeur Java

## Sources \ Merci à eux !

- ▶ cours de David Mazières (Stanford),  
<http://www.scs.stanford.edu/10wi-cs140/>
- ▶ cours d'Arnaud Legrand (IMAG),  
<http://mesca1.imag.fr/membres/arnaud.legrand/teaching/20>
- ▶ illustrations supplémentaires par A. Tanenbaum, E. Skoglund, R. Javaux (Wikipédia), T. Petazzoni & D. Decotigny (Éditions Diamond), L. Courtès, Ron Minnich et al.

## Références

- ▶ *Simple Operating System*, articles dans GNU/Linux Magazine France, Diamond Éditions, T. Petazzoni et D. Decotigny, 2004–2007, <http://sos.enix.org/>
- ▶ *Modern Operating Systems*, Andrew Tanenbaum
- ▶ *Towards a New Strategy of OS Design*, Thomas Bushnell, 1994, <http://www.gnu.org/software/hurd/hurd-paper.html>
- ▶ Manuels des processeurs Intel, <http://www.intel.com/products/processor/manuals/>