

# Principes des systèmes d'exploitations

L3 – Info32B

Année 2015-2016



Nicolas Sabouret – Université Paris-Sud

## Organisation

- 10 séances de cours (lundi 10h45-12h15)
- 10 séances de TD/TP en E204
  - Groupe 1 : lundi 13h30-16h45
  - Groupe 2 : mardi 13h30-16h45
  - M. Gleize et R. Bonaqueà partir de la semaine du 14/09
- Partiel : lundi 19/10 10h30-12h00 (à confirmer)
- Évaluation : examen écrit 60 % + CC 40 %
  - (CC = partiel ½ + note TP ½)
  - 2e session pour l'examen, max avec 1ère
  - CC conservé en 2e session
  - Compensations entre UE
- Langages support : C, Java et bash sous Linux

## Plan du cours

- Introduction, structure d'un S.E (cours 1)
- Processus (cours 2, 3 et 4)
- Mémoire (cours 5 et 6)
  - == Partiel ==
- Systèmes de fichier (cours 7, 8 et 9)
- Entrées-sorties (cours 10)
  - == Examen ==

## Bibliographie

- Support du cours :
  - Silberschatz et al., 2004. *Principes des systèmes d'exploitation avec Java*. Ed. Vuibert (2008)
- Les « bibles » des SE :
  - Tanenbaum, 2001. *Modern Operating Systems*. Ed. Prentice Hall
  - Stallings, 2000. *Operating Systems*. Ed. Prentice Hall
  - Nutt, 2000. *Operating Systems: a modern perspective*. Ed. Addison-Wesley

## Cours 1

### Introduction Structure générale d'un SE

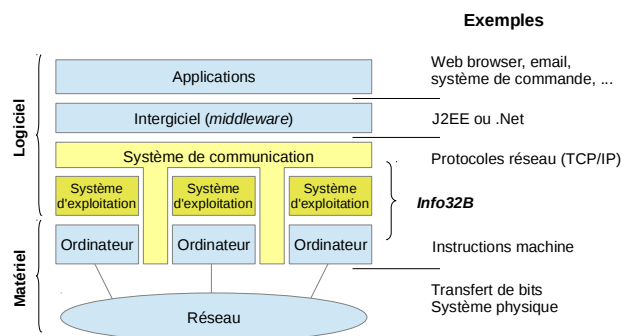
L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Architecture d'un système



Source : S. Krakowiak

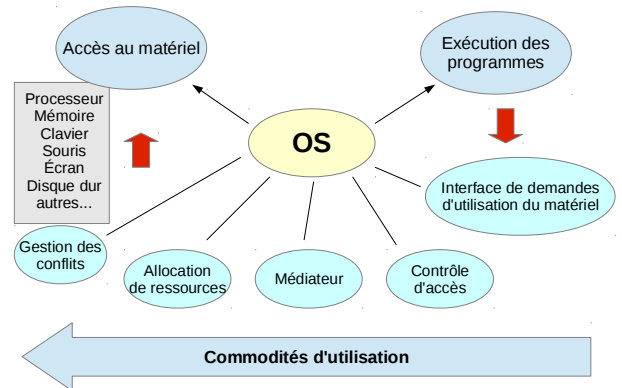
## Définition

Un Système d'Exploitation (ou Operating System) est :

- Un programme (ou ensemble de programmes)
- Qui gère la partie matérielle
- Qui sert de socle pour les applications

→ L'OS est l'intermédiaire entre les applications (l'utilisateur) et le matériel (l'ordinateur)

## Rôle d'un système d'exploitation

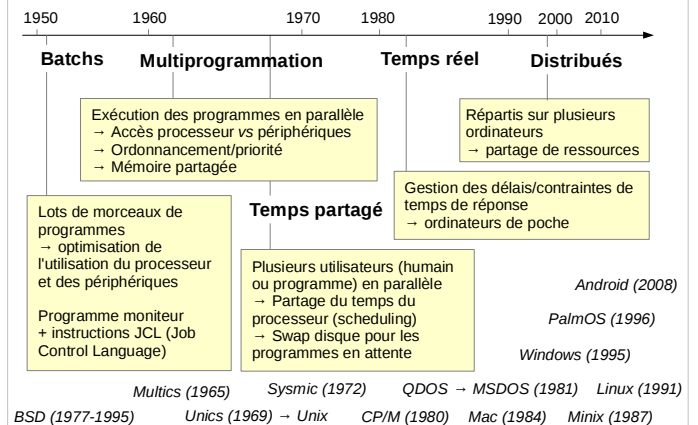


## Plan

### Cours d'introduction

- Définition et rôle de l'OS
- Un peu d'histoire : différents types d'OS
- Fonctions principales
  - Processus
  - Entrées/Sorties
  - Accès mémoire et fichiers
- Structure et fonctionnement d'un SE :
  - Interruptions
  - Échange de données
  - Gestion de la mémoire
  - Interpréteur de commandes
  - Structures en couches et en modules

## Bref historique



## Historique (suite)

### • Batch (traitements par lots), années 50

#### - Un problème d'E/S

Programme = ensemble de fiches représentant les données en entrée et les traitements à effectuer sur ces entrées pour produire des sorties

→ entre chaque fiche, un opérateur (humain) doit intervenir sur l'ordinateur (*diverses opérations de contrôle dont le changement de tâche...*)

#### - Solution

Décrire les opérations de manipulation des fiches (changement de tâche, chargements des données, chargement du programme, exécution, écriture des sorties) dans un programme (fiche de traitement du « lot » (*batch*))

→ les premiers OS étaient nés !

## Historique (suite)

### • Multiprogrammation et temps partagé (années 60)

#### - Un problème d'optimisation :

Plusieurs programmes peuvent utiliser des parties différentes du système en même temps (processeur vs imprimante)

Du coup, l'ordinateur pourrait effectuer plusieurs programmes en parallèle

#### - Solution : le scheduling et la mémoire partagée

L'OS détermine quel programme doit avoir accès à quelle partie de l'ordinateur à quel moment et pour combien de temps

En particulier : le processeur → parallélisme d'exécution

→ Unix était né !

## Historique (suite et fin ?)

- Micro-ordinateurs (années 80)
  - Système CP/M
  - IBM PC → MSDOS
- Interfaces graphiques et réseau (80-90)
  - Xerox → Apple Macintosh
  - Windows 95
  - Linux
- Systèmes embarqués : OS spécifiques
- PC de poche & smartphones (années 2000)

et après ?...

## Fonctions d'un OS

- Utilisation des périphériques
  - interface d'utilisation
- Accès aux fichiers
  - gestion des formats
- Accès aux ressources
  - gestion de l'authentification
- Partage des accès (processeur, ressources)
  - gestion des processus
- Gestion des erreurs
- Contrôle du système (logs)

## Utilisation des périphériques

- Chaque périphérique a une interface spécifique
  - Structure de données en entrée et en sortie
  - Protocole d'interaction et jeu d'instruction
- L'OS a pour rôle de rendre l'utilisation du périphérique transparente
  - i.e. indépendante du constructeur, du jeu d'instruction
    - simple lecture/écriture dans un fichier !
- Les pilotes de périphériques sont des modules de l'OS qui assurent cette transparence
  - et l'OS offre les fonction d'E/S standard...

## Accès aux fichiers

- Chaque support a un format propre (NFS, NTFS, EXT4FS...)
  - l'OS est capable de gérer ces différents formats de fichier
- Les commandes d'E/S de l'OS sont standard (généralement sous forme d'octets)
  - l'OS traduit et dé-traduit les données dans le format de stockage utilisé
- L'OS assure aussi le contrôle d'accès
  - qui a le droit d'accéder à quel fichier, à quelle partie de la mémoire...

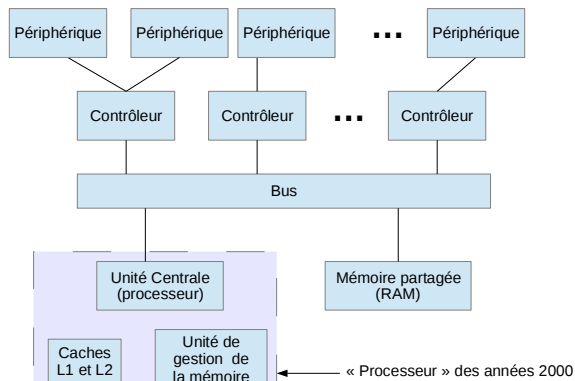
## Partage des ressources

- L'OS assure le partage des accès
  - Lorsque plusieurs programmes veulent utiliser le processeur
  - Lorsque plusieurs programmes veulent écrire sur le disque
  - Lorsqu'un programme utilise en entrée la sortie d'un autre programme
  - Etc.
- L'OS gère aussi les erreurs
  - Timeouts, relances, retour utilisateur
  - Logs systèmes (pour le contrôle)

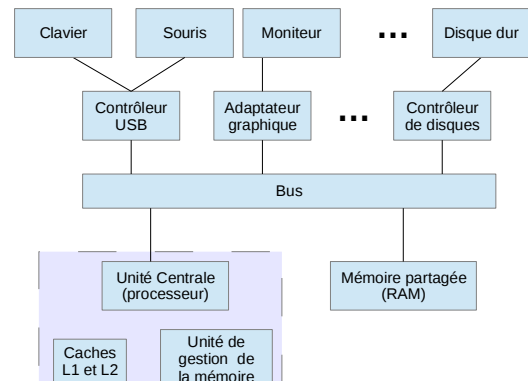
## Plan du cours (suite)

- Cours 2-3-4 : gestion des processus
  - gestion des ressources
- Cours 5-6-7-8 : gestion des données
  - structure de mémoire
  - système de fichiers
- Cours 9-10 : entrées-sorties
  - gestion de l'accès
  - sécurité

## Structure d'un système info.



## Structure d'un système info.



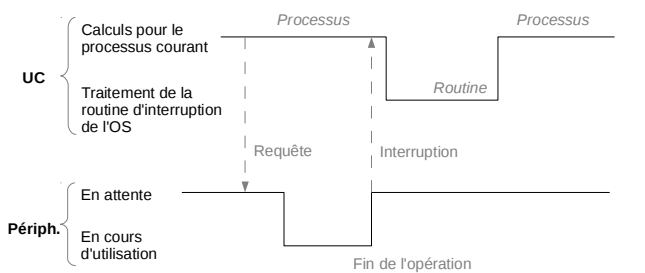
## Fonctionnement (1/2)

- Bus de communication → accès à la mémoire
  - Processeur et contrôleurs
  - contrôleur de mémoire (synchronisation)
- Programme d'amorçage
  - dans une ROM (ou EEPROM)
  - initialisation des contrôleurs, du contenu de la mémoire, du processeur...
  - puis attente d'une **interruption** (matérielle)

## Fonctionnement (2/2)

- Systèmes dirigés par les interruptions
  - à chaque interruption correspond une routine de traitement par l'OS
  - pas d'interruption + pas de processus → l'OS reste inactif
- Interruptions matérielles
  - exemple : fin d'E/S, requête service
- Interruptions logicielles (exceptions)
  - exemples : division par zéro, accès mémoire invalide...

## Interruption de l'UC



Source : A. Silberschatz et al. 2004

## Gestion des interruptions

- Nombre fini d'interruptions possibles
  - table de pointeurs vers des routines d'interruptions stockée en mémoire basse
  - ex : PC sous MSDOS ou Linux → de 00 (1er octet) à F0 (239e octet)
- Sauvegarde de l'adresse de l'instruction interrompue + état courant du processeur (si nécessaire) → pile système
- Mode superviseur : opérations de l'OS différenciées des opérations des processus (ex : archis x86)

## Entrées/Sorties

- Requête : l'UC charge les registres dans le contrôleur
  - instruction + données
  - (ex : lecture → chargement des données dans le tampon)
- Utilisation d'un tampon
  - = mémoire locale au gestionnaire
- L'UC attend la réponse du contrôleur (instr. wait)
  - Synchrone : le processeur reste en attente jusqu'à la fin du traitement par le périphérique
  - Asynchrone : le contrôleur rend la main et utilise une interruption lorsque le périph. a terminé son travail
- File d'attente (périph. occupé) gérée par l'OS
  - table de statut des périphériques

## Direct Memory Access (DMA)

- Périphériques rapides (ex : clavier, disque)
  - Le contrôleur transfère un bloc de données directement vers la mémoire, sans intervention de l'UC
  - une seule interruption pour le bloc (au lieu d'une par octet)
- Pilote de périphérique
  - positionne les registres du contrôleur DMA pour utiliser les adresses sources et destination correspondantes (bloc et mémoire)
- Attention : la mémoire ne peut traiter qu'un transfert à la fois
  - Le DMA « vole » des cycles mémoire à l'UC...
  - l'UC est donc « ralentie » par l'utilisation du DMA

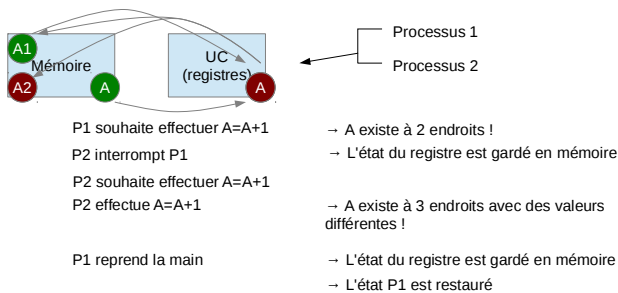
## Gestion des données

- L'OS gère la mémoire :
  - Savoir quelles sont les zones mémoire actuellement utilisées, et par quel processus
  - Décider quel processus doit être chargé en mémoire
  - Allouer et désallouer la mémoire
- L'OS gère les fichiers
  - Création et destruction (fichiers et répertoires)
  - Primitives de manipulation des fichiers
  - Correspondance fichier ↔ mémoire secondaire
  - Enregistrement sur disque

## Problème de consistance

- Structure de stockage hiérarchique
  - Disque → Mémoire → Cache → Registre
  - une même donnée peut être recopiée dans ces différents endroits
- Problème :
  - Si plusieurs processus utilisent la donnée, comment s'assurer que chacun a bien accès à la version la plus récente ?
- algorithmes de consistance

## Problème de consistance



**P1 calcule sur une mauvaise valeur de A !**

## Partage de temps et de mémoire

- N processus devant s'exécuter
  - partager le temps de l'UC et l'accès à la mémoire
- Partage de temps
  - solution à base de minuterie
  - Horloge centrale (fournie par l'ordinateur)
  - Registre de timeout (modifié par l'OS)
  - Interruption vers l'OS lorsque cette durée est atteinte
- Partage de mémoire
  - solution à base de registres d'adressage
  - Chaque processus a son propre espace mémoire
  - Registre de base + registre limite (modifiés par l'OS)
  - Tout accès en dehors de la zone provoque une exception

## Interpréteur de commande

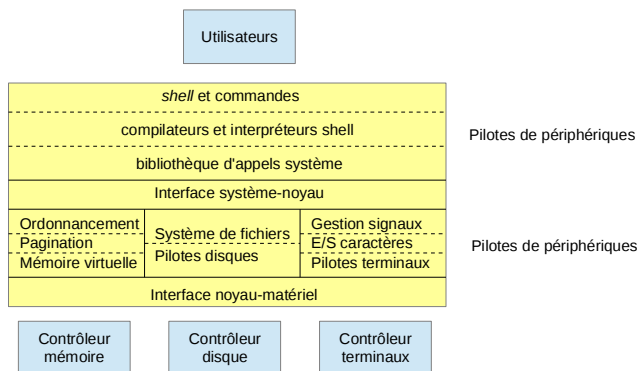
- Programme en charge de l'interface avec l'utilisateur (aussi appelé **shell**)
  - MSDOS/Unix → console + clavier
  - Mac/Windows → souris + clavier
- Rôle : interpréter les opérations de l'utilisateur sous forme de commandes pour l'OS
  - Exemple : `rm chemin` → suppression d'un fichier
- Différent des appels système !
 

Primitives dans un langage de programmation qui sont compilées sous forme d'interruptions pour l'OS  
... qui forment la base des shell

## Appels système (exemples)

- Contrôle de processus
  - Chargement, exécution, création et terminaison de processus
  - *wait*, terminaison (fin ou erreur, tâche ou lot) et autres interruptions
  - Allocation de mémoire
- Gestion de fichiers
  - Création, ouverture, fermeture, destruction
  - Lecture, écriture, positionnement
  - Gestion des attributs
- Gestion des périphériques
  - Réquisition, libération lecture, écriture, attributs
  - Montage
- Gestion de l'information système
  - Date, heure, données système, attributs de processus
- Communications
  - Connexion réseau
  - Envoi et réception de messages

## Structure d'un OS



La structure varie d'un OS à l'autre

## Structure d'un OS


- Approche en couche
  - chaque couche masque la couche suivante
  - Facilite la modularité mais structuration difficile
- Approche par micro-noyau (ex : WindowsNT)
  - noyau réduit aux fonctions primitives (processus, mémoire, communication)
  - pilotes = processus utilisateurs
  - Augmente la sécurité mais ralentit l'exécution → souvent couplé avec une approche par couches (ex : MacOS X)
- Approche par modules (ex : Linux)

# Cours 2

## Processus (1/3)

L3 – Info32B

Année 2014-2015



UNIVERSITÉ PARIS SUD  
Comprendre le monde, construire l'avenir

Nicolas Sabouret – Université Paris-Sud

# Plan

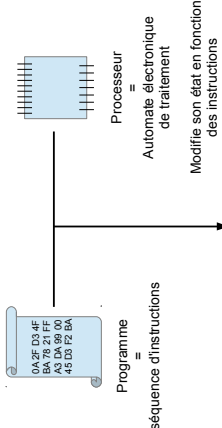
Cours 2

- Définition
- Structure et cycle de vie
- Problème de l'ordonnancement
- Communication entre processus
- Thread : définition
- Threads linux et windows
- Threads Java

Cours 3 : Ordonnancement

Cours 4 : Synchronisation

# Définition



Programme =  
séquence d'instructions

Processeur =  
Automate électronique  
de traitement

Modifie son état en fonction  
des instructions

**Processus = un programme exécuté par un processeur**

# Définition

- Programme ≠ processus
  - Programme = ensemble d'instructions
  - Exécution d'un programme par l'UC = processus  
→ un programme peut générer plusieurs processus
- Exemples de processus :
  - Logiciel de traitement de texte en cours d'utilisation
  - Compilation d'un code source
  - Tâche système (envoi de données vers l'imprimante)

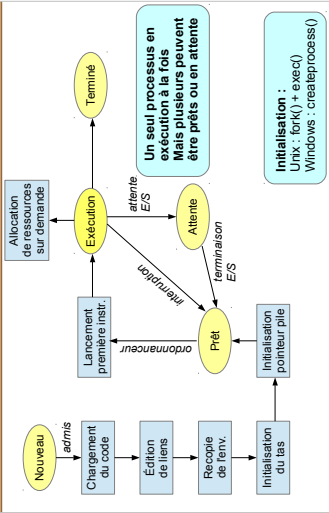
# Concurrence

- Dans un OS moderne, plusieurs processus s'exécutent en parallèle
  - Même lorsque le système est mono-utilisateur  
ex : navigateur + vidéo + traitement de texte
- On distingue les processus utilisateur des processus système (mode superviseur)
  - mais tous sont des processus, en concurrence pour l'accès aux ressources de l'ordinateur
- Rôles de l'OS :
  - Création et suppression des processus
  - Suspension et reprise, gestion de la mémoire
  - Gestion de la communication et de la synchronisation

# Structure d'un proc. en mémoire

- Environnement
  - Variables héritées du parent (ex : PATH=/usr/bin)  
(voir getenv en C ou TD1)
- Tas
  - Données globales statiques + variables allouées dynamiquement  
→ partagé par tous les threads du processus
- Pile
  - Variables temporaires du programme (blocs C)
  - Contexte (ex : compteur d'instruction)
- Code
  - Recopie du programme machine : fixe et protégé en mémoire
  - C'est le segment « exécutable » du processus

# Cycle de vie du processus



# Création d'un processus

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t fils;
    printf("Starting ... \n");
    fils = fork();
    if (fils==1) printf(stderr, "fork failed");
    else if (fils==0) /* c'est le fils */ {
        printf("id: %d \n", getpid());
        sleep(5);
        printf("id: %d \n", getpid());
        exit(5);
    } else /* c'est le pere */ {
        printf("id: pere de %d \n", getpid(), fils);
        int r; waitpid(fils, &r, 0); /* attente du fils */
        printf("id: fils %d sort (code %d)\n",getpid(),fils,r);
    }
    return 0;
}
```

# Recouvrement d'un processus

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

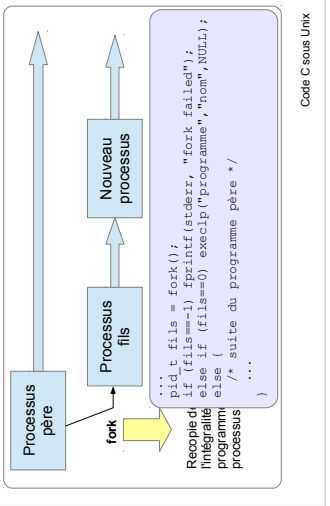
int main() {
    char *argu[3] = {"", "-a", NULL};
    execv("/bin/uname", argu);
    printf("Ce code n'est jamais atteint!\n");
    return 0;
}
```

**Le nouveau processus (ici uname -a) recouvre l'ancien  
→ on ne ressort pas d'un recouvrement !**

Code C sous Unix

Linux blink 3.5.0-21-generic #32-Ubuntu SMP  
Tue Dec 11 18:51:59 UTC 2012 x86\_64 x86\_64 GNU/Linux

# Lancement d'un processus



# Bloc de contrôle (PCB)

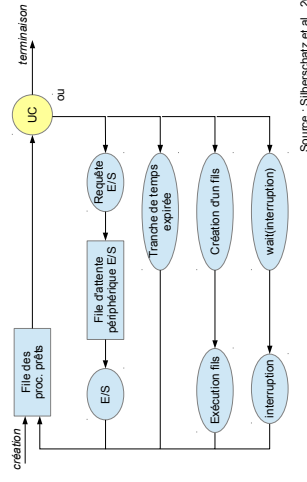
- Représente les informations sur un processus :
- État (prêt, en attente, en exécution)
  - Compteurs d'instruction
  - Registres du processeur
  - Données d'ordonnancement (priorité, pointeur vers la file d'attente, etc)
  - Informations mémoire (ex : registres base et limite)
  - Périphériques d'E/S alloués
  - Usage (ex : total temps processeur)

# Problème d'ordonnancement

- Chaque processus utilise des ressources  
→ problème de partage de ressources
- Ordonnancement = sélection d'un processus parmi tous les processus prêts + en exécution
- Principe général : file d'attente  
File de PCB des processus prêts  
+ chaque périphérique a une file de PCB pour les processus en attente chez lui



## Ordonnancement : principe



Source : Silberschatz et al., 2004

## Communication

L'OS doit assurer l'indépendance des processus, mais...

Deux processus peuvent avoir besoin de s'échanger de l'information

(autrement qu'un code de retour du fils qui termine vers son père)

- Plusieurs formes de « communication »
  - Partage de données en mémoire
  - Lecture/écriture dans un tampon (mémoire ou fichier) partagé
  - Utilisation de fichiers
  - Envoi de signaux
  - Etc.

## Communication

## Choix du modèle !

- Une boîte utilisable par plus de 2 processus ?
- Un seul processus peut faire receive à chaque instant ?
- Sinon, quel processus reçoit le message ?  
(cf TP1 sur les terminaux)
- Envoi bloquant ou non ?  
Jusqu'à ce que le message soit lu (receive)
- Réception bloquante ou non ?  
Receive *null* si non bloquant
- Utilisation d'un tampon ? Capacité bornée ?

## Ordonnancement

- Équilibrer entre les processus E/S et les processus de calcul
  - meilleure utilisation du système
- Swapping :
  - mise en attente d'un processus sur le disque
    - plus long terme (vs ~100 ms pour ordonnanceur)
- Commutation de contexte
  - = changement du processus en exécution
    - Enregistrer le PCB du processus courant
    - Charger le PCB du nouveau processus
    - surcoût (UC non utilisée)

## Communication (suite)

- Communication inter-processus (IPC)  
Mécanisme de haut niveau qui peut faire communiquer des processus distribués sur deux systèmes différents (ex : chat)
- Difficultés
  - Les processus doivent pouvoir se nommer  
send(P.message) – receive(Q.message)
  - Utilisation de boîtes aux lettres  
Adresse unique  
Boîte aux lettres partagée entre les processus  
→ socket

## Exemple : RPC windows XP

- Chaque processus P1 a son propre canal de communication
  - Un port de connexion (cnx1), visible par tous
  - Un port de communication (com1), privé
    - Les ports servent aussi bien à émettre qu'à lire
- Chaque processus P2 qui veut communiquer contacte le port de connexion de P1
  - P2 envoie (via « son » com2) une demande de connexion vers cnx1 et attend la réponse (sur com2)
  - P1 crée 2 ports com2n1 et com12 qui sont reliés : les entrées de com12 sont les sorties de com2n1 et vice-versa
  - P1 envoie com2n1 à P2 via com1, sur com2.
  - P1 lit et écrit sur com12, P2 lit et écrit sur com2n1

### Thread : définition

- Une thread est l'unité de base du processus  
→ un processus peut avoir plusieurs threads qui partagent le même code et les mêmes données, mais qui ont chacun une pile propre

code

données

droits

tas

fichiers

exécution

pile

registres (contexte)

code

données

droits

tas

fichiers

pile

registres

pile

registres

Processus mono-thread

Processus multi-thread

### Thread : exemple

- Serveur web
  - Reçoit des requêtes
  - Traitement (envoyer le code HTML de la page demandée)
  - Traite plusieurs demande en parallèle→ processus multi-threadé
- 1 seul ensemble de données/fichiers
- Création de threads à chaque demande  
Comme des processus fils mais partageant les données.

### Threads : avantages

- Réactivité  
Un processus multi-threadé peut continuer à s'exécuter pendant qu'une de ses threads est bloquée  
Ex : navigateur web → chargement des images
- Partage de ressources  
Ex : le code n'est pas recopié N fois
- Économie de temps  
Allocation mémoire, changement de contexte → opérations coûteuses !
- Parallélisme  
Sur les architectures multiprocesseurs

### Thread et processus

- Création de processus fils (fork)  
→ faut-il dupliquer aussi les threads du père ?
  - Oui : si on veut cloner le processus
  - Non : si on veut simplement lancer un nouveau sous-processus par exec après
- Traitement des signaux  
→ faut-il envoyer le signal à toutes les threads du processus ?
  - Oui : si c'est un signal global (ex : SIGKILL)
  - Non : si c'est un signal spécifique à certaines threads (ex : op. d'E/S)
- Données spécifiques au thread  
→ tout sur la pile !

### En TD


- Semaine 3  
→ processus et threads en C
- Semaine 4 et 5  
→ threads en Java  
→ synchronisation

# Cours 3

## Processus (2/3)

L3 – Info32B

Année 2014-2015



UNIVERSITÉ PARIS SUD  
Comprendre le monde, construire l'avenir

Nicolas Sabouret – Université Paris-Sud

# Plan

Cours 2 : Processus et threads

Cours 3 :

- Objectifs de l'ordonnancement
- Algorithmes d'ordonnancement
  - Méthode FIFO
  - Méthode « plus court d'abord »
  - Notion de priorité
  - Algorithme round-robin
- Ordonnancement de threads
- Exemples d'OS

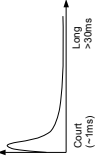
Cours 4 : Synchronisation

# Problème (rappel)

- Flot continu : utiliser l'UC pendant qu'un processus est en attente d'E/S
  - répartir les processus sur les ressources
- Plusieurs processus peuvent vouloir les mêmes ressources
  - problème de partage de ressources
- Ordonnancement = sélection d'un processus parmi tous les processus prêts + en exécution
- Principe général : file d'attente de PCB
- Équilibrer entre les processus E/S et les processus de calcul
- Coût de la commutation de contexte

# Objectifs

- Durée des cycles UC (mono-processus)
  - les E/S causent beaucoup d'interruption et des cycles UC très courts !
- La sélection (ordonnancement) doit être :
  - Équitable
    - Tous les processus ont accès aux ressources dont ils ont besoin, en fonction de leurs priorités
  - Efficace
    - Bonne répartition des processus sur les ressources, par exemple entre E/S et calcul
  - Réactive
    - Un processus ne doit pas attendre trop longtemps



# Systèmes préemptifs

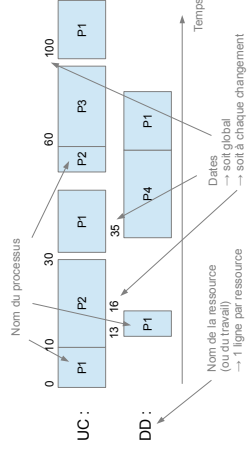
- Ordonnancement possible :
  - Lorsqu'un processus passe en attente (ex : E/S) ou se termine
    - ordonnancement non-préemptif (ou coopératif) le processus garde l'UC tant qu'il en a besoin
  - Lorsqu'un processus passe dans l'état « prêt »
    - ordonnancement préemptif le processeur est partagé entre plusieurs processus actifs qui en ont besoin simultanément
- OS modernes : Windows≥95, MacOS≥X, Linux
  - inconvénient : **coût de coordination entre les processus !**

# Critères d'évaluation

- Taux d'utilisation de l'UC
  - = proportion du temps pendant lequel l'UC est utilisée et fait des calculs
- Débit
  - = nombre (moyen) de processus terminés par unité de temps
- Temps d'attente
  - = durée passée dans la file « prêt » (donc sans compter le temps perdu en E/S)
- Rotation
  - = durée réelle d'un processus (date fin – date début)
  - Variante : temps de réponse = temps temps de calcul (hors E/S) pour traiter une requête
  - Critère **min-max** (minimiser le tps de rép. du proc. le + lent)

## Diagramme de Gantt

- Représentation de l'exécution des processus



## Algorithme « plus court d'abord »

- Choisir le processus qui a la durée la plus courte  
Dans l'exemple précédent, quel que soit l'ordre d'arrivée de P1, P2 et P3, on aura :
- Avantage  
Pour un ensemble de processus donnés, temps d'attente moyen optimal (preuve par récurrence)
- Inconvénients
  - On ne peut pas toujours prédire la durée du prochain cycle d'utilisation de l'UC par un processus
  - Famine pour les gros processus si de nouveaux petits s'insèrent dans la file

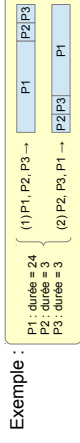


## Algorithme « plus court d'abord »

- Exemple
    - P1 : arrivé à 0, temps estimé 8
    - P2 : arrivé à 1, temps estimé 4
    - P3 : arrivé à 2, temps estimé 9
    - P4 : arrivé à 3, temps estimé 5
- Diagramme de Gantt montrant l'exécution de processus P1, P2, P3, P4 sur une ressource UC. Les axes sont le temps (0 à 10) et le nom du processus. Les dates de début et de fin sont indiquées pour chaque processus.
- | Processus | UC     |
|-----------|--------|
| P1        | 0 - 8  |
| P2        | 1 - 5  |
| P4        | 3 - 8  |
| P3        | 2 - 11 |
- Au temps 0, il n'y a que P1 → P1 sélectionné  
Au temps 1, P2 arrive. Reste 7 pour P1 et 4 pour P2 → P2 préempté  
Au temps 2, P3 arrive. Reste 9 pour P3 et 3 pour P2 → P2 reste  
Au temps 3, P4 arrive. Reste 5 pour P4 et 2 pour P2 → P2 reste  
Au temps 5, P2 termine. Reste P1(7), P3(9) et P4(5) → P4 élu  
Au temps 10, reste P1(7), P3(9) → P1 élu

## Algorithme simple : FIFO

- Premier arrivé = premier servi
  - Code simple à écrire et principe facile à comprendre
  - Pénalise les processus courts



Si on suppose que les processus étaient tous arrivés dans la file avant le début du premier d'entre eux :

- (1) temps d'attente moyen =  $(0+24+27)/3 = 17$
- (2) temps d'attente moyen =  $(0+3+6)/3 = 3$

+ effet d'accumulation si le système est chargé !

## Algorithme « plus court d'abord »

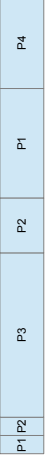
- Estimation du temps UC (pour un processus)  
moyenne exponentielle des cycles précédents :  
$$d_{n+1} = \alpha d_n + (1 - \alpha) t_n$$
  
 $d_n$  = durée estimée à la  $n^{\text{ème}}$  utilisation de l'UC par le proc.  
 $t_n$  = durée réelle (mesurée) de la  $n^{\text{ème}}$  utilisation de l'UC  
 $\alpha$  = facteur de pondération (généralement  $1/2$ )

- Ordonnancement **préemptif** au **temps restant** le plus court

Lorsqu'un nouveau processus arrive dans la file, on compare son temps restant à celui du processus en cours d'exécution

## Notion de priorité

- Munir chaque processus d'une priorité pour sélectionner plus souvent les processus les plus urgents
- Algorithme FIFO avec des files séparées par priorité
- Exemple (en mode préemptif) :
  - P1 : arrivé à 0, temps estimé 8, priorité 0
  - P2 : arrivé à 1, temps estimé 4, priorité 1
  - P3 : arrivé à 2, temps estimé 9, priorité 2
  - P4 : arrivé à 3, temps estimé 5, priorité -1



## Notion de priorité

- Remarque

L'algorithme « plus court d'abord » est un algorithme avec priorité dans lequel la priorité est l'inverse de la durée prédite pour le prochain cycle

- Dans un système réel, la priorité est définie :

- Par l'OS (interne)

Type du processus, Limite de temps, besoins en mémoire, nombre de fichiers ouverts...

- Par l'utilisateur (externe)

En fonction de l'urgence du travail

## Notion de priorité

- Avantages

- Algorithme simple à comprendre
- Souplesse (selon calcul priorité interne et externe)

- Inconvénients

- Famine pour les processus peu prioritaires
- Choix du calcul interne (OS) de la priorité

- Méthode de vieillissement

Pour éviter les famines, on fait baisser la priorité des processus « longs ». Tous les N unités de temps, la priorité d'un processus descend de 1

## Round robin (ou tourniquet)

- Limiter arbitrairement la durée d'un cycle d'utilisation de l'UC par un processus

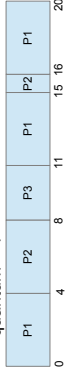
→ quantum de temps

- Algorithme FIFO non préemptif (mais RR est préemptif...)

- Exemple (sans priorité)

- P1 : arrivé à 0, durée 12
- P2 : arrivé à 1, durée 5
- P3 : arrivé à 1 (après P2), durée 3

quantum = 4



## Round robin

- Avantage

- Algorithme équitable et relativement simple

- Inconvénients

- Augmente les commutations sur l'exemple précédent, on commute alors qu'il ne reste qu'une unité de temps
- Temps d'attente moyen long

sur l'exemple précédent, 8,33 à comparer avec 3,66 pour un algorithme « plus court préemptif »

- Utilisé en pratique dans les OS modernes

avec un quantum → 2 à 10 fois le temps de commutation de contexte

→ empiriquement, RoundRobin est efficace si ~80 % des cycles sont inférieurs au quantum

## Ordonnancement de threads

- Comme les processus, les threads pose un problème d'ordonnancement

- Les threads POSIX (vues au TD2 en C) peuvent être ordonnancées en FIFO ou en RR

```
pthread_attr_t attr;
pthread_attr_t attr;
pthread_create(&id, &attr, *fonction, NULL);
```

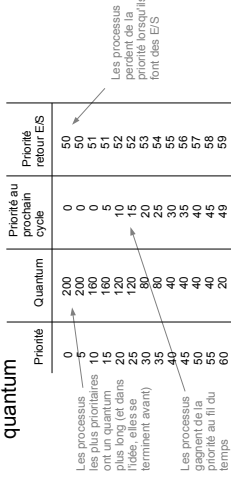
- Les threads Java (prochain TD) utilisent des priorités

```
Thread.currentThread().
setPriority(Thread.NORM_PRIORITY+10);
```

## Exemple : solaris

- Solaris est un OS distribué sur les machines Sun entre 1993 et les années 2000

- Ordonnancement basé sur les priorités avec quantum



## Exemple : Windows (XP, Vista...)

### Multi-level feedback queue

- Priorité au niveau des threads des processus
- Ordonnancement préemptif avec priorités
  - Tout thread de priorité supérieure prend la main sur les threads de priorité inférieure (files différentes)
  - 32 niveaux de priorité
- Au sein d'une même priorité: RoundRobin
- La priorité est baissée à la fin du quantum  
sauf pour les threads a priorité fixe, comme les threads temps réel
- La priorité est montée à la fin d'une E/S  
→ meilleur temps de réponse interface graphique

## Exemple : Linux

### Deux algorithmes

- Temps partagé équilibrable → système de crédits  
Chaque processus dispose d'un crédit (= priorité).
  - Le plus élevé l'emporte (préemptif)
  - Le processus exécuté perd un crédit à la fin du quantum
  - Si aucun des processus « prêts » n'a de crédit, tous les processus sont réexécutés (y compris ceux « en attente »):  
crédit = creditiz + priorité (= credit initia)  
→ combine historique et priorité  
→ favorise les processus d'E/S (ex : interface utilisateur)
- Ordonnancement temps réel (pour les tâches temps réel)
  - Préemptif avec priorités (sans changement de priorité)
  - Au sein d'une même file de priorité, sélection du plus « ancien » (sans quantum → FIFO, ou avec → RR)



<h2>Section critique</h2> <p>Propriétés</p> <ul style="list-style-type: none"> <li>- Exclusion mutuelle <ul style="list-style-type: none"> <li>Si T effectue sa section critique, alors aucune autre thread ne peut exécuter sa section critique</li> </ul> </li> <li>- Déroulement <ul style="list-style-type: none"> <li>Une thread qui souhaite rentrer en section critique ne peut pas décider qui rentre en section critique /i.e. seules les threads qui ne vont pas entrer en section critique peuvent participer au choix</li> </ul> </li> <li>- Attente bornée (ou vivacité) <ul style="list-style-type: none"> <li>Une thread qui souhaite rentrer en section critique y rentre en un temps borné (en terme de nombre de fois où elle s'est fait prendre la place par une autre demandé)</li> </ul> </li> </ul>	

<h2>Objectif</h2> <ul style="list-style-type: none"> <li>• Écrire du code pour gérer les sections critiques <ul style="list-style-type: none"> <li>- Solution à 2 tâches : MUTEX</li> <li>- Sémaphores</li> <li>- Moniteurs</li> </ul> </li> <li>• Algorithmes génériques écrits en Java</li> <li>• Application sur des exemples concrets en TD</li> </ul>	

<h2>Mutex</h2> <ul style="list-style-type: none"> <li>• Solution pour deux threads</li> <li>• Un contrôleur central qui autorise les sections critiques chacun à son tour, mets en attente le reste du temps</li> </ul>	<pre> public interface Mutex {     /* id prend ses valeurs dans {0,1} */     public abstract void commencerSectionCritique(int id);     public abstract void finirSectionCritique(int id); }  Interface Java  public class MaClasse extends Thread {     MaClasse(..., Mutex mutex, int id) { ... }     public void run() {         ... section critique ...         mutex.commencerSectionCritique(id);         ... section critique ...         mutex.finirSectionCritique(id);         ... section non critique ...     } }  Code thread mutex et id passés au constructeur, avec la dans {0,1} 2 instances max ! </pre>

<h2>Mutex : algorithme simple</h2> <ul style="list-style-type: none"> <li>• Mettre en attente la thread demandeuse si l'autre a déjà pris la section critique</li> </ul>	<pre> public class Mutex1 implements Mutex {     boolean estEnSC[] = new boolean[] {false, false};      public void commencerSectionCritique(int id) {         estEnSC[id] = true;         while (estEnSC[1-id])             ; /* ou Thread.sleep(1); */     }      public void finirSectionCritique(int id) {         estEnSC[id] = false;     } } </pre> <p>Problème : risque d'interblocage (selon coupure scheduler)</p>

<h2>Mutex : algorithme</h2> <ul style="list-style-type: none"> <li>• Mettre en attente la thread demandeuse si l'autre a déjà pris la section critique</li> <li>+ gérer les tours de priorités</li> </ul>	<pre> public class Mutex2 implements Mutex {     boolean estEnSC[] = new boolean[] {false, false};     int tour = 0;      public void commencerSectionCritique(int id) {         while (estEnSC[1-id])             ; /* ou Thread.sleep(1); */     }      public void finirSectionCritique(int id) {         estEnSC[id] = false;     } } </pre> <p>Si 2 threads entrent en SC, selon le point d'interruption, l'une des deux passera son tour mais récupérera la main après quoi arrivera (même si l'autre demande une SC).</p>

<h2>Algorithme de Dekker</h2> <ul style="list-style-type: none"> <li>• Généralisation du Mutex à N threads</li> </ul>	<pre> public class MutexN implements Mutex {     private final int[] Etat = {STAT_RIEN, ETAT_DEMANDE, ETAT_TIENT};     int etat[] = new int[N] {RIEN,...,RIEN};     int tour = 0;      public void commencerSectionCritique(int id) {         int n;         do {             etat[id] = DEMANDE;             while (tour != id)                 ; /* ou Thread.sleep(1); */             tout = id;             etat[tout] = TIENT;             n = 0;             while ((n &lt; N) &amp;&amp; (n == id)    (etat[n] != TIENT))                 n++;         } while (n &lt; N);          public void commencerSectionCritique(int id) {             etat[id] = RIEN;         }     } } </pre> <p>Attendez son tour</p> <p>Aucun autre thread en SC</p> <p>Attendez la fin de la section critique et faites vos demandes des autres</p>



## Limites

- Avantages
  - Pas de famine
- Inconvénients
  - Il faut connaître à l'avance le nombre de processus (N) + les identifier (id)
  - Les processus font de l'attente active: boucle while sans rien

→ vers les sémaphores

## Sémaphores

Dijkstra, 1960

- Généraliser le principe du Mutex à plusieurs threads (ou processus), nombre inconnu
  - Définir un objet partagé (le *sémaphore*) qui met en attente ceux qui le demandent lorsque quelqu'un l'a déjà acquis et rend la main dans l'ordre des demandes.
- Utilisation :

Toutes les threads en concurrence sur des données en section critique partagent un sémaphore pour gérer cette section critique

Utilisation d'un sémaphore

```
public class MaClasse extends Thread {
    MaClasse(..., Semaphore S) { ... }
    public void run() {
        ...
        S.acquire(this);
        ... section critique (S) ...
        S.release(this);
        ... section non critique (S) ...
    }
}
```

## Sémaphores (suite)

- Principe général :

```
public class Semaphore {
    int S = 0;
    public void acquier() {
        while (S>0);
        S++;
    }
    public void relacher() { S--; }
}
```

**Sémaphore** (principe)
- Trois problèmes
  - Méthodes **acquier** et **relacher** doivent être atomiques !
  - Relâcher les processus dans l'ordre (vivacité)
  - Attente active: boucle while sans rien
    - le sleep réduit la charge mais c'est un artifice !
- Solutions
  - file d'attente
  - 1) et 3) → appels systèmes fournis par l'OS

## Sémaphore : code (abstrait)

```
public class Semaphore {
    int S = 0;
    LinkedList <Thread> file = new LinkedList<Thread>();

    public void acquier() { /* entrer SC */
        disableInterrupt();
        while (S>0)
            block(Thread.currentThread());
        S++;
        enableInterrupt();
    }

    public void relacher() { /* finir SC */
        disableInterrupt();
        S--;
        if (S>0) {
            Thread t = file.poll();
            wakeup(t);
        }
        enableInterrupt();
    }
}
```

## Interblocage & sémaphores

- L'utilisation des sémaphores peut introduire des interblocages lorsque deux processus utilisent les mêmes deux sémaphores
- Exemple

```
public class Proc1 ... {
    Semaphore S1, S2;
    ...
    S1.acquire();
    S2.acquire();
    ...
    S1.release();
    S2.release();
}

public class Proc2 ... {
    Semaphore S1, S2;
    ...
    S2.acquire();
    S1.acquire();
    ...
    S2.release();
    S1.release();
}
```

## Sémaphores : conclusion

Reste 2 problèmes :

- Risque d'interblocage lorsqu'on utilise plusieurs sémaphores
- Nécessite des appels systèmes (OS) (block, wakeup, disableInterrupt et enableInterrupt)

Ces problèmes ne peuvent jamais être complètement résolus...

→ il faut programmer proprement !

Moniteurs

Hoare, 1974

- Définition d'objets partagés et synchronisés au niveau du langage de programmation
  - Notion issue de la programmation par objet
  - Première implémentation : Concurrent Pascal, Hansen, 1975
- Principe
  - À tout instant, une seule thread peut exécuter une méthode d'un moniteur
  - Le moniteur prévient les threads appelantes quand il a été libéré

Moniteurs : exemple

- Pseudo-code Java :

```
monitor class CompteurQuatre {
    int valeur;
    public void provisionner(int s) {
        valeur = valeur + s;
    }
    public boolean retrait(int s) {
        if (valeur>s) {
            valeur = valeur - s;
            return true;
        }
        return false;
    }
}
```

Dès qu'une thread invoque l'une des deux méthodes *provisionner* ou *retrait*, plus aucune autre thread ne peut utiliser l'objet, jusqu'à ce que la thread est terminée la méthode

→ Le compilateur peut implémenter ce moniteur à l'aide d'un sémaphore

Moniteurs : principe

- Comme pour les sémaphores, la mise en attente et le réveil des threads est géré au niveau de l'OS
- On associe les moniteurs à un ensemble de *variables conditions* :  
chaque thread indique qu'elle veut accéder au moniteur lorsqu'une condition (sur l'état du moniteur) est vérifiée
- La solution dans le moniteur doit garantir la vivacité dans la gestion des threads de la file (mais pas forcément en mode FIFO)

Moniteurs : utilisation

- Un moniteur est défini par :
  - Une liste de variables conditions
  - Une file de thread en entrée (qui ont demandé le moniteur)
  - Une file de thread en signal (pour qui la condition est vérifiée)
  - Pour chaque variable condition, une file de thread en attente de cette condition (pour les prévenir)
- Utilisation  
Le programme déclare ses conditions :

```
condition x, y, z
```

Avant d'entrer en section critique, il attend une conditions :

```
x.wait
```

La thread est bloquée jusqu'à ce qu'une autre thread invoque :

```
x.signal
```

via le moniteur → cette 2e thread quitte le moniteur

Synchronisation en Java

- Généralisation des moniteurs  
→ chaque objet a un verrou qui lui est associé  
(i.e. chaque objet est une condition de moniteur)
- Mot clef synchronized  
→ définit une condition sur une portion de code

```
class CompteurQuatre {
    public synchronized void provisionner(int s) {
        ...
    }
    public synchronized boolean retrait(int s) {
        ...
    }
}
```

Ne résout pas le problème de risque d'interblocage !

Synchronisation Java (suite)

- On peut déclarer une portion de code synchronisée sur un objet

```
public void maMethode() {
    ... section non critique ...
    synchronized (monObjet) {
        ... section critique ...
    }
}
```

N'importe quel objet peut servir de verrou
- Attention !
  - Une thread qui possède un verrou peut rentrer dans n'importe quelle méthode (verrou recursif)
  - Une thread peut verrouiller plusieurs objets (risque d'interblocage)
  - Tout bloc non synchronized peut être appelé par n'importe qui n'importe quand
  - Méthodes **wait** et **notify** de la classe **Object** appelables dans un bloc synchronized (uniquement)

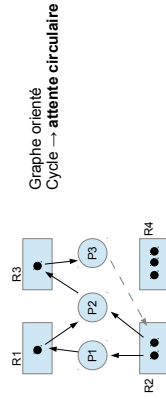
## Interblocage

- Une ressource (variable ou matériel) au moins est protégée par un mécanisme de synchronisation
- Un processus possède une ressource et est en attente d'autres qui sont détenues par d'autres processus
- Ces ressources ne peuvent pas être préemptées : seul leur possesseur peut les libérer
- L'attente est circulaire : il existe un sous ensemble  $P_1, \dots, P_N$  tel que chaque  $P_i$  attend une ressource détenue par  $P_{i+1}$

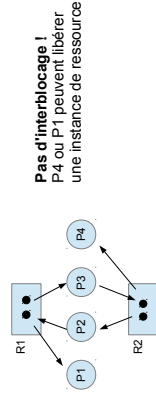
Il y a risque d'interblocage seulement si toutes ces conditions sont réunies

## Graphe d'allocation

- Ensemble de processus P
- Ensemble de ressources R/n (arête)
- Arêtes  $R \rightarrow P$  : affectation
- Arêtes  $P \rightarrow R$  : demandes



## Graphe d'allocation (2)



Attention : interblocage possible ne signifie pas qu'il y aura interblocage ! Cela dépend de l'ordre des appels dans les sémaphores

## Gestion des interblocages

- Ignorer
  - Majorité des OS (dont Unix et Windows) et des langages (dont Java)
  - C'est au programmeur de bien gérer son code (voir Thread.suspend et Thread.resume en Java)
- Détection et réparation
  - Autoriser l'interblocage mais offrir des mécanismes de résolution (termination de processus ou préemption)
- Prévenir
  - Définir des protocoles qui assurent qu'on n'aura pas d'interblocage (il suffit d'invalider l'une des 4 conditions nécessaires)
  - Utiliser des méthodes de preuve de programme

## Cours 5

### Gestion de la mémoire

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Plan

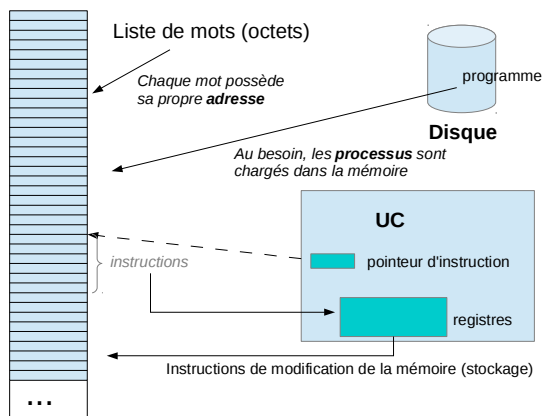
### Cours 5

- Fonctionnement de la mémoire
- Méthodes d'allocation
- Pagination
- Politique de remplacement

### Cours 6

- Segmentation
- Mémoire virtuelle
  - Gestion des pages
  - Utilisation des fichiers

## Fonctionnement de la mémoire



## Liaison d'adresses

- Programme compilé
  - adresses symboliques
- Processus en mémoire
  - adresses réelles dans la mémoire (adresses « logiques »)
  - à partir du début de la zone allouée au processus
- édition de lien
  - relier adresses symboliques et adresses mémoire
- 3 possibilités
  - Compilation : si on connaît la zone mémoire qui sera utilisée (ex : processus OS)
  - Chargement : code compilé translatable
  - Exécution : processus déplacé en cours d'exécution (par exemple parce qu'il a besoin de plus de mémoire) → réédition de lien

Registre de translation dans l'UC

## Chargement dynamique

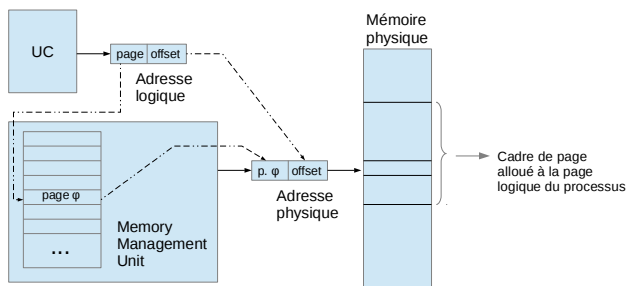
- Chargement du processus en mémoire
  - Charger tout le code, données, bibliothèques...
  - Problèmes : taille mémoire du processus + multi-processus → répétition (inutile) de code (bibliothèques)
- Solutions :
  - Charger les fonctions (routines) « à la demande »
    - Routines conservées au format translatables
    - Chargement + édition de lien (adresses) des routines lorsqu'elles sont appelées
  - Édition de liens dynamique
    - But : partager les bibliothèques
    - Bibliothèque vue comme une routine, chargée en mémoire sans répétition
    - Utilisation de **stub** (code remplaçable) dans le processus pour l'appel de la bibliothèque
      - Édition de lien à l'exécution

## Allocation mémoire

- Objectif : maintenir plusieurs processus en mémoire simultanément
  - réduit le temps dû au recouvrement ou à la permutation de processus
- Méthodes
  - Partitions fixes, swapping
  - Pagination
  - Segmentation
- Contourner la limite de taille :
  - Mémoire virtuelle et pagination à la demande



## Pagination (suite)

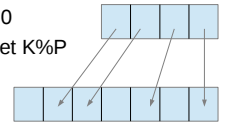


Taille des pages définies par le matériel

Source : Silberschatz et al., 2004

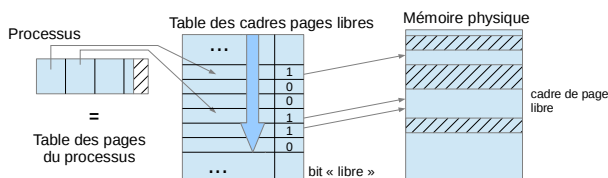
## Cadre de page

- Mémoire physique
  - N cadres de page (de taille P)
- Processus
  - Mémoire de M pages logiques (de taille P)
  - Table de pages associant page logique et cadres de pages (pour chaque processus, maintenu par l'OS)
- Adresse logique
  - Adresse 0 = page logique 0, octet 0
  - Adresse K = page logique K/P, octet K%P
  - Table de page :
    - p. logique 0 → p. physique X
    - p. logique K/P → p. physique Y



## Difficultés (1/2)

- Allocation
  - Liste de cadres de pages libres (au niveau MMU)
  - Donner X pages logiques au processus
    - Méthode : premier cadre de page libre
      - parcours de la table des pages
    - Structure contiguë pour la table des pages
      - mais adresses physiques possiblement entrelacées (selon algorithme de parcours de la table des cadres libres)

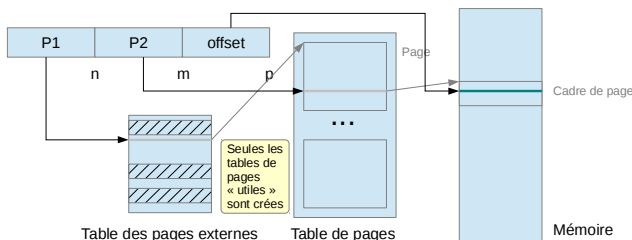


## Difficultés (2/2)

- Taille des pages : commutation/fragmentation
  - Pages plus petites → moins de fragmentation (moy =  $\sim \frac{1}{2}$  p.)
  - Pages plus petites → plus de pages
    - Allocation/commutation/compactage plus coûteux
    - Structure d'adressage plus grosse !
- Solutions
  - Pagination hiérarchique
  - Table des pages inversée
- Partage de code
  - Systèmes multi-utilisateurs (utilisation une même application)
    - Plusieurs pages logiques pour une même page physique

## Pagination hiérarchique

- Gestion d'un grand espace d'adressage
  - paginer la table des pages
    - Table de pages de niveau 1 (N grandes pages)
    - N tables de niveau 2 (petites pages)
  - structure d'adresse à deux niveaux (ex : PIII)

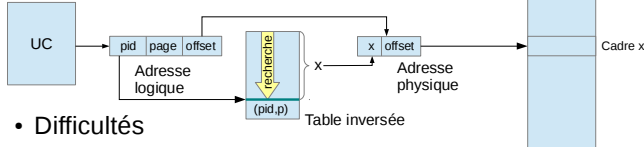


## Exemple

- Processeur 32 bits
  - $2^{32}$  adresses physiques possibles (4Go)
- Pagination simple, pages 4 Ko ( $2^{12}$  : p=12)
  - Table des pages  $2^{20}$  entrées (1Mo)
  - 1 entrée = 32 bits = 4 octets
    - 4 Mo par processus rien que pour l'espace d'adressage !
- Pagination double niveau, n=10, m=10
  - K+1 table externe de  $2^{10}$  entrées (4Ko) / processus avec  $K \ll 2^{10}$
- Proc 64bits : pagination à 3 niveaux !

## Tables des pages inversées

- 1 processus = 1 table de pages  
→ coûteux (commutation, mémoire utilisée...)
- Une unique table de pages
  - Indexée par les cadres de pages physique
  - page  $\phi$  → (processus, page logique)



- Difficultés
  - Translation plus coûteuse
  - Partage de page plus difficile au niveau OS
- Utilisé par les archis 64 bits début 2000

## Implémentation

- Peu de pages → registres matériels dédiés
- En général

- Table des pages en mémoire (RAM)
- Registre de base de la table des pages (PTBR)

Problème :

2 accès mémoires pour chaque accès (table des pages puis accès réel)

→ Utilisation d'un cache matériel :

**TLB** (translation lookaside buffer)  
= stocke des couples (page logique, page  $\phi$ )

- Optimisation

Chaque processus n'utilise pas toutes ses pages à la fois !

→ avec un buffer suffisant, la multiprogramation peut être très efficace

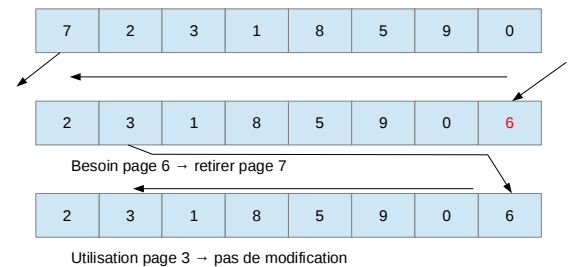
## Remplacement

- Problème de « cache »
  - Gestion du TLB, ou de n'importe quel cache
  - Gestion de la mémoire virtuelle (cf. cours 6)
- N processus, K pages par processus, espace de stockage de taille  $P < N * K$ 
  - Lorsqu'une donnée manque, il faut la charger dans le cache → quelle donnée retirer ?
  - Optimiser l'utilisation du cache
- Politiques de remplacement
  - FIFO
  - LFU/NRU/LRU

## First-in, first-out (FIFO)

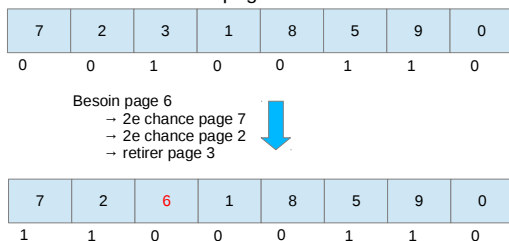
- Principe : retirer la page la plus « ancienne » de la liste

Lorsqu'une page est utilisée, elle **n'est pas** remise en fin de file



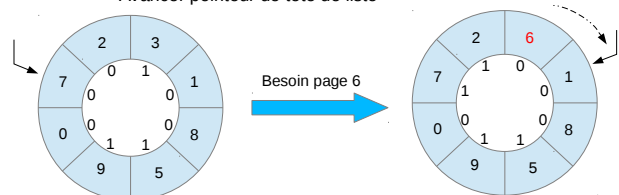
## Seconde chance

- Principe : FIFO + bit de remplacement
  - Lorsqu'une page doit sortir, on vérifie d'abord son bit de remplacement
    - S'il n'est pas positionné, on le positionne
    - On remet à 0 si la page est utilisée



## Clock-based

- Principe : implémentation de l'algorithme de seconde chance avec une file circulaire
  - Pointeur de tête (horloge)
  - Lorsqu'une page utilise sa chance ou lorsqu'elle est utilisée, on la met à la fin
  - Avancer pointeur de tête de liste



## Least Frequently Used (LFU)

- Principe : compter les utilisations et jeter la moins utilisée
  - coûteux à gérer (compter les pages à chaque appel...)

7	2	3	1	8	5	9	0
4	2	8	6	12	1	4	5

Besoin page 6



7	2	3	1	8	6	9	0
4	2	8	6	12	0	4	5

## Not Recently Used (NRU)

- Principe : LFU sans « compter » : juste noter l'utilisation des pages
  - 2 bits par page : référencée (R) + modifiée (M)
  - R remis à 0 à intervalle régulier
  - Priorité : RM > R > M > .

7	2	3	1	8	5	9	0
(0,0)	(0,0)	(0,1)	(0,0)	(0,0)	(0,0)	(0,1)	(0,0)

Utilisation de la page 7 en lecture → (1,0)

Utilisation de la page 3 en écriture → (1,1)

etc...

Horloge → cycle de nettoyage des bits R

Utilisation de la page 7 en lecture → (1,0)

etc...

Remplacement → 2<8<7<0<3<5<9<1

## Least recently used (LRU)

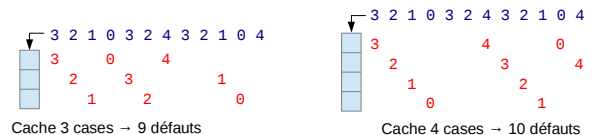
- LFU et NRU sont très coûteux en temps, pour un gain faible par rapport à FIFO
- Principe : mémoriser ce qui est utilisé et quand
  - Matrice triangulaire NxN
  - Quand la i<sup>è</sup> case est utilisée, la ligne i est mise à 1 et la colonne à 0
  - Remplacer la case dont la ligne est entièrement égale à 0 et la colonne remplie de 1, sauf au 0 (c'est la LRU)
  - Très facile à implanter en hardware (and/or)

0 0 0 0	0 1 1 1	0 0 1 1	0 0 1 0	0 0 0 0	0 1 1 1
0 0 0 0	0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 1
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
Ici, LRU=0	Ici, LRU=1	Ici, LRU=2	Ici, LRU=2	Ici, LRU=0	Ici, LRU=1

## Évaluation

- Nombre de « défauts de cache »
  - = la page cherchée ne figure pas dans le cache, il faut donc appliquer la politique de remplacement
- FIFO → 15 à 20 % plus de défauts que LRU
- Anomalie de Belady
  - avec FIFO, plus de défauts sur un cache à N+1 cases que sur un cache à N cases

Exemple : séquence 3 2 1 0 3 2 4 3 2 1 0 4, FIFO



## Questions avant le partiel ?

### Programme :

- Principe général d'un OS, structure, interruptions
- Processus (définition, ordonnancement, synchronisation)
- Mémoire (allocation, pagination, remplacement)



## Cours 6

# Gestion de la mémoire (2/2)

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Plan

### Cours 5

- Fonctionnement de la mémoire
- Méthodes d'allocation
- Pagination
- Politique de remplacement

### Cours 6

- Segmentation
  - Gestion des pages
  - Allocation

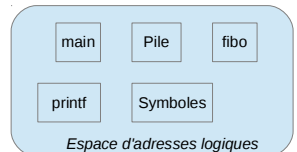
## Rappels

- Maintenir plusieurs processus en mémoire
  - Réduction des coûts
  - Représentation de la structure mémoire  
ex : listes chaînées
- Méthodes
  - Allocation contigüe
    - partitions fixes, swapping
  - Pagination
    - table des pages
- Mesure : taux de fragmentation

## Segmentation

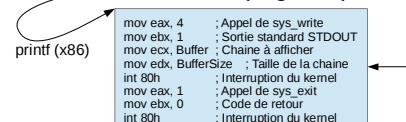
Représentation de la mémoire sous forme de blocs de données/routines indépendants

- Pile
- Table des symboles
- Programme principal
- Fonction f1
- Sous-routine
- Etc.



### • Donnée ou instruction

= adresse de base (segment) + offset (décalage)

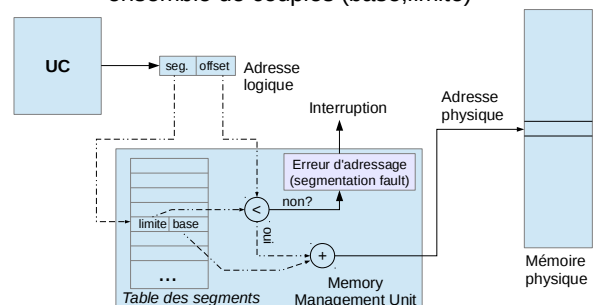


## Compilation

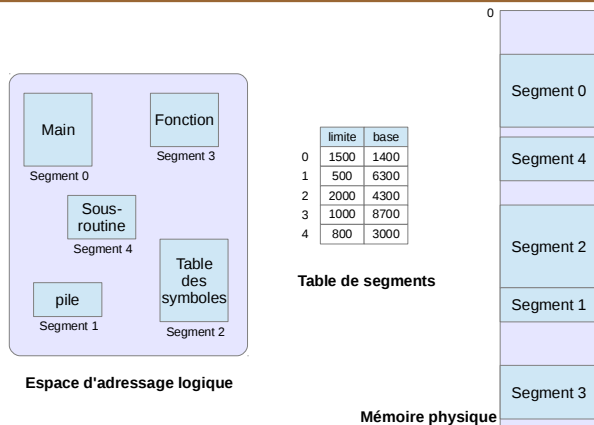
- Construction des segments
  - chaque segment est référencé par un numéro
- Exemples :
  - Segments d'un programme Java
    - Méthodes
    - Tas
    - Piles pour chaque Thread
    - Class loader
  - Segments d'un programme C
    - Variables globales
    - Fonctions de bibliothèques
    - Programme principal

## Memory Management Unit

- Adresse = numéro segment + offset
- Table des segments
  - ensemble de couples (base, limite)



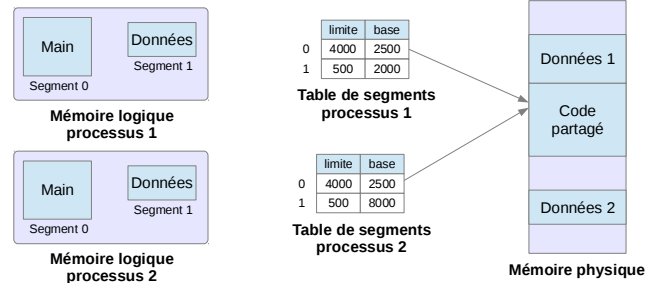
## Exemples



## Avantages & limites

- Partage de segments

- 1 processus = 1 table
- Partage de code (bibliothèques, programmes multi-utilisateurs)



## Avantages & limites

- Partage de segments
  - 1 processus = 1 table
  - Problème : référencement du segment « commun »
    - Appel de la routine = saut d'adresse
    - Tous les codes doivent référencer le segment commun de la même manière !
    - Trouver une référence qui convienne à tout le monde...
- Protection
  - Segments R, W ou RW (ex : instructions vs données)
  - Table d'association → bits de protection RW
- 1 tableau = 1 segment → vérification automatique des débordements !

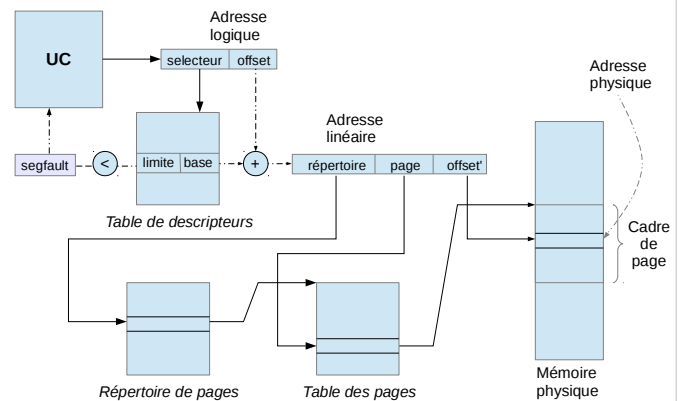
## Avantages & limites

- Partage de segments
    - 1 processus = 1 table
    - Problème : référencement du segment « commun »
  - Protection
    - Lecture/écriture
    - Protection de la mémoire
  - Allocation
    - Segments de longueur variable
      - problème d'allocation (cf. mémoire contigüe)
    - Fragmentation + risque d'attente (ou compactage)
- combiner pagination et segmentation !

## Segmentation avec pagination

- Principe
  - Paginer les segments
    - réduit la fragmentation et les problèmes d'allocation
    - permet le partage et l'adressage des segments
  - Adresse logique = segment + offset
  - Adresse linéaire = répertoire + page + offset
  - Adresse physique = cadre de page + offset
- Memory Management Unit
  - La table de descripteurs convertit les adresses logiques en adresses linéaires : **segmentation**
  - Le répertoire + la table des pages permettent de déterminer les cadres de page : **pagination**

## Segmentation avec pagination



## Segmentation avec pagination

- Exemple : intel 80386
  - 16 384 ( $2^{14}$ ) segments possibles par processus, 1 segment = max 4Go
    - 8192 « globaux » (partage de segments) et 8192 locaux (données & code propres au processus)
  - Adresse logique = 16 bits **selecteur** + 32 bits offset

Segment (13)	G (1)	RW (2)
--------------	-------	--------

- Table Descripteurs (G ou L) = 8 Ko ( $2^{13}$ )
  - 1 entrée = 8 octets dont base et limite (4)
    - adresse linéaire 32 bits
- 1 page = 4 Ko → pagination double

Répertoire (10)	Page (10)	Offset* (12)
-----------------	-----------	--------------

## Segmentation avec pagination

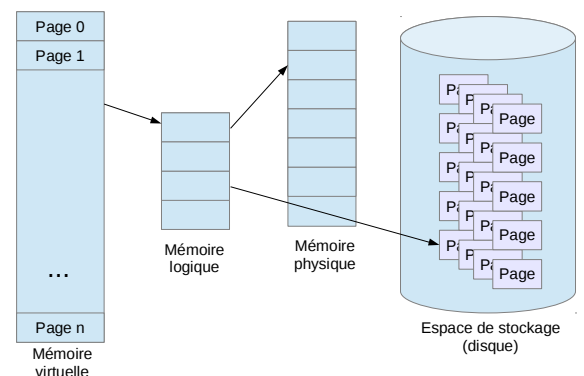
- Exemple (suite) : linux 2 sur 80386
  - Segmentation réduite (compatible plusieurs architectures) → 6 segments
    - Code du noyau
    - Données du noyau
    - Code utilisateur
    - Données utilisateur
    - État de la tâche
    - Local Descriptors Table (LDT) par défaut
  - Protection RW = 4 niveaux
    - Linux : 2 niveaux seulement (utilisateur/noyau)

## Mémoire virtuelle

- Mémoire d'un programme > mémoire physique
  - on n'a pas besoin de tout le programme en même temps !
  - Une partie des instructions et données seulement doivent être en mémoire physique
- Mémoire virtuelle
  - Le programmeur voit un espace d'adressage « virtuel »
  - L'espace d'adressage peut être plus grand que la mémoire physique
  - La mémoire logique **doit** être plus petite !
  - Utilisation du disque → pagination à la demande

## Mémoire virtuelle

- Schéma général :

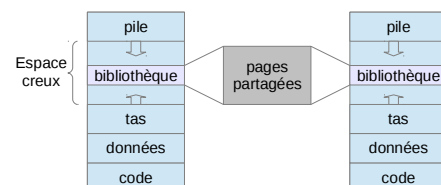


## Avantages

- Exemples
  - Code d'erreurs, fonctions rarement exécutées
  - Table des symboles d'un programme
  - Tableau (généralement sous-utilisé)
- Moins de mémoire utilisée
  - plus de programmes en parallèle !
  - Moins de perte E/S lors des commutations de contexte
- Grand espace d'adressage virtuel
  - le programmeur ne se préoccupe plus de la mémoire
  - gestion souple de la pile et du tas (tailles variable)
    - « Espace creux » variable entre les deux

## Avantages (suite)

- Partage de pages
  - Bibliothèques (utilisées en lecture seule)
  - Fork et création de processus

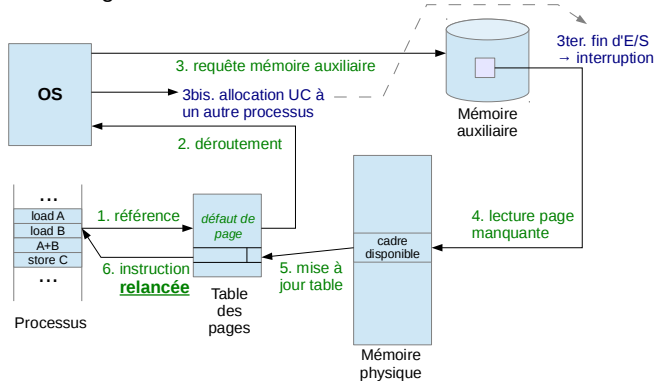


- Bit de validité de page → déchargement des pages
- Recharger seulement les nouvelles pages requises

Pagination à la demande

## Fonctionnement

### Pagination à la demande



## Temps d'accès

### Un peu de calcul...

- $p$  = probabilité de défaut de page
- $M$  = temps d'accès à la mémoire
- $D$  = temps de traitement défaut de page
- temps d'accès =  $(1-p) \cdot M + p \cdot D$

$$= M + p \cdot (D - M)$$

le temps d'accès est directement proportionnel à la probabilité d'avoir un défaut de page !

→ d'où l'intérêt d'avoir un bon algorithme de remplacement !

$p \cdot D / M \approx$  ralentissement dû à pagination à la demande

## Remplacement de pages

- Rappel des algorithmes
  - FIFO, deuxième-chance, horloge
  - LFU, LRU, NRU (*peu utilisés en pratique*)
  - Bit de modification
    - noter les pages non-modifiées et les remplacer en priorité
- Défaut de pages : anomalie de Belady
- Réduire les défauts → sur-allocation
  - hypothèse = les processus n'utilisent pas tous les cadres qui leur sont alloués
  - charger plus de processus sans augmenter le nombre de défaut de pages
  - Les cadres supplémentaires requis correspondent en réalité à des pages non utilisées

## Allocation (1/3)

- Mémoire physique → cadres de page

Allouer  $N$  cadres sur  $P$  processus  
 → plus de cadres → moins de  $P$  possibles  
 → moins de cadres → défauts de page → ralentissement de l'exécution

- Allocation équitable
  - Chaque processus reçoit  $N/P$
  - Le reste  $N \% P$  sert de **tampon**
    - lors du remplacement de page, permet de charger une page et de relancer l'instruction, sans avoir à vider la page en mémoire tout de suite

## Allocation (2/3)

- Allocation proportionnelle
  - En fonction de la taille de la mémoire virtuelle de chaque processus
  - Soit  $M_i$  la taille du processus  $i$  (en mémoire virtuelle) et  $M = \sum M_i$
  - $P_i$  reçoit  $N * M_i / M$  cadres
- Allocation basée sur la priorité
  - En fonction de la priorité de chaque processus
  - Soit  $prio_i$  la priorité du processus et  $prio_{tot}$  la somme des priorités
  - $P_i$  reçoit  $N * prio_i / prio_{tot}$

## Allocation 3/3

- Remplacement local
  - un processus ne peut « remplacer » que dans les cadres de pages qui lui sont alloués (plus le cache)
- Remplacement global
  - un processus peut utiliser n'importe quel cadre de page, y compris ceux utilisés par d'autres
  - Avantage : augmenter la vitesse d'exécution des processus prioritaires
  - Inconvénient : risque de provoquer de nouveaux défauts de pages pour les autres processus

### → écrasement

Allocation → ralentit → revoir le taux de l'UC → mettre de nouveaux processus → encore plus de défauts...

## Cours 7

### Système de fichiers (1/3)

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Rôle du système de fichiers

- Partie « visible » de l'OS :
  - Mécanisme de stockage
  - Accès aux données
  - Accès aux programmes de l'OS
  - Accès au système informatique
- Structure
  - Collection de fichiers
  - Structure de répertoires
  - Partitions (éventuellement)
  - Mécanisme de protection

## Plan

- Cours 7
  - Notion de fichier, structure et fonctionnement général
  - Méthodes d'accès
  - Structure des répertoires
  - Montage
  - Partage et protection
- Cours 8
  - Implémentation, structure des disques, blocs
  - Allocation, Ordonnancement
- Cours 9
  - Disques SSD
  - Systèmes distribués : NFS, RAID

## Notion de fichier

- Unité de stockage logique
  - Indépendant du support (disque, bande, carte...)
  - Abstraction des propriétés physiques
  - Correspondance périphérique ↔ fichier
- Définition

Collection nommée d'informations, enregistrée sur un stockage secondaire (*i.e.* hors RAM)

Sous forme de bits, lignes ou enregistrements... dont la signification est définie par le créateur et l'utilisateur de ce fichier

## Structure d'un fichier (1/2)

Dépend de son type :

- Fichier texte (.txt) : succession de caractères, organisés en lignes
- Fichier source (.c) : succession de fonctions et sous-programmes, composés d'instructions
- Fichier objet (.o, .class) : succession d'octets organisés en blocs interprétables par l'éditeur de lien
- Fichier exécutable (.exe) : succession d'instructions que l'OS peut charger en mémoire et exécuter dans l'UC
- etc

## Structure d'un fichier (2/2)

- **Nom** : conservé dans un format compréhensible (chaîne de caractères), indépendant de l'OS  
Fichier nommé → indépendance vis-à-vis du processus créateur et de l'utilisateur !
- **Identifiant** : en général numérique, pour l'OS
- **Type** : information sur la structure du contenu  
Souvent visible dans le nom par l'extension, ou dans le début du contenu, utilisable par l'OS (ex : MSDOS n'exécute que les fichiers .com, .exe ou .bat)
- **Emplacement** : pointeur sur un périphérique
- **Taille** : en octets ou en blocs, peut être bornée
- **Protection** : contrôle d'accès (R/W, exec)
- **Date(s)** : de création, de modification, d'accès...
- **Utilisateur** : qui a créé ce fichier

## Opérations & répertoires

- Opérations de base sur un fichier
  - Création : allocation espace + entrée répertoire
  - Écriture : pointeur d'écriture
  - Lecture : pointeur de lecture
  - Repositionnement : déplacement des pointeurs
  - Suppression : retrait de l'entrée dans le répertoire
  - Troncature : vider le contenu mais garder l'entrée
- Opérations composées (copie, renommage...)
- Répertoire = structure de stockage des informations des fichiers
  - Entrée répertoire = nom+identifiant
  - Structure des fichiers → plusieurs Ko !
  - Non scindables (chargé en mémoire par bloc)

## Ouverture de fichier

- Opération dans un fichier
  - recherche de l'entrée dans le répertoire  
coûteux si répété !
  - opération préliminaire imposée par l'OS : appel système **open**
- Table des fichiers ouverts
  - open = ajout dans la table
  - close = retrait de la table
- Gestion par l'OS
  - Open implicite au premier accès vs exception
  - Close lors de la fin du processus
    - une table par processus (avec pointeurs lecture/écriture)
    - 1 table globale avec « compteurs »

## Verrouillage

- Problème lecteurs/écrivains (voir TD4)
  - bloque l'accès aux autres processus
- Exemple en Java :

```
java.nio.channels.FileLock s = null, e = null;
java.io.RandomAccessFile f =
    new java.io.RandomAccessFile("toto.txt", "rw");
java.nio.channels.FileChannel c = f.getChannel();

Écrivain → exclusif { e = c.lock(0, f.length() / 2, false);
                    /* écriture */
                    e.release();

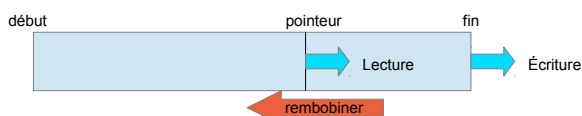
Lecteur → partagé { s = c.lock(f.length() / 2 + 1, f.length(), true);
                    /* lecture */
                    s.release();
```

## Blocs & méthodes d'accès

- Notion de bloc
  - Unités d'échange de données avec l'OS  
Le disque transfère les données par bloc
  - fragmentation interne  
ex : blocs de 512o, 1949 octets  
→ 99o perdus dans le dernier bloc
- Accès
  - Séquentiel
  - Direct
  - Index

## Accès séquentiel

- Modèle « lecteur de bande »
  - Pointeur de lecture séquentiel  
Lit la donnée et avance
  - Pointeur d'écriture séquentiel  
Pointeur = fin de fichier  
Ajoute à la fin du fichier et définit la nouvelle position de la fin
  - Possibilité de « rembobiner »  
Reculer le pointeur de lecture ou d'écriture



## Accès direct

- Modèle « disque » ou « base de données »
  - Fichier = séquence d'enregistrements de taille fixe (blocs)
  - Lecture/écriture d'enregistrements dans n'importe quel ordre
- Instruction read/write + n (enregistrement)
  - Positionnement sur l'enregistrement
  - Read/write séquentiel sur 1 enregistrement
- Adresses relatives au début du fichier
  - 0 = premier enregistrement du fichier
  - Sauts de taille T à partir de l'emplacement du fichier
- Capable de simuler l'accès séquentiel (à l'aide d'un compteur)

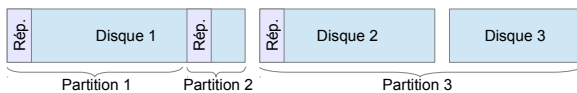
## Index

- Modèle « livre »
  - Index = ensemble de pointeurs vers un groupe d'enregistrements, regroupés en blocs
  - Lorsque bloc fichier > taille enregistrement
- Fichier d'index
  - Relation d'ordre entre les enregistrements
    - classement dans l'ordre dans les blocs
  - Index → pointeur vers le premier enregistrement du bloc
- Indexation multi-niveaux
  - Si le fichier d'index devient trop gros
    - voir pagination multi-niveaux

## Structure des répertoires

- Disque : structure physique
- 1 ou plusieurs partitions
  - structure logique (« disque virtuel »)
    - Périphérique de stockage individuel virtuel
    - Selon les OS, 1 disque =  $N \geq 1$  partitions, ou 1 partition =  $N \geq 1$  disques
  - Exemple : partitions Linux/Windows
- Répertoire : 1 par partition
  - Nom → identifiants
  - Fonctionnalités :
    - Recherche d'un fichier
    - Création/suppression/renommage
    - Lister les entrées / parcours du système de fichiers

## Répertoire : structure de base



- Structure à un niveau : nom → fichier
  - 1 seul répertoire → taille proportionnelle au nombre de fichiers
  - Limiter la taille des noms
    - réduit l'espace du répertoire
    - problème d'unicité
  - Peu pratique pour l'utilisateur (mémoriser les noms)
- Exemple : MSDOS (taille limitée à 110)
  - Unix, taille limitée à 2550

## Répertoires multi-niveaux (1/2)

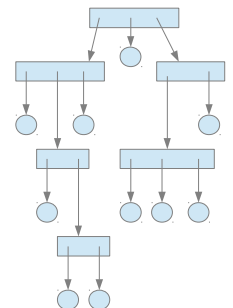
- Structure à deux niveaux :
    - un répertoire par utilisateur
- 
- Les noms de chaque UFD doivent rester uniques
    - mais 2 UFD différents peuvent avoir des fichiers différents qui portent le même nom
    - isole les utilisateurs !

## Répertoires multi-niveaux (2/2)

- Accès autorisé aux fichiers d'un autre utilisateur
    - il faut pouvoir nommer les fichiers
- [Partition +] [utilisateur +] nom
- toto.txt (nom seul → utilisateur courant)
- /batman/toto.txt (style UNIX)
- c:\robin\toto.txt (style MSDOS)
- Fichiers de l'OS
    - chargeurs, assembleurs, bibliothèques... → ne pas les recopier dans chaque UFD !
    - répertoire d'utilisateur spécial (user0) + recherche par défaut si fichier non trouvé chez l'utilisateur
    - Ordre de recherche = chemin (PATH)

## Répertoires arborescents

- MSDOS, Unix, MacOS...
  - Répertoire racine
  - Entrée = fichier ou sous-répertoire
    - bit « répertoire » dans la table
    - Création/Suppression = appels systèmes
  - Chemins uniques depuis la racine
    - noms uniques : chemin *absolu*
- Répertoire courant
  - Appel système → changement
  - Recherche à partir du répertoire courant
    - chemin *relatif*
- Suppression :
  - Répertoire vide (MSDOS) ou Destruction des entrées (Unix *rm*)



## Répertoires en graphes (1/2)

- Partage de fichiers/répertoires (entre plusieurs utilisateurs)
  - graphe acyclique

- Mécanisme

- Mécanisme de liens

1 seul fichier/rep. référencé dans deux répertoires différents

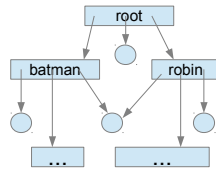
→ bit de « lien » + chemin absolu

- Duplication

Contrairement au lien, original et copie indiscernables !

- Suppression

- Laisser les liens (Unix *ln -s*, Windows)
  - Compteur de liens dans les fichiers (Unix *ln*)



## Répertoires en graphes (2/2)

- Graphe acyclique

- parcours facile (ex : recherche de fichier)
  - difficulté de garantir l'absence de cycle

- Cas général

- Détecter les cycles (créés par les liens)
  - Ramasse-miette (suppression des liens vides)
  - Algorithmes coûteux

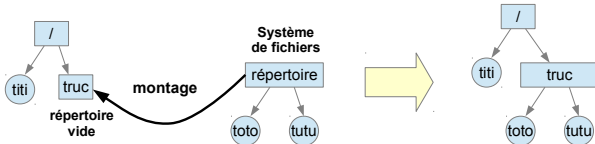
→ La majorité des OS ne gèrent pas le cas général

## Montage

- Chargement des répertoires → montage

- Lien système de fichier ↔ point de montage  
(ex : répertoire vide)
  - le répertoire est chargé et les fichiers deviennent accessibles

- Exemple :



- Montage implicite (sur interruption) ou explicite (sur appel système)

## Partage de fichiers (1/2)

- OS multi-utilisateurs

Fichiers d'un utilisateur accessibles à un autre

- partage de fichiers
  - politique de protection
  - attributs supplémentaires sur les fichiers

- Notion de propriétaire

→ Peut changer les attributs de protection

- Identifiant utilisateur (unique)
    - associé à tous ses processus
    - contrôle d'accès sur le propriétaire du fichier
  - Montage → attribution à l'utilisateur qui charge le système de fichier

## Partage de fichiers (2/2)

### Gestion de la cohérence

- Algorithmes de synchronisation (Semaphore, Mutex...) → problème des lecteurs/écrivains
  - En général pas utilisé dans les OS
  - gestion à la charge des utilisateurs
- Exemple : Unix
  - Écritures visibles immédiatement
  - Partage sur le pointeur de lecture/écriture
- Modèle par session : Andrew FS
  - Écritures « retardées » jusqu'à la fermeture du fichier
  - co-existence de versions différentes d'un même fichier !
- Autre méthode : fichiers partagés non modifiables

## Protection (1/2)

- Contrôle d'accès

→ quel utilisateur (ID) peut accéder à quel fichier ?

Lecture, Écriture, Exécution, Ajout, Suppression, Visualisation (nom & attributs)

- Access Control List (ACL)

- Utilisateur → type d'accès autorisé
  - Pb : liste d'ID à connaître a priori + taille du répertoire = fonction du nombre d'ID !

- Classes d'utilisateurs

- Ex : Propriétaires vs Groupe (cf. page suivante) vs Autres
  - 3xN bits par fichier

- Mot de passe

- Fichiers → mot de passe → accès limité
  - Mots de passe par type d'accès (lecture, exécution, ajout, etc)



## Protection (2/2)

- Notion de groupe
  - Définition d'un ensemble de groupes
  - 1 fichier = 1 propriétaire + 1 groupe
  - 1 utilisateur (ID) = N groupes
- Protection :
  - Accès en fonction du groupe
    - Est-ce que l'utilisateur ID (dont le processus veut accéder au fichier) a dans ses groupes le groupe du fichier ?
  - Généralement, accès propriétaire  $\neq$  accès groupe

## Protection : Unix

- Contrôle d'accès sur 3 bits :
  - `r` Lecture / Lister fichiers du répertoire
  - `w` Écriture / Suppression / création dans répertoire
  - `x` Exécution / accès en tant que répertoire courant
- Méthode par « classes d'utilisateurs »
  - Propriétaire (= utilisateur/créateur), Groupe, Autres
- Exemple

```
$ ls -l
total 248
drwx----- 6 nico prof      4096 nov.  20 12:06 private
-rw----- 1 nico prof      2356 déc.   5 15:30 notes.ods
drwx----- 2 nico prof      4096 juin  30 00:17 mail
drwxrwxr-x  2 nico prof      4096 déc.   4 17:31 doc
-rw-rw-r--  1 nico student    36 jan.  21 16:49 programme.c
-rwxrwxr-x  5 joe  student    996 jan.  28 10:53 programme
drwxr-x---  1 root users     1630 fév.  10 12:14 lib
```

## Cours 8

### Système de fichiers (2/3)

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Plan

- Cours 7
  - Notion de fichier, structure et fonctionnement général
  - Méthodes d'accès, montage, répertoires
  - Partage et protection
- Cours 8
  - Implémentation
  - Structure des disques
  - Gestion des blocs
  - Allocation
  - Ordonnancement
- Cours 9
  - Disques SSD
  - Systèmes distribués : NFS, RAID

## Structure d'un FS (1/2)

- Mémoire auxiliaire
  - Grande quantité de données
  - Accès lent (par rapport à la RAM)
  - Support le plus utilisé : disque magnétique
    - Capacité de réécriture (comme la bande)
    - Accès direct n'importe où (contrairement à bande)
- Notion de bloc
  - Unité de base du support
  - Généralement 512o (peut varier de 32o à 4Ko)
  - Organisation des fichiers en blocs
    - **Système de fichiers**

## Structure d'un FS (2/2)

### Structuration en couches :

- Système de fichiers logique
  - Structure des répertoires
  - Blocs de contrôle de fichiers (FCB)
  - Gestion de la protection
- Module d'organisation des fichiers
  - Fichiers → blocs logiques
  - Blocs logiques → blocs physiques (allocation)
  - Gestionnaire d'espace libre
- Système de fichiers de base :
  - Identification des blocs physiques : unité, cylindre, piste, secteur
  - Commandes haut niveau (ex : extraire le bloc 456)
- Contrôle des E/S : pilotes de périphériques
  - ex : extraire le bloc 456 → instructions matériel

## Implémentation (1/2)

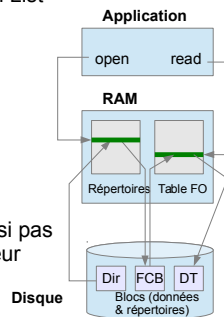
- Structures de contrôle **sur le disque lui-même**
  - Bloc/secteur de démarrage
    - Informations nécessaires au système informatique pour charger et démarrer l'OS depuis cette partition
  - Bloc de contrôle de partition
    - Informations sur la partition (nombre de blocs, taille, compteur de blocs libres, FCB libres...)
    - superbloc ou Master File Table (MFT)
  - Structure de répertoires
  - Blocs de contrôle de fichiers (FCB)
    - Informations sur les fichiers : propriétés, permissions, taille, emplacement des blocs de données...
    - **inode** (Unix) = bloc FCB
    - stockés dans la MFT (Windows) = structure BD

## Implémentation (2/2)

- Informations dans la mémoire
  - Gestion du FS par l'OS
  - Table des partitions montées
  - Structure des répertoires (cache → optimisation de l'accès)
  - Table générale des fichiers ouverts + copie des FCB des fichiers ouverts
  - Table des fichiers ouverts par processus = pointeurs vers l'entrée dans la table générale
- Utilisation de descripteurs de fichiers
  - L'OS ne manipule pas les noms, mais les identifiants de fichiers

## Descripteur de fichier

- File Control Bloc
  - Dates : création, accès, modification
  - Propriétaire, groupe, Access Control List
  - Permissions associées
  - Taille du fichier
  - Blocs de données du fichier
- Open
  - Nom de fichier → FS
  - Recherche dans les répertoires
  - Chargement du FCB dans la table (si pas déjà ouvert) et mise à jour du pointeur



## Répertoires

- Liste linéaire
  - Séquence de (nom, pointeurs FCB)
  - Coût : recherche d'un fichier (à l'ouverture, à la création, à la suppression...)
  - Amélioration : liste triée (dichotomie) + cache
- Table de hachage
  - Liste linéaire
  - Table de couples (clef, pointeur(s))
  - Clef = fonction du nom
  - Problème = nb clefs limité
    - Recalcul lorsqu'on a besoin de plus de clefs (coûteux)
    - Clef → liste de pointeurs (ralentit la recherche)

## Allocation

- Problème
  - Placer les bloc sur le disque
  - Accès direct (comme la RAM) → fragmentation
- Méthodes
  - Contiguë
  - Chaînée
  - Indexée

## Allocation contiguë

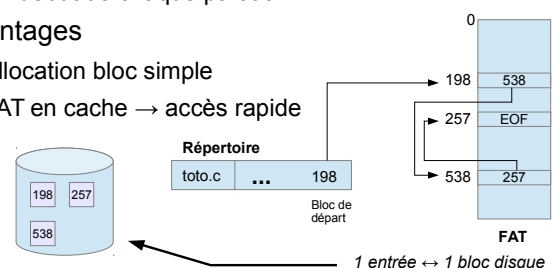
- Principe
  - Tous les blocs du fichiers sont rangés les uns derrière les autres, dans l'ordre
- Avantage
  - Accès de B à B+1 → aucun coût (sauf changement de cylindre, voir plus loin)
  - Entrée FCB simplifiée (début+taille vs liste blocs)
- Inconvénients
  - Connaître à l'avance la taille des fichiers (ex : données)
  - Recherche de l'espace nécessaire → coûteux
  - Allocation (cf. mémoire) : premier, plus petit, plus grand → aucune stratégie n'est meilleure !
  - Fragmentation → compactage (coûteux et bloquant)

## Allocation chaînée

- Principe
  - Fichier = liste chaînée de blocs
  - FCB → pointeur vers le premier et le dernier blocs
  - Chaque bloc contient un pointeur vers le suivant
- Avantages
  - Pas de fragmentation
  - Fichiers taille variable
- Inconvénients
  - Accès séquentiel : Nième bloc → N accès
  - Espace utilisé par les pointeurs
  - Compromis : groupes (cf. pages RAM), mais augmente fragmentation
  - Fiabilité : bloc endommagé → fichier perdu (on ne sait plus comment trouver les blocs suivants)

## FAT

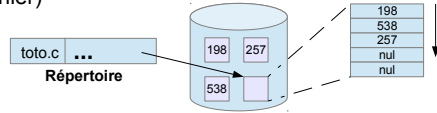
- Table d'allocation de fichiers (FAT)
  - Utilisée sous MSDOS et OS/2 (IBM)
  - Liste chaînée d'index des blocs
  - En début de chaque partition
- Avantages
  - Allocation bloc simple
  - FAT en cache → accès rapide



## Allocation indexée (1/2)

- Principe

Rassembler tous les pointeurs dans un bloc d'index  
(1 bloc par fichier)



- Avantages

- Pas de fragmentation (comme avec liste chaînée)
- Accès direct rapide (comme avec FAT)

- Inconvénients

- Blocs « gaspillés » (ex : 1 bloc pour 1 fichier vide)  
→ déterminer la taille du bloc d'index
- Taille fichier limitée par la taille du bloc

## Allocation indexée (2/2)

- Extension

→ fichiers nécessitant plus d'un bloc d'index

- Liste chaînée de blocs d'index
- Indexation multi-niveau (comme pagination)

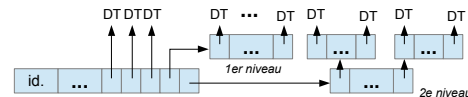
- Schéma combiné

- FCB contenant l'index des N premiers blocs

→ pas de bloc d'index pour les « petits » fichiers

+ K blocs d'indirection

1er bloc = index simple, Nième bloc = index supérieur



## Gestion de l'espace libre (1/2)

- Problème

Accès rapide aux blocs libres

→ liste d'espace libre

- Méthodes (cf. mémoire)

- Vecteur binaire

1 bloc = 1 bit (1 si libre)

→ rapide, code spécifique dans le processeur

ex : Intel 80386 ou Motorola (Mac)

→ mais coûteux (maintien du vecteur en mémoire)

s'il est gardé uniquement sur le disque, l'accès est très ralenti !

Méthode Mac : regrouper les blocs (réduction de l'espace)

NB : la FAT contient une telle liste de blocs libres !

## Gestion de l'espace libre (2/2)

- Méthodes (suite)

- Liste chaînée des blocs libres

- Pointeur vers le premier bloc libre

→ accès immédiat au premier bloc libre

- Allocation de plusieurs blocs compliquée

→ parcours de liste... mais cela n'arrive pas souvent

- Groupage

- Pointeur vers premier bloc libre

- Contient les adresses des N-1 prochains blocs libres + un pointeur vers le Nième (N = taille bloc / taille adresse)

→ permet de trouver plusieurs blocs libres rapidement

- Comptage

En pratique, on libère souvent des blocs contigus

- Liste chaînée + nombre de blocs à sauter

## Notion de cache

- Performance

- Coût d'accès au blocs (cf. matériel)

→ conserver les blocs à réutiliser

- cf. mémoire : algorithmes de remplacement

- FIFO, clock-based

- LRU

→ Linux et Windows : cache commun pages et blocs disque

Évite de mettre deux fois en cache les mêmes données (ex : lecture d'un fichier)

- Gestion du cache

- Lecture/écriture asynchrone

→ gestion de la cohérence (cf. processus)

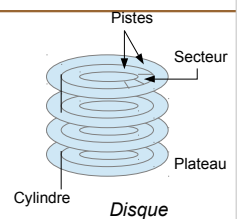
## Structure de disque (1/2)

- Structure générale

- Disque = ensemble de cylindres

- Cylindre = ensemble de pistes

- Piste = suite de secteurs



- Vue logique

- Tableau de blocs de 512o

- Bloc 0 = premier secteur de la première piste du cylindre extérieur

par secteur, puis par piste, puis par cylindre :

→ N° de bloc traduit en adresse physique (secteur, piste, cylindre)

## Structure de disque (2/2)

- Secteurs défectueux
  - mécanisme transparent pour l'OS
  - Le matériel les remplace par d'autres secteurs libres
- Vitesse angulaire (rotation des disque)
  - le nombre de secteur par piste ne serait pas constant
  - Solutions :
    - vitesse **linéaire** constante : le disque ralentit lorsque la tête de lecture se rapproche du centre  
ex : lecteurs de CD/DVD
    - Vitesse angulaire constante, densité d'octets différente  
ex : disques durs

## Blocs et disques (1/3)

- Formatage bas niveau
  - définition des secteurs
    - En-tête (ex : numéro de secteur)
    - Zone de données (512o)
    - Terminaison (ex : code correcteur d'erreur)
- Partitionnement
- Formatage logique
  - structures de données initiales (ex : FAT)
- Bloc d'amorçage (boot)
  - initialisation de l'UC, des contrôleurs de périphériques, de la mémoire (noyau OS)
  - partition spécifique du disque
  - programme de chargement du bloc d'amorçage en ROM

## Blocs et disques (2/3)

- Code correcteur
  - Fonction de l'ensemble des octets du secteurs
  - recalculée et comparée au CC à chaque E/S
  - Détection des blocs défectueux
  - Identification des octets à corriger (code correcteur)
- Principe = ajout d'information sur la zone
- Exemples :
  - Codes en blocs : vote majoritaire  
 $F(0)=000$  et  $F(1)=111$
  - Somme de contrôle : bit de parité de la somme  
 $F(00)=000$ ,  $F(01)=101$ ,  $F(10)=110$ ,  $F(11)=011$
  - Codes de Hamming (code linéaire)  
optimal en taille pour max. 1 bit d'erreur → cours de Master !

## Blocs et disques (3/3)

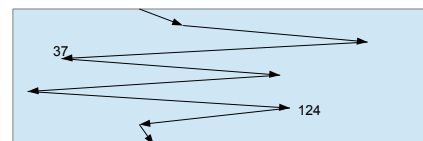
- Gestion des secteurs défectueux
  - Par l'OS : disque IDE (années 90)
    - ex : noter dans la FAT les blocs correspondant aux secteurs défectueux pour ne pas les utiliser
  - Par le matériel : disques SCSI
    - remplacement des secteurs par le contrôleur
  - Table de blocs défectueux
  - Groupe de secteurs « réservés » pour les remplacements, lors du formatage bas niveau
- Principe
  - Le contrôleur prévient l'OS que le bloc est défectueux (lorsqu'il est détecté)
  - L'OS utilise une requête pour demander un remplacement (en général : **glissement**)
    - l'accès est ensuite transparent pour l'OS

## Ordonnancement

- Problèmes
  - Temps d'accès
    - Temps de positionnement de la tête de lecture au dessus de la bonne piste
    - Temps de latence de rotation (max. attendre un tour)
  - Bande passante
    - Nombre d'octets transférés / temps de traitement
- programmer les requêtes d'E/S dans un ordre « intelligent »
- Plusieurs méthodes :
  - Premier arrivé = premier servi
  - Plus court positionnement d'abord
  - SCAN et C-SCAN
  - LOOK et C-LOOK

## FCFS

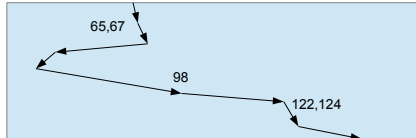
- First-Come, First-Served
  - Algorithme simple
  - Pas très rapide
- Exemple
  - Cylindres 53, 98, 183, 37, 122, 14, 124, 65, 67



→ 640 cylindres parcourus, beaucoup de va et viens

## SSTF

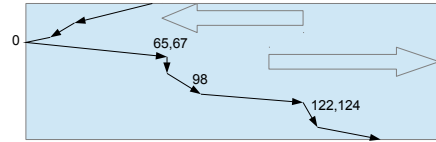
- Shortest-Seek-Time First  
→ servir d'abord le cylindre le plus proche de la position courante
- Même exemple (53, 98, 183, 37, 122, 14, 124, 65, 67)



- 236 cylindres parcourus
- risque de famine !  
Tant qu'il arrive de nouvelles requêtes « proches » de ce qui est servi actuellement, tous les autres attendent
- pas optimal (ex : servir 14 ou 37 en premier → 208 total)

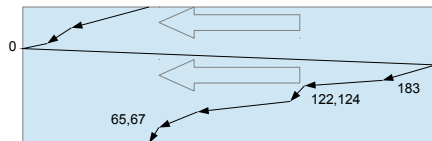
## SCAN

- Principe = ascenseur d'un immeuble  
La tête parcourt les cylindres dans l'ordre en va et viens et sert les requêtes lorsqu'elle passe sur le cylindre correspondant
- Bon compromis temps/latence
- Problème : on vient de servir les requêtes près du bord lorsqu'on fait demi-tour, donc on risque peu d'en avoir de nouvelle dans cette partie du disque !



## C-SCAN

- Circular-SCAN  
Principe = ascenseur uniquement montant (ou descendant)  
→ temps d'attente plus uniforme



## LOOK & C-LOOK

- Amélioration de SCAN et C-SCAN  
→ le bras ne va pas jusqu'au bout  
→ on repère lorsqu'on n'a plus de requête supérieure (resp. inférieure) à la requête courant et on fait demi-tour
- En pratique
  - La plupart des OS utilisent un SSTF  
sauf pour les OS qui utilisent beaucoup le disque → LOOK
  - Algorithme de calcul de l'ordonnancement optimal → coûteux
  - Temps d'accès dépend :
    - De la méthode d'allocation de fichiers (ex : contiguë vs chaînée)
    - Des positions des répertoires et blocs d'index (cf. ouverture des fichiers puis lecture)
    - Attention à la latence de rotation ! → contrôleur matériel
    - L'OS peut décider ce qui est important (ex : pagination > E/S)

## Cours 9

### Système de fichiers (3/3)

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Plan

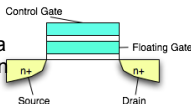
- Disques (fin)
  - Disques SSD
- Systèmes de fichiers répartis
  - Principe et problèmes
  - NFS
  - RAID

## Disques « Flash » (1/4)

- Solid State Drive
    - Clef USB, smartphone... → le dispositif se généralise !
    - Moins rapide et plus cher que les DD (fin 2013)
      - 110 Mo/s en test pour un DD 500 Go à 80 €
      - 50 Mo/s pour un SSD 120 Go à 80 €
- mais performances et coût s'améliorent rapidement !
- jusqu'à 500 Mo/s pour un SSD de 500 Go à 1000 €

- Principe : mémoire flash NAND
  - Effet tunnel Fowler-Nordheim (mécanique quantique)
  - Transistor avec une « grille flottante » en plus

- Courant sur la grille de contrôle
  - Courant +/- entre les deux bornes de la grille
- une partie des électrons va sur / part de la **grille flottante** (effet tunnel) jusqu'à saturation  
→ elle devient isolante (bit 0) ou cond. (bit 1)



## Disques « Flash » (2/4)

- Problème
  - Donnée = état grille flottante
  - Il reste toujours des électrons → usure entre 10 000 et 50 000 cycles d'écriture
- Blocs 512 Ko
  - Découpés en pages de 4 Ko+128 bits de contrôle
  - Bus en série → charger les 64 pages dans une RAM seule zone « accessible » + joue le rôle d'un cache
  - Contrôleur interne d'écriture → détecter les erreurs lors de l'écriture RAM → transistors + bit de contrôle (voir RAID semaine prochaine)
- Bien géré par les OS récents (Windows >=7, Linux, MacOS)

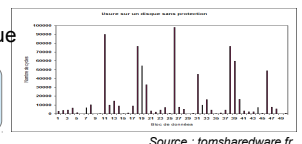
## Disques « Flash » (3/4)

- Stratégie de gestion de l'usure (*Wear Leveling*)
  - Mécanisme au niveau du contrôleur
  - Pas de WL : @logique (OS) → @physique fixe
    - Remplacement = effacer le bloc + réécrire → lent
    - Fichiers très lus → blocs plus utilisés → très vite inutilisables
    - C'est idiot car tous les blocs sont accessibles à la même vitesse !  
*pas de tête de lecture, de rotation...*
  - WL dynamique : modification sur écriture
    - Table d'adresse des blocs logiques (LBA)
    - Remplacement → choix nouvelle @physique + marquer bloc « invalide »
    - Table du nombre d'écritures + liste triée des blocs libres/invalides

## Disques « Flash » (4/4)

- WL dynamique (suite)
  - Problème : seule l'usure des blocs remplacés est égalisée (les blocs lus n'usent pas)  
→ usure inégale sur le disque

Le disque est inutilisable dès que X blocs ont atteint leur limite, quelque soit l'état des autres blocs !



Source : tomshardware.fr

- WL statique (ou « global »)
  - Idée : déplacer les blocs qui ne sont pas écrits
  - Table LBA + enregistrer date utilisation + nombre écritures
  - Écriture → chercher le bloc le moins utilisé
    - Bloc libre → utiliser
    - Bloc dynamique = utilisé depuis moins de x → trouver un autre bloc
    - Bloc statique (pas utilisé depuis x) → déplacer la donnée et utiliser ce bloc

## Systèmes répartis

## Partage de fichiers

- Partage de fichier → user ID
  - Droits sur le fichier associés à l'ID stocké dans le FCB
  - Montage : attribution du répertoire à l'ID du propriétaire du processus
    - conserve les informations sur les fichiers (en terme d'ID, de droits...)
- Partage de fichiers à distance
  - Transfert (FTP, mail...)
    - Le web est un cas particulier de transfert (HTTP, via le navigateur (client))
  - Systèmes de fichiers distribués (ou répartis)

## Transfert de fichiers

- Approche client-serveur
  - Serveur = machine sur laquelle sont stockés les fichiers
    - En réseau, le serveur est un processus, donc un espace plus restreint
  - Client = machine qui souhaite accéder aux fichiers
- Accès anonyme
  - ex : HTTP, ou FTP anonyme
    - le client n'a pas besoin d'UID sur le serveur
    - un nouvel UID est attribué à chaque demande
- Identification
  - Adresse IP, login → risque d'usurpation
  - Clef → interopérabilité (même algo de cryptage (ex : RSA), échange de clefs, etc)

## Systèmes répartis

- NFS (Network File System, Unix)
  - Informations (ID) fournies par le client
    - ID sur le client et sur le serveur doivent correspondre
  - Monter le FS distant (authentification réseau)
    - communication via le protocole DFS
      - Les opérations sur les fichiers sont les mêmes, à partir des ID fournis par le client vs ceux stockés sur le serveur
- Service de nommage distribué
  - unifier l'accès aux informations
    - ex réseau : DNS = nom d'hôte → adresse IP

## Services de nommages

- Unix : NIS (Network Information Service)
  - Anciennement YP (Yellow Pages)
  - fournit des informations complètes (login, MDP, UID, GID) pour les fichiers et le matériel (ex : imprimantes)
  - identification via IP + MDP non crypté !
- Windows : Samba
  - CIFS (Common Internet File System)
  - Protocole SMB (Server Message Block)
  - identification (login, MDP) → login réseau
  - espace de nommage unique, partagé client & serveur
  - Nom de domaine (XP, 2000) puis Active Directory (>vista)
  - L'espace de nommage est partagé par tous les C&S
- LDAP (Lightweight Directory-Access Protocol)
  - Le plus utilisé aujourd'hui (Active Directory, Unix...)
  - Principe : informations type NIS avec authentification sécurisée & unique comme dans CIFS

## Gestion des pannes

- FS local
  - cause matérielle...
    - fichier considéré comme perdu
    - échec/exception immédiatement
- FS distant
  - causes variées : perte de paquet, coupure réseau, arrêt planifié du serveur
    - ne pas considérer le fichier comme perdu !
  - DFS : retarder les opérations sur les fichiers
    - idée : l'hôte peut redevenir disponible
    - ajout d'information « d'état » (fichier ouvert, point d'accès, position de lecture...) sur le client et le serveur (ex : NFS)
      - pb de sécurité (n'importe quel processus peut accéder au fichier s'il fait croire qu'il est celui qui l'a ouvert...)



## NFS (1/8)

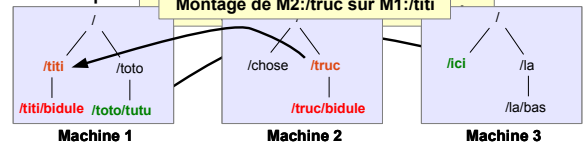
- Network File System
  - Système de fichier réseau type « client-serveur »
  - Très utilisé (serveurs Unix)
  - Basé sur protocoles IP+TCP/UDP
  - Implémentation variant d'un OS à l'autre (Solaris, BSD, Linux...)
- Principe général
  - Machines indépendantes, FS indépendants
  - Partage sur demande explicite
  - Partage transparent
  - N vers N (une machine peut être à la fois client et serveur)
    - chaque paire client-serveur voit un partage différent

- Protocole DFS avec primitives RPC (Remote Procedure Call)  
→ Protocole de montage + Protocole NFS d'accès aux fichiers

## NFS (2/8)

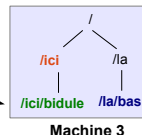
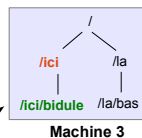
- Montage
  - d'un répertoire du FS distant (serveur)
    - identifié par hôte + nom unique du répertoire sur le serveur
  - dans un répertoire du FS local (client)
    - le répertoire distant est alors vu comme un répertoire local (transparence)

### Exemple



## NFS (3/8)

- Avantage
  - Station de travail sans disque (on peut monter la racine depuis le serveur)
  - Montage en cascade
    - Monter M2:/truc dans M3:/ici
    - Puis M1:/toto/tutu dans le nouvel M3:/ici/bidule
- Mobilité
  - Le client n'accède pas aux FS montés par le serveur
    - Monter M1:/titi sur M3:/la/bas, alors que M2:/truc/bidule est déjà monté sur M1:/titi
      - on récupère les données bleues (de M1, pas de M2)



## NFS (4/8)

- Client-serveur
  - Chaque machine possède un processus serveur, extérieur au noyau, capable de traiter les RPC pour accomplir le protocole
- Protocole de montage
  - Client → nom du répertoire distant (+ hôte)
  - Serveur : liste d'exportation
    - Linux : /etc/exports    Sun : /etc/dfs/dfstab
    - liste de répertoires + machines autorisées à les monter + droits d'accès (RW)

L'authentification se fait uniquement sur la base des noms (ou adresses) des machines sur le réseau local !

## NFS (5/8)

- Protocole de montage (suite)
  - Le serveur renvoie un **File Descriptor** qui sera utilisé comme clef pour les accès au répertoire monté
    - Contient les informations d'un répertoire nécessaires pour accéder aux fichiers (= inode du répertoire local (serveur))
  - Le serveur note quelle machine a monté quel répertoire
- Protocole NFS
  - Accès aux fichiers d'un répertoire distant monté
    - Recherche un fichier dans un répertoire
    - Lister les entrées d'un répertoire, les attributs
    - Lecture/écriture de fichiers
    - Manipulation et création de liens

→ via des RPC, avec un Descripteur de Fichier (FD)

## NFS (6/8)

- Protocole NFS
  - pas d'opération open/close !
    - Le serveur NFS n'a pas d'état : pas de table de fichiers ouverts côté serveur (mais côté client)
    - L'identifiant (FD) est fourni à chaque appel (RPC)
      - robuste (en cas de panne, rien à refaire)
  - Identifiants de requête (unique)
    - Permet au serveur de vérifier les requêtes perdues ou dupliquées (erreurs réseau)
- Asynchronisme
  - Toute les opérations sur le serveur son synchrones
    - le client peut mettre des données fichiers en cache, mais le RPC est synchrone (mise en attente)
    - le contrôleur confirme l'écriture → latence !

## NFS (7/8)

- Propriétés
  - Synchronisme
  - Atomicité des opérations (pas d'interruption par un processus local)
    - Attention : max données = 8Ko
    - un appel système « write » peut être découpé en plusieurs RPC
    - utilisation de verrous (pb des lecteurs/écrivains) externes au NFS
  - Robustesse (pannes courtes réseau ou serveur)
- Virtual FS interface
  - Couche de communication OS – NFS
  - traduit les appels systèmes en RPC et vice et versa
  - Symétrique, géré par un ensemble de processus dans le noyau

## NFS (8/8)

- Gestion des noms/chemins de fichiers
 

L'OS parcourt le chemin → chaque point de montage franchi donne lieu à une RPC

```

      /usr/share/lib/java/java7/bin/java
      ↗
      share monté depuis un serveur
      → la suite de l'arborescence
      n'existe que sur le serveur donc
      nécessite une RPC
      ... y compris pour accéder
      à un 2e répertoire monté
      en NFS (ici java7)
      
```

  - Le serveur ne peut pas savoir que java7 est monté dans truc/lib/java chez le client !
  - En général, utilisation de caches
- Caches des fichiers (client)
  - Cache des attributs (informations inodes)
  - Cache des blocs
  - Les RPC vérifient s'il faut mettre à jour + timeout des informations dans les caches (60 sec)

## RAID

- Redundant Array of Inexpensive Disks
  - Utilisation en parallèle des disques (locaux) sur lesquelles les données sont réparties et répétées
- Problème de performance et de fiabilité des données
  - Fiabilité
    - 1 disque tombe en panne toutes les 100 000 heures (11 ans) → dans un parc de 100 disques, on a une panne tous les 41 jours !
    - Méthode classique = sauvegardes → pertes de données
    - RAID → puisque les disques sont peu chers, en avoir plus et utiliser de la **redondance**
      - stocker de l'information non-nécessaire mais qui permet de récupérer les pannes (cf. code correcteur)
  - Performance
    - Améliorer le matériel → coût exponentiel
    - RAID → puisque les disques sont peu chers, en avoir plus et accéder au **parallèle** aux données

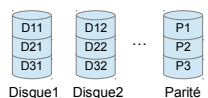
## RAID 0

- RAID0 : performance (via parallélisme)
  - Idée : entrelacer les données sur les disques
  - Entrelacement de bits
    - chaque bit est écrit sur un disque différent, modulo N
    - chaque disque virtuel peut être vu comme un disque dont les secteurs sont N fois plus grands... et avec un débit N fois supérieur !
  - En général, N = 2, 4 ou 8
  - Entrelacement d'octets, de blocs...
- Avantages
  - Meilleure capacité de traitement des « petits » accès (ex : les pages en swap)
  - Réduit le temps de réponse des grands accès

## RAID 1

- RAID1 : enregistrer en double toutes les données
  - Mirroring : données recopiées 2 fois sur le même disque
  - Shadowing : chaque disque est dupliqué
- Avantages/inconvénients
  - Simple mais coûteux
    - 1 disque logique = 2 disques physiques
    - Requêtes disque doublées (2 fois plus lent)
  - Dépend du temps moyen avant réparation !
    - Exemple : temps de dépannage = 10 heures
    - une panne tous les 57 000 ans ( $100k \times 100k / (2 \times 10)$ )
    - ... à condition que les pannes soient indépendantes
- shadowing meilleur que mirroring (meilleure indépendance des pannes)

## RAID 2 à 4

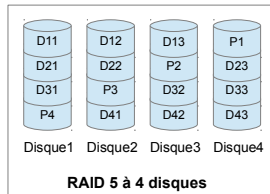
- Code correcteur d'erreur
- RAID2 : RAID0 (volume en bande) + code de Hamming
    - 1 bit de parité par octet + bits de correction
    - 8 disques de données + 4 disques de code correcteur
    - sécurité mais coûteux → obsolète
  - RAID3 et RAID4 : bande + 1 seul disque de parités
    - N disque de données + 1 disque de parité
    - Principe : si 1 secteur est défectueux, son bit sera faux mais si ceux des autres disques sont corrects, avec le bit de parité, on peut reconstruire le bit faux
    - bit de parité par octet (RAID3) ou bloc (RAID4)
- Problèmes :
- Disque de parité très sollicité
  - Max. 1 disque de DT en panne
- Remplacé par RAID5 maintenant
- 

## RAID 5

- Volume agrégé par bandes à parité répartie  
La parité est distribuée tout au long des données, au lieu d'être placée sur un disque spécifique  
→ pour chaque bloc, 1 disque stocke la parité et les N autres stockent les données  
→ le disque de parité change pour chaque bloc

- RAID 6

Même principe avec N disques supplémentaires au lieu de 1



## RAID : combinaison

- RAID N + M = combiner les deux niveaux  
Pb : coûte autant de disques en plus !
  - RAID 01
    - Chaque disque est optimisé en RAID0 (bande)
    - Le tout est doublé (RAID1)
      - combine performance et sécurité (mais très coûteux)
  - RAID 10
    - Chaque disque est sécurisé par miroir (RAID1)
    - Le tout est stocké en RAID0
    - Plus fragile que 01 (1 disque → perte de la bande)
- Remplacement à chaud  
Disque non utilisé sauf en cas de panne  
→ recopie automatique

## Cours 10

### Entrées et sorties

L3 – Info32B

Année 2014-2015



Nicolas Sabouret – Université Paris-Sud

## Plan

- Principe général
- Architecture
- Gestion du matériel
- Interfaces : données, types d'accès
- Services : ordonnancement, tampon, cache, etc.
- Requêtes d'E/S
- Flux
- E/S en C et en Java

## Principe

- Contrôler les services connectés à l'ordinateur
  - Souris/clavier
  - Disque dur
  - CD-ROM

→ chaque périphérique a son propre mécanisme d'entrée-sortie (E/S ou I/O)
- Méthodes d'E/S
  - Rendre « transparent » les E/S pour le reste des processus de l'OS

## Pilote de périphérique

- 2 approches
  - Standardisation des dispositifs d'E/S
    - permet d'intégrer les méthodes d'E/S dans l'OS
  - Variété croissante des dispositifs
    - impossible d'intégrer
    - code « externe » au noyau de l'OS = pilote de périphérique
- Pilote de périphérique
  - Interface uniforme d'accès au sous-système d'E/S

(cf. appels systèmes pour accès O/S)

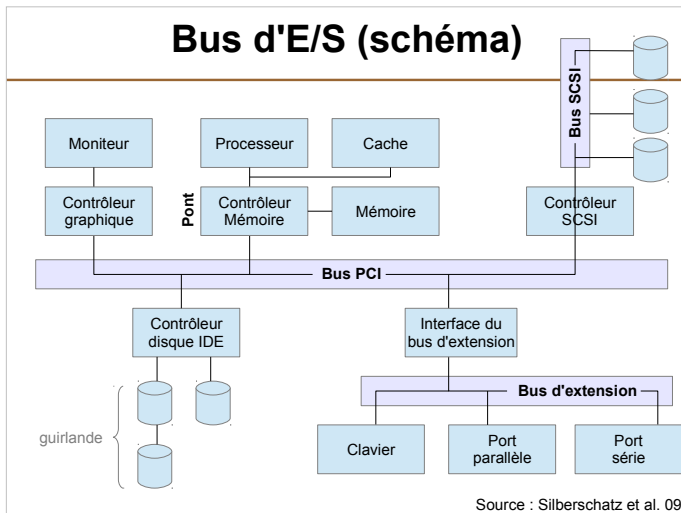
## Matériel d'E/S

- Périphériques standards
  - Stockage (disques, bandes)
  - Transmission (réseau, modems)
  - Interface utilisateur (écran, clavier, souris)
- Périphériques spécifiques
  - Manette (joystick d'avion) : transmission d'un signal manette en signal volets
  - Capteurs (médical) : transmission et agrégation de signaux
- Connexion des périphériques
  - Point de connexion périphérique-ordinateur : **port**
  - Connexion partagée : **bus**
    - protocole spécifique au bus

## Bus d'E/S

- Ensemble de lignes de communication
  - cf. « lignes électriques »
- Protocole
  - Ensemble de messages possibles
  - = application spécifiques de tensions aux lignes
- Bus PCI
  - Relie UC, mémoire et périphériques rapides
- Bus d'expansion
  - Périphériques lents (clavier, ports série/parallèle)
- Bus spécifiques
  - Ex : contrôleur SCSI
- Guirlande (daisy chain) = périphériques connectés entre eux
  - Fonctionne comme un bus

## Bus d'E/S (schéma)



## Contrôleur

- Définition
  - Composant électronique
  - Agit sur un port, un bus ou un périphérique
- Contrôleur simple : contrôleur série
  - Puce contrôlent les signaux des lignes (messages) du port série
- Contrôleur complexe : contrôleur SCSI
  - Protocole complexe → carte séparée (adaptateur)
  - Processeur, microcode, mémoire propre pour gérer les messages
- Contrôleur intégré au périphérique
  - Ex : contrôleur de disque avec microcode + processeur
  - implémente côté périphérique (disque) le protocole correspondant (IDE ou SCSI)
  - gestion propre au périph. (secteurs défectueux, tampon...)

## Commande du contrôleur

- Côté contrôleur
  - Registres données (x2 : en entrée et en sortie)
  - Registres pour signaux de contrôle
    - Registre d'état (commande en cours, terminée, données en attente d'être lues...)
    - Registre de contrôle (envoi de commandes, modification des paramètres (ex : taille du mot 7 ou 8 bits sur port série))
- Côté processeur
  - À l'aide de messages, écrit des bits dans ces registres
  - instructions d'E/S demandant le transfert d'un octet vers/depuis un port d'E/S (adresses d'E/S)
  - Exemple PC : 040-043 = horloge, 2F8-2FF = port série...
  - les lignes du bus sélectionnent le périphérique concerné puis font le transfert

## Contrôleur (suite)

- Projection mémoire
    - Mapper les registres du contrôleur avec des adresses mémoires (adressées par le processeur)
    - le processeur utilise des instructions standard pour E/S
  - Contrôleur graphique d'un PC
    - Ports d'E/S pour contrôle de base
    - Ensemble d'adresses = contenu de l'écran
    - le processeur écrit dans la mémoire
- Avantage :**  
accès plus rapide
- Inconvénient :**  
erreur d'adressage → vulnérabilité du périphérique

## Coordination

- N contrôleurs sur le bus
  - concept de « poignée de main » (problème type producteur-consommateur)
- Contrôleur : 2 bits du registre d'état
  - Bit occupé → modifié par le contrôleur
  - Bit commande → modifié par le système (signal bus)
  - 1. lire bit occupé jusqu'à ce qu'il soit à 0 (**scrutation**)
  - 2. mettre bit commande à 1
  - 3. le contrôleur remet le bit occupé à 1
  - 4. E/S via registres de données
  - pour chaque octet !
- Problème d'attente active
  - utilisation des interruptions

## Interruptions

- Gestion asynchrone des périphériques
- Ligne d'interruption
  - Ligne spécifique pour l'UC
  - Consultée après chaque instruction !
- Signal émis
  - sauvegarde de l'état courant
  - routine du gestionnaire d'interruption (adresse fixe en mémoire)
    - Déterminer la cause (cf. plus loin)
    - Servir le périphérique concerné (E/S)
    - Libérer l'interruption
  - poignée de main « à l'envers » (périph → UC)

## Contrôleur d'interruptions

- Problèmes
  - Déterminer la cause → parcourt des périphériques : long !
    - Vérifier bit occupé
    - Vérifier bit données en attente
  - Différer l'interruption (sections critiques)
    - 2 lignes d'interruption sur UC moderne
      - Non-masquable (interruption directe sur UC). ex : erreurs RAM vecteurs 0-31 sur 255 pour l'Intel Pentium
      - Masquable (pour les contrôleurs de périphériques)
  - Priorités d'interruptions
- Utilisation d'un contrôleur d'interruptions !

## Mécanisme d'interruption

- Adresse
  - Nombre (offset) désignant l'une des routines
  - Transmis sur N lignes « d'adresse »
    - le gestionnaire d'interruption sait quel périphérique a demandé quoi
- Limite
  - Nombre d'adresses (en général trop petit pour tous les périphériques)
    - chaînage d'interruptions
      - Adresse = renvoie vers une liste de gestionnaires
      - Invocation des gestionnaires pour « rechercher » celui qui peut servir la requête

## Interruptions (suite)

- Chargement
  - L'OS teste les bus matériel pour connaître les périphériques présents
    - installation des gestionnaires d'interruptions dans le vecteur d'interruption (mémoire)
- Interruption
  - Le périphérique prévient dès qu'une données est disponible, qu'une sortie est achevée, etc.
    - Ou pour les exceptions (erreur d'adressage, division par 0, défaut de page...)
    - l'OS devrait alors exécuter la routine correspondante
- Appels systèmes
  - Interruption logicielle (ou déroutement)

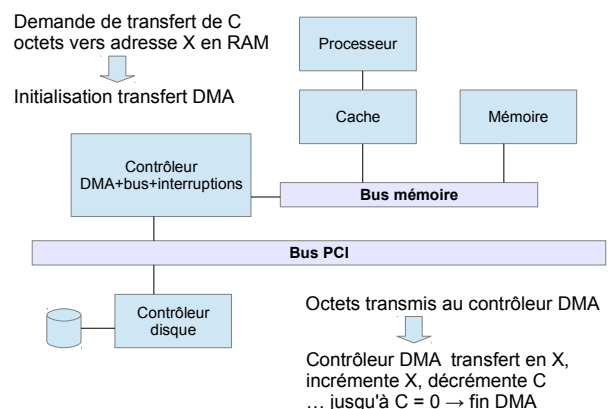
## Priorités

- Exemple : lecture disque
  - processus (lent) de recopie des données dans le tampon utilisateur
  - possibilité d'initier la prochaine E/S en parallèle (priorité plus élevée)
- Utilisation de threads
  - privilégier les threads de priorité élevée
  - intervalle de priorité réservé aux gestionnaires d'interruption

## DMA

- Transfert disque ↔ mémoire
  - Passer par le processeur (gestionnaire d'interruption) → coûteux
  - Direct Memory Access → passer par le bus directement
- Bloc de commande DMA
  - Pointeur source, pointeur destination, compteur d'octets
  - Stockés en mémoire
  - Pointeur vers le bloc dans le registre du contrôleur DMA
  - le contrôleur DMA s'adresse directement au bus mémoire dédié
- Gestion des bus
  - Lignes spécifiques bus PCI : DMA-request, DMA-acknowledge
  - Processeur interdit d'accès au bus pendant le DMA
    - vol de cycles

## DMA (suite)



## À retenir

### Mécanisme d'entrée-sortie

- Les bus
- Les contrôleurs
- Ports d'entrée/sortie avec registres
- Protocole « poignée de main »
  - Réalisé en boucle de scrutation ou à l'aide d'interruptions
- Transferts volumineux via le DMA

## Interface d'E/S

- Abstraction des différences entre les périphériques
  - identification de types de données
  - fonctions standardisées dans la couche d'E/S de l'OS
  - Chaque OS possède ses propres standards*
- Pilotes de périphériques
  - Masquer les différences pour la couche d'E/S de l'OS
  - Permettre l'utilisation de nouveau matériel sans attendre que le gestionnaire soit intégré au noyau

## Propriétés

- Transfert des données : Caractères (octets) vs blocs
  - Ex : terminal vs disque
- Positionnement : Séquentiel vs direct
  - Ex : modem vs CD-ROM
- Modèle : Synchrone vs asynchrone
  - Ex : bande vs clavier
- Partage : Dédié vs partageable (plusieurs processus en parallèle)
  - Ex : bande vs disque
- Direction : R, W, RW
  - Ex : CD-ROM, Écran, disque
- Vitesse : latence, temps de transfert, délais...

L'OS masque ces différences aux applications

## Accès direct

- Système d'échappement (escape route)
  - Permettre à une application d'envoyer directement une commande à un périphérique
- Ex Unix : appel système `ioctl`
  - Descripteur de fichier → lien application – périphérique (via le pilote)
  - Entier → commande du pilote
  - Pointeur → structure de données en mémoire (pour communication d'information)

## Blocs ou caractères

- Interface par blocs
  - Périphériques orientés blocs (ex : disque)
  - Appels système **read/write** + **seek** (si accès direct)
    - En général, interface du système de fichier (FS)
    - Accès direct possible (raw access)
  - Fichier représenté en mémoire
    - Tableau de caractères
    - Mécanisme d'accès type « mémoire virtuelle »
- Interface par caractères
  - Périphériques caractères (ex : clavier, modem, carte son)
  - Appels systèmes **get** et **put**
  - Bibliothèques pour accès par ligne, service de tampon...
    - Ex : `getc` → `gets` → `scanf` en C

## Réseau

### Interface à base de socket

- Créer une socket
- Connecter une socket locale à une adresse distante
  - branchée sur une socket créée par une application sur la machine distante
- Écouter les connexions sur la socket locale
- Envoyer et recevoir des paquets sur la connexion
- Lister (**select**) les socket prêtes à recevoir ou à émettre
  - attente passive

## Horloge

- Fonctions principales
  - Donner l'heure actuelle
  - Donner le temps écoulé
  - Déclencher une opération à une heure donnée (minuterie programmable)
    - interruption à une date donnée, éventuellement répétable
    - Ex : ordonnancement round-robin, purge des buffers du disque, timeout réseaux...
- Pas d'appel système standardisé
- Multi-utilisateurs

L'OS gère des « horloges virtuelles » à l'aide de la minuterie de l'horloge, transparent pour l'application

## E/S non bloquantes

- E/S bloquante
    - mise en attente (file « en attente »)
    - retour dans la file « prêt » lorsque l'appel système est terminé + récupération des données résultat
  - Besoin d'E/S non-bloquantes
    - Ex : lecture de données vidéo décompressées et affichée sur l'écran
      - on veut commencer la décompression sans attendre la fin de l'E/S !
    - Utilisation de threads et découpage des requêtes
    - Appel système asynchrone : rend la main + pointeur résultat
      - résultat ultérieur signalé par interruption ou attente active
      - Ex : **select** réseau
- OU

## Ordonnancement E/S

- 1 file par périphérique
  - Appel bloquant → entrée file
  - Ordonnancement (ex : lecture blocs disque SSTF)
    - réorganisation de la file à chaque entrée
- Compromis
  - Favoriser les applications urgente
  - Qualité de service (en particulier famine)
  - Requetes « temps-réel » : délai fixé
- Amélioration
  - Utilisation de tampons ou cache

## Tampon (buffer)

- Définition
  - Zone mémoire servant à stocker des données pendant leur transfert entre deux périphériques ou entre un périphérique et une application
- Objectifs
  - Gestion des différences de vitesse entre producteur et consommateur du flux de données
    - Ex : modem → disque (1000 fois plus rapide)
      - Accumulation dans le tampon puis purge sur le disque en seule opération
    - Double buffer : producteur dans B2 pendant que B1 est lu/purgé
  - Gestion des différences de volumes de données
    - Ex : réseau (fragmentation et réassemblage des messages)
  - Copie
    - Application modifie données pendant leur lecture
    - Le tampon stocke les données à copier, pas les données modifiées

## Ordres de grandeur

- Clavier = 1 ; Souris = 3
- Modem = 10k, imprimante = 30k
- Réseau = 100k
- Disque dur = 1M
- Bus SCSI = 2M
- SBUS = 10M
- Bus PCI moderne = 300M

## Cache

- Définition
  - Zone de mémoire rapide contenant des copies des données
- Tampon vs cache
  - Toute donnée en cache existe ailleurs (c'est juste une copie, contrairement au tampon qui peut être la donnée unique)
- Une même zone mémoire peut servir à la fois de tampon et de cache
  - Ex : données disques en tampon dans la mémoire → joue le rôle de cache !



## Mise en attente (spooling)

- Définition

Tampon contenant une sortie destinée à un périphérique incapable d'accepter des flux intercalés (ex : imprimante)

- L'OS récupère les sorties et les mets en attente dans le tampon (spool)

- 1 fichier par application
- Utilisation d'une file d'attente pour le périph.
- Appels systèmes de manipulation de la file  
lister, supprimer, etc.

- Implémentation

- Thread interne au noyau
- Démon (processus réveillé sur interruption)

## Erreurs

- Valeur de retour d'une entrée sortie

Permet à l'OS de savoir comment s'est déroulée la requête et comment s'adapter en cas d'erreur (relance, attente, arrêt...)

- Implémentation

- 1 bit → ok vs échoué
- errno (Unix) → type d'erreur
- Clef d'erreur (SCSI) → informations détaillées (quel paramètre a causé l'erreur)

## Requête E/S

Exemple : requête lecture bloquante

- Appel système **read** (bloquant) sur un fichier préalablement ouvert
- Vérification des paramètres
- Recherche des données dans le cache tampon  
→ renvoi au processus et fin de la requête d'E/S
- Lancement de l'E/S physique → processus mis en « attente »
- E/S dans la file d'attente du périphérique + envoi de la requête au pilote de périphérique (sous-programme ou appel noyau)
- Le pilote alloue un espace tampon et programme l'E/S
- Envoi des commandes au contrôleur de périphérique (écriture dans les registres de contrôle, via les lignes du bus)
- Le contrôleur de périphérique demande au matériel d'effectuer le transfert de données (DMA + interruption ou scrutation)
- Gestionnaire d'interruption (table des vecteurs d'interruption) → lancement de la routine :
  - Stocker les données
  - Prévenir le pilote de périphérique
  - Fin de l'interruption
- Le pilote prévient de la fin de l'E/S, les données + code de retour sont transmises à l'adresse prévue pour le processus demandeur et le processus retourne dans la file « prêt »

## E/S en C

- Par octets

`fgetc` → `fgets` (chaîne terminée par '\n')

`fputc` (et `fputs`)

- Groupes d'octets

`fread` et `fwrite`

→ pointeur buffer, taille d'un élément, nombre d'éléments

- Entrées et sorties formatées

`fscanf` et `fprintf` → dans un flux

`sscanf` et `sprintf` → dans une chaîne de car.

## E/S en Java

- Par octets (équiv. `char` en C)

- `InputStream` → fonctions de lecture
- `OutputStream` → fonctions d'écriture
- Classe abstraite contenant les fonctions d'écriture  
`FileOutputStream` → écriture dans un fichier

- Par caractères :

- `Reader` → lecture et `Writer` → écriture
- Classes abstraites  
`BufferedReader` → lecture de chars dans un tampon

- Passer de l'un à l'autre :

- `InputStreamReader`  
Ex : lecture dans un fichier, en tampon  
`toto = new InputStreamReader(new FileInputStream(File))`
- `OutputStreamWriter`

Création → open

Merci !