Les Systèmes d'Exploitation

Introduction

Gérard Padiou

Département Informatique et Mathématiques appliquées
ENSEEIHT

Septembre 2010

plan

Plan

La raison d'être des systèmes d'exploitation

Origine

- Idée : la gestion de l'exécution des programmes sur un ordinateur peut être gérée par ...

  un programme (un peu spécial néanmoins)

- Problème de base d'un système d'exploitation : assurer l'exécution de programmes divers et variés
  - faire en sorte que l'exécution soit la + rapide possible :

    ☞ il faut utiliser au mieux les ressources ;
  - résister aux erreurs du programmeur ... : le programme exécuté peut être erroné (exemple : il boucle)
  - assurer la gestion des données utilisées/produites pas les programmes :

    ☞ notion de fichiers ;

Qu'est-ce qu'un programme « exécutable » ?

Sous quelle forme mémoriser, représenter un programme exécutable ?

Notion de programme exécutable

Deux formats

- Approche interprétative : Un programme exécutable est représenté sous forme de lignes de texte dans un langage.
  Il n'est bien sûr exécutable que par un interpréteur capable de décoder les instructions (commandes) qu'il contient.
- Approche directe : Un programme exécutable est représenté sous un format binaire contenant des instructions d'une machine donnée.
  Il est le résultat d'une compilation et d'une édition des liens.

La notion de format binaire exécutable ou ré-éditable

Principes

- Spécifique d'une famille de systèmes : Unix, Windows, . . . ;
- Qualifié de portable : format identique $\forall$ le processeur ;
- Deux formes : exécutable et ré-éditable (statiquement ou dynamiquement).

Exemple

- Les vieux formats Unix :
  ☞ Formats a.out et COFF (Common Object File Format)
- La famille Unix (Linux, Solaris, Irix, System V, BSD)
  ☞ Format ELF (Executable and Linking Format) :
- La famille Windows :
  ☞ Format PE (Portable Executable Format) et COFF.

Les formes de programmes

Programme source

```
int main() { printf("coucou") ; }
```

Fichier binaire exécutable

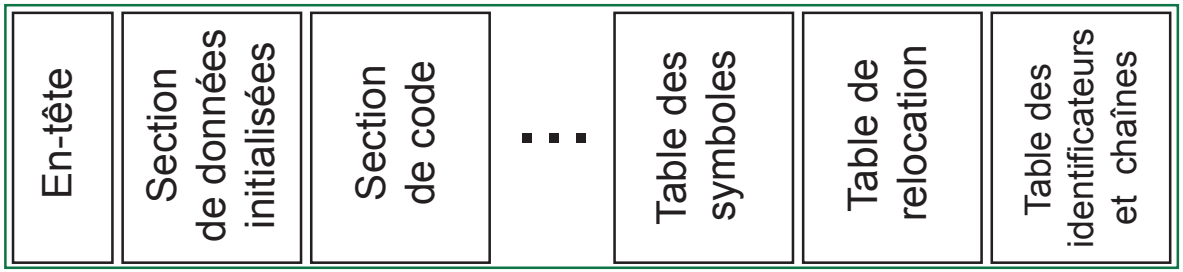
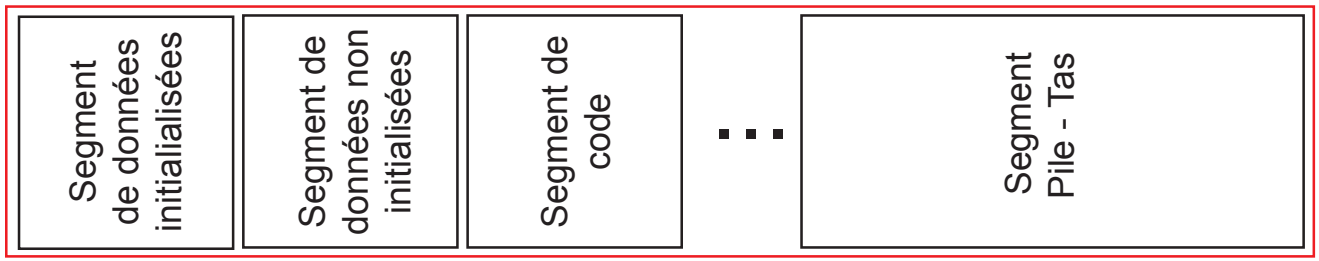


Image en mémoire



Une longue histoire ...

Quelques systèmes

- avant les PC's :
  ☞ OS/360, CP-67, MCP, MVS, Multics, GECOS, ...
- après les PC's :
  ☞ CP/M, Unix, MS-DOS, Windows, Linux, ...

Evolution

- Unification progressive des concepts ;
- Quelques évolutions marquantes : mémoires virtuelles, temps partagé, multi-programmation, etc

Un « mal » nécessaire

☞ Le système d'exploitation est indispensable

- pour gérer efficacement l'exécution des programmes ;
- pour assurer le confinement des erreurs en cours d'exécution ;
- pour gérer efficacement les ressources : processeurs, mémoires, etc ;
- pour offrir une interface d'utilisation «agréable» aux usagers ;
- pour garantir une protection des données entre usagers ;

Un programme difficile à développer

Des programmes complexes qui ont eu un impact

- ☞ sur le génie logiciel :
  - Modularité ;
  - Couches de logiciel ;
  - Machine virtuelle ;
  - Parallélisme ;
- ☞ sur les architectures matérielles
  - Notion de mode d'exécution ;
  - Mécanismes de protection mémoire ;
  - Notion d'interruption, de déroutement.

Définition d'un système d'exploitation

Fonctionnalité

Un ensemble de programmes
qui apportent un environnement d'exécution
pour exécuter des programmes en assurant la sécurité des données

☞ Deux contextes d'usage :

- Contexte de développement : environnement de développement de nouveaux programmes ;
- Contexte d'exploitation : support d'exécution d'un ensemble d'applications dédiées : bureautique, transactions bancaires, contrôle d'un réacteur, video à la demande, etc.

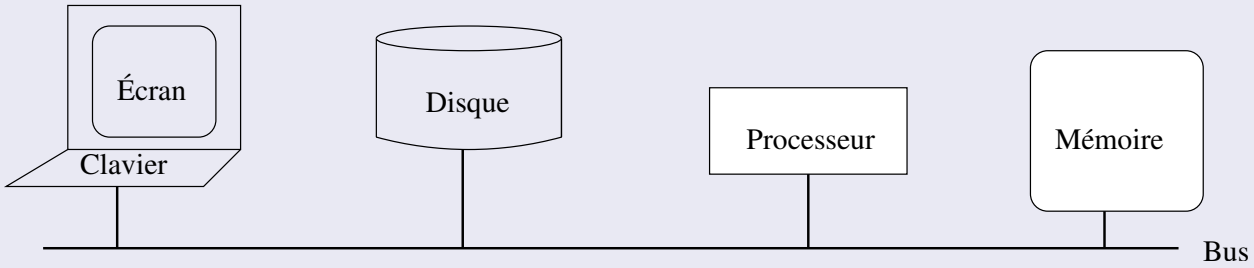
Un mélange des deux est toujours possible

Plan

Un système d'exploitation minimaliste

Architecture matérielle et spécification du système

Architecture matérielle



☞ Exécution « continue » d'instructions

Spécification du système

- Interface simple textuelle via un écran-clavier ;
- Soumission de requêtes d'exécution de programmes (commandes) ;
- Des programmes exécutables «commandes» sont prêts dans des fichiers sur disque.

L'interface de dialogue avec l'utilisateur

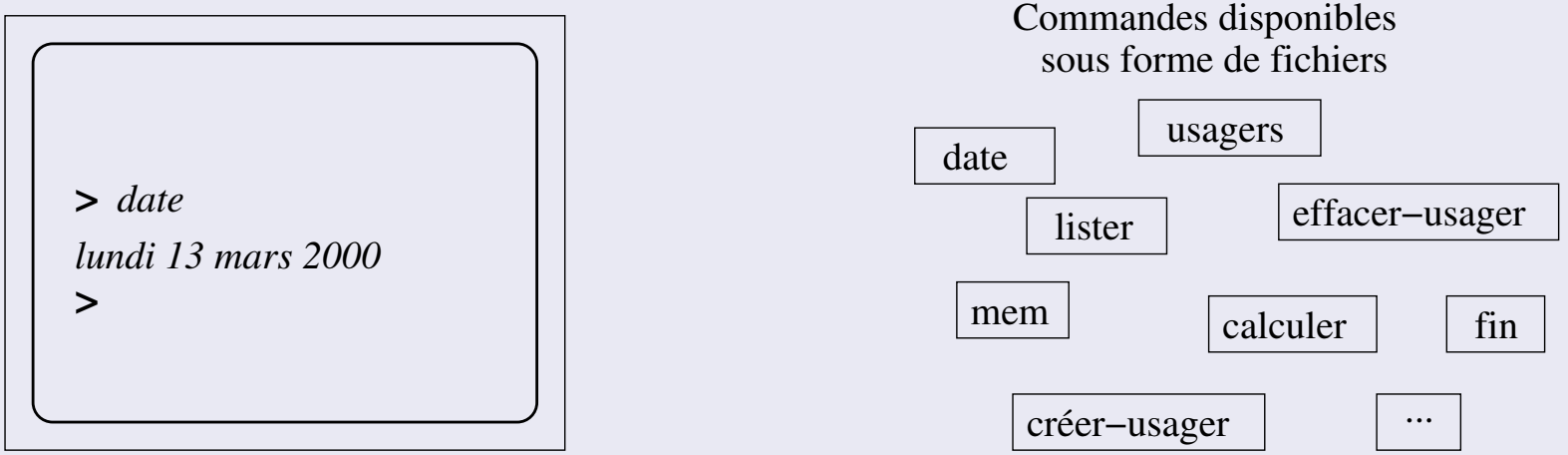
Quelques commandes possibles

lister	liste les commandes de base qui existent ;
date	affiche la date présente ;
expliquer <com.>	affiche la fonction de la commande en paramètre ;
calculer <expr.>	calcule l'expression arithmétique fournie en paramètre et affiche le résultat ;
mem	affiche l'espace mémoire centrale disponible ;
fin	arrêt du système ;
...	

L'interface de dialogue avec l'utilisateur

Une vision des choses

Machine abstraite usager



- Réactivité : commande → exécution d'un programme ;

Le traitement des commandes

L'interpréteur de commandes : algorithme

```
Interpréter() {
   while (true) {
     écran.Afficher(">");
    Commande c = Ligne.Lire();
    if (c.valide()) c.Exécuter();
    else écran.Afficher(c.Erreur);
  }
}
```

Le traitement des commandes

Exécuter une commande

```
Exécuter() {
    Proc prog = Programme.Charger(c.nom) ;
    prog(c.arg) ;
}
```

- Il faut charger le programme en mémoire centrale ;
- Simple relation de programme à sous-programme

  ☞ Appel procédural ? Plutôt :

```
Process p = Lancer(prog,c.arg) ;
AttendreFin(p) ;
```

- Et si le programme usager boucle ? ou contient une erreur ?

Plan

Un programme . . . mais avec quelques spécificités

- Réactif : requête $\rightsquigarrow$ réponse ;
- Indispensable : sans lui, difficile de faire exécuter un programme à une machine ;
- Service commun utilisable par tous les autres programmes ;
- Cohabitation de programmes en mémoire centrale ;
- Robustesse vis-à-vis des programmes exécutés ;
- Parallèle : entrelace l'exécution de plusieurs programmes ;
- De taille «respectable» ;
- Si possible discret ;
- Si possible très fiable ;

Un programme . . . mais avec quelques spécificités

Réactivité

Objectif : assurer l'enchaînement des commandes soumises

- Si erreur d'exécution d'une commande :
  ☞ arrêter et passer à la suivante
- Si une commande dure . . . trop longtemps
  ☞ arrêter et passer à la suivante

Objectif : garantir un temps de réponse acceptable

- Allocation optimisée des ressources (processeur, mémoire, entrées/sorties)

Le « système » contrôle, optimise, supervise l'exécution

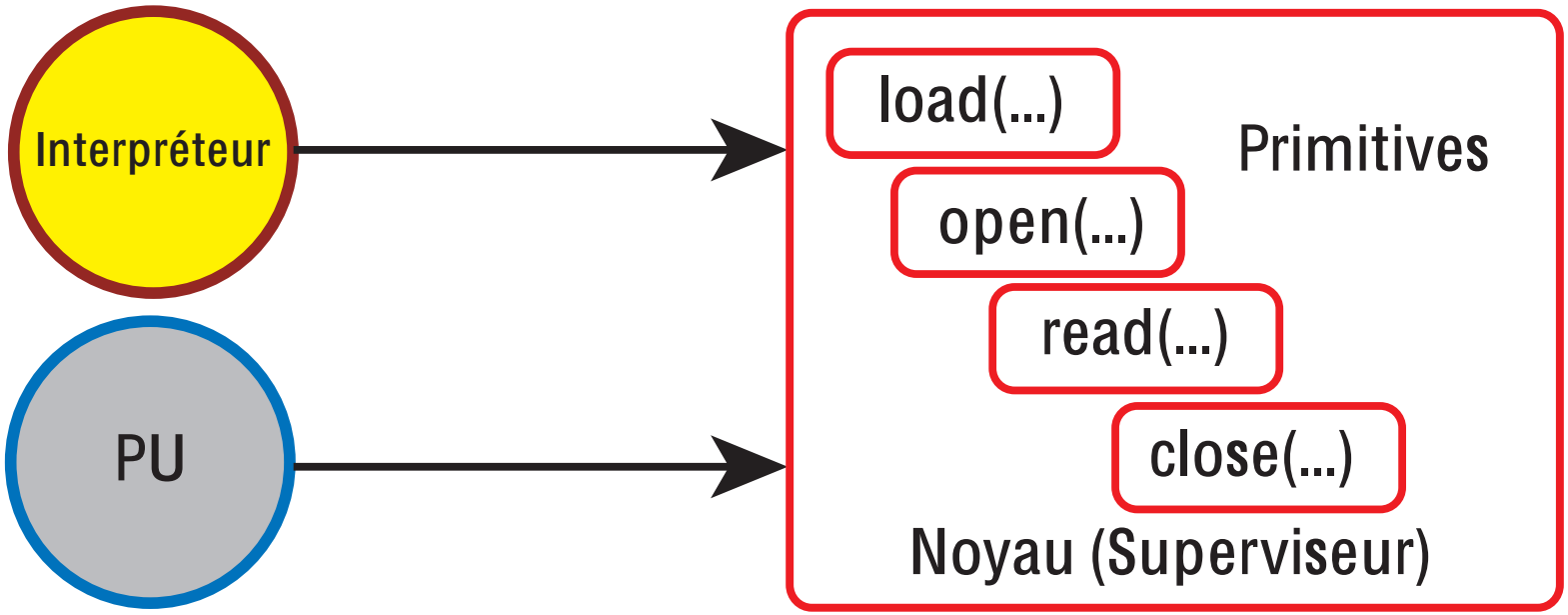
Un programme . . . mais avec quelques spécificités

Indispensable

- Le noyau du système doit être le premier programme chargé en mémoire ;
- Un programme spécifique le précède cependant :
  ☞ chargeur initial (loader/debugger) en mémoire morte (PROM)
- Le noyau est lui-même un exécutable :
  ☞ il occupe de la place en mémoire centrale
- Après initialisation, le contrôle passe à un programme de connexion (login)
- Puis, l'usager connecté passe sous le contrôle de l'interpréteur de commande.

Un programme . . . mais avec quelques spécificités

Service commun utilisable par tout programme



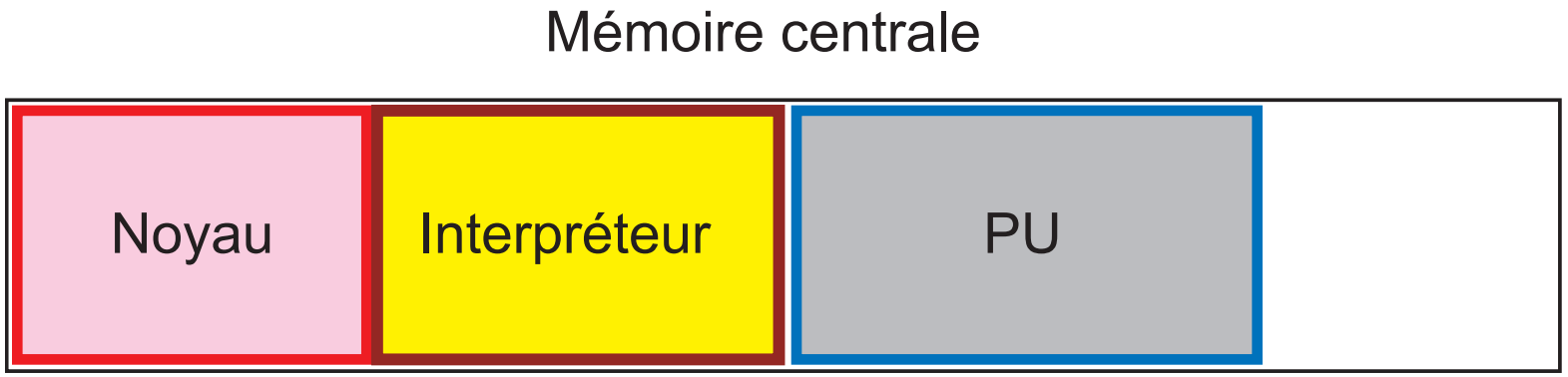
Un programme . . . mais avec quelques spécificités

Service commun utilisable par tout programme

- Il implante une machine abstraite « système » ;
- Tout programme exécutable peut faire appel aux « instructions » du noyau durant son exécution :
  ☞ primitives du noyau
- Ces primitives implantent une machine « système » permettant d'utiliser des concepts de haut niveau : processus, exceptions, mémoire virtuelle, fichiers, pipes, sockets, etc
- Ces primitives définissent une machine abstraite indépendante de la configuration matérielle sous-jacente ;
  ☞ Portabilité des programmes
- Ces primitives sont du code partagé par tout programme qui s'exécute.

Un programme . . . mais avec quelques spécificités

Cohabitation avec d'autre « programmes »



Soulève un problème :

- de protection mémoire :
  ☞ ne pas détruire le programme noyau ;
- d'accès au noyau : ☞ contrôle des appels aux primitives
- Asymétrie Superviseur/Programme : + de pouvoir au superviseur !

Cohabitation avec d'autre « programmes »

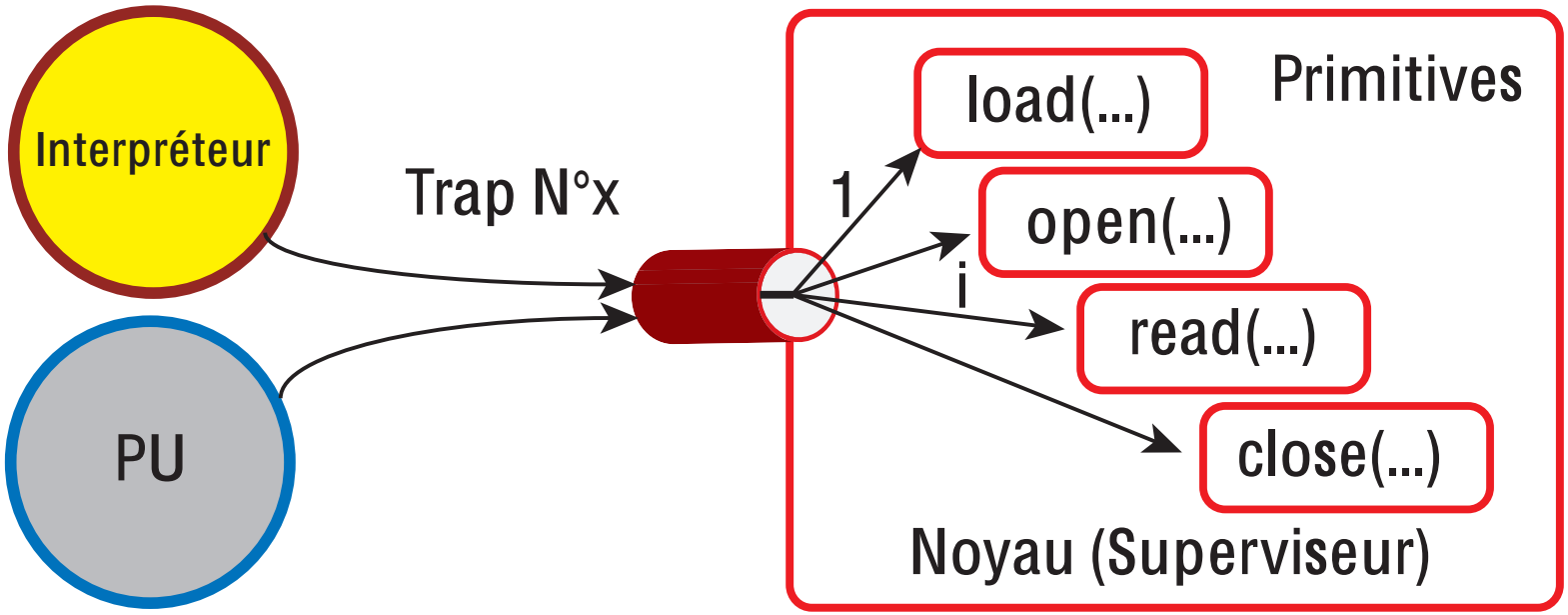
Protection mémoire

Assurer le respect de « chacun chez soi »

- Les références mémoire d'un programme au cours de son exécution doivent rester dans une zone fixée :
  celle où se trouve l'image du programme qui s'exécute
- Problèmes :
  - Il faut quand même pouvoir appeler le noyau
    ☞ autoriser les branchements dans la zone noyau
  - Le noyau doit pouvoir accéder à toute zone mémoire ;
- ☞ Nécessité de mécanismes de protection mémoire.

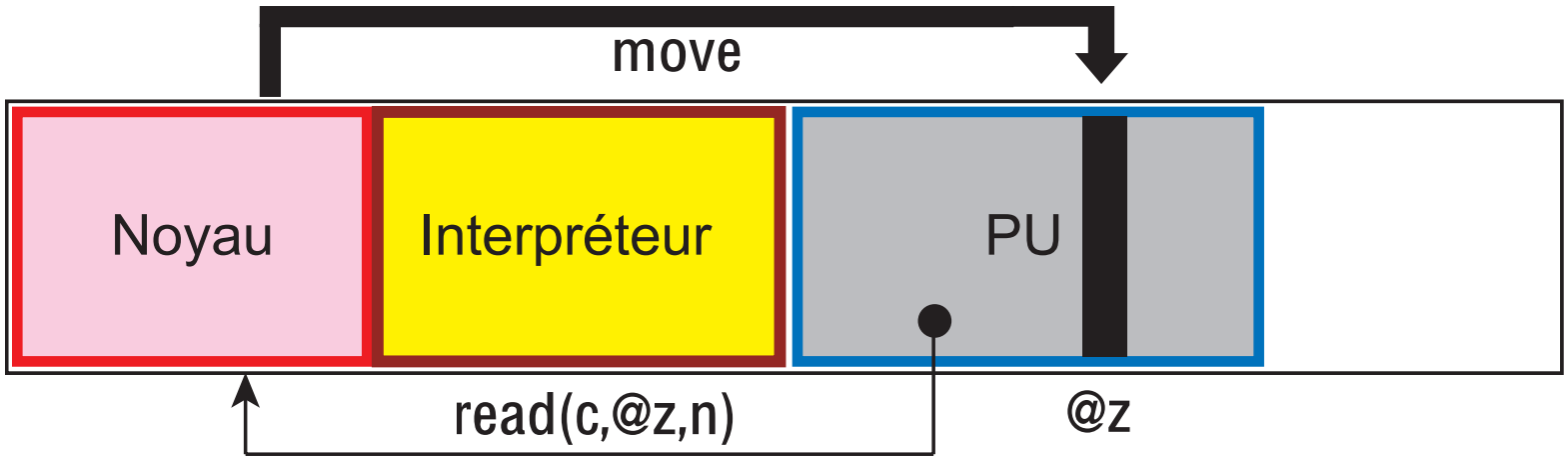
Cohabitation avec d'autre « programmes »

Contrôle des appels aux primitives



Cohabitation avec d'autre « programmes »

Asymétrie Noyau/Programme



☞ Mécanisme de mode d'exécution Superviseur/Programme

Un programme . . . mais avec quelques spécificités

Robustesse vis-à-vis des programmes usagers

Confinement des erreurs locales à une exécution

L'enchaînement de l'exécution des programmes doit être assurée même si des programmes contiennent des erreurs.

- Un programme boucle . . .
- Un programme contient un code instruction inexistant ;
- Un programme tente d'écrire dans la zone « noyau » ;
- . . .

Réaction

Par défaut : Abandonner l'exécution du programme fautif.

Un programme . . . mais avec quelques spécificités

Parallélisme

Même si un seul processeur central . . .

- Durant son exécution, un programme n'a parfois rien d'utile à faire : il attend un événement, une ressource ;
- Les échanges mémoire centrale $\leftrightarrow$ périphériques peuvent être exécutés en parallélisme réel avec l'exécution d'un programme sur le processeur central ;

Exemple

- un programme appelle la primitive read pour lire la frappe de caractères au clavier ;
- un programme appelle la primitive wait pour attendre la FIN de l'exécution d'un autre.

Un programme . . . mais avec quelques spécificités

Complexe, de taille « respectable »

- Historiquement, les systèmes d'exploitation ont été longtemps les programmes développés les plus complexes ;
- Ce sont les premiers programmes gérant le parallélisme ;

Si possible discret

- Le noyau consomme des ressources : mémoires, processeur . . .
- Primitives les plus rapides possibles

Si possible très fiable

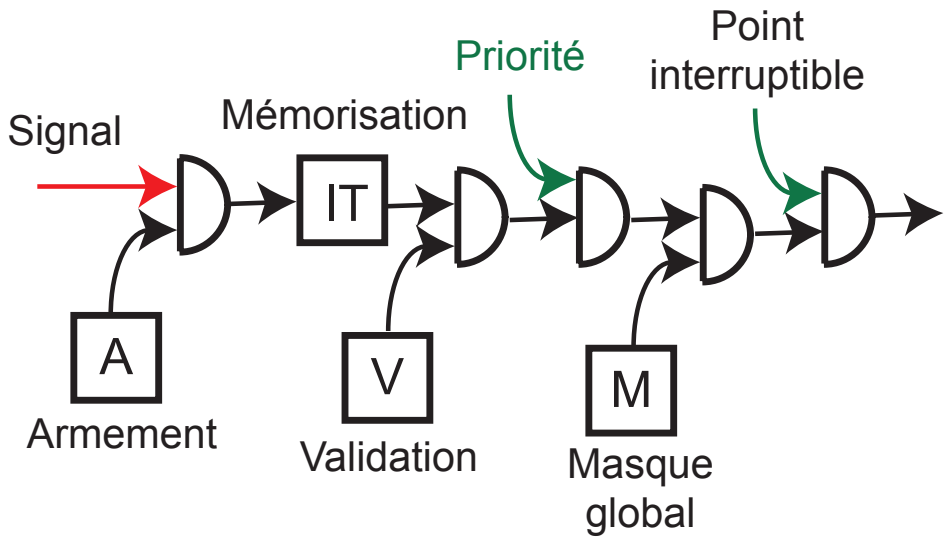
- C'est le problème de base : un système ne doit pas s'arrêter . . . avant qu'on l'arrête.

Plan

Mécanismes de base

Les interruptions

Evénements externes au programme en cours



- Armement : interruption mémorisée (sinon perdue) ;
- Validation : interruption acceptée (sinon retardée) ;
- Masque global : ininterruptibilité du processeur ;
- Priorité éventuelle entre interruptions.

Mécanismes de base

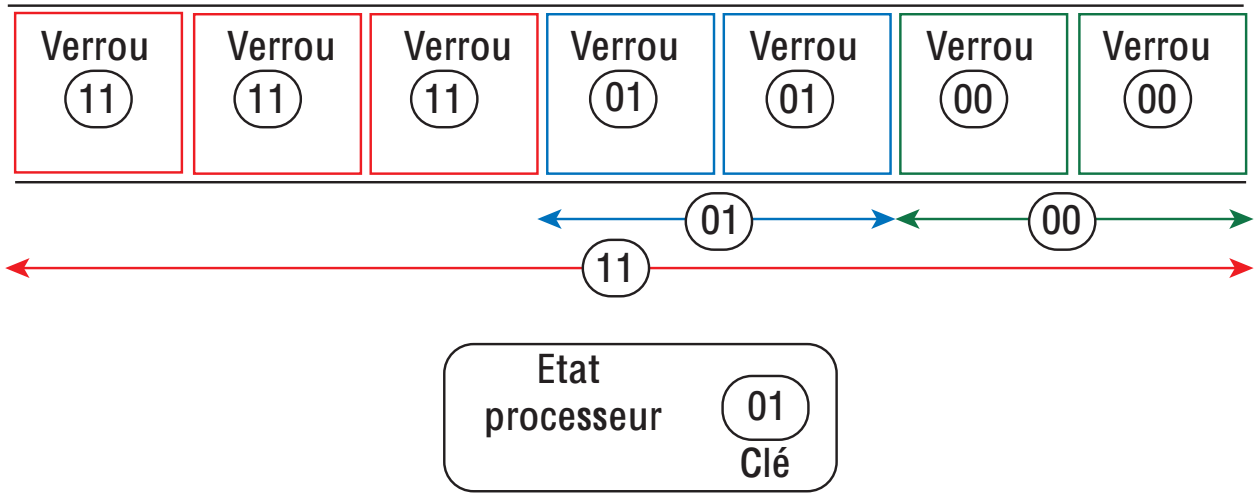
Les déroutements

Evénements internes au programme exécuté

- Erreur d'exécution au niveau du processeur ;
- Réaction immédiate : déroutement vers une routine ;

Mécanismes de base

Protection mémoire : un exemple élémentaire



- Besoin d'une clé « passe-partout »
- Nécessité d'interdire la programmation des verrous et du registre clé : ☞ Modes d'exécution

  Solution actuelle : les Unités de Gestion Mémoire (UGM)

Mécanismes de base

Les modes d'exécution

Objectif

- Protection contre les erreurs des « usagers »
- Droits distincts entre superviseur et programme applicatif
- Instructions privilégiées ;
- Au moins 2 modes : superviseur/programme ;

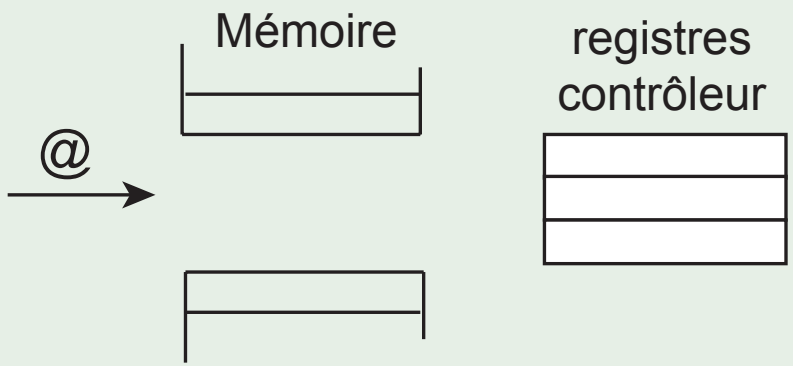
☞ Généralisation : anneaux de protection

Les échanges (entrées/sorties)

Deux technologies

- Par instructions privilégiées (ex : Intel) PMIO (Port-mapped Input/Output)
- Par projection en mémoire (ex : Motorola) MMIO (Memory-mapped Input/Output)

Exemple



Plan

Conception d'une machine « système »

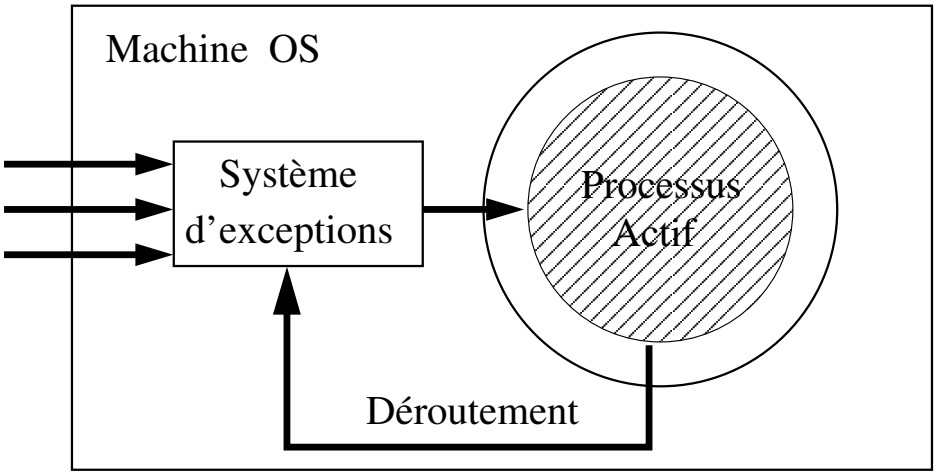
Machine support d'exécution des programmes

Objectif

- Fournir un support, un environnement d'exécution indépendant du matériel réel ;
- Nécessite d'abstraire les mécanismes de base d'une configuration matérielle :
  - ☞ son système d'exceptions (interruptions et déroutements)
  - ☞ ses spécificités de gestion des entrées/sorties avec les périphériques.

La machine « système »

Machine support d'exécution des programmes



Abstraire les interruptions et déroutements

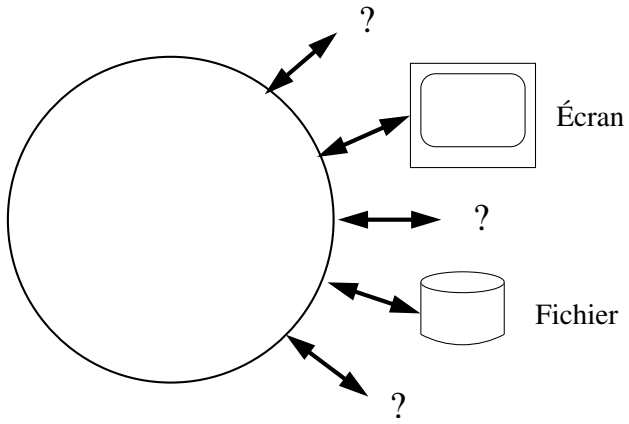
- Définition d'un système logique d'exceptions ;
- Recherche d'une interface générique et simple ;
- Définition de primitives de programmation.

Exemple : les signaux Unix

La machine « système »

Machine support d'exécution des programmes

☞ Chaque couche implante une machine abstraite ayant une interface caractérisée par un jeu d'opérations.

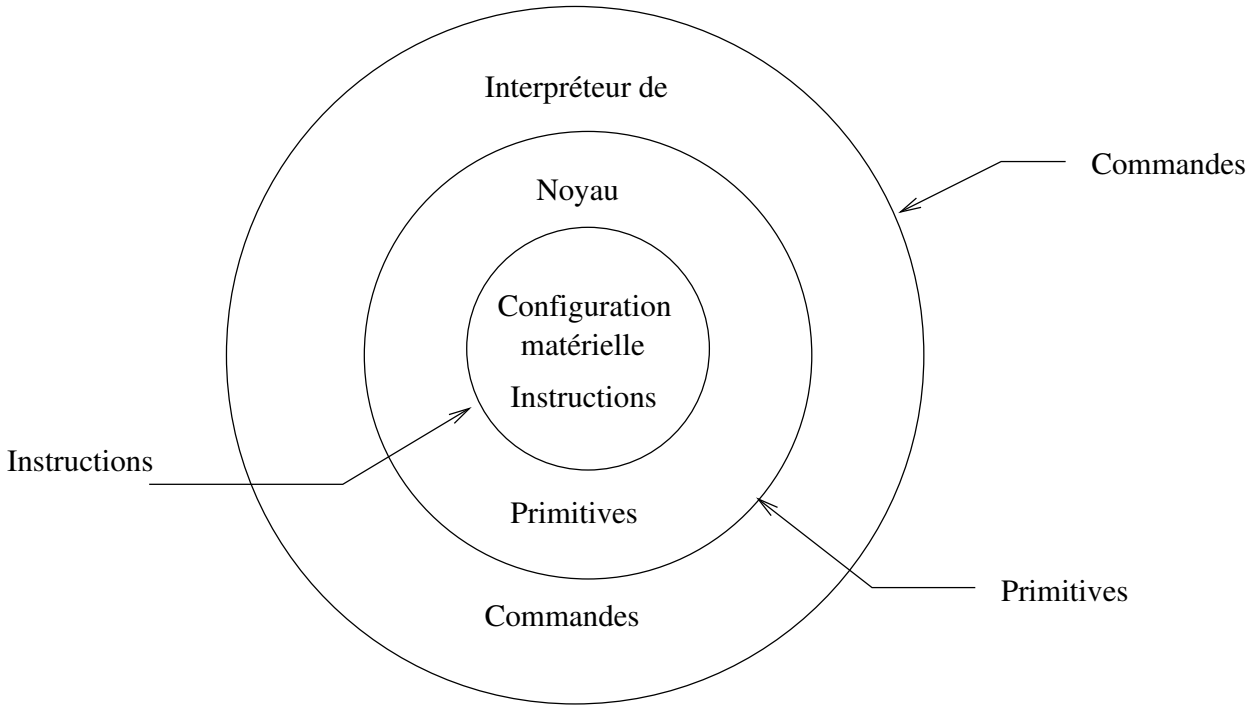


Environnement de communication

- Abstraction de la communication : flots de données ;
- Connexion dynamique durant l'exécution ;
- Interface standardisée quel que soit le périphérique réel final.

Principes de conception

Structure en couches



Principes de conception
Facteur d'échelle

☞ Dans la hiérarchie des couches successives, les opérations sont de plus en plus complexes.

Niveau	Ressource	Type d'opération	Durée
Matériel	Processeur	Instruction	μsec
Noyau	Primitives	Sous-programme	$>$ millisec
Interpréteur	Commandes	Programmes	quelconque

Principes de conception

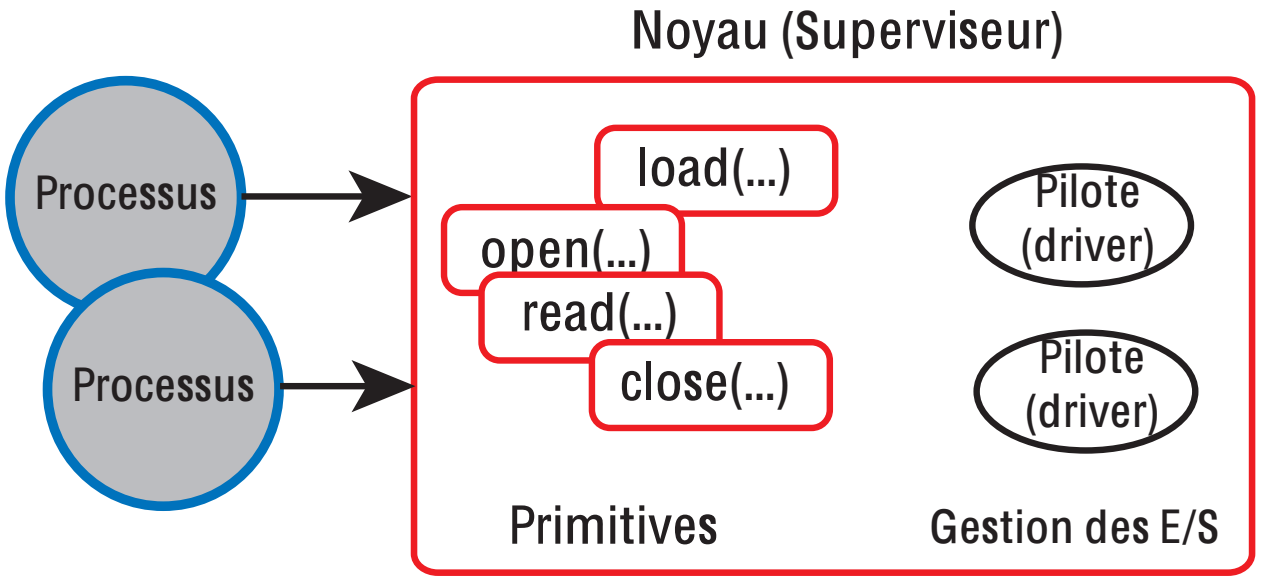
Le principe de liaison dynamique

- Connexion dynamique aux ressources logiques :
  ☞ fichiers, pipes, ports, etc
- Objectif : optimiser l'usage des ressources ;
- Approche : retarder la liaison effective le plus tard possible
  ☞ Delay binding time

Exemple

Attendre la première référence à une instruction d'un programme pour charger en mémoire centrale cette instruction.
☞ Mémoires virtuelles avec pagination à la demande

Conclusion



Ensemble de sous-systèmes

- Gestion des Entrées/Sorties (Basic Input/Output System) ;
- Gestion des exceptions ;
- Gestion des processus et des fichiers ;
- Gestion de la mémoire virtuelle.