Systèmes d'Exploitation

Un Système d'Exploitation (SE): ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les usagers.

Bibliographie : divers livres sur le sujet, par exemple

S. Krakowiak : Systèmes d'Exploitation

A. Tamenbaum : Architecturedes ordinateurs

A. Tamenbaum : Systèmes d'Exploitation des ordinateurs

G. Nutt : Operating Systems

(et beaucoup d'autres...)

Aperçu historique

- Fortement liés à l'architecture des machines (matériel)
- L'évolution du matériel $\Rightarrow$ évolution des SE
- On distingue quatre générations :

Débuts(1946-55)

- les premières machines: 20 000 tubes à vide, taille gigantesque;
- la plus connue = l'ENIAC: 20 tonnes, 160 m^2, 18000 tubes, 6000 commutateurs, 70000 résistances, ...
- pas (ou très peu) de mémoire (lecture et chargement – par des interrupteurs – d'une instruction dans un registre + exécution)
- un seul programme à la fois, écrit en langage machine,
- chargement manuel des registres (bit par bit),
- manoeuvre d'interrupteurs (load de programme sur plusieurs jours!)

→ on met dans le compteur ordinal (PC) l'adresse de la première instruction

→ on suit l'exécution à l'aide de voyants lumineux qui représentent les valeurs des registres

→ si problème ⇒ arrêt, consultation des registres.

- on gère soi même les opérations d'entrée sortie (E/S) ⇒ il faut bien connaître les périphériques)
- on écrit une fois pour toutes des *modules* (pas encore compilé) qui réalisent certaines opérations ⇒ Ex : chargeurs, assembleurs, éditeurs de liens, ...
- les fonctions mathématiques regroupées dans des bibliothèques de programmes
- drivers de périphériques (utilisables par d'autres programmes)
- plus tard :
  - compilation, utilisation de langages évolués (Fortran, Cobol, ...).
  - clavier hexadécimal,
  - lecteur de cartes/ruban perforé.

Transistors, traitement par lot (1955-65)

transistor: une grande évolution

⇒ industrialisation / commercialisation de la fabrication d'ordinateur (au lieu des machines conçues individuellement)

- coût élevé : plusieurs M$
- utilisation de Fortran et Assembleur
- programmation sur carte perforée :
  - l'opérateur charge les cartes du compilateur Fortran
  - Le compilateur transforme le programme en e.g. langage d'assemblage
  - Si erreur ⇒ vidage du contenu de la mémoire (dump) sur papier pour débugguer
- Il faut charger les cartes de l'assembleur
- Programme traduit en langage machine

Ces manipulations prennent du temps → l'unité centrale mal utilisée

⇒ solution : traitement par lot (batch processing) :

regrouper et traiter les travaux similaires

e.g. les programmes Fortran regroupés dans un lot.

- L'enchaînement automatique des travaux se fait par un **moniteur d'enchaînement** ou "moniteur résident".
- Le moniteur est en fait l'ancêtre des S.E.
- cartes de contrôle insérées au début/fin des paquets
- Il a une place privilégiée en mémoire (zone protégée du moniteur par opposition à la zone utilisateur),
- On remarque bien (dans le traitement par lot) la différence de vitesse entre le calcul et les E/S:

e.g. 50 secondes pour lire 250 cartes, 2 secondes pour les exécuter!

⇒ Solution : gagner du temps en faisant recouvrir la lecture d'un programme avec l'exécution d'un autre.

⇒ Les périphériques : DMA (ou circuits dédiés aux E/S)

3- Circuits intégrés (IC) et la multi programmation (1965-80)

- Les IC remplacent les transistors → baisse des coûts
- problème de compatibilité: 1 hardware → 1 SE
- nouveaux matériels → nouveaux programmes.
- 1971 : 8008 d'Intel (après 4004)
- IBM lance (avec OS/360) l'idée d'un système unique qui s'adapte à tout type de machines
  → différents hardware mais 1 SE.
- Arrivée des disques:
  - amélioration
  - les bandes étaient séquentielles et lentes
  - Avec les accès par secteur (sur disques), on peut mieux recouvrir les (phases de) E/S d'un programme avec les calculs d'un autre
  - La **Multi Programmation**

La Multi Programmation :

- Plusieurs programmes sur disque et en mémoire
- Quand un programme décide d'une E/S, l'unité centrale lance l'opération et sans attendre la fin de l'E/S, lance l'exécution d'un autre programme.
- Si aucun programme est prêt, on en charge un autre depuis le disque.
- Le passage d'un programme à un autre est fait par des signaux d'interruption (sauvegarde de contexte,...)
- La multi programmation permet l'interactivité entre le programme et l'utilisateur : **Temps Partagé**

  $\Rightarrow$ Chaque programme utilise l'UC pendant une tranche de temps avant de passer la main à un autre.
- Les usagers sont (le peuvent !) devant leurs terminaux.
- **MULTICS** a été conçu dans cet esprit (système pionnier mais peu répandu)

  Multics : multiplexed Information & computing service

- Miniaturisation des IC

  ⇒Mini ordinateurs (DEC PDP-1 en 1961).

- Prix: environ 120 000 $ (environ % 5 d'un IBM 7094).

- Le développement des PDP caractérise cette génération :

  ⇒ c'est sur un PDP-7 que les premières versions d'**UNICS** sont nées .

  ⇒ **UNICS** rebaptisé **UNIX ,** réécrit en C, implanté sur tous ordinateurs (VAX, SM90, SUN, ...)

4- La miniaturisation et les réseaux d'ordinateurs (1980-..)

- Caractéristique: les micros ordinateurs
- Système interactif, graphique, logiciels (MS DOS, IBM PC, INTEL 80/88) ...
- Système graphique (Appel, ATARI, AMIGA, Microsoft, ...)
- Développement des systèmes Multiprocesseurs, réseaux (ARPANET, SNA, TRANSPAC, ETHERNET...) pour relier les ordinateurs (éloignés)
- meilleur délai d'acheminement des messages
- plus grande possibilité de calcul, de stockage,...
- INTERNET, WEB , ...
- Multiprocesseurs et parallélisme , Transputers, Hyper-Threading ...

Typologie des SEs

1- Système à soumission de travaux (système batch)

- A chaque instant; 1 seule tâche est exécutée
- La tâche dispose de toutes les ressources
- Ex : micro-ordinateurs (primaires) ou un terminal relié à un hôte distant

2- Système mono/multi tâches

- Multi tâches : le temps d'utilisation du processeur est réparti entre plusieurs tâches
- Les ressources affectées alternativement aux tâches selon les priorités (mécanisme transparent aux utilisateurs)
- Ex. de multi tâches : Unix
- Ex. de mono tâche : CPM, DOS

3- Système mono/multi utilisateur (Unix, Win, MacOs, ..)

- Un système mono utilisteur (mais mono ou multi tâches) traite les commandes d'un seul utilisateur.

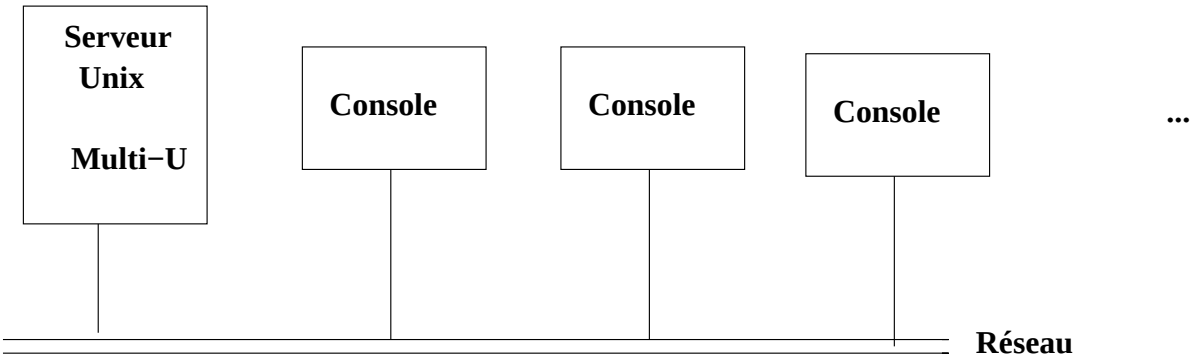
  $\Rightarrow$ le SE est multi tâches

4- Système mono/multi traitements

- plusieurs processeurs, multi tâches
- répartition des tâches entre les processeurs, en fonction de la spécialité ou de la charge $\Rightarrow$ optimisation des performances
- Unix est multi tâches, multi utilisateur , voire multi traitement

$\Rightarrow$ **Multi tâches = multi programmation** $\neq$ **multi traitement**

Exemple UNIX :

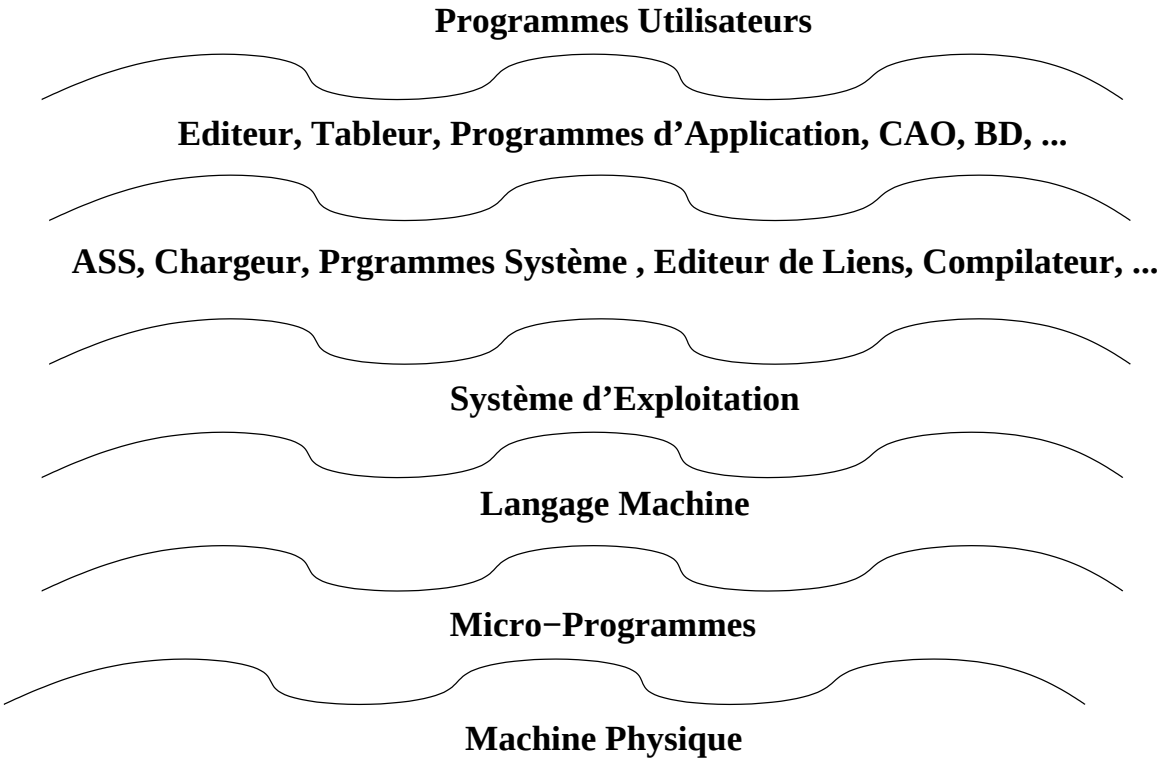


Objectifs et fonctions essentielles d'un S.E.

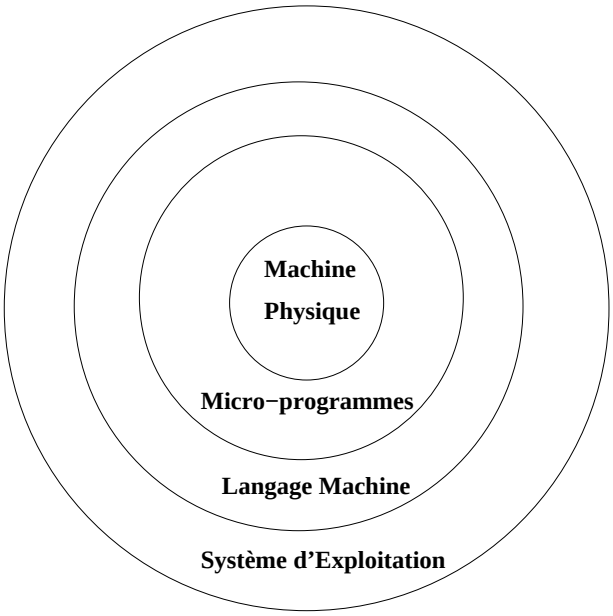
1- Construire, sur la machine physique telle qu'elle est livrée, une machine VIRTUELLE plus facile d'emploi et plus conviviale.

2- Prendre en charge la gestion complexe des ressources en optimisant leur utilisation et permettre leur partage entre les utilisateurs (fig 1.4 SE-2bis).

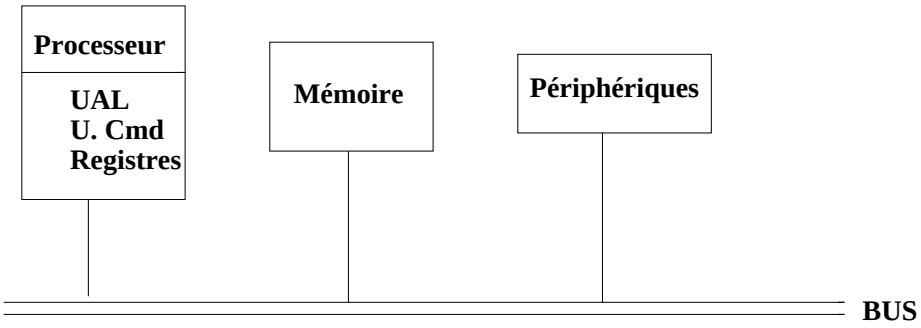
⇒ Les systèmes d'exploitation modernes sont conçus sur une **architecture par couches.**



Architecture par couches :



niveau 0 : la machine physique : *bus, mémoire, UAL, périphériques physiques, ...*



niveau 1 : micro programmes : opérations réalisées directement par le matériel

niveau 2 : langage machine : les instructions spécifiques à l'ordinateur.

⇒instructions interprétées par les micro programmes.

- **niveau 3** : SE

$\Rightarrow$ interpréteur d'instruction plus complexe qui correspond aux services supplémentaires offert par SE *(revoir la figure)*

Hiérarchie de langages :

On peut définir une hiérarchie de langages :

celui d'un niveau *i* est exécuté par une machine logique ou virtuelle *Mi*

$\Rightarrow$ en fait, exécuté par les machines d'un niveau $<$ i)

Par exemple :

un programme Pascal $\Rightarrow$ machine virtuelle $\Rightarrow$ Programmes $\Rightarrow$ Machine physique

> *$\Rightarrow$ Le programme Pascal est exécuté par une machine virtuelle "PASCAL".*

Le découpage en couches ne doit pas être figé (sauf aux plus basses couches) :

$\Rightarrow$ certaines fonctions peuvent être confiées aux logiciels et puis (plus tard) aux matériels.

$\Rightarrow$ Fondamentalement, toute fonction logicielle peut être confiée au matériel et vice versa.

$\Rightarrow$ Le choix est d'ordre économique ou d'efficacité.

$\Rightarrow$ De nos jours, les fonctions d'un SE sont plutôt exécutées par le matériel.

$\Rightarrow$ il faut considérer l'approche **fonctionnelle** du SE, indépendamment de son implantation.

ELEMENTS DE BASE D'UN SE

Processus : notion de base introduite par MULTICS

$\Rightarrow$ C'est une abstraction de l'activité d'un processeur.

Dans un système multi programmé avec un processeur (ressource non partageable), le *processus* permet d'exprimer qu'un programme est en cours d'exécution bien qu'il ne progresse pas (en attente du processeur).

Exemple :

- *recette* : l'algo (code du programme)
- *la dame (la maman)* : le processeur
- *les ingrédients* : données du programme
- $\rightarrow$ lire la recette, trouver les ingrédinets et cusisiner (processus) un gâteau
- $\rightarrow$ le fils bléssé : interruption, sauvegarde de l'état du processus (de cuisine) et changement de contexte pour exécuter un autre processus (soins)
- $\rightarrow$ le processeur (la dame) est passé d'un processus (gâteau) à un autre , plus prioritaire.

- chaque processus a son propre programme (recette)
- la dame reprendra la cuisine après les soins (voir la description page suivante)

⇒ **1 processus** : une activité qui possède un code (programme), des données (In/out) et **un état** courant.

On distingue un programme de son exécution ⇒ notion d'**état**

⇒ un programme : entité statique associée à une suite d'instructions ;

⇒ un processus : entité dynamique associée à la *suite des actions* réalisées par un pro gramme lors d'une exécution particulière.

→ suite indépendante du nombre de fois et des instants où le processus a été mis en attente de processeur.

⇒ Dire qu'un processus P est parvenu à l'action **lire la case M** signifie :

- soit le processeur est réellement entrain d'exécuter l'instruction correspondant à l'action,

- soit P est mis en attente, il exécutera l'action quand il aura le processeur),

- soit P a exécuté l'action et a été mis en attente (pour la suite).

- *début d'un programme* = la 1ère instruction (sur le papier !)
- *début d'exécution d'un programme* = l'exécution de la 1ère instruction (début du processus)
- La suite d'instructions exécutées ne permet pas de d'écrire entièrement comment le programme est exécuté.

  ⇒ information d'**Etat**.

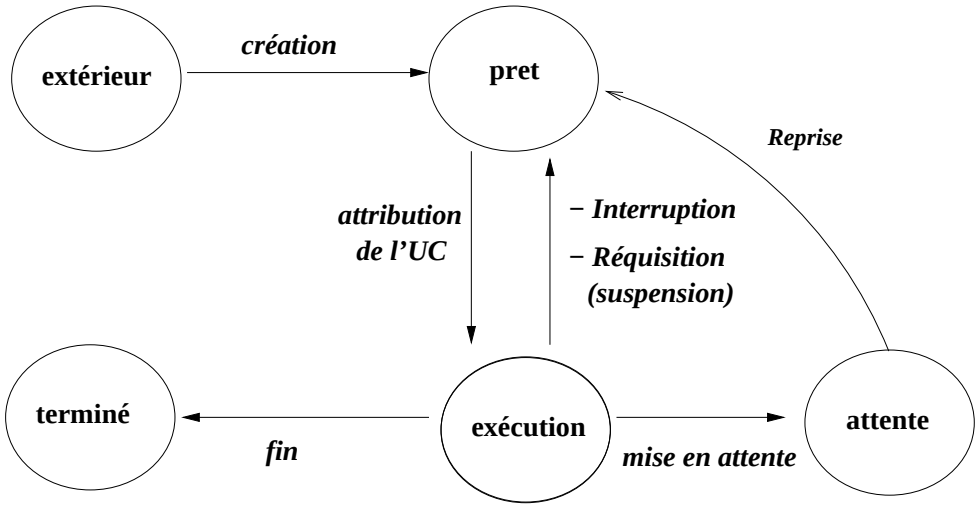
Etats possibles d'un processus

- est réellement exécuté;

- est prêt à être exécuté (en attente du processeur);

- en attente de ressource/événement (e.g. fin d'un E/S);

$\Rightarrow$ diagramme d'états (simplifié):



- Les états importants : "en exécution", "prêt", "en attente" (d'autre ressource que le processus)
- Il y a d'autres états : "terminé", "extérieur", ...
- Il y a des états associés : "bloqué", "suspendu",...
- On pourra aussi distinguer les causes d'attente, attente en mémoire centrale/secondaire,...
- Les états sont modifiés par le SE, sous l'effet d'un **événement**.

- Les événements peuvent être internes au processus : une demande d'E/S fait passer de l'état "en exécution" → "en attente".
- Les événements peuvent être extérieurs (depuis le SE) : l'attribution d'une ressource fait passer de l'état "en attente" → "prêt".

Nota bene :

Thread = processus léger (souvent associé à une procédure).

Crée par le père (un processus), un thread partage les données du père.

On préfère un thread à un processus lorsque les données (d'un processus) doivent être partégées pour un traitement annexe (e.g.: correction pendat la frappe d'un texte.)

Exemple de tâches en ADA/JAVA

Représentation interne de processus (PCB)

- Rappel : la nécessité de conserver un *état* pour tout processus
- L'état d'un processus permet sa reprise
- Les données (de reprise) qui caractérisent un processus sont stockés dans la structure appelée *Processus Control Bloc (PCB)*
- **Le PCB contient :**

  - l'état du processus (cf. le diagramme), ID, PC (Program Counter = l'adresse de la prochaine instruction), registres , mémoire allouée (+ pointeurs vers la mémoire) , autres (temps d'UC utilisé, temps restant autorisé,...), fichiers et opérations d'E/S , ...

- Les PCB sont en mémoire (vitesse d'accès)
- Certaines machines possèdent des registres spéciaux vers les PCB.
- Souvent, les instructions câblées effectuent la sauvegarde/restauration de PCB.

Opérations sur les processus

Réalisés par le SE

- *créer / détruire*

- *mettre en attente / réveiller*

- *suspendre / reprendre*

- *modifier la priorité*

• Un processus peut en créer un autre (père - fils)

• Le fils peut à son tour en créer d'autres. Dans ce cas, le graphe de création du père et ses descendants est une structure partiellement ordonnée.

• On appelle par fois "job" l'ensemble du processus père et ses descendants.

Exemple d'Unix : *fork*

- Permet de duppliquer le processus exécutant (copie identique)

- Renvoie 0 au fils et le numéro d'ID du fils au père

- Copie le code et les données du père

- On peut charger le fils d'exécuter une partie du code mais en général, on utilise un **exec** pour exécuter un (autre) code.

- **exec** remplace le code et les données du processus (qui l'exécute) par ceux du fichier chargé.
- le processus reste le même mais le programme exécuté change
- **fork + exec** permettent l'exécution en parallèle du père et du fils.
- Exemple :

```
idfils = fork ();
if (idfils == 0) // en est dans le fils
        exec(fichier disque)
else // on est dans le père
        <continuer la travail du père>
```

• Dans certains systèmes, le père attend la fin du fils.

Par exemple :

* Sous DOS : on charge un fichier binaire en mémoire pour l'exécuter comme un processus fils mais le père se met en attente de la fin du fils (pas d'exécution en parallèle).

* Sous Unix :

- Le père et le fils ont des contextes mémoires séparés (les variables sont dupliquées), mais le fils a accès aux fichiers du père.

- Pour communiquer, le père et le fils passent par des fichiers (ou par la mémoire partagée) .

• Un processus fils peut se terminer de différentes façons:

- exécution de la dernière instruction

→ état Zombie → Il fait lire sa valeur de retour

- instruction d'auto destruction (exit)

- il est détruit par un autre processus (*kill* d'Unix)

- Un fils ne peut être détruit que par son père.

- Dans certains systèmes, la destruction d'un processus entraîne la destruction de ses descendants. Dans d'autres, les fils peuvent continuer.

- A la destruction, les ressources sont libérées, le PCB est effacé et le processus disparaît des tables et des files d'attente.

→ Pour sortir un processus de l'état **zombie**, sa valeur de retour peut être lu (par le père) via *wait()*, *wait4()* ou *waitpid()*, ...

Un exemple : Le père crée des processus de communication et attend leur fin :

```
int idfils  = fork ();
if (idfils == -1)
  {perror("pb. de création"); exit(-2);}
if (idfils == 0)       // en est dans le fils
  execlp("/home/alex/comm", "comm", NULL);
else // on est dans le père
  {// close(sock);
   int pid;
   while (pid=waitpid(-1, NULL, WNOHANG))
     if (pid == -1)
       {perror("pb. de waitpid"); exit(-1);}
  }
```

Attendre / Réveil

- “en attente” : demande de ressource (autre que l’UC) non disponible;
- “réveil” : quand le S.E. peut lui affecter les ressources; le processus passe à l’état “prêt”.
- **Exemple :**

  Unix donne la priorité au processus qui fait une E/S par rapport à celui qui va faire une demande d’E/S.

  $\Rightarrow$ permet de garder le processeur et le périphérique occupé plus souvent et maximise le parallélisme.

Suspendre / reprendre

- similaire à une mise en attente (save contexte)
- permet de donner “son tour” à chaque processus.
- à la repise $\Rightarrow$ “prêt” ou “en attente” (selon l’état avant la suspension)
- Equité : ne pas suspendre un processus trop souvent/trop long temps;
- un processus en attente de données (d’un autre processus) peut être suspendu (Attente $\Rightarrow$ suspension)
- le SE peut suspendre un processus (pour efficacité : trop de processus, performances, ..)

Changement de priorité

Utilisé par l'ordonnanceur (augmentation/baisse)

Phases de calcul et d'E/S

- Un processus (v. un programme) : phases de calculs / E-S

  ⇒ phase de calcul extrêmement courte, E-S trop longue (statistiques).

- Cause : différence de vitesse UC/périphériques.

Exemple-1 : `$ cat f1 f2 | grep "main"`

⇒ Deux processus (faire *ps* pour les voir)

- *grep* calcule sur des sorties de *ca*

- *grep* (à l'état **prêt**) risque d'attendre les données de *cat*

⇒ attente

La lecture du *pipe* est comme l'attente au clavier.

Exemple-2 (pire !) : `$ cat f1 - | grep main`

grep attend *cat* qui attend le clavier.