Systèmes d'Exploitation Avancés

Pablo Oliveira

ISTY

Informations Administratives (1)

- Cours Systèmes Exploitations Avancé (SEA)
- Site du cours : https://www.sifflez.org/lectures/SEA/
  - Emploi du temps, documents et notes de cours
- Bibliographie :
  - *Operating System Concepts, 8th Edition*, de Silberschatz, Galvin, and Gagne
  - *Modern Operating Systems, 4th Edition*, de Tanembaum
  - *Stanford CS 140 lectures Winter'14*, de David Mazières

Informations Administratives (2)

- Prof. : pablo.oliveira@uvsq.fr
- Group de discussion (inscription obligatoire)
  - https://groups.google.com/group/iatic4-os/
  - Poser des questions sur les cours, TDs, projets.
  - Tout le monde est encouragé à répondre !
- Dates clés :
  - Cours : Mercredi matin
  - TDs : Mercredi après-midi
- Contrôle Continu
  - 30% Mini-Projets notés
  - 35% QCMs en début de cours
  - 35% Contrôle

Sujets abordés

- Threads et Processus
- Concurrence et Synchronisation
- Ordonnancement
- Mémoire virtuelle
- Entrées/Sorties
- Protection & Securité
- Systèmes de Fichiers
- Machines Virtuelles
- Note : Nous prendrons souvent Unix comme example
  - SE actuels fortement influencés par Unix
  - Windows est l'exception

Systèmes d'Exploitation 1

- Présentation des SE du point de vue de *l'utilisateur*
  - Utilisation du shell (commandes de base, redirections, tubes/pipes)
  - Création de threads et processus
  - Introduction à l'ordonnancement
  - Utilisation des principaux appels systèmes
  - Projet : sytème de fichiers virtuel

Objectifs du cours

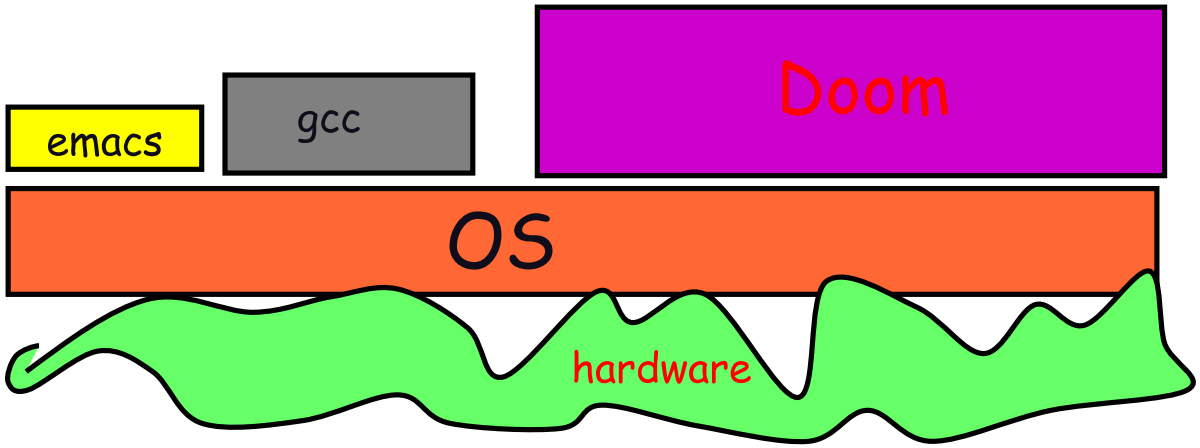
- Présentation des SE du point de vue du *concepteur*
- Approfondir les concepts fondamentaux des SE
  - Comprendre le SE fait de vous de meilleurs programmeurs
- Comprendre les enjeux et l'implémentation des méchanismes des SE
  - Caching, concurrency, memory management, I/O, protection
- Vous apprendre à travailler avec un projet logiciel complexe
  - Attention : ce cours est considéré difficile par de nombreux étudiants
  - TDs : beaucoup de code à rendre
  - Travail régulier et soutenu nécessaire

TDs

- Ne pas utiliser/récupérer les solutions des autres groupes
  - Vos rendus sont comparés avec un logiciel anti-plagiat
  - Ne publiez pas vos solutions
  - Respectez la charte anti-plagiat de l'UVSQ (document disponible sur le portail de l'ISTY)
- Citez tout code dont vous vous inspirez
  - Si c'est cité, c'est pas de la triche
  - Des points seront déduits si une grosse partie rendu est du code extérieur. Mais les emprunts cités n'entrainerons pas de sanctions.

Qu'est ce qu'un système d'exploitation ?

- Interface entre les applications et le materiel

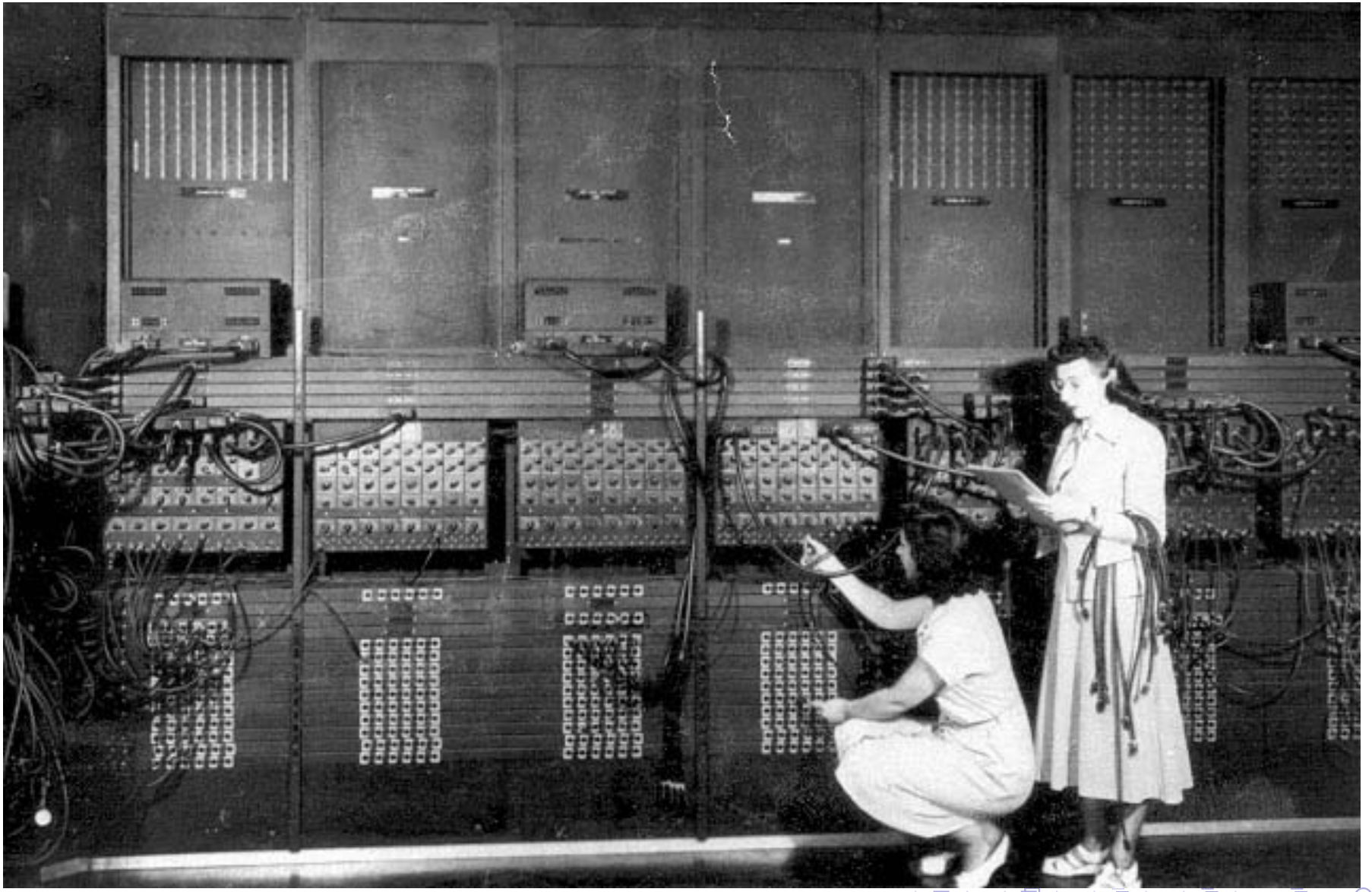


- Rends le matériel utilisable par le programmeur
- [Souvent] Abstrait le matériel
  - Gère et cache les détails du matériel
  - Accède au matériel à travers des interfaces bas niveau interdites aux applications
- [Parfois] Protège
  - Evite qu'un utilisateur/processus puisse lire/détruire les données d'un autre utilisateur/processus

Pourquoi étudier les systèmes d'exploitation ?

- Concepts système importants pour bien programmer la machine
  - Important pour HPC ou applications critiques
  - Algorithmes réutilisables dans d'autres contextes
  - Important pour comprendre l'intéraction avec le matériel
- Comprendre les Serveurs Haute Performance
  - Mêmes problèmes que dans un SE
- Comprendre le Partage des Resources
  - Durée de la batterie, spectre Hertzien, etc.
- Comprendre la Sécurité
  - Isolation, capabilités, DoS, etc.
- Apparition de nouveaux SE embarqués (eg. Android)
- Navigateur ressemble de plus en plus à un SE (eg. Chrome isolation des tabs)

Années 40-50 : Super calculateur ENIAC



Années 40-50 : Super calculateur ENIAC

- 1946 : P. Eckert et J. Mauchly, simulation des tirs d'artillerie
  - fréquence 100kHz
  - Turing-complet
  - Programmable avec des interrupteurs et tableau de connexion

Système Exploitation des premiers Calculateurs ?

- Calculateurs gigantesques :
  - Dizaines de milliers de relais mécaniques
  - Remplacés par des lampes
- Programmation :
  - Manuelle
  - En langage machine avec des panneaux de connexion
  - Pas de SE, le programme tourne directement sur la machine
  - 1950 : généralisation des cartes perforées

IBM 701

Traitements par lots (55-65)

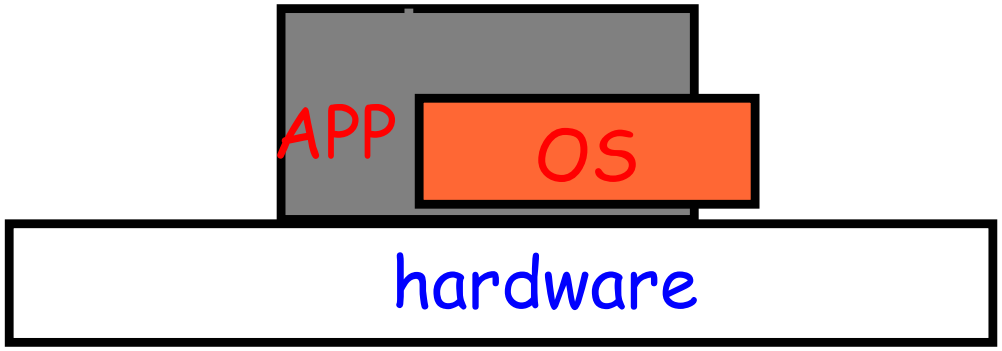
- Invention des transistors en silicone (1954) :
  - plus petit et rapide que les lampes
  - Bell Labs TRADIC (1MHz) et IBM 608
- Opérateurs :
  - Chargés de la surveillance du système
  - Allocation et supervision des tâches
- Les tâches sont traitées par lots :
  - Des opérateurs chargent les cartes perforées pour chaque programme

Moniteur Résident

- Bandes magnétiques remplacent les cartes perforées
- Moniteur résident : premier SE
  - Il charge automatiquement les tâches successivement `$LOAD` et `$RUN`
  - Problème : inactivité du CPU pendant les chargements depuis la bande.

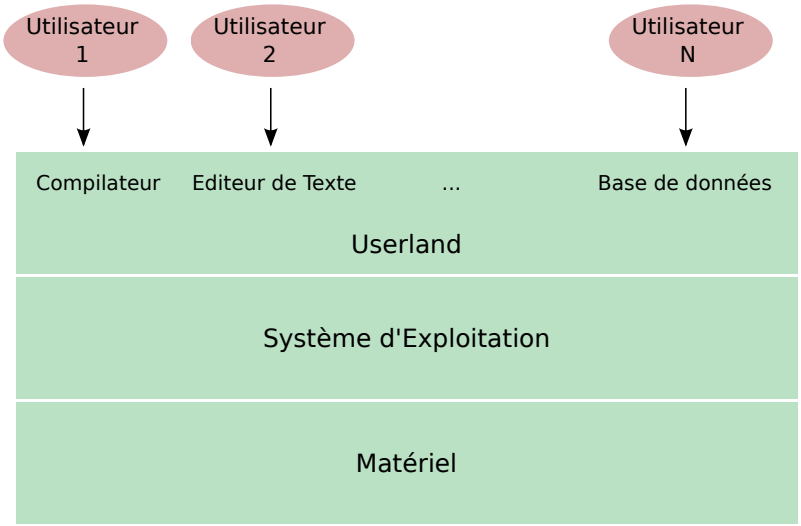
Systèmes d'Exploitation Primitifs

- Juste une librairie de services standards [pas de protection]



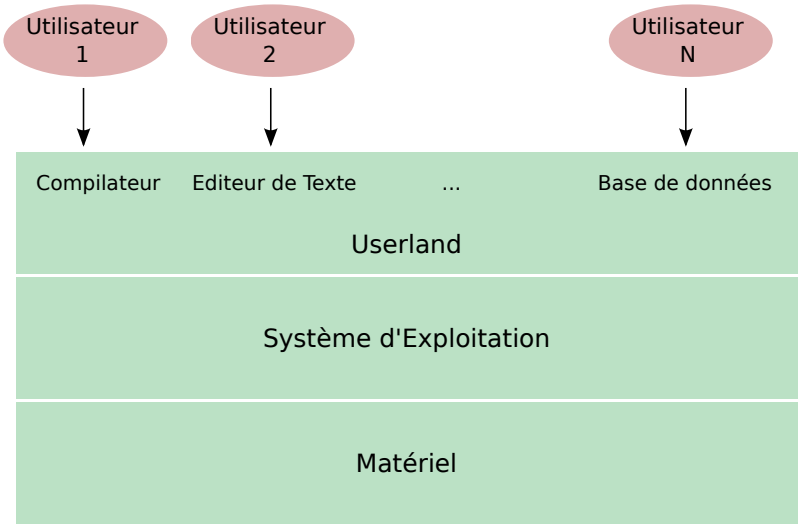
  - Interface standard communique avec des pilotes matériels
- Hypothèses simplificatrices
  - Un seul programme tourne à la fois
  - Pas de programmes ou utilisateurs malicieux (mauvaise hypothèse)
- Problème : Mauvaise utilisation des ressources
  - . . . du matériel (e.g., CPU attends que le disque envoie les données)
  - . . . de l'humain (doit attendre la fin d'un programme pour en lancer un nouveau)

Execution multi-tâche



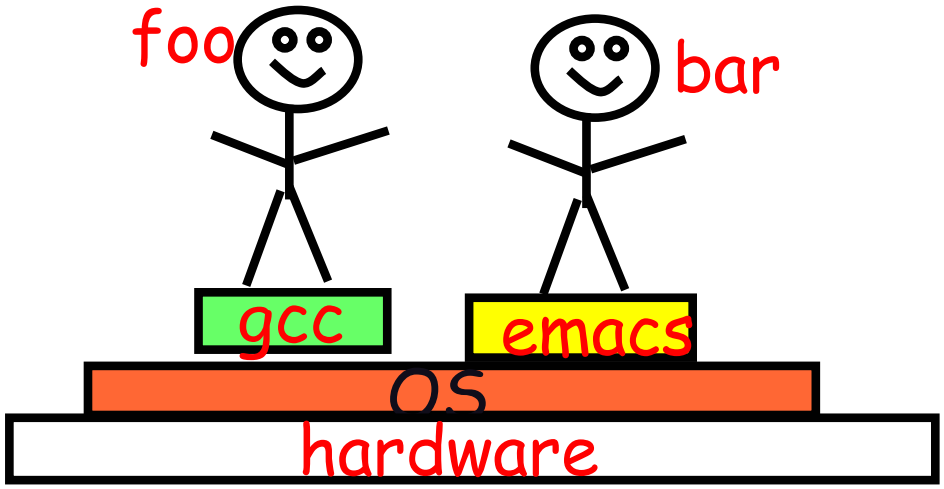
- Idée : Executer plusieurs processus en "même temps"
  - Lorsque un processus bloque (attente disque, réseau, entrée clavier, etc.) on execute un autre processus
- Problème : Que peut faire un processus malicieux/mal écrit ?

Execution multi-tâche



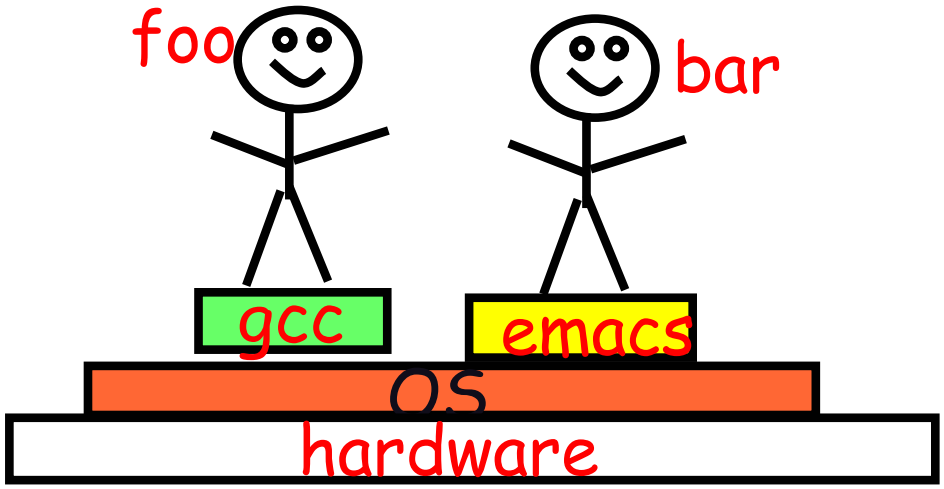
- Idée : Executer plusieurs processus en "même temps"
  - Lorsque un processus bloque (attente disque, réseau, entrée clavier, etc.) on execute un autre processus
- Problème : Que peut faire un processus malicieux/mal écrit ?
  - Boucle infinie et ne jamais lâcher le CPU
  - Écraser la mémoire d'autres processus pour les faire cracher
- SE propose des mécanismes pour empêcher ces problèmes
  - *Préemption* – interrompt régulièrement le processus boucle infinie
  - *Protection mémoire* – Chaque processus ne peut écrire que sur sa propre mémoire

SE Multi Utilisateurs



- Idée : Avec N utilisateurs, système n'est pas forcément N fois plus lent
  - Les demandes de CPU, mémoire, etc. sont intermittentes
  - Tous les programmes n'ont pas besoin de la même ressource simultanément
- Où ça peut coincer ?

SE Multi Utilisateurs

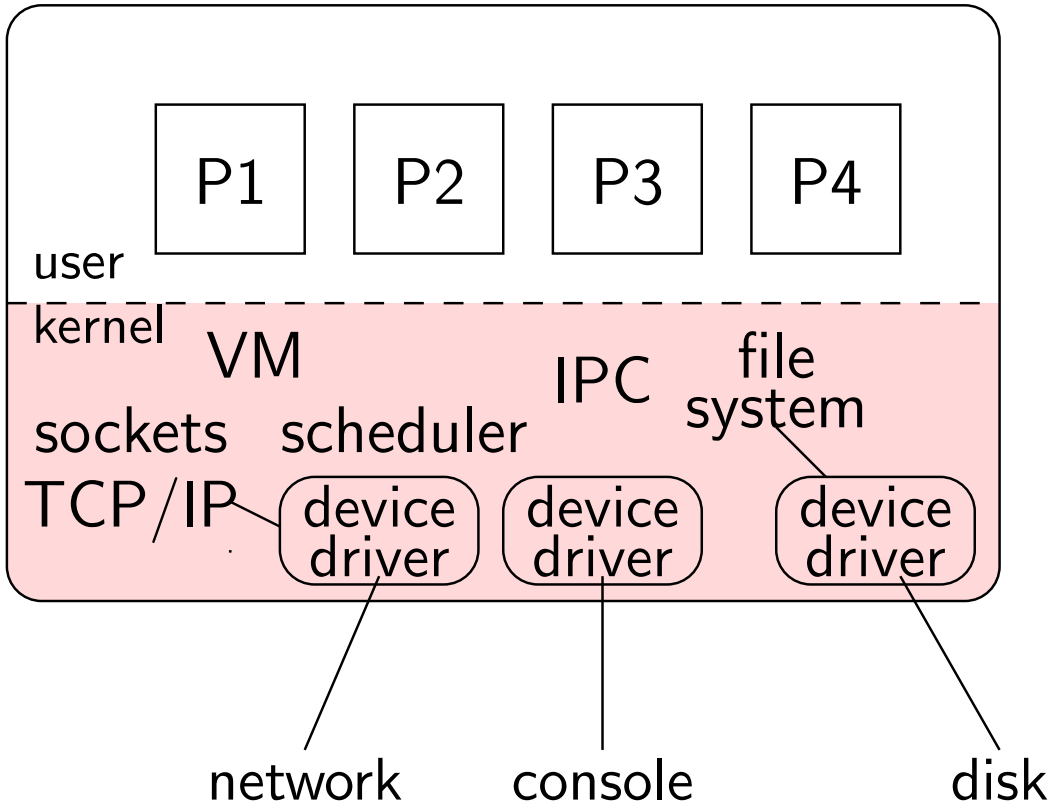


- Idée : Avec N utilisateurs, système n'est pas forcément N fois plus lent
  - Les demandes de CPU, mémoire, etc. sont intermittentes
  - Tous les programmes n'ont pas besoin de la même ressource simultanément
- Où ça peut coincer ?
  - Utilisateurs gloutons (mise en place de politiques d'allocation)
  - Mémoire utilisée par l'ensemble des processus supérieure à la mémoire disponible (virtualisation)
  - Ralentissements super-linéaire (trashing)
- Les SE mettent en place des *protections* pour éliminer ces problèmes

Protection

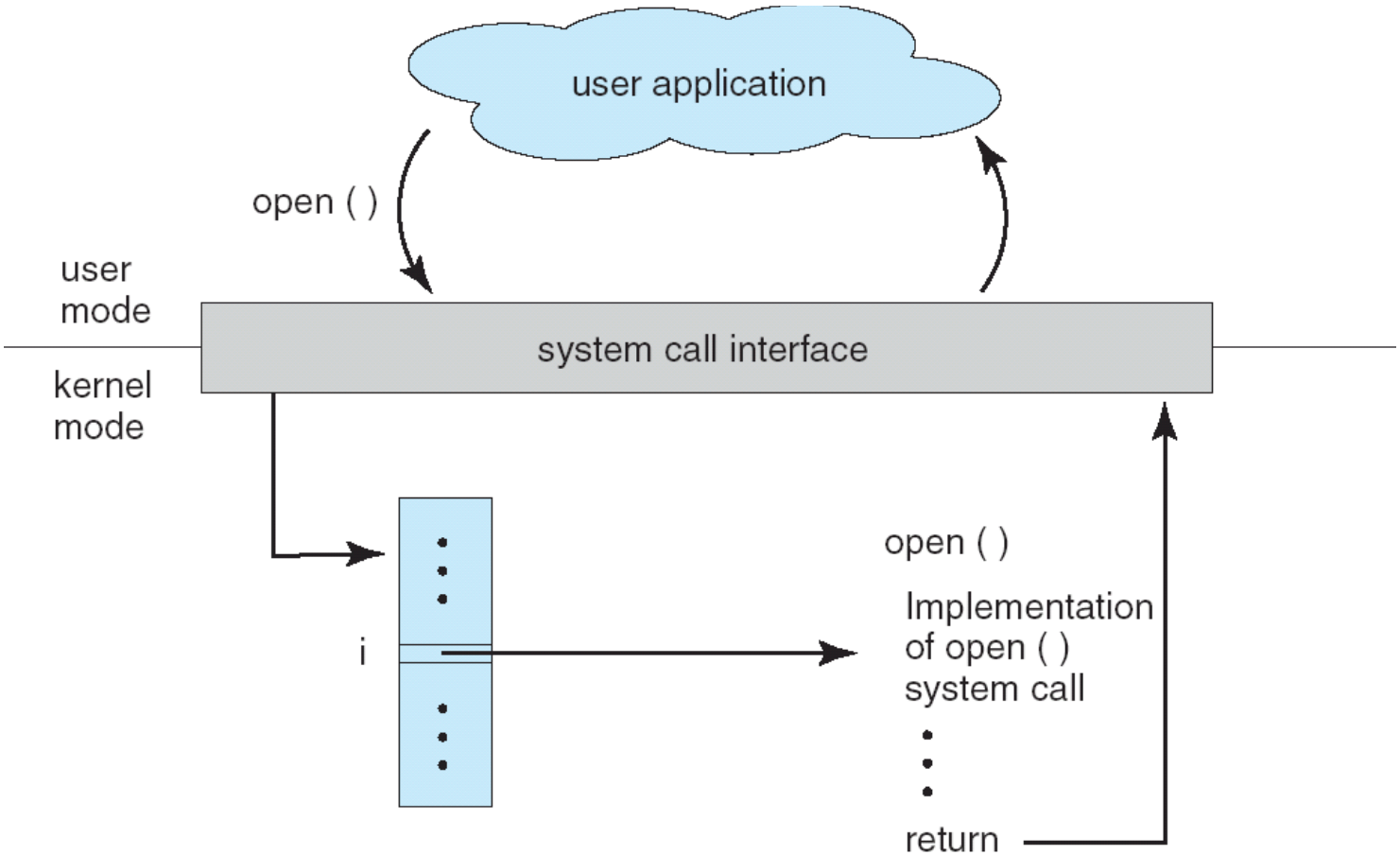
- Mécanismes pour isoler les utilisateurs/processus malicieux
- Préemption :
  - Donner une ressource, mais la reprendre au bout d'un certain temps
- Médiation :
  - SE est le médiateur entre les processus et les ressources
  - Controle toutes les ressources qu'une application peut utiliser (table de capabilités)
  - Pour chaque demande le SE vérifie que l'application à le droit de faire cette demande
- Mode privilégié dans le CPU
  - Applications tournent en *mode utilisateur*
  - SE tourne en *mode privilégié* (ou mode noyau)
  - Les opérations de protection ne sont disponibles qu'en *node privilégié* (par exemple editer la table de capabilités).

Structure d'un SE typique



- Les applications tournent en mode utilisateur (P[1-4])
- Le noyau tourne en mode privilégié [en grisé]
  - Crée et détruit les processus
  - Décide et vérifie qui accède au matériel

Appel Système

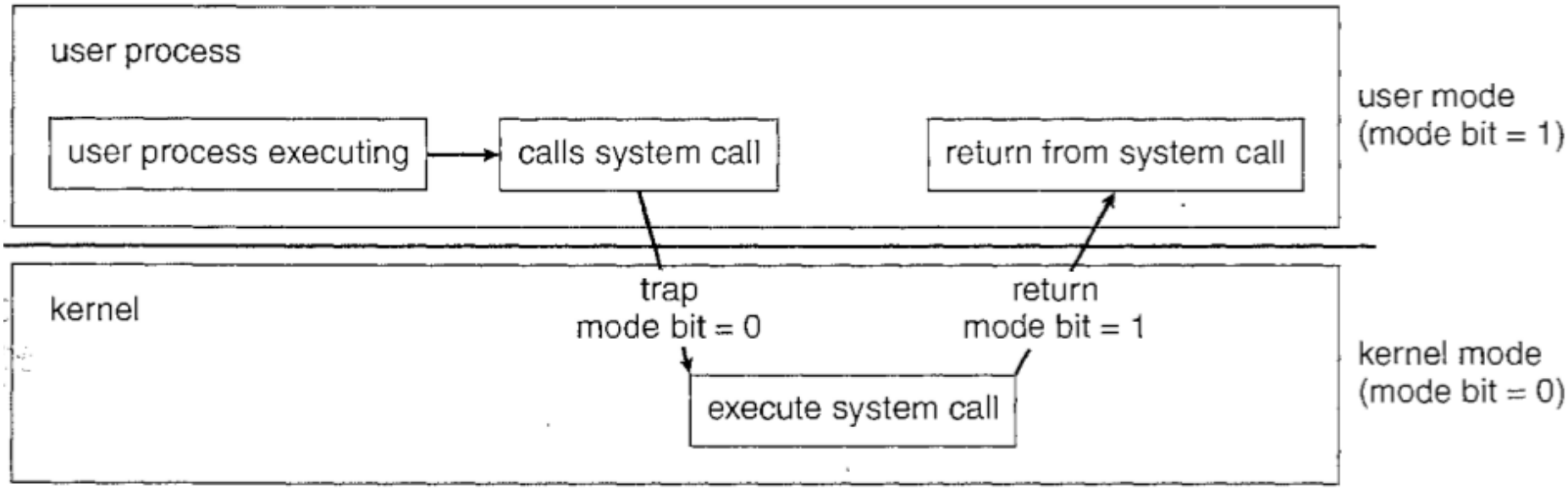


- Les applications invoquent le noyau avec des *appels système*
  - Des instructions assembleur spéciales transfèrent le contrôle au noyau
  - . . . qui transfère l'appel à l'une des 100+ routines gestionnaires

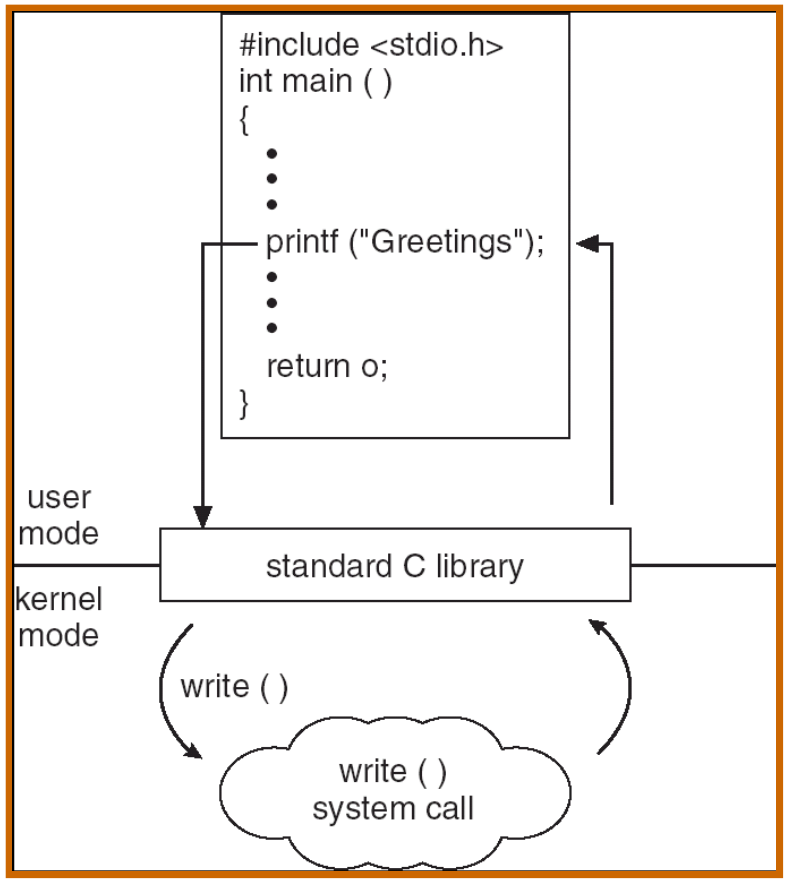
Appel Système (suite)

- Objectifs : Faire des choses que l'application ne peut pas faire en mode utilisateur
  - Semblable à un appel de bibliothèque mais en mode privilégié
- Noyau expose une interface d'appels systèmes bien définie
  - L'application configure les arguments de l'appel et appellent une "trappe" (ou interruption logicielle)
  - Le noyau répond à la demande et retourne le résultat
- Exemple : interface POSIX/UNIX sont des appels systèmes
  - `open`, `close`, `read`, `write`, `...`
- Fonctions de plus haut niveau sont implémentées à l'aide d'appels système
  - `printf`, `scanf`, `gets`, etc.

Mode privilégié



Exemple d'un appel système



- Librairie standard implémentée à l'aide d'appels systèmes
  - *printf* – dans la libc, mêmes privilèges que l'utilisateur
  - appelle *write* – dans le noyau, qui peut écrire sur le disque dur, ou envoyer des bits sur les ports de sortie

UNIX appels système pour les Entrées/Sorties

- Les applications ouvrent des fichiers (ou des périphériques)
  - Les E/S opèrent sur des descripteurs de fichiers (des entiers)
- `int open(char *path, int flags, /*mode*/...);`
  - `flags` : `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT` : crée le fichier s'il n'existe pas
  - `O_TRUNC` : tronque le fichier
  - `O_APPEND` : ouvre le fichier et positionne le curseur d'écriture en fin de fichier
  - `mode` : avec `O_CREAT` donne les permissions du fichier
- Retourne un descripteur de fichier

En cas d'erreur ?

- Que se passe t'il si `open` rencontre une erreur ?
- La majorité des appels systèmes retournent -1 en cas d'erreur
  - L'erreur spécifique est renseignée dans la variable globale int `errno`
- `#include <sys/errno.h>` contient la liste des codes possibles
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` permet d'afficher un message adapté
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"

Operations sur des descripteurs de fichier

- `int read (int fd, void *buf, int nbytes);`
  - Retourne le nombre d'octets lus
  - Retourne 0 lorsque la fin du fichier (EOF) est atteinte et -1 en cas d'erreur
- `int write (int fd, void *buf, int nbytes);`
  - Retourne le nombre d'octets écrits, -1 en cas d'erreur
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence` : relatif au 0 – début, 1 – courant, 2 – fin
    - Retourne la position précédente dans le fichier, et -1 en cas d' error
- `int close (int fd);`

Numéros de descripteurs de fichiers

- Les descripteurs de fichiers sont propres à un processus
  - ... mais ils sont hérités par ses fils
  - Quand un processus crée un processus fils, ils se partagents des descripteurs
- Les descripteurs 0, 1, et 2 ont un sens special
  - 0 – "entree standard" (`stdin` ANSI C)
  - 1 – "sortie standard" (`stdout` ANSI C)
  - 2 – "erreur standard" (`stderr` ANSI C)
  - Normalements ils sont rattachés à un terminal
- Exemple : `cat.c`
  - Ecrit le contenu d'un fichier sur `stdout`

cat.c

```
void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}
```

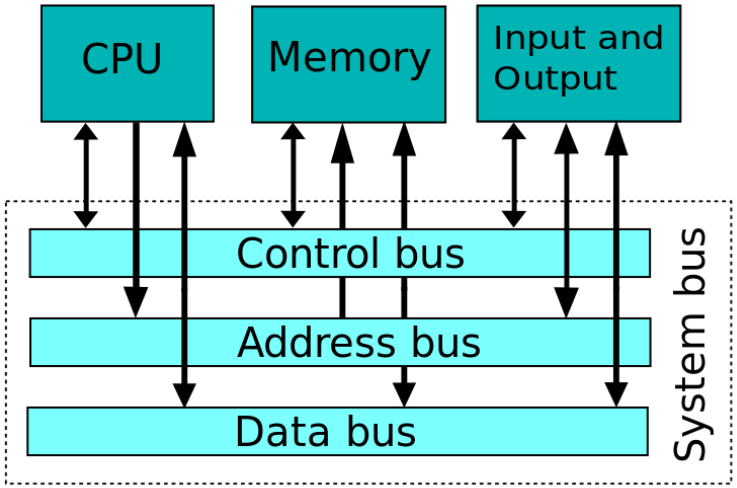
Traitement des interruptions

- Une interruption suspend la tâche en cours.
- Chaque interruption est associée à une routine de traitement
- Un vecteur d'interruption situé à un endroit fixe en mémoire contient les adresses de chaque routine
- On exécute la routine de traitement de l'interruption, puis on revient au traitement précédent.
- Les routines doivent sauvegarder l'état du processeur et l'adresse du traitement interrompu.

Deux interruptions en même temps ?

- Masquage simple
  - Le traitement des interruptions est différé jusqu'à la fin du traîtement de l'interruption en cours.
- Masquage sélectif
  - Les interruptions ont des niveaux de priorité.
  - Les interruptions les plus prioritaires peuvent interrompre celles de niveau inférieur.

Exemple de communication périphérique



1. Le CPU écrit sur la ligne d'*adresse*, la position du disque à lire
2. Le CPU active la ligne de *controle* pour signaler au contrôleur disque
3. Le contrôleur disque écrit le contenu de la position demandée sur la ligne de *données*
4. Il signale avec une interruption que les données sont prêtes

- Inefficace pour des gros transferts

Différents contextes d'exécution

- Le système se trouve habituellement dans un des contextes suivants
- *Processus Utilisateur :* execution d'une application en mode utilisateur
- *Processus Noyau* :
  - Execution de code noyau à la demande d'une application
  - eg. traîtement d'un appel système
  - Exception (segmentation fault, division par zéro, etc.)
  - Execution d'un processus noyau (serveur de fichiers nfs)
- *Code noyau hors processus*
  - Interruption d'horloge
  - Interruption matérielle
  - Interruption logicielle, "Tasklets" (dans linux)
- Changement de Contexte – Changer les espaces d'adresse
- En attente (Idle) – rien à faire (souvent on "éteint" ou "ralentit" le CPU)

Mécanismes de changement de contexte

- Utilisateur $\rightarrow$ Processus Noyau : appel système, exception
- Processus Noyau $\rightarrow$ Utilisateur/Changement Contexte : return
- Processus Noyau $\rightarrow$ Changement Contexte : sleep
- * $\rightarrow$ Gestionnaire d'interruption matérielle : IRQ matériel
- Changement de Contexte $\rightarrow$ Processus Utilisateur/Noyau

Préemption CPU

- Éviter la monopolization du CPU
- E.g., un timer noyau reprends la main tous les 10 ms
  - Interruption matérielle d'horloge
  - Nécessite le mode privilégié pour reprogrammer l'horloge hardware
- Le gestionnaire d'interruption du noyau
  - Prends le contrôle quand l'interruption d'horloge se déclenche
  - Si d'autres processus sont en attente du CPU, il leurs donne la main
  - Protection : mode privilégié nécessaire pour definir l'adresse du gestionnaire
  - L'utilisateur ne peux pas "hacker" la routine de traîtement
- Resultat : un processus malicieux ne peux pas affamer tous les autres
  - Pire cas : tout le monde aura $1/N$ du CPU si N processus CPU-gloutons

Protection != Securité

- Comment monopoliser le CPU quand même ?

Protection != Securité

- Comment monopoliser le CPU quand même ?
- Utiliser plusieurs processus : Fork bomb
- Jusqu'à encore récemment, le code suivant "met à genoux" plusieurs SE

```
int main() { while(1) fork();}
```

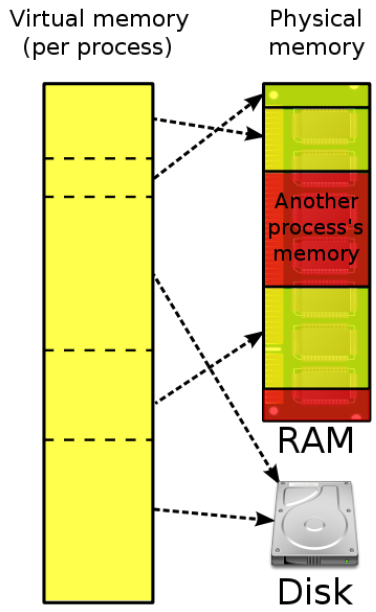
  - Crée des processus jusqu'à épuisement des ressources
- Utiliser toute la mémoire : Malloc bomb
- Résolu par une combinaison de technique + social
  - Solution technique : limiter les processus par utilisateur (ulimit,)
  - Solution technique : limiter le débit de création (grsec)
  - Social : Rebooter et crier sur les utilisateurs embêtants :-)
  - La bonne solution dépend de l'usage
  - Serveur de la NSA vs. Serveur Imprimante dans une PME

Allocation et Protection de la mémoire

- Le noyau alloue la mémoire
- Éviter qu'une tâche utilisateur détruise les données d'une autre tâche
  - Protection mémoire
- Gérer efficacement la mémoire :
  - Si la RAM est pleine, le SE va déplacer la mémoire de certains processus sur un stockage secondaire (disque)
  - swap

Protection Mémoire (1/2)

- Comment éviter que les processus écrivent dans la mémoire d'autres processus
- Idée Mémoire Virtuelle : Chaque processus ne voit que sa mémoire à lui.



Protection Mémoire (2/2)

- Le CPU traduit les adresses virtuelles en adresse physiques en utilisant des tables de traduction à chaque accès.
- La table de traduction est changée à chaque changement de contexte
- Elle ne contient que des traductions vers des adresses autorisées
- Le SE peut marquer comme invalides certaines adresses virtuelles
  - Détecter les fautes de segmentation et interrompre les programmes
  - Allocation paresseuse + SWAP
    (e.g., ramener les pages du SWAP uniquement lorsqu'elles sont accédées)
- Le SE peut marquer certaines adressses comme lecture seule
  - Partage de données entre processus (par exemple le code executable de la libc)
  - De nombreuses autres optimizations (copy on write par exemple)
- Le SE peut marquer certaines adresses comme non exécutables
  - Rends difficile les attaques par injection de code

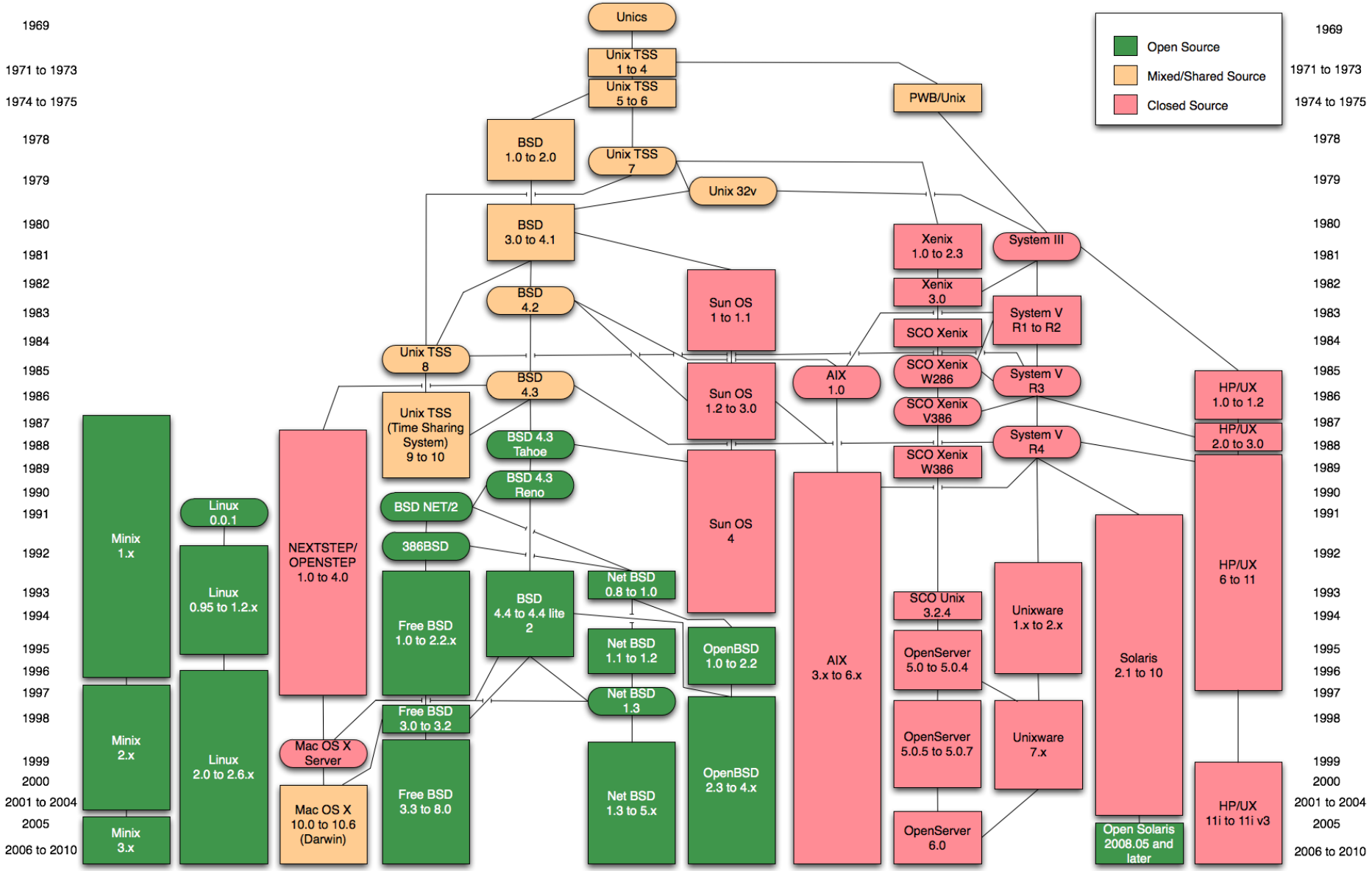
Temps partagé (premier vrai SE)

- 1961 CTSS (Compatible Time Sharing System)
  - Plusieurs utilisateurs en simultané
  - Ordonnanceur de tâches avec priorité
  - Mémoire segmentée : sépare SE et userland
  - Interrupteur d'horloge pour interrompre les tâches
- Suivi de MULTICS (Multiplexed Information and Computing System)
- IBM 7094

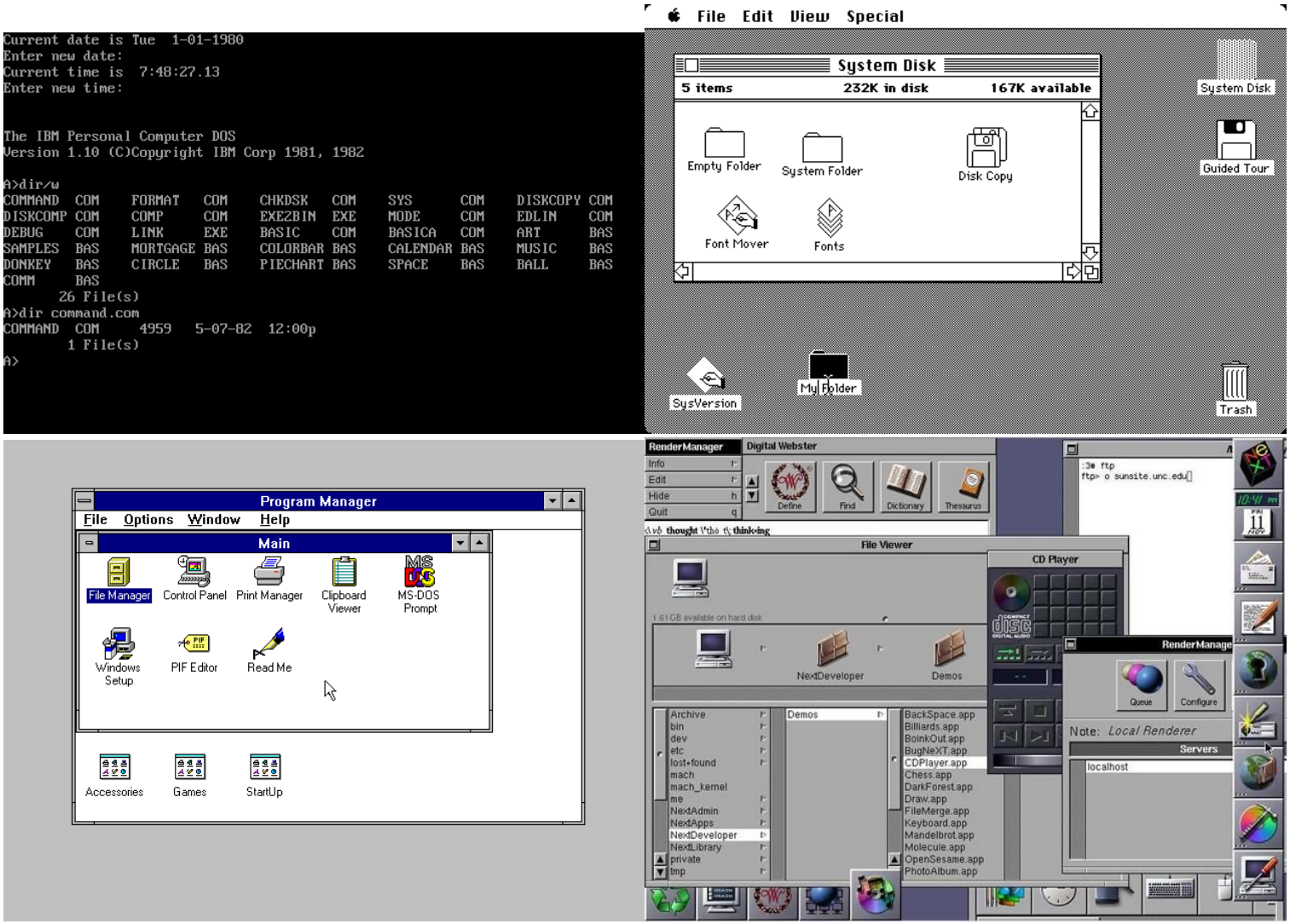
Developpement des SE

- Ken Thompson adapte MULTICS sur un PDP-7 (mini-ordinateur),
- 1969 : ce travail devient UNIX (Bell Labs), qui donnera naissance à SYSTEM V et BSD
- 1981 : Bill Gates écrit MS-DOS (Microsoft Disk Operating System)
  - Modification du DOS de Seattle Computer Products
- 1987 : Tanembaum écrit MINIX (SE micro-noyau) qui servira d'inspiration à Linux
- 1991 : Torvalds libère la première version de Linux

Famille Unix



Quelques captures d'écran "vintage"



(1) DOS '81 (2) Classic '84 (3) Window 3.1 '92 (4) OpenStep '89