Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 2

Anne Benoit

February 4, 2015

1 Structure des systèmes d'exploitation

1.1 Organisation et architecture d'un système informatique

1.2 Services fournis par l'OS

1.3 Appels systèmes et programmes systèmes

1.4 Conception, implémentation et structure d'un OS

La structure interne d'un OS peut varier grandement; nécessaire de bien définir d'abord les buts de l'OS et les spécifications, ce qui va dépendre en particulier du matériel utilisé.

- Buts utilisateurs: OS facile à utiliser, fiable, et rapide;
- Buts système: OS facile à concevoir, à implémenter, à maintenir, mais aussi flexible, sans erreurs, et efficace.

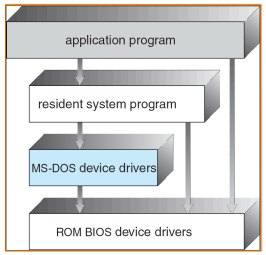
Attention à bien séparer les politiques (qu'est ce qu'on veut faire?) des mécanismes (comment c'est fait?), ce qui permet une plus grande flexibilité.

Exemple d'un timer: c'est un mécanisme pour protéger le CPU, par exemple empêcher une boucle infinie: interruptions à intervalles réguliers (compteur décrémenté par l'OS, interruption à 0). La politique peut être modifiée: décider combien de temps donner à un utilisateur avant de l'interrompre.

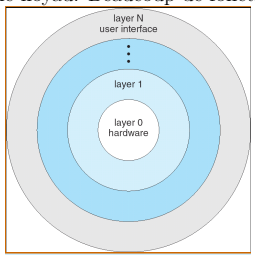
Implémentation. Traditionnellement en langage assembleur, maintenant en langage de haut niveau C, C++: plus rapide, plus compact, plus facile à déboger, et plus portable!

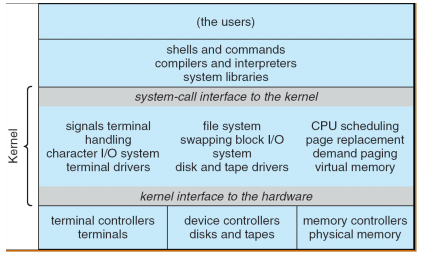
Structure des OS. Résumé des structures classiques utilisées.

1. Structure simple: exemple de MS-DOS, avec le plus de fonctionnalités possible dans le moins d'espace possible. Pas vraiment de séparation entre interface et niveaux de fonctionnalité. Pas de mode utilisateur/noyau, pas de protection.



2. Structure en couches: chaque couche s'appuie sur celle du dessous. Exemple d'UNIX qui a une structure limitée, avec 2 parties distinctes: les programmes systèmes et le noyau. Beaucoup de fonctions à un même niveau.

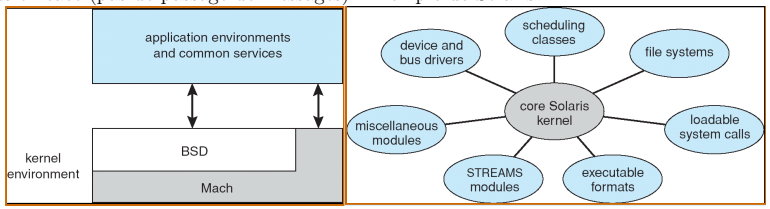




3. Structure avec un micro-noyau: avoir le plus de fonctionnalités possible dans l'espace utilisateur plutôt que dans le noyau. Le noyau gère le minimum sur la gestion des processus et de la mémoire, et permet les communications (échange de message). Avantages: facile à étendre, facile à porter, plus fiable (peu de code tourne en mode noyau), plus sécurisé. Par contre, perte de performance car communications entre espace utilisateur et noyau.

Exemple de Mac OS X: approche hybride (couches et micro-noyau) avec des extensions possibles (modules). Mach microkernel (Carnegie Mellon University) pour gestion mémoire, communication interprocessus. BSD Kernel (Berkeley) pour les pthreads, Posix API.

4. Structure avec des modules noyau: indépendance des composants, qui communiquent via des interfaces connues. Proche du modèle en couches, mais plus flexible et plus efficace (pas de passage de messages). Exemple de Solaris.

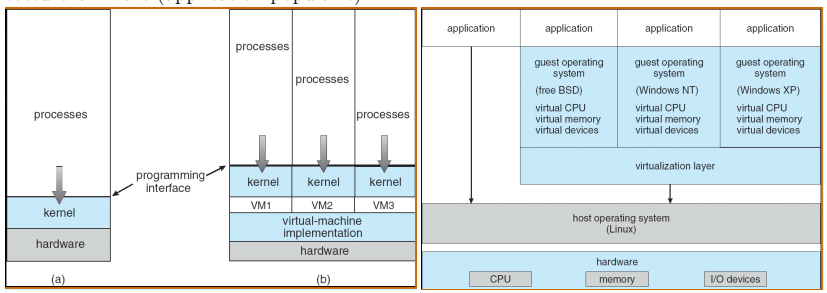


Génération de l'OS et boot. OS: peut tourner sur toute une classe de machines différentes; doit être configuré pour la machine spécifique sur laquelle il est installé. Programme SYSGEN qui retourne l'information matérielle nécessaire.

Booting: démarrer l'ordinateur en chargeant le noyau. *Programme bootstrap:* code stocké dans la ROM qui peut localiser le noyau, le charger en mémoire et l'exécuter.

Machines virtuelles. Une machine virtuelle fournit une interface identique au matériel sous-jacent: matériel et noyau de l'OS considérés comme du matériel. L'OS crée l'illusion de multiples processus s'exécutant chacun sur son propre processeur (avec sa propre mémoire (virtuelle). Les ressources de la machine physique sont partagées pour créer les machines virtuelles.

Exemple (a) sans mémoire virtuelle, et (b) avec mémoire virtuelle. A droite, architecture VMware (application populaire).



Protection totale des ressources du système: les VM sont isolées les unes des autres. Par contre, pas de partage direct des ressources! VM: parfait pour la recherche et le développement des OS, pour ne pas perturber les opérations systèmes de base. Concept de VM difficile à implémenter: doit fournir un dupliqua exact de la machine physique.

1.5 Conclusion

- OS: gère le matériel, fournit un environnement pour que les programmes utilisateurs puissent s'exécuter, fournit un ensemble de services (appels systèmes);
- Programmes: chargés dans mémoire principale (volatile), et autres moyens de stockage; IO pour accéder aux périphériques de stockage;
- Multiprogrammation (plusieurs jobs en mémoire simultanément) et timesharing (partage du temps: le CPU passe très rapidement d'un job à un autre pour être réactif);
- Modes utilisateur/noyau: instructions privilégiées s'exécutent en mode noyau;
- Bien séparer les politiques des détails d'implémentation (mécanismes): les mécanismes doivent être en place et la politique peut être choisie ensuite librement;

- Langage haut-niveau pour faciliter l'implémentation, la maintenance, et la portabilité.

Petites questions d'application du cours.

1. Plusieurs utilisateurs se partagent le système. Problèmes de sécurité? Peut-on garantir la même sécurité que sur une machine dédiée?
2. Que sont les interrupts? Différence entre trap et interrupt? Les traps peuvent-ils être générés intentionnellement par un programme utilisateur?
3. Comment faire pour obtenir un profil statistique du temps passé dans chaque portion du code d'un programme qui s'exécute? Pourquoi de telles statistiques sont importantes?
4. Discuter les avantages et désavantages de l'approche micro-noyau (microkernel).

Et ensuite? On détaillera la notion de processus puis celle de threads.

2 Gestion des processus

Objectifs: introduire la notion de processus = un programme en exécution, et décrire leurs caractéristiques: ordonnancement, création, terminaison, communication.

2.1 Concept de processus

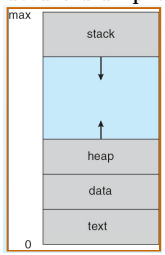
OS: execute des programmes. Programme = passif, c'est un bout de code. Un processus est un programme en exécution; on peut avoir plusieurs processus pour un même programme (par exemple un navigateur Web).

Process = task = job.

Un processus progresse séquentiellement.

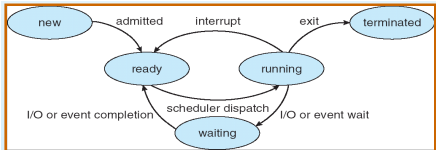
Processus = compteur de programme (PC), pile (stack), section du texte (code du programme), section des données, tas (heap, mémoire allouée dynamiquement).

Structure d'un processus dans la mémoire .



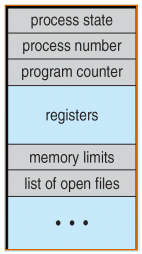
Etat d'un processus. Lors de son exécution, un processus change d'états:

- new: processus créé;
- running: instructions exécutées;
- waiting: en attente d'un événement;
- ready: en attente d'être assigné à un processeur;
- terminated: exécution terminée.



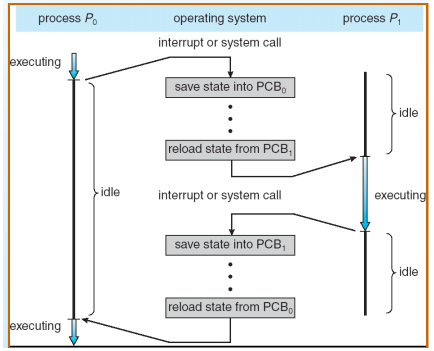
Process Control Block, PCB. A chaque processus, on associe un dépôt avec toute l'information qui le concerne, le PCB:

- état du processus (new, ready, running, ...);
- PC, registres du CPU;
- information pour l'ordonnancement (voir chapitre ordo);
- information pour la gestion de la mémoire (voir chapitre mémoire);
- informations de compte (temps CPU...);
- statut des IOs: fichiers ouverts...

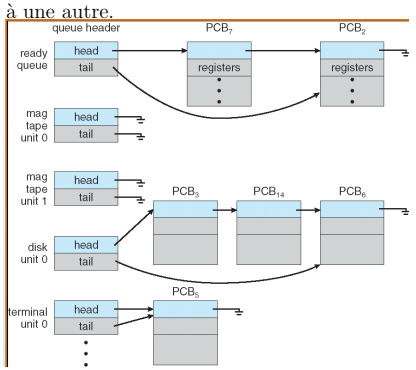


2.2 Ordonnancement des processus

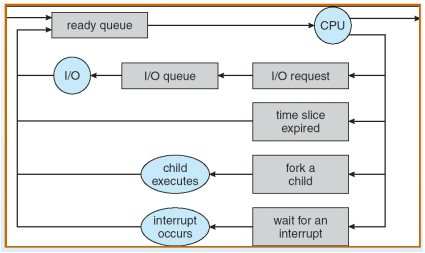
Rappel: multi-programmation pour maximiser l'utilisation du CPU, et partage du temps pour être interactif → le CPU passe d'un processus à un autre, et l'ordonnanceur doit choisir un processus prêt. Le système doit sauvegarder l'état du processus interrompu (son PCB), et charger le PCB du nouveau processus. Pas de travail utile lors d'un changement de contexte, c'est du temps perdu, doit être très rapide. Dépend du matériel: si le CPU a plusieurs ensembles de registres (Sun UltraSPARC), on n'a qu'à changer un pointeur vers l'ensemble utilisé... Il faut aussi faire attention à la mémoire...



Files d'ordonnancement. Une file avec tous les processus du système, la file des processus dans l'état ready, et des files pour les périphériques. Un processus migre d'une file à une autre.



Un processus en cours d'exécution peut être interrompu par une requête d'IO (mise dans la file d'IO correspondante, et ready quand IO terminée), une expiration du timer (directement ready), une création d'un fils (ready quand exécution du fils terminée), une attente d'interruption (ready quand l'interruption a eu lieu). Terminaison: disparaît des files et PCB et ressources désalloués.

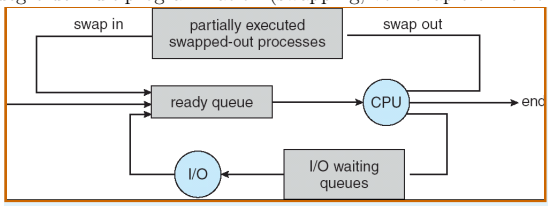


Ordonnanceurs. Ce sont les ordonnanceurs qui décident comment les processus migrent d'une file à une autre.

- Ordonnanceur court-terme (ordonnanceur du CPU): sélectionne le processus qui doit être exécuté, invoqué très fréquemment, doit être rapide!
- Ordonnanceur long-terme (ordonnanceur des tâches): sélectionne les processus à mettre dans la file ready; contrôle le degré de multiprogrammation: nombre de processus en mémoire; nombre constant: invoqué quand un processus termine uniquement, pas besoin d'être rapide.

Classification des processus: gourmands en IO (multiples utilisations du CPU courtes), ou en temps CPU (quelques longues utilisations du CPU). L'ordonnanceur long-terme doit choisir un bon mélange des deux catégories.

Ajout d'un ordonnanceur moyen-terme: l'ordo long-terme est parfois absent ou minimaliste, et il peut être avantageux de supprimer des processus de la mémoire → réduire le degré de multiprogrammation (swapping, voir chapitre mémoire).



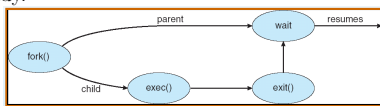
2.3 Opérations sur les processus

Les processus peuvent être concurrents, et peuvent être créés/supprimés dynamiquement. L'OS doit fournir un moyen de créer/ supprimer des processus.

Création. Appel système create-process, un processus parent crée un processus fils, on obtient un arbre des processus. Chaque processus a un unique pid (process id). Plusieurs politiques possibles:

- Partage de ressources (temps CPU, mémoire, fichiers, IO): père et fils peuvent tout se partager, partage d'un sous-ensemble des ressources du père, pas de partage (le fils obtient ses ressources de l'OS);
- Exécution: concurrente, ou bien le père attend que le fils ait terminé son exécution;
- Espace d'adressage: le fils peut être un dupliqua du père (même programme et données), ou être un nouveau programme.

Exemple d'UNIX: l'appel système **fork** crée un nouveau processus; **exec** charge un nouveau programme dans l'espace mémoire du processus. Processus fils: diffère du père par son pid (et ppid, pid du père), et statistiques d'utilisation remises à zéro. Linux: copie à l'écriture: duplication d'une page mémoire lorsqu'un processus en modifie une instance (pour avoir un fork très rapide). Appel système **wait** pour être enlevé de la file ready.



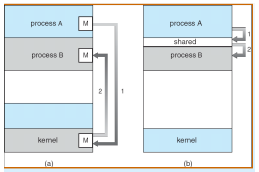
Terminaison. Appel système **exit** pour demander à l'OS de supprimer le processus; ressources désallouées (mémoire, fichiers ouverts, buffers d'IO). Un père peut terminer l'exécution d'un de ses fils avec **abort**, par exemple si le fils utilise trop de ressources, ou sa tâche n'est plus nécessaire. Si le père quitte, ses fils peuvent être tous terminés en cascade (exemple de l'OS VMS), ou bien alors le ppid des fils devient le processus initial init (exemple de Linux).

2.4 Processus coopérants

Le contraire des processus indépendants: ils peuvent affecter/ être affecté par l'exécution d'un autre processus.

Avantages:

- Permet l'accès concurrent à un fichier partagé;
- Accélérer les calculs si plusieurs CPU;
- Permet la construction d'un système modulaire;
- Commode même pour un seul utilisateur (éditer, compiler, imprimer).



Deux principaux modèles de communication: (a) envoi de messages (b) mémoire partagée