Systeme Exploitation

Fiche

0.1 Introduction Systeme Exploitation

Un OS (Operating System) = machine virtuelle plus facile à programmer et gestionnaire de ressources
Le système d'exploitation est

•Une machine virtuelle : pas de souci à comment invoquer les commandes des peripheriques. Les specificites materielles cachees aux programmeurs. : -Vue Uniforme des entrées/sorties -Une mémoire virtuelle et partageable -Gestions Fichiers et Répertoires -Gestion droits d'accès, sécurité et traitement d'erreurs -Gestions Processus (Ordonnancement, Communication,...
•Un gestionnaire de ressources : Gestion d'accés aux différents périphériques (clavier, écran, imprimante, interfaces réseaux, etc.) -Fonctionnement des Ressources -Contrôle l'accès aux Ressources -Gestion des erreurs -L'évitement des confits (ressource critique)
•Le système d'exploitation se charge -Des fichiers -Des processus -De la mémoire -Des E/S -Des utilisateurs

Types d'OS
Différents types d'OS pour différentes fonctionnalités :
Mono utilisateur : votre téléphone par exemple
Temps réel (Nucléaire, Chimie) : réactif
Général (Linux, Windows, Android, Mac OS,...) Multi-tâches et Multi-utilisateurs

Concepts d'un OS Les programmes en exécution sont encapsulés dans des processus.: Gestion multi-programmation. Gestion des communications entre programmes. Gestion des attentes de réponse des programmes. Droits associés aux restrictions fixés aux utilisateurs.

(Systèmes de) Fichiers : Les données stockées dans des objets appelés fichiers et on s'abstrait des disques. Gestion des droits d'accés aux fichiers.

Mémoire : Gestion de la mémoire d'exécution des programmes par espaces d'adressage. Mémoire transformée en mémoire virtuelle.

Un ensemble de fonctions systèmes avec des super-droits et une sémantique d'appel particulière appelée appels systèmes. L'interpréteur Shell est l'exemple type utilisant beaucoup les appels systèmes.

Appel Systeme
Principales Fonctionnalités d'un système Attention : 1 seule instruction par temps CPU
1 Proposer des services pour accéder au matériel: fonctions systèmes read, open, fork, dup, etc.
2 Traiter les erreurs matérielles des processus division par zero, seg fault, etc.
3 Traiter les interruptions matérielles erreur de lecture disque, ecran, souris, clavier, etc.
4 Entretien global: accés au processeur, allocation de mémoire, etc.
Par le mécanisme des appels systèmes

Problématiques
1 Protection matérielle.
2 Certaines instructions sont réservées (pour la protection par exemple) ou accés à certaines parties de la mémoire. Mémoire virtuelle et 2 modes d'exécution: utilisateur et noyau Mémoire virtuelle
1 Les adresses mémoire des programmes ne peuvent référencer les adresses physiques.
2 Les processus ont des espaces d'adressage virtuel
3 Lors du chargement les adresses virtuelles sont traduites en adresses physiques (changement de contexte) Un circuit Memory Management Unit fait la conversion à l'aide de registres Une table de conversion pour chaque processus

Modes d'Exécution
Utilisateur
1 Processus peut accéder uniquement à son espace d'adressage et à un sous-ensemble du jeu d'instructions. =) pas de corruption du système
2 L'accés à l'espace noyau est protégé et on y accède par une instruction protégée. Noyau
1 Accès à tous les espaces : noyau et utilisateur Code et données du SE accessible seulement en mode noyau: les segments mémoire sont inclus seulement lors du passage en mode mémoire.
2 Accés à toutes les instructions protégées (qui ne peuvent exécutées qu'en mode noyau) Instructions de modification segments de mémoire: un processus ne peut pas modifier ses droits d'accés à la mémoire. Accés aux périphériques: E/S, réseaux, allocation mémoire, etc.

Noyau?
Machine virtuelle
1 Vue uniforme des E/S
2 Gestion de la mémoire et des processus, réseau
3 Système de fichiers
Gestionnaire de ressources
1 Fonctionnement des ressources (processeur, délais, ...)
2 Contrôle d'accés aux ressources (Allocation CPU, disque, mémoire, canal de communication réseau, ...)
3 Gestion des erreurs
4 Gestion des conflits

Modes d'exécution
Mode noyau != mode root
1 Mode noyau = gestion par le matériel via des interruptions (matérielle et logicielle)
2 Mode root = gestion logicielle (par le code du SE) et est souvent en mode utilisateur. Mode noyau par le matériel
1 non connaissance lors de la compilation des segments de mémoire où se trouvent les fonctions systèmes.
2 Raisons: maintenabilité et portabillité du SE

Standard POSIX: Portable Operating System Interface
1 Est une interface de programmation système. Un ensemble de fonctions disponibles sur tous les SE *IX et pratiquement implémentées par tous. Un ensemble de types : time_t, size_t, dev_t, ...
2 Beaucoup de fonctions libc sont des wrappers: font juste appel à la fonction système (ex: time, E/S, etc.) Stocker les arguments dans les bons registres Invoquer l'appel système Interpréter la valeur de retour et si possible positionner la variable errno.

Principe Exécution Appels Systèmes (POSIX)

SCHEMA

1 Lors de l'initialisation on installe les codes des appels systèmes dans une table Interrupt handler et à chaque fonction système on associe un numéro interrupt
2 Dans le code de l'appel système on a une instruction de passage en mode noyau (sous Linux: int) qui prend en arguments le numéro de la fonction système et les différents arguments de la fonction.
3 Depuis le mode noyau

1 On appelle le gestionnaire d'exception trap handler: sauvegarde du contexte et transfert des données vers espace noyau.
2 Ce dernier à son tour appelle la vraie fonction système (indexée par son numéro)
3 Après calcul, transmission valeur de retour au trap
4 transmission de la valeur de retour et des données et retour en mode utilisateur (encore instruction protégée) après restauration du contexte

Types d'Appels Systèmes
Appel bloquant. Le processus appelant ne pourra continuer son travail que lorsque l'appel système a terminé (lorsque les données demandées sont prêtes par exemple). Ex: appels système: open, read, write
Appel non bloquant. On fixe un délai . La main est redonnée automatiquement au processus appelant si au bout de temps l'appel système n'a pas terminé. Ex: read, write. Il existe des fonctions pour passer d'un mode bloquant à un mode non bloquant ou inversement.
Attente active. Le processus simule lui-même un mode bloquant sur un appel non bloquant. Ex: while (1) / r= read (...); if (r$\geq$0) break;/

Gestion des erreurs
1 Une variable globale errno dans errno.h qui permet de transmettre les erreurs des fonctions systèmes aux codes utilisateurs
2 Un appel système qui réussit et alors le retour de la wrapper est un entier $\geq$0
3 Un appel système qui échoue et alors le retour de la wrapper est un entier $<$ 0 et un positionnement de la variable errno : numéro de l'erreur Les fonctions strerror(int) et perror(string) pour avoir/afficher le texte associé à l'erreur : perror("ouverture") affiche : "ouverture:"+message associé à errno (man 2 intro pour la liste des valeurs possibles de errno)

0.2 Memoire

Gestion de la Mémoire
Mémoire physique correspond à la mémoire du processus. Un seul processus à la fois en mémoire.
Multi-processing lourd = copies dans le disk de l'image d'un processus avant de switcher un nouveau.

SCHEMA
Chaque processus a son propre espace d'adressage dans la mémoire
2 problèmes : protection mémoire des processus et accés mémoire
Chaque processus a son propre espace d'adressage dans la mémoire
2 problèmes : protection mémoire des processus et accés mémoire
Solution 0 (IBM). Le processus a une adresse de base. Mais pose problème.

SCHEMA

Pas d'abstraction de la Mémoire Chaque processus a son propre espace d'adressage dans la mémoire
2 problèmes : protection mémoire des processus et accés mémoire
Solution 1. Le processus a une adresse de base et peut-être une taille limite. Protection (IBM): on peut donner des codes aux processus et coder dans les mots ces codes des processus.
Accés mémoire : on recalcule les adresses à partir des addresses de base.
Logiciel : à la compilation on détermine les accés adresses et on ajoute l'adresse de base. Qui est adresse?
Matériel : 1 registre qui transforme les accés adresse en ajoutant l'adresse de base.
Comment protéger les autres processus ? Adresse limite Lenteur : addition et comparaison avant chaque instruction d'adressage. Extension des processus ? Comment fixer la taille des processus ? Certains processus nécessitent plus que la mémoire physique.

SCHEMA Va et Vient

Gestion Mémoire Libre
Diviser la mémoire en blocs
Questions. Taille des blocs ? Gestion des blocs utilisés ? libres ?
Par bitmap : tradeoff taille bitmap/taille bloc ? Discuter recherche mémoire.
Par listes chaînées : discuter de différentes possibilités et de comment optimiser le choix pour un processus.

Mémoire Virtuelle par Pagination
Taille des processus trop grands pour la mémoire
Diviser le processus en pages = un bloc contigu d'adresses du processus.
Ces pages virtuelles sont mises en correspondance avec des parties contigües de la mémoire physique.
Expliquer au tableau : table des pages, mapping par TLB, changement de page.

SCHEMA

MMU
SCHEMA

Ramasse-Miettes aka Garbage Collector
On alloue de la mémoire statiquement (tableaux) ou dynamiquement (les pointeurs ou tableaux dynamiques). A certains moments certains ne sont pas utilisés et il faut libérer l'espace (sinon on risque de manquer d'espace). Une libération par le programmeur peut ne pas être complète et dans ce cas on a des fuites de mémoire.
Solution. Gérer au niveau système (OS ou machine virtuelle utilisateur) la libération de la mémoire Ramasse-Miettes.
Principe.
Déterminer les objets qui ne sont plus (ou peuvent plus être) utilisés par le programme. Non faisable à la compilation, par contre on peut identifier ceux non référencés pendant l'exécution. Libérer la mémoire utilisée par ces objets.

Algorithmes de Ramasse-Miettes
Plusieurs familles
Comptage de références : A tout bloc alloué est associé un compteur qui compte le nombre de références de ce dernier.
Algorithmes traversants : Les blocs de la mémoire forment un graphe orienté où chaque objet a 2 arêtes sortantes au plus :
-Un bloc référence au plus un bloc non utilisé et au plus un bloc utilisé.
-Ce graphe évolue : on déplacement de pointeur. L'algorithme consiste à faire un parcours à partir d'objets racines.
Générationnels : On hiérarchise les données suivant leur durée de vie et le libérateur commence souvent par les plus jeunes. Exemple : celui de .Net

Exemple d'Algorithme Traversant (Mark and Sweep)
Algorithme de Dijkstra •

0.3 Systeme de fichier

Implémentation Fonctions
Systèmes POSIX

0.4 Processus

Processus.
1 C'est un programme qui tourne en machine (le chargement d'un code en mémoire) Ensemble d'instructions et de données: code + données statiques + données alloués dynamiquement
2 Et une structure allouée par le système pour le contrôler = Environnement Une partie pour la gestion du processus (appartient au noyau) Une partie constituant le paramétrage du processus: arguments, variables d'environnement, etc.
3 Arborescence : Chaque processus a un unique père (sauf le 1er init).

Blocs de Contrôle d'un Processus (POSIX)
•Pid •État •Compteur ordinal •Allocation mémoire •Fichier ouverts... •Tout pour suspendre/reprendre le processus
Etats d'un Processus On est dans un environnement multi-processus.
1 Le processus utilise un laps de temps très bref le(s) processeur(s).
2 Particulièrement vrai pour un processus en attente d'une ressource (éviter de surcharger le système)
3 Ces différentes étapes d'un processus sont appelés Etats.
4 On a les états: Prêt pour l'exécution, Actif (utilise le processeur), bloquer/endormi (attend une ressource), suspendu (un utilisateur le désactive), zombie (réside en mémoire, mais ne peut être réactivé: par exemple pas de contrôleur de tâches pour le supprimer de la liste des processus)
5 Un processus peut demander intentionnelement à passer à l'état suspendu avec la fonction système sleep(unsigned int)

Création d'un Processus
1 Allocation d'un nouveau processus par clonage (appel système fork()) le fils hérite l'environnement du père (environnement d'exécution, variables système) C'est une copie et non un partage : faire attention aux effets de bords (partage de tampon par exemple), descripteurs de fichiers dupliqués (partagent le même décalage par exemple). Le fils reçoit un identifiant (numéro): pid différent de celui de son père ppid Si un processus devient orphelin, il est adopté par le processus initial (le 1)
2 Remplacement du code père par un autre si besoin (exec*(chemin,arg,...)) Par le passé recopier tout, d'où le remplacement Aujourd'hui: environnement seulement est copié et on récupère au fur et à mesure ce qui est important
3 Exemple au tableau
4 Synchronisation parfois nécessaire
Les différences (après duplication) entre père et fils sous POSIX
1 pid différents de tous les autres pids etpgid
2 ppid fils = pid père
3 Mesures de temps consommés initialisé à 0 [normal car pas encore consommé du processus]
4 Les verrous posés par le parent ne sont pas hérités [un verrou est relatif à un pid] Un fichier verrouillé ne sera pas dupliqué
5 Minuterie désactivée
6 Ensemble des signaux pendants du père au moment de la duplication est initialisé à $\emptyset$

Chargement d'un exécutable
1 Construction d'un nouveau code à partir d'un exécutable (précédent supprimé)
2 Initialisation de la liste d'arguments à transmettre au main
3 Initialisation de la variable globale environ (peut être une copie du processus)
4 Placement dans le CO l'adresse du main (on appelle le main en gros)
5 Six fonctions dans POSIX: execl, execvl, execle, execve, execlp, execvp : Descripteurs de fichiers verrouillés fermés (au niveau descripteur) répertoires ouverts fermés On ignore les signaux ignorés et le reste on prend le comportement par défaut Les éléments d'environnement restants sont inchangés (répertoire courant, répertoire racine, masque, signaux pendants, etc.) Si bit de positionnement (le fameux s à la place du x), alors on change uid avec celui de l'exécutable (de même pour gid)

Terminaison (1)
1 exit(int) (vidage tampons, flots ouverts fermés, etc.) ou _exit(int etat) (plus bas niveau)
2 L'état d'un processus arrêté peut être retourné à son parent si gestionnaire de tâches existe.
3 Utiliser les fonctions wait (bloquant) et waitpid (bloquant ou non et on peut demander une notification). Dans ce dernier cas on peut demander des infos d'arrêt sur plusieurs processus.
4 Faire attention aux processus fils zombis (en particulier il faut vraiment gérer les processus qui ne sont pas vraiment morst, mais juste par exemple stoppés)

Le père est averti de la terminaison du fils (un signal SIGCHLD)
1 Cas 1: Le père récupère le code de retour du fils (donné à return ou exit) et appelle wait pour l'obtenir afin de libérer les ressources associées au fils
2 Cas 2: le père ne récupère pas le code de retour, le fils devient un zombie: Lorsque le père meurt, les processus sont définitivement tués.
3 Cas 3: le père meurt avant le fils, le fils orphelin est adopté par init

Processus Zombie
Bref un père qui n'attends pas son fils Un processus zombie et orphelin est adopté puis attendu !
Avec par exemple le code suivant : while(waitpid(1, NULL, WNOHANG))
Que faire pour éviter les zombis ? = Déléguer l'attente à un processus
1 On crée un processus fils
2 Le processus fils crée le processus que l'on voulait créer et meurt aussitôt
3 Celui que l'on voulait créer est adopté par init
4 Il ne peut pas être zombi car init nettoie (il est l'initial)
5 C'est la seule solution desfois car on n'a pas accés direct à init

Manipulation des processus et Gestion des Processus
EXEMPLE EN C

Les threads
Un thread est un chemin d'exécution d'un processus
Les threads sont inclus dans un processus

Un thread est un chemin d'exécution d'un processus
Les threads sont inclus dans un processus:
Les threads d'un même processus partagent les ressources du processus sont partagés entre ces threads) Code exéc., mémoire, fichiers, périphériques...
Chaque thread possède en plus : sa zone de données, sa pile d'exécutions, ses registres et son compteur ordinal

SCHEMA
Pourquoi les threads ?
Rapide à créer : pas de contexte à créer
Communication inter-thread super simple et rapide !
Réactivité : un thread pour attendre l'user les autres bossent Parallélisation d'une tâche !
Plus diffcile à manier :Synchronisation et Crash

Les fonctions de la bibliothèque pthread :
int pthread_create(pthread_t * thread, pthread_attr_t * attr, (taille de la pile, priorité, ...) void *nomfonction, void *arg);
int pthread_join(pthread_t *thid, void **valeur_de_retour);
void pthread_exit(void *valeur_de_retour);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);)

sert à établir l'état de terminaison d'un processusléger :
PTHREAD_CREATE_DETACHED : le processus léger libérera ses ressources quand il terminera
PTHREAD_CREATE_JOINABLE : le processus léger ne libérera pas ses ressources

0.5 Pilotes aka Drivers

0.6 Architecture Modulaire

0.7 TD/ TP

tab titre

...	...	...