
Système d'exploitation

Copyright et Licence

- Copyright © 2007-2019 Brice Goglin.
- Ce support de cours est diffusé sous licence *Creative Commons CC-BY-NC-SA*.
 - <http://creativecommons.fr/licences/les-6-licences/>

Ceci n'est PAS un polycopié de cours.

Ces « notes » guident simplement le cours.
De nombreuses explications et illustrations manquent, les détails seront donnés au tableau et à l'oral pendant le cours magistral.

Pour me joindre

- Bâtiment Inria, 200 avenue de la vieille tour
- Bureau B425, 05.24.57.40.91
- Envie de découvrir la recherche en système ou HPC ? Stage, thèse, ...

- Brice.Goglin AT inria.fr
- <http://people.bordeaux.inria.fr/goglin/>
- Facile à trouver dans Google !

Programme du module

- ~~15~~ **11** séances de cours
- ~~10~~ **8** séances de 2h de TD
 - Prévoyez vos laptops pour les deux dernières
- Slides du cours et sujets de TD en ligne
 - <http://people.bordeaux.inria.fr/goglin/teaching/Systeme.html>
 - Beaucoup d'illustrations et explications y manquent
- Examen d'environ 2h
 - Tous documents autorisés
- Projet système (IT202) début février

Sondage

- Qui n'a pas suivi de **prog système** ?
- Qui n'a pas suivi de cours d'**architecture** ?
- Qui n'a jamais **utilisé** Linux ?

- Qui a déjà **compilé** un noyau Linux ?
- Qui a déjà **regardé** les sources de Linux ?
- Qui a déjà **modifié** les sources de Linux ?
- Qui a déjà **écrit** un système d'exploitation ?

Positionnement du cours

- Structure en couche
 - En dessous de la programmation système
 - Au dessus de l'architecture
- Domaine vaste
 - des concepts logiciels...
 - ... aux implémentations matérielles
 - en passant par les algorithmes mis en jeu
- Illustration surtout par le noyau Linux
- Pré-requis : architecture et prog système

Plan du cours

- Introduction
- Concepts généraux
- **Processus et exécution**
- **Gestion mémoire**
- Concurrence et synchronisation
- Gestion du temps
- ~~Systèmes de fichiers~~
- ~~Entrées-sorties~~
- Virtualisation

Bibliographie

- Polycopié de l'ancien prof
- Livres sur les systèmes d'exploitation
- Livres sur Linux
- Ressources en lignes diverses

- Voir la page web du cours !

Système d'exploitation

Introduction

Pourquoi des systèmes d'exploitation?

- OS (*Operating System*)
- Il arrive parfois qu'on n'en utilise pas
 - Quelques systèmes embarqués
 - Mais souvent une bonne idée d'en utiliser un
- Exposer interface virtuelle indépendante du matériel
 - Gérer les différents matériels et leurs limites
 - Rendre les choses simples et uniformes pour l'utilisateur, le programmeur et les applications
- Gérer de multiples tâches et utilisateurs

Pourquoi est-ce intéressant?

- Connaître son fonctionnement permet de mieux exploiter les machines

- Inutile de réinventer la roue, l'OS fait déjà plein de choses pour vous

Probablement le logiciel le plus complexe

- Des aspects très techniques (matériel)
- Du génie logiciel (les interfaces doivent être pratiques, efficaces et fiables)
- De l'optimisation (performances cruciales)
- Des compromis
 - Supporter tout à peu près bien?
 - ... ou supporter certains cas parfaitement?
- De l'algorithmique, des heuristiques
 - Deviner ce que l'utilisateur/application veut
 - S'y adapter, s'y préparer à l'avance, ...

2018-2019

Brice Goglin

13

Ceci n'est PAS le système d'exploitation

- Compilateurs
 - et les bibliothèques ?
- Système de fenêtrage (X.org, ...)
 - même si il touche aux périphériques
- Interpréteur de commandes (bash, ...)
- Outils en ligne de commande (cp, mv, ...)
- Le super-utilisateur (root)
- Un processus « homme en noir » qui surveille et contrôle tout depuis le fond

2018-2019

Brice Goglin

14

Ceci peut être un système d'exploitation

- Le noyau
 - Le coeur, qui s'exécute en mode privilégié
- Eventuellement des bibliothèques autour
 - La bibliothèque standard (libc sous Linux)
 - Fournit les appels système
 - ex: GNU/Linux vs. Linux
- Eventuellement des outils, services, démons, ...
 - Qui rendent l'OS plus facile à utiliser

2018-2019

Brice Goglin

15

root vs. noyau

- Le super-utilisateur n'est pas le noyau
 - Il a juste un accès privilégié au noyau
 - Le noyau le laisse faire plus de choses
- Le noyau contrôle tout
 - Il y accède par les appels-système
 - Il vérifie ce que les utilisateurs veulent faire
- root privilégié **logiciellement** par le noyau
 - Le noyau est privilégié **matériellement**
 - par le processeur

2018-2019

Brice Goglin

16

Exemples d'OS

- Linux (sur des dizaines d'architectures)
- Windows (x86 et ARM)
- Solaris (Sparc et x86)
- FreeBSD, NetBSD, OpenBSD, ...
- GNU/Hurd
- MacOS X (x86 et PowerPC)
- PalmOS, Plan9, Symbian, VxWorks, ...

Système d'exploitation

Petite digression :
Ordre de grandeurs

Exécution

- Processeur
 - Dizaine(s) de coeurs
 - Fréquence
 - ~3 Ghz
 - Instruction (cycle)
 - 1 ns
 - Instruction atomique
 - 100 ns
 - Appel-système
 - 50-1000 ns

Mémoire et I/O

- Mémoire
 - 10 Go, 50-100 ns, 50 Go/s
- Mémoire cache
 - 10 ko - 100 Mo, 1-10 ns
- Bus PCI
 - Go/s, 100+ ns
- Interruption
 - 5 μ s

Stockage et réseau

- Disque
 - Mécanique
 - 10 To, 10 ms, 10-100 Mo/s
 - SSD, NVMe, ...
 - 0,1-10 To, 1-100 μ s, 0,1-10 Go/s
- Réseau
 - A la maison
 - 10-1000 ms, 1-100 Mo/s
 - Réseaux hautes performances
 - 1 μ s, 10 Go/s

Système d'exploitation

Encore une petite digression :
Que se passe-t-il pendant le boot ?

Boot matériel

- Au démarrage, les processeurs ne savent pas quoi faire
 - Et la mémoire est vide
- Le premier coeur (*Bootstrap Processor*) est configuré pour exécuter le code stocké à une adresse spéciale
 - Le système est configuré pour avoir de code de démarrage à cette adresse
 - Cette adresse pointe vers une mémoire spéciale
 - ROM (*Read Only Memory*), non volatile

BIOS, UEFI, OpenFirmware, ...

- Le processeur exécute le BIOS (*Basic Input Output System*)
 - ou UEFI (*Unified Extensible Platform Interface*) sur machines récentes
 - ou Open Firmware sur les architectures non-x86
 - plus ou moins portable
- C'est l'interface bas-niveau avec le matériel
 - Détecte les composants
 - Autres processeurs, mémoire, périphériques, ...
 - Et les ressources (mémoire embarquée, ports, ...)

BIOS, UEFI, OpenFirmware, ... (2/2)

- Sait faire plus ou moins de choses
 - Configurateur graphique
 - Réglages du processeur, choix du périphérique de démarrage, etc.
 - Exécution de code depuis différents endroits
 - Début du premier disque, CD, USB, réseau (PXE), ...
- Une fois terminé, il charge le code de l'OS
 - Depuis le disque ou le réseau
 - Souvent au début du premier disque dur
 - *Master Boot Record*
 - Puis l'exécute

Boot Loader

- Après le BIOS, on est pas obligé de démarrer un vrai OS
- Le *Boot Loader* (Lilo ou Grub) est un mini-OS conçu pour choisir
 - Quel OS lancer ?
 - Sur quelle partition, avec quel fichier de noyau, ...
 - Impose de savoir lire ce fichier
 - Il peut être stocké de manière discontigue sur le disque
 - Avec quels paramètres, ...

Boot du noyau

- Le *Boot Loader* saute à un autre programme
- Le noyau se lance
 - détection des ressources matérielles via le BIOS
 - création des ressources logicielles de base
 - chargement du code
 - table de page
 - descripteurs de périphériques
 - ...
 - démarrage des autres coeurs
 - quelques démons (threads noyau)
 - travaux système en arrière-plan

Boot du système d'exploitation

- La tâche « chargement du noyau » se transforme en un vrai processus utilisateur : *Init*
 - Le noyau est fonctionnel, l'OS démarre
- *Init* charge les services logiciels configurés par l'administrateur
 - Gestionnaire de périphériques
 - Chargement de modules pilotes
 - Démons utilisateurs
 - Services utilisateur en tâche de fond
 - Gestionnaire de session
 - ...

Concepts généraux

Plan

- Les principaux concepts
- Structure des systèmes
- Exemples de systèmes

Concepts généraux :

Les principaux concepts

Objectifs des systèmes d'exploitation

- Rendre le système plus facile à utiliser
 - Abstraction des périphériques
- Améliorer l'efficacité du système
 - Fournir des ressources efficacement
- Facilité d'évolution
 - Continuer à fournir les mêmes services de la même façon
- Protection des différents programmes
- Sécurité vis-à-vis des autres utilisateurs

Interface entre utilisateur et machine

- Interface uniforme d'accès aux périphériques
 - Détails techniques cachés
- Exécution de programmes
- Accès contrôlé aux données
 - Montrer le contenu structuré des périphériques de stockage et réseau
 - Fournir des fonctions d'édition, développement, communication, ...

Interface entre utilisateur et machine (2/2)

- Gérer les erreurs proprement
 - Signaler aux programmes les erreurs matérielles
 - Réagir en cas d'erreur logicielle
 - Punir ou corriger
- Fournir des statistiques d'utilisation et fonctionnement
 - Permet à l'utilisateur de mieux adapter la configuration du système

Évolution historique

- *Serial Processing* (~1950)
 - Lancement manuel d'un nouveau programme à la terminaison du précédent
- *Simple Batch Systems* (~1960)
 - File d'attente de programmes
- *Multiprogrammed Batch Systems* (~1970)
 - Exploiter le matériel en alternant les processus
- *Time Sharing* (~1980)
 - Alternance automatique entre processus
 - Interactivité

Processus

- Contexte mémoire propre
 - Exécution indépendante des autres processus
- Une file d'exécution (registres + pile)
 - Eventuellement plusieurs si multithreading
 - Permet I/O bloquante sans bloquer tout le processus
- Exécution concurrente pour maximiser l'utilisation des ressources matérielles

Systeme multi-tache et preemption

- Systeme multi-taches
 - Existence et execution de plusieurs taches simultanement
- Systeme preemptif
 - Ordonnement non-contraint a la bonne volonte du processus en cours
 - Ordonnement de force par le systeme

Gestion memoire

- Isolation des processus
 - Pas de collision entre leurs memoires
 - Donnees et instructions propres
 - Grace a la memoire virtuelle et au support materiel
- Allocation et gestion transparente
 - Pas de contraintes sur le programmeur
- Programmation modulaire
 - Bibliothèques réutilisables
- Partage de memoire avec protection
- Stockage en memoire persistante

Protection des donnees et securite

- Controle des autorisations
 - Acces aux donnees critiques du systeme
 - Acces et modification des donnees des utilisateurs
- Authentification des utilisateurs
- Survie a un probleme technique ?

Ordonnement et gestion des ressources

- Acces aux ressources equitable
 - Pour les travaux de meme type
 - Quid des processus multi-threadés
 - En distinguant des classes de travaux
 - Interactif ? I/O ? Calcul pur ?
- Controle dynamique
- Efficacite
 - Utilisation du materiel et reactivite

Ordonnancement et gestion des ressources (2/2)

- Ordonnancement de processus et/ou d'entrées-sorties
 - Dans quel ordre envoyer les lectures/écritures sur le disque ?
- Contraintes *Deadline*
 - Événement doit arriver avant...
 - Systèmes temps réel, ...
- Contraintes matérielles
 - Ex: *Elevator*

Les architectures sont parallèles : SMP et NUMA

- Plusieurs processeurs
 - SMP (*Symmetric MultiProcessor*)
 - Accès concurrents
 - Affinité pour un processeur
 - Cache, TLB, ...
 - Affinité des threads d'un même processus
- Contraintes NUMA
 - NUMA (*Non-Uniform Memory Access*)
 - Affinité pour certaines zones mémoire
- Nouvelles contraintes d'ordonnancement

Système d'exploitation

Concepts généraux :

Structure des systèmes

Structure du système d'exploitation

- Noyau
 - Coeur
 - Pilotes de périphériques
 - Interface vers l'utilisateur
- Bibliothèques utilisateur de bas niveau
 - Appels système pour parler au noyau
 - Incontournable !
 - Assuré par le matériel
- Et au dessus il y a vos applications

Structure (2/2)

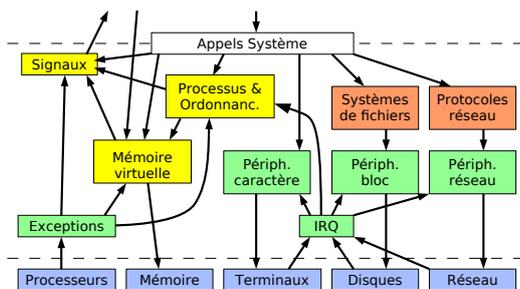
- Gestion mémoire au coeur du système
 - Processus
 - Systèmes de fichiers
 - Réseau
- Systèmes de fichiers et stockage
- Ordonnancement
- Communications entre processus
- Réseau

Noyaux monolithiques

- Ensemble de procédures pouvant toutes s'appeler les unes les autres
 - Très peu d'organisation
- Souvent découpé en
 - Procédures principales
 - Appels-système
 - Procédures de services
 - Traitement effectif
 - Procédures utilitaires
 - Gestion de listes, tables de hachage, ...

Linux monolithique ?

(cf aussi l'*Interactive Map* en lien sur le site du cours)



Noyaux en couche

- Tentative d'organisation des monolithiques
- Limites entre couches difficile à définir
 - La gestion mémoire a besoin des I/O pour remplir les pages
 - Les I/O ont besoin de mémoire pour allouer des tampons intermédiaires
- Traversée des couches contraignantes pour les performances

Noyaux modulaires

- Chargement/déchargement dynamique de code optionnel
 - Pilotes de périphériques
 - Fonctionnalités spécifiques
- Réduction du noyau initial
- Possibilité d'évolution dynamique
 - Sans redémarrage

Micronoyaux

- Noyaux monolithiques trop gros ?
 - Pas organisés, même avec des couches
 - Difficile à organiser clairement et efficacement
 - Sécurité difficile à intégrer car trop d'interactions possibles
 - Difficile à maintenir et faire évoluer
- Micronoyaux
 - Uniquement le strict nécessaire
 - Le reste dans des processus serveurs dédiés
- Éternelle dispute entre Torvalds et Tanenbaum

Micronoyau (2/3) : Design

- Un serveur pour chaque tâche
 - Processus, gestion mémoire, ordonnancement, réseau, systèmes de fichiers
 - Dans des espaces mémoire séparés
 - Passage de message entre applications et serveurs
 - Validés par le micronoyau
- Interfaces simples et uniformes
- Extensible, flexible
- Fiable

Micronoyaux (3/3) : Performance

- Passage de message plus lent qu'un appel direct de procédure du noyau ?
- Trop d'interactions critiques entre composants du système d'exploitation ?
 - Mémoire et systèmes de fichiers ?
- Difficile de comparer avec noyaux monolithiques
 - Pas de vrai OS fonctionnel entièrement basé sur un micronoyau
 - GNU/Hurd ?

Système d'exploitation

Concepts généraux :
Exemples de systèmes

Windows

- Très modulaire
 - Ensemble de *Managers (Executive)*
 - *I/O, Cache, PnP, Power, Security, VM, Process, ...*
 - Couche d'abstraction des périphériques
 - Pilotes de périphériques
 - Fenêtrage et graphisme
- Noyau hybride (pseudo micro-noyau)
 - Quasiment tout tourne dans un seul espace noyau
 - Pour des raisons de performance

Windows (2/3)

- *Local Procedure Calls*
 - Échange de message entre applications et managers
 - Modèle *Client-Serveur*
 - Permet communication entre modules, qu'ils soient en mode noyau ou utilisateur
- Design orienté objet (Polymorphisme)
 - Les *Handles* permettent de manipuler tout et n'importe quoi de la même façon

Windows (3/3)

- Support SMP, threads, communications entre processus, ...
- Ordonnanceur de threads préemptif avec priorités
 - Avec affinité pour cache et NUMA
 - Priorité ajustée dynamiquement selon attente passive (GUI) et consommation CPU
- Conçu pour bureautique (initialement)

Unix

- Créé à la fin des années 60
- Design pour serveur et réseau
- Portable car rapidement écrit en C
- Tout est fichier
- Noyau (*kernel*) + Ensemble de bibliothèques et applications (*libc*)
 - Appels système (*System Call Interface*)

Unix (2/3)

- Noyau monolithique
 - Fonctionnalités communes
 - Gestionnaire mémoire
 - Périphériques bloc
 - Périphériques caractère
 - Ordonnanceur
 - Interface Vnode/VFS pour les fichiers

Unix (3/3)

- *System V Release 4*
 - Académique et commercial (AT&T et Sun)
- *Solaris (SunOS)*
 - Distribution commerciale (Sun) basé sur SVR4
- *Berkeley Software Distribution (*BSD)*
 - Très répandu dans le monde académique
 - Base de Mac OS X
- *Minix*
 - Clone à but éducatif
- Voir lien vers arbre généalogique

Linux

- Apparue en 1991
- Basé sur *Minix*
 - Pas trop cher pour usage personnel
- Libre (GPL-2)
- Développement collaboratif sur Internet
- Supporte de nombreux architectures et périphériques matériels
- Conçu pour serveurs, stations de travail, bureautique, embarqué, ...

Linux (2/2)

- Chargement dynamique de modules noyau
 - Chargement automatique selon les besoins des applications
 - Hiérarchie basée sur dépendances de symboles
- Structure monolithique particulière
 - Seules certaines fonctions sont accessibles aux autres modules
 - EXPORT_SYMBOL()
 - Pas d'interface fixée

The GNU Hurd

- Le seul vrai OS basé sur un micronoyau ?
 - Unix sur micronoyau *Mach*
 - Pilotes de périphériques limités
- IPC entre composants via le micronoyau
 - Processus serveur et applications
 - L'utilisateur peut lancer ses propres serveurs
 - Réseau, système de fichiers, ... sans être root
 - Sans impacter le reste du système
- Notion de *Translator* (voir lien)
 - Fonctionnalité (fichier, répertoire, accès réseau, ...) gérée par processus
 - Parcours aisé d'un site web, serveur ftp, archive, ...

Système d'exploitation

Processus et exécution

Plan

- Description des processus
- Exécution des processus
- Processus et threads
- Vie et mort des processus
- Ordonnancement
- Algorithmes d'ordonnancement
- Exécution du système

Système d'exploitation

Processus et exécution

Description des processus

Objectifs des processus

- Toutes les applications semblent progresser simultanément
 - Les processeurs physiques les exécutent en alternance
 - Chaque application a l'illusion d'être seule sur la machine
- Partager les ressources disponibles entre de multiples applications
- Le processeur et les périphériques sont utilisés efficacement

Définitions d'un *Processus* ?

- Programme en exécution
- Instance d'un programme s'exécutant sur un ordinateur
- Entité pouvant être assignée et exécutée sur un processeur
- Unité d'activité caractérisée par l'exécution d'une suite d'instructions, un état courant et un ensemble de ressources système

Caractérisation

- A l'initialisation
 - Code du programme
 - Ensemble de données en entrée
- Durant l'exécution
 - Bloc de contrôle
 - Contient un identifiant
 - PID (*Process Identifier*)

Bloc de contrôle d'un processus

- Identifiant
- Adresses mémoire
 - Données utilisateur, programme, pile, ...
- *Program Counter*
- État d'ordonnement
 - En exécution, prêt, bloqué, mort, ...
 - Priorité
- État des entrées/sorties
- Privilèges
- Statistiques

Généalogie

- Le processus a un père
 - Et peut avoir des fils
 - Le père est notifié de la mort des fils
 - Et peut l'attendre et savoir s'il a réussi
- Cas des orphelins
 - Il leur faut un père
 - Dans Linux, c'est *init* (PID=1) par défaut
 - Modifiable avec `prctl()`

Système d'exploitation

Processus et exécution
Exécution des processus

Beaucoup d'architectures différentes

- x86/x86_64
- IA64
- Alpha
- PowerPC
- Sparc
- ARM
- MIPS
- ...
- Quel est le plus vendu ?

Exécution dans les processeurs

- Une instruction par cycle (par pipeline)
 - Durée du cycle fixe, reliée à la fréquence
- Interruption régulière par horloge
 - Permet de donner la main au système régulièrement
 - Environ 1 fois par milliseconde sous Linux
- Différents modes d'exécution avec différents privilèges

État du processeur et contexte d'exécution

- Registres utilisateurs
 - Données
 - Adresses
 - Segment, pile, ...
- Registres de status et contrôle
 - *Program Counter* (PC)
 - Status arithmétique

Modes d'exécution

- Plusieurs niveaux de privilèges
 - Mis en place matériellement
 - Par le processeur
 - Accès aux registres de contrôle
 - Instructions d'entrées/sorties de bas niveau
 - Gestion mémoire
 - Ex: Accès à CR3 pour la table des pages sur x86
 - Accès à certaines zones mémoire
 - Limité par les structures pointées par certains registres

Modes d'exécution (2/3)

- Différents niveaux de privilèges
 - Jeu d'instructions réduit selon les privilèges
- Le noyau peut tout faire
 - *Protection Ring 0* sur x86 (mode protégé)
 - Sauf en cas de virtualisation matérielle
 - Seul l'hyperviseur peut vraiment tout faire

Modes d'exécution (3/3)

- L'utilisateur est très limité
 - *Protection Ring 3* sur x86
 - Pas d'accès à la mémoire du noyau
 - Non visible dans la table de pages
 - qui n'est pas modifiable sans privilèges
 - Pas d'accès bas niveau au matériel
 - Sauf si autorisation préalable (ex: ioperm, iopl)

Exceptions et interruptions

- Événements inattendus qui interrompent temporairement l'exécution en cours
- Le processeur saute automatiquement et immédiatement à un traitant
 - Fonction dont l'adresse a été définie par le système d'exploitation au démarrage
 - Dépend du type d'exception ou interruption
- Déroutement
 - Suspension du programme en cours
 - Retour au code initial à la fin du traitant

Exceptions

- Interruption par le processeur lui-même
 - En cas d'erreur
 - Erreur arithmétique (division par zéro, ...)
 - Mauvais accès mémoire (segfault, ...)
- Processeur ne sait pas comment continuer !
 - Il demande à l'OS de l'aider
 - Déroutement vers traitant qui va corriger l'erreur
 - Reprise ensuite de l'exécution au même endroit
 - L'instruction fautive est re-exécutée
 - L'application ne se rend compte de rien

Exceptions (2/2)

- Exception uniquement possible quand le processeur travaille
 - Nécessite contexte d'exécution
 - Processus, thread noyau, ...
 - Qu'il soit dans le noyau ou en espace user
- Reprise de l'exécution uniquement si le traitant a réparé l'erreur avec succès
 - Ex: Défaut de page
 - Sinon la tâche est tuée
 - Ex: Segmentation Fault

Interruptions

- IRQ = *Interrupt Request*
- Interruption = Message d'un périphérique
 - Signal électrique sur broche dédiée du processeur
 - ou Ecriture à adresse mémoire spéciale (si matériel récent)
- Evénement asynchrone
 - Le processeur pourrait continuer à tourner en l'ignorant pendant un certain temps
 - Mais OS et/ou applications auraient des problèmes
 - Entrée-sortie non terminée, carte réseau saturée...

Interruptions (2/2)

- Interruption possible n'importe quand
 - Même si le processeur ne fait rien
 - Il suffit qu'un périphérique ait été initialisé
- Déroutement temporaire dans le pilote du périphérique
 - Traitement de la requête
 - Réception paquet réseau, ...
- Pas d'influence directe sur la tâche qui s'exécutait
 - L'application ne se rend compte de rien
 - Mais peut réveiller des tâches qui attendaient la fin d'une entrée/sortie

Exceptions et interruptions (2/2)

- Traitement très similaire
 - Passage temporaire en mode privilégié
 - Un seul cœur (une seule tâche suspendue)
 - Exécution du traitant spécifique
 - Selon exception ou interruption et paramètres
 - Retour à l'exécution initiale
- Exception spéciale pour appels système
 - Changement de mode d'exécution
 - Passage du mode utilisateur au mode noyau
 - Augmentation temporaire des privilèges
 - **Instruction spéciale pour revenir**

Cas particulier d'exception : Les appels-système

- Exception forcée par l'application
 - Pour effectuer opération privilégiée
 - Changement de mode d'exécution du processeur à la demande
- Se produit uniquement depuis l'espace utilisateur
 - A la demande des applications
- Traité dans le contexte d'un processus
 - En mode privilégié

Cas particulier d'exception : Les appels-système (2/2)

- Déroutement
 - Passage à un mode plus privilégié par instruction spéciale
 - Exception dédiée (int80, ...)
 - *Callgate*
 - Instruction dédiée (sysenter, syscall, ...)
 - Appelé par l'application en mode utilisateur
 - Généralement par des fonctions de la libc
- Passage obligatoire de toute application
 - Impossible de tricher
 - Vérification des paramètres en mode privilégié

2018-2019

Brice Goglin

85

Appels-système (1/4) Comment ça marche ?

- Traitement de l'appel système dans le traitant de l'instruction spéciale
 - Adresse du traitant défini par le noyau au boot
 - Comme pour les traitants d'interruption/exception
 - Le processeur y saute immédiatement lors de l'appel système
 - Exécution temporaire en mode privilégié
 - Le processus n'est pas endormi, son exécution est déournée
 - Appel d'un sous-traitant dans la table des appels système
 - Table non modifiable depuis l'espace utilisateur

2018-2019

Brice Goglin

86

Appels-système (2/4) : L'appel

- Convention de passage des paramètres
 - Numéro dans %eax sur Linux/x86
 - Arguments: %ebx, %ecx, %edx, %esi et %edi
 - Et dans la pile si nécessaire (mmap)
- Sauvegarde des paramètres
- Sauvegarde du contexte utilisateur
- Appel du traitant
 - Après mise des paramètres dans registres
 - Saut à `syscall_table[numéro]`

2018-2019

Brice Goglin

87

Appels-système (3/4) : Le retour

- Restauration des registres utilisateurs
- Convention pour le passage de la valeur de retour de l'appel système
 - Ex: %eax pour Linux/x86
- Possibilité de passer plus de paramètres ou valeurs de retour
 - Pointeurs dans les arguments
 - Le noyau peut lire/écrire en espace utilisateur
- Retour concret en espace utilisateur par instruction spéciale (`iret`, `sysexit`, ...)

2018-2019

Brice Goglin

88

Appels-système (4/4) : Concrètement dans Linux

- Code assembleur caché dans la glibc
 - On appelle des fonctions « normales »
- 386 appels-système dans Linux 4.21/x86
 - De nombreux sont dépréciés
 - 52 non implémentés sur x86_64
 - La glibc s'adapte à ceux qui sont disponibles dans le noyau utilisé sur la machine
- Voir le code de la glibc et du noyau Linux

Appels-système virtuels

- Appels système chers
 - 100ns sur x86 modernes, parfois plus de 1µs
- Parfois idiot
 - Le retour de getpid() varie « peu »
- Optimiser certains appels en espace utilisateur
 - Mapping de code et données simples du noyau (gettimeofday, getcpu, ...)
 - Voir <http://lwn.net/Articles/615809/>
 - Se rappeler de certaines valeurs (getpid, getppid, ...)
- Risques de sécurité?
 - Appel-système virtuel pour setuid() ?

Contextes d'exécution

- Processus en mode utilisateur
- Processus en mode noyau pendant un appel-système
- Processus en mode noyau pendant un traitement d'interruption/exception
 - Mode privilégié temporaire
 - Processus suspendu temporairement
 - Reprend la main à la fin du traitement
 - Exécution restreinte car contexte très limité
 - Interruptions désactivées, ...

Ajouter des appels-systèmes?

- Dangereux pour des raisons de sécurité
 - Ex: Ajouter appel-système « je deviens root »
- Table statique
 - Difficile à modifier
 - Souvent la cible des pirates
 - Modification ou remplacement
- Numéros fixés par compatibilité binaire
- Comment faire des commandes à la demande ? Les IOCTL (cours I/O)

Système d'exploitation

Processus et exécution

Vie et mort des processus

Création de processus (Windows)

- CreateProcess(...) crée un processus à partir d'une ligne de commande
 - Nouvel espace d'adressage (et ressources)
 - Nouvelle file d'exécution dans cet espace
- Si on veut modifier des attributs, il faut passer plein d'arguments
 - Ou utiliser CreateProcessAsUser(...)
 - Pas très flexible/extensible

Création de processus (Unix)

- Création du nouveau processus PUIS exécution d'une commande
 - fork() + exec() au lieu de CreateProcess()
 - On peut configurer plein de choses entre les deux
- Fork() pour dupliquer
 - Duplication de l'espace d'adressage
 - Partage de certaines ressources
 - Nouvelle file d'exécution dans cet espace
 - Fils identique au père, sauf le PID
- Exec() pour transformer
 - Ligne de commande remplace le processus

Création de threads

- On rajoute une file d'exécution dans l'espace d'adressage courant
 - pthread_create()
 - clone()
 - Partage de certaines ressources

Lancement d'une tâche

- Le noyau crée un contexte utilisateur
 - Pointant vers la fonction à exécuter
 - main() pour un processus
 - Fonction dédiée pour un thread
 - Avec nouvelle pile
 - Quid de l'ancien contexte ?
 - Détruit en cas d'exec
 - Conservé si en cas de nouvelle tâche
 - Inexistant dans le cas de Init
- Restauration du contexte puis passage en espace utilisateur

Terminaison d'une tâche

- La terminaison se passe dans le noyau
 - Il faut d'abord un moyen d'y entrer
 - Appel système exit ou exception
- La file d'exécution est détruite
 - Elle relâche les ressources qu'elle utilisait
 - Le dernier utilisateur détruit la ressource
 - Identique pour partage entre threads ou processus

Terminaison d'un thread

- pthread_exit() est appelé implicitement ou explicitement
 - Les ressources spécifiques sont détruites
- Le dernier fait l'appel système exit qui détruit le processus
 - Les ressources partagées sont détruites

Terminaison d'un processus

- La tâche qui termine renvoie un code de retour
 - Destiné au père
- Le père peut attendre la terminaison d'un ou n'importe quel fils
 - wait() et waitpid()
 - Récupère le code de retour
- Tâche *Zombie* tant que le père ne s'en occupe pas
 - La plupart des ressources peut être libérée

Vie des processus et états d'ordonnement

- Lors de l'exécution « normale »
 - Alternance régulière entre *Ready* et *Running* selon décisions de l'ordonnanceur
- Lors d'un appel-système bloquant
 - Sommeil sur file d'attente en attendant événement
 - Retour en *Ready* lors de l'événement
 - Ex: interruption d'un périphérique, action d'un autre processus, ...

La fonction de changement de contexte

- **2** tâches mis en jeu à chaque changement de contexte
- **3** processus du point de vue de la tâche désordonnée puis réordonnée
 - La tâche elle-même
 - Un successeur avant changement
 - Un prédécesseur après retour
 - La tâche voit 2 demi-changements différents
- `schedule()`, `switch_mm()` et `switch_to()` dans Linux

Runqueues

- Les processus prêts à s'exécuter sont dans une file d'attente spéciale (*Runqueue*)
 - Soit une seule globale
 - Soit une par processeur
- Uniquement les processus Ready !
 - Les processus non prêts sont dans files spéciales, avec un moyen de les retrouver
 - Ne pas parcourir une liste de 1000 processus pour trouver le seul prêt à s'exécuter !
 - Cf Projet Système IT202 sur les threads

Files d'attente

- Les processus non prêts sont dans files spéciales, avec un moyen de les retrouver
 - File d'attente d'un pilote si attente I/O
 - File dédiée à attente événement logique
 - Ex: Attente de données dans un tube
 - Ex: `pthread_join()` dans votre projet IT202
 - Aucune uniquement si réveillé par PID
 - Ex: Suspendu par `^Z` et réveillé par `SIGCONT`
 - L'endroit où les signaux arrivent pour chaque PID est une pseudo file d'attente

Attente active d'un événement

- Boucle infinie
 - Tant que l'événement n'arrive pas...
- Monopolise le processeur jusqu'à l'événement
 - Très bonne réactivité
 - Gaspillage de cycles processeur
 - Acceptable pour μ s, gaspillage si trop long

Attente semi-active

- Attente active en rendant la main
 - yield() à chaque itération de boucle while
 - Réactivité imprévisible
 - Pas de garantie de reprise de main rapide
 - Peut gaspiller autant de temps CPU
 - Dépend de l'activité des autres tâches
 - A n'utiliser que quand une attente passive est totalement impossible
 - A éviter dans vos join(), sémaphores, ... dans le projet IT202

Attente passive d'un événement

- Mécanisme de réveil lors d'un événement
 - La tâche est placée sur une file d'attente
 - Elle n'est plus dans la runqueue
 - Ignorée par l'ordonnanceur
- Lors de l'événement
 - Celui qui produit l'événement déplace la tâche de la file d'attente vers la runqueue
 - Le premier de la file (ou plusieurs si nécessaire)

Attente passive d'un événement (2/2)

- Endormissement+réveil assez cher
 - Changements de contexte
 - Délai si non prioritaire
 - Ne peut pas préempter immédiatement
- Cela vaut-il le coup de dormir?
 - Pas si le délai d'attente est très court
 - Attente active plus réactive
 - Mais très coûteux si trop long
 - Trouver le bon compromis

Système d'exploitation

Processus et exécution

Ordonnancement

États des processus

- Automate de transition entre les états
 - *Running*
 - *Ready* (Prêt mais pas en train de s'exécuter)
 - *Blocked* (Non-prêt, en attente d'un événement)
 - *New* (En cours de création)
 - *Exit* (En cours de destruction, *Zombie*)
 - *Suspend* (Suspendu par l'utilisateur)

Exécution de l'ordonnanceur

- Le noyau n'existe pas
 - Il n'y a pas d'homme en noir pour ordonnancer les tâches en arrière plan
- Ordonnancement nécessite l'exécution du code de l'ordonnanceur
 - Sur le bon processeur
 - En mode noyau
 - On ignore les threads utilisateur dans le reste du cours
- Invocations explicites ou implicites
 - Ou invocations de force

Ordonnancement coopératif

- Ordonnancement explicite par les applications
 - Processus qui rend la main
 - `sched_yield()`
 - Processus qui attend
 - `sleep()`
- Ordonnancement implicite
 - Appel système bloquant
 - `read()`, `poll()`, ...
 - Retour de l'appel système nécessite fin de l'I/O

Ordonnancement coopératif (2/2)

- Aucune garantie d'ordonnancement régulier
 - Dépend du bon vouloir des applications
- Problème d'équité et de réactivité
 - Un processus peut conserver le processeur indéfiniment
 - Les tâches doivent coopérer

Exécution de l'ordonnanceur (suite)

- Il faut tâche s'exécutant en mode noyau
 - Sur le bon processeur
 - Et elle doit appeler le code de l'ordonnanceur
- Trouver un moyen d'appeler l'ordonnanceur même si l'application ne fait pas d'ordonnancement explicite ou implicite
 - Ordonnancer depuis code qui n'a rien à voir !
 - À chaque fois que le système prend la main
 - Appels système
 - Interruptions

Préemption

- Désordonner une tâche de force
 - Si une tâche s'exécute depuis trop longtemps
 - La mettre sur à la fin de la file *Ready*
 - Remplacer par la première de la file *Ready*
 - Si une autre tâche est prioritaire
 - Mettre la tâche courante sur *Ready*
 - À la fin ou au début ?
 - Remplacer par la première tâche prioritaire
 - Risque de famine ?

Préemption (2/2)

- La préemption a un léger surcoût
 - On doit vérifier de temps en temps s'il faut désordonner de force
- Mais gain énorme en équité et réactivité
 - Plus besoin de coopération des processus !

Préemption dans Linux

- L'ordonnanceur vérifie régulièrement s'il faut passer la main à une autre tâche
 - Lors du retour en espace utilisateur
 - Fin d'appel-système
 - Fin d'interruption
 - Utilité des ticks d'horloge périodiques !
- Ici on parlait de préemption uniquement pour le code utilisateur
 - Le code du noyau peut être long et nuire à la réactivité du système
 - Linux préempte aussi le code noyau

2018-2019

Brice Goglin

117

Qui préempte qui ?

- Pas de préemption directe en espace utilisateur
 - Il faut être dans le noyau pour exécuter le code de l'ordonnancement
 - Peut se produire n'importe où dans code utilisateur via les interruptions
- Processus préemptés dans le noyau de la même façon
 - Par exemple juste avant leur retour en espace utilisateur
 - Après appels système, exceptions, interruptions
- Les interruptions peuvent préempter n'importe quand
 - Sauf si désactivées

2018-2019

Brice Goglin

118

Signaux

- Message (avec numéro) d'un processus à un autre
 - Reçu par exécution d'un traitant
 - Déroutement du programme puis retour
- Délivré quand on revient en espace user
 - Traité par l'ordonnanceur comme préemption
- Délivré uniquement s'il y a des appels-système ou interruptions
 - Heureusement qu'il y a l'horloge de temps en temps

2018-2019

Brice Goglin

119

Signaux (2/2)

- Le noyau modifie le contexte d'exécution du processus avant de le laisser retourner en exécution
 - Force l'exécution du traitant
 - Avec segment de pile dédié
 - Force l'exécution d'une fonction « retour de traitant » à la sortie du traitant
 - Modification de l'adresse de retour
- Le « retour du traitant » remet le contexte d'exécution initial en place
 - Reprise de l'exécution à l'endroit initial

2018-2019

Brice Goglin

120

Système d'exploitation

Processus et exécution

Processus et threads

Intérêt des threads

- Partage de ressources entre tâches
 - Mémoire partagée triviale
 - Pas d'initialisation complexe
- Meilleure exploitation des processeurs
 - Un seul programme sur plusieurs processeurs
 - Critique depuis l'avènement des multicores
 - Mais pas trivial à utiliser et/ou implémenter...
- Recouvrements des entrées/sorties
 - Exécution d'un thread pendant qu'un autre bloque en attendant un événement

Définition des *threads* ?

- File d'exécution
 - Interne à un processus
- Le processus devient un conteneur
 - Au moins un thread
 - Des ressources partagées

Données partagées ou non

- Contextes d'exécution distincts
 - Piles, registres, ...
- Partage possible de
 - Espace d'adressage mémoire
 - Fichiers ouverts
 - Signaux
 - Identifiant des processus
 - Sous-identifiant par thread
- Définition du partage à la création
 - Voir la manpage de clone()

Modèle 1-on-1

- Un thread noyau pour chaque thread
- Aussi coûteux que les processus
 - Création, destruction, changement de contexte
- Processus multi-threads avantageés
 - Une *Timeslice* normale pour chaque thread
- Ex: NPTL (Native Posix Thread Library)
 - Création : 6 μ s
 - Changement de contexte : 0,5 μ s

Threads en espace utilisateur

- Un seul thread noyau propulse plusieurs threads utilisateurs
- Coût d'ordonnancement très réduit
 - On reste en espace utilisateur
- Multiples threads ignorés par le système
 - Une seule *Timeslice* pour tout le processus
 - Si un thread bloque, il bloque tous les autres
- Ex: Marcel
 - Création : 0,21 μ s
 - Changement de contexte : 0,22 μ s

Modèle M-on-N

- Plusieurs threads utilisateurs sur plusieurs threads noyau
- Si un thread bloque une tâche, les autres tâches peuvent continuer à travailler
- Ajout de threads noyau à la volée
 - *Scheduler Activations*
 - Collaboration entre ordonnanceurs noyau et utilisateur pour ajouter tâche noyau si nécessaire
- Ex: Thr (Solaris), NGPT (*Next Generation Posix Threading*), Marcel

Ordonnancement des processus multi-threadés

- Dépend du modèle de thread
 - Le système est-il au courant de l'existence des threads?
 - Avantageé en 1-on-1
 - Désavantageé en threads utilisateur

Changement de contexte en espace utilisateur

- Comment sauver/restaurer le contexte sans casser le contexte qui le fait?
- set/get/make/swapcontext
- setjmp/longjmp

- Projet système IT202 sur les threads

Comportement des programmes multithreadés ?

- Dupliquer tout ou un seul thread lors d'un fork ?
- Et en cas d'exec ?
- Un seul thread a-t-il un sens sans ses copains ?

Système d'exploitation

Processus et exécution

Algorithmes
d'ordonnancement

Besoins

- Progression de toutes les tâches
 - Équité
 - Interactivité
- Ne pas dépendre de la bonne volonté des tâches
 - Prémption

Durée d'ordonnancement

- Choix de la durée des *Timeslices*
 - Quantum de temps d'exécution
 - Dépend de la priorité
 - 100 ms par défaut sous Linux
 - Varie entre 5 et 800 ms selon priorité
 - Pas trop court
 - Sinon le coût d'ordonnancement devient prohibitif
 - Pas trop long
 - Sinon les processus n'avancent pas assez régulièrement
 - Problème avec les contraintes temps réel

Mesure du temps d'ordonnancement

- Difficile de savoir précisément si un processus est gourmand en temps processeur
- Ordonnancement essentiellement basé sur les ticks d'horloge
 - Pas très précis
 - 4 ms sous Linux/x86
 - Utilisation du compteur de cycle s'il existe
- Comment comptabiliser le temps passé à traiter les interruptions ?

Critères d'ordonnancement pour l'utilisateur

- Performance
 - Rapidité d'exécution ou de terminaison
 - Réactivité
 - *Deadlines*
- Équité ?
- Prédicabilité ?

Critères d'ordonnancement pour le système

- Performance
 - Quantité de travail
 - Utilisation du processeur
 - et des périphériques
- Équité
- Respect des priorités

Idées pour l'ordonnancement

- FIFO (*First In First Out*)
- *Round-Robin*
 - Exécution tour à tour pendant *Timeslices*
- *Short First* ou *Short Remaining Next*
 - Impose de prévoir la durée d'exécution
- *Feedback*
 - Pénalité pour processus gourmand
 - Ajuster la priorité initiale

Priorités dans Unix

- Priorité de base dynamique
 - *Feedback*
- Facteur ajustable par l'utilisateur (*nice*)
 - Privilège nécessaire pour rendre prioritaire
- *Swapping* très prioritaire
 - En gros, la priorité augmente en allant des applications aux périphériques physiques

Adapter l'ordonnancement aux priorités ?

- Exécuter les tâches prioritaires d'abord ?
 - Peu utile ?
 - Sauf pour processus très courts qui s'exécutent d'un seul coup ?
- Exécuter les prioritaires plus souvent
- Exécuter les prioritaires plus longtemps

Liste triée par priorité

- Liste de tâches triées par priorité ?
 - Complexité linéaire en le nombre de tâches
 - Chaque opération d'ordonnancement coutera cher
 - Yield, Création, Destruction, ...
 - Attention aux famines
 - Les processus non-prioritaires ne doivent pas rester indéfiniment en fin de liste
 - Ajuster priorité selon privilèges précédents ?
 - En temps constant ?

Listes par priorités

- Listes de tâches de même priorité
 - Manipulation en temps constant
- Les listes sont parcourues plus ou moins souvent selon la priorité
 - Limité à un nombre fini de priorités
 - 40 sous Linux
 - Sinon utiliser des *Skip List* ?

Ajuster la timeslice aux priorités

- Définir un quantum de temps d'exécution variable (*Timeslice*) selon la priorités
 - Éventuellement dynamique
 - Privilégié si interactif
 - Entrées/sorties font dormir avant la fin de la timeslice
 - Désavantagé si abuse du processeur
 - Préempté quand la timeslice est terminée
 - Calculable en temps constant

Que change les multicoeurs et multiprocesseurs dans tout ça ?

- Prémption = Concurrence entre tâches sur un seul processeur
 - Évitable en désactivant les interruptions
- Multiprocesseur = Concurrence réelle entre tâches
 - N'importe quand
 - Synchronisation critique
- Sur quel processeur/coeur exécuter quelles tâches ?

Ordonnement pour multiprocesseurs

- Une seule runqueue (*Load Sharing*)
 - Tous les processeurs y piochent
 - Contention si beaucoup de processeurs
- Une runqueue par processeur
 - Pas de contention
 - Mais il faut les équilibrer (*Load Balancing*)

SMP, NUMA et migration

- Affinité entre processus, threads, processeur et/ou mémoire
 - Notion de préférences dans l'ordonnanceur
- Impose de garder une trace des affinités
 - Difficile
 - Faire une file par coeur et garder le processus dessus à la fin de sa *Timeslice*?
 - Compromis avec équilibrage de charge
- Quand casser l'affinité ?
 - Si un processeur est libre et un est surchargé
 - Ca dépend...

2018-2019

Brice Goglin

145

Ordonnements spéciaux pour multiprocesseurs

- Dédier un processeur à une tâche
 - Les autres tâches ne peuvent pas venir
 - 100 % de temps CPU
 - Affinité de cache et mémoire garantie
 - A ne pas confondre avec l'assignation d'un processeur
 - Processeur libre, mais tâche contrainte
- Par groupe (*Gang Scheduling*)
 - Intéressant si affinité de cache/mémoire
 - ou si communication inter-tâches

2018-2019

Brice Goglin

146

Ordonnement dans Linux

- Ordonnanceur Unix assez classique
 - mais évolue très vite
 - Le fameux *O(1)-Scheduler* de Linux 2.6 a déjà été remplacé plusieurs fois
- Bonne interactivité et équité
 - Priorité effective avec bonus ou pénalité
 - Crédit d'interactivité si le processus dort
 - Priorités parfois respectées uniquement à l'intérieur des sessions (cf vos TD)

2018-2019

Brice Goglin

147

Ordonnement dans Linux (2/2)

- Support de beaucoup de processus et processeurs
 - Optimisation avancée des structures de données et algorithmes pour passer à l'échelle
 - Masque de processeur accepté pour chaque processus
 - et/ou préféré selon affinité
- Enormément de petites d'optimisations
 - Ex: Le fils de `vfork()` préempte le père
 - Évite un *Copy-on-Write* inutile

2018-2019

Brice Goglin

148

Complexité d'ordonnancement

- Les coûts des opérations ne peut pas dépendre du nombre de tâches
 - Toujours beaucoup de tâches *Blocked*
 - Celles-ci ne doivent pas être sur la Runqueue
 - Potentiellement beaucoup de tâches *Ready*
 - Ne jamais parcourir la Runqueue

Complexité d'ordonnancement (2/2)

- Insertion d'une tâche désordonnée
 - Ne pas trier la liste
 - Comment gérer les priorités ?
- Retrait d'une tâche ordonnée
 - Ne pas parcourir la liste
 - Comment choisir rapidement la tâche à exécuter ?
- Calcul des timeslices
 - Ne pas parcourir la liste pour calculer la somme du poids des tâches
 - Maintenir la somme au fur et à mesure des ordonnancements de tâches

Cas des contraintes temps réel

- Réactivité
- Priorité
- *Deadline* (exécution avant une date)
- Utilise un cas spécial d'ordonnancement
 - *Timeslice* pouvant être infinie
 - Nécessite des privilèges
 - SCHED_RR et SCHED_FIFO sous Linux
 - Manpage de sched_setscheduler()

Ordonnanceur temps réel

- Ordonnanceur non-généraliste
 - Conçu pour le temps réel
- Maximiser la réactivité
 - Minimiser la désactivation des interruptions
 - Minimiser les verrous
- Routines dédiées à la gestion des contraintes temps réel
 - Retardement, pause, alarmes (*Timer*), ...

Système d'exploitation

Processus et exécution

Exécution du système

Le noyau n'existe pas

- Pas de processus noyau qui surveille tout le monde
 - Pas d' « homme en noir » qui prend les processus, les fait s'exécuter, vérifie ce qu'ils font, ...
- Le système d'exécute essentiellement dans le contexte des processus lorsqu'ils sont en mode noyau
- Juste éventuellement des démons dédiés à des tâches périodiques

Exécution du noyau dans le dos des processus

- Le noyau s'exécute à travers des *hooks* un peu partout dans le code qui n'a rien à voir
 - Ex: Ordonnancer au retour de `getpid()`
- Pendant les appels système
 - Traitement des appels système
 - Traitements annexes par la même occasion
- Dans le cas d'interruptions/exceptions
 - Traitement immédiat nécessaire
 - Traitements différés si nécessaire
 - Travaux coûteux pourraient nuire à la réactivité

Exécution en mode noyau

- Changement de contexte
 - Changement des privilèges
 - Utilisation d'une pile spéciale
 - Éviter de stocker des informations critiques en espace utilisateur
 - Les espaces mémoire utilisateur et noyau n'ont pas les mêmes contraintes
 - Pas de défaut de page noyau sous Linux
 - Sauvegarde et restauration des registres utilisateurs

Cas des micro-noyaux

- Appels système minimaux
 - Passage de message entre composants en espace utilisateur
- Gestion effective du système d'exploitation en dehors du noyau
 - Dans les processus serveur dédiés

Threads noyau

- Noyaux monolithiques ont besoin de traitements annexes
 - Appels-système imprévisibles
 - Traitants d'interruption dans contexte trop limité
- Utilisation de threads dédiés
 - Toujours en mode noyau
 - Pas d'espace utilisateur
 - Travaux périodiques et/ou programmés

Les threads noyau sont partout

- Tout processus est propulsé par un thread noyau
 - Il passe en mode utilisateur juste après sa création
 - Il revient en mode noyau de temps en temps
 - Appels système
 - Et juste avant sa mort
- Et un processus multithread est propulsé par 1 ou plusieurs threads noyau
 - Selon le modèle 1-on-1, M-on-N, ...

Comment savoir qui je suis ?

- Le noyau doit souvent savoir qui il est
 - Tous les processus peuvent être amenés à exécuter la plupart du code noyau
 - Et aussi les threads noyaux
 - Voir les traitants d'interruption
- Comment savoir quelle tâche s'exécute ?
 - Quels droits a-t-elle ?
 - Utilisateur, permissions, ...
 - Quelles ressources a-t-elle ?
 - Mémoire virtuelle, ...

Comment savoir qui je suis ? (2/2)

- Sacrifier un registre ?
 - On a déjà un registre pour localiser la pile
- Linux utilise un joli hack
 - Le descripteur de tâche est placé sous la pile
 - Tâche = Adresse de la pile tronquée
 - `current = StackPointer & ~STACK_SIZE`
 - Imposer d'aligner et limiter les piles

Système d'exploitation

Gestion mémoire

Objectifs

- Fournir une abstraction aisée pour le programmeur
- Séparer et protéger les différents processus et utilisateurs
 - Mémoire utilisateur et système
- Maximiser l'utilisation du matériel
 - Processeurs, disques, ...
 - La mémoire sert d'intermédiaire

Objectifs (2/2)

- Supporter l'organisation logique des programmes
 - Programmation modulaire
- Supporter l'organisation hiérarchique de la mémoire
 - Caches, RAM, disques, ...
 - Avec les performances adaptées

Hierarchie mémoire

- Registres du processeur
- Mémoire centrale
 - Stockage volatile (disparaît au reboot)
 - Relativement lent
- Disque, bande, ...
 - Stockage persistant
 - Très lent

Adressage de la mémoire centrale

- Adressage 32 ou 64 bits actuellement
 - Imposé par le processeur
 - La taille des pointeurs en découle
 - 4 ou 8 octets
- Les OS utilisent des adresses de moins de 64bits en pratique
 - Et les processeurs 64bits ne supportent pas vraiment 64 bits en fait

Cache

- Zone de stockage intermédiaire
 - plus petite mais plus rapide
- Conserve zones récemment accédées
- Précharge zones proches qui pourraient être accédées peu après

Exemples de cache

- Cache dans les contrôleurs disques
 - Entièrement géré par le matériel
- Mémoire cache, jusqu'à trois niveaux
 - Très rapide, placé entre processeur et mémoire
 - Essentiellement géré par le matériel
- Cache de pages et espace de *Swap*
 - Entièrement géré par le système d'exploitation

Plan

- Programmation modulaire
- Partitionnement
- Mémoire virtuelle
- Tables de pages
- Pagination à la demande et défauts de page
- Heuristiques de gestion mémoire
- Support matériel
- Mémoire du noyau

Système d'exploitation

Gestion mémoire
Programmation modulaire

Pourquoi de la programmation modulaire ?

- Réutilisation de code
 - Ne pas inclure les mêmes bibliothèques dans tous les programmes
 - Économie d'espace disque
 - Et mémoire
- Changement de bibliothèque sans relinker
 - Il suffit que l'ABI soit la même
- Modification du comportement au chargement
 - LD_PRELOAD, ...

Chargement des programmes

- Assemblage de modules
 - Programme, bibliothèques, ...
- Calcul des adresses mémoire
 - Chargement absolu
 - par le programmeur, le compilateur ou l'assembleur
 - Chargement dynamique
 - Adresses relatives calculées à l'exécution

Édition de liens

- Utilisation de « symboles » pour référencer les adresses inconnues
- Références non-résolues provoquent chargement d'autres modules
 - Mise à jour de modules sans recompiler
 - Fournir la même liste de symboles avec même ABI
 - Partage de code facile

Emplacement d'un programme en mémoire

- A l'origine, pas de mémoire virtuelle
 - Tous les programmes et bibliothèques étaient chargés dans le même espace
- Un programme n'est pas toujours chargé au même endroit
- Les bibliothèques sont rarement chargées au même endroit
 - Liste de bibliothèques variables
 - Chargement multiple possible
 - Même avec de la mémoire virtuelle

Relocalisation

- Comment stocker les adresses si on peut charger à différents endroits ?
 - Adresses absolues simples à gérer
 - Impose emplacement du code en mémoire
 - Impose un seul programme ou mémoire virtuelle
 - Et une seule copie par programme
 - Adresses relatives ajustées plus tard
 - Au chargement ou à l'exécution
 - Edition de liens entre modules
 - Fixer les adresses à l'intérieur des modules

Code indépendant de la position

- Uniquement des adresses relatives
 - Facile à charger
 - Pas besoin de modifier le code
 - Partageable entre plusieurs processus qui le chargent à différents endroits
 - Un peu coûteux en temps, registre et mémoire
- Compilation avec -fPIC (*Position-Independent Code*)
 - Bibliothèques dynamiques sous Unix
- Uniquement pour références internes au module
 - Toujours édition de liens avec autres modules

La pile et le tas

- Intrinsèquement des adresses relatives
 - Augmentations imprévisibles
 - Équivalence entre différents endroits
- Spécificités de la pile
 - Références au cadre précédent (appelant)
 - Un même code peut utiliser différents cadres
 - Appels récursifs, ...
- Spécificités du tas
 - Augmentation imprévisible
 - Pas forcément contigu
 - mmap à partir d'un certain seuil

Sous Linux

- Le compilateur précise l'interpréteur
 - Dans les en-têtes de l'objet
- Quand on lance un programme
 - Interpréteur lancé en premier (ld.so)
- Chargement des modules en mémoire
 - Programme
 - Bibliothèques
 - Création dynamique du tas et de la pile

Sous Linux (2/2)

- Chargement configurable
 - Bibliothèques stockées dans emplacements standard ou non
 - /etc/ld.so.conf, LD_LIBRARY_PATH, ...
 - Bibliothèques de surcharge (LD_PRELOAD)
 - Résolution de symboles à l'exécution ou au chargement (LD_BIND_NOW)
- Voir la manpage de ld.so

Système d'exploitation

Gestion mémoire
Partitionnement

Pourquoi du partitionnement ?

- Une seule mémoire physique
- Plusieurs programmes, bibliothèques, modules
- Comment les répartir ?
 - Découper en segments contigus ?
 - Une partition pour chaque module à allouer
 - Allouer par morceaux ?
 - Pagination
 - Plusieurs pages pour chaque module

Partitionnement fixe

- Division de la mémoire physique en N partitions de même taille
 - Un module par partition
 - N = degré de multiprogrammation maximum
- Inconvénients
 - Fragmentation interne
 - Mémoire inutilisée dans chaque partition
 - Programmes limités en taille
 - Compromis entre multiprogrammation et taille

Partitionnement fixe (2/2)

- Adresses finales du programme décalées
 - La relocalisation peut gérer le décalage mais pas la taille limitée de la partition
 - Le matériel peut gérer le décalage
 - Registre de base pour traduction transparente
 - $@phys = @virt + base$
 - Base adaptée par l'OS lors du changement de contexte
 - Le matériel peut vérifier qu'on ne déborde pas
 - $@virt < taille/N$
 - Protection inter-processus

Partitions de tailles différentes

- Découpage fixe en partitions différentes
 - File de partitions libres
 - Prendre la plus petite suffisamment grande pour le programme à charger
 - Algorithmes *First Fit* or *Best Fit*
 - Peu important si peu de partitions
- Support matériel
 - Registre base + limite
 - Traduction en utilisant base
 - Protection en utilisant limite

Partitionnement dynamique

(variable dans le temps)

- Allocation de partitions selon la taille du programme
 - Algorithmes *First Fit* or *Best Fit*
 - Important si beaucoup de programmes ?
- Support matériel
 - Registre base + limite
 - Traduction en utilisant base
 - Protection en utilisant limite

Partitionnement dynamique (2/2)

- Avantages
 - Pas de fragmentation interne
 - On alloue l'espace exactement nécessaire au chargement du programme
- Inconvénients
 - Fragmentation externe
 - Petits résidus inutilisés entre les partitions
 - Dépend de l'algorithme d'allocation
- Solutions
 - Compactage dynamique

Pagination

- Au lieu d'allouer un bloc par programme, on alloue plusieurs pages de même taille par programme
 - Pas de fragmentation externe
 - Un peu de fragmentation interne
 - Taille des pages à bien choisir
- On peut gérer la mémoire à faible granularité
 - Charger, swapper, protection, ...

Pagination (2/2)

- Support matériel beaucoup plus complexe
 - Programmes répartis dans pages physiques discontinues !
 - Relocalisation ne suffit plus
 - Un objet contigu à cheval sur deux pages devient discontinu en mémoire physique
 - *p++ ne fonctionne plus
 - Il faut traduire toutes les adresses à la granularité de la page
- Ce qu'on va utiliser plus loin dans le cours

Autres applications du partitionnement

- Utile dès qu'on doit stocker plusieurs choses dans un espace borné
 - Malloc
 - Partitionnement dynamique
 - L'utilisateur veut des buffers contigus
 - Système de fichiers
 - Similaire à la pagination
 - Matériellement découpé en bloc dans les disques
 - Pas directement accédé par l'utilisateur, donc discontiguïté invisible
 - Rendu contigu par mmap ou read/write

Système d'exploitation

Gestion mémoire

Mémoire virtuelle

Besoins

- Chaque processus voit un espace linéaire
 - Pas forcément linéaire physiquement
 - Pas forcément utilisé intégralement
 - Utilisation dynamique
- Taille non limitée par la mémoire physique
 - Toute la mémoire virtuelle n'est pas utilisée en même temps
- Protection, partage

Intérêts de la mémoire virtuelle

- Stockée en mémoire physique ou ailleurs
 - Plus de mémoire virtuelle que de mémoire physique disponible
- Code et données non-utilisées peuvent rester sur le disque
 - Chargés par l'OS quand ils deviennent nécessaires
- Cohabitation de nombreux processus
 - même si très gourmands en mémoire
- Isolation des processus
 - Espace virtuel des autres est inaccessible

Différents types d'adresses

- Virtuelle ou Logique (@virt)
 - Sélecteur de segment + Décalage
 - Pointeurs déréférencés par le programme
 - Manipulés par le processeur
- Linéaire (@lin)
 - dans l'espace global du processus
- Physique (@phys)
 - dans la mémoire physique de la machine
- De bus (@bus)
 - Manipulable par les périphériques (DMA)

Adresses vers quoi ?

- Les adresses pointent vers des octets
 - Pas vers des bits particuliers
- Les processeurs manipulent des octets
 - Par paquets de 1/2/4/8/...
 - Avec parfois des contraintes d'alignement
- Accès à certains bits d'un octet ?
 - Instruction arithmétique après avoir chargé octet entier dans registre

Adresses virtuelles

- Indépendant du stockage physique sous-jacent
 - RAM, disque, I/O, ...
 - Géré par l'OS via les défauts de page
- Les instructions processeur manipulent des adresses virtuelles
 - Traduites en adresses physiques au dernier moment
 - Par le matériel (MMU)
- Espace d'adressage d'un processus
 - L'ensemble des @virt valides

Intérêts de la mémoire virtuelle (2/2)

- Pas de contraintes pour le programmeur
 - Abstraction linéaire et uniforme du stockage physique
 - Pas (Peu?) de limite en espace
- Partage facile
 - Plusieurs @virt virtuelles vers même @phys
- Protection
 - Droits d'accès associés à chaque @virt

Inconvénients

- Performance
 - Consommation d'espace mémoire
 - Stockage des tables de pages
 - Consommation en temps
 - Gestion fine donc complexe
- Assistance du matériel très importante

Implémentation

- Traduction des adresses virtuelles
 - Pagination et segmentation
 - Table des pages
- Politiques de gestion
 - Chargement, swap, ...
 - Quand ? Qui ?
- Support matériel et OS conjoint
 - *Memory Management Unit* (MMU)
 - *Translation Lookaside Buffer* (TLB)

Pagination : Le retour

- Permet forte non-contiguïté de l'espace virtuel en mémoire physique
- Division de la mémoire physique en blocs de taille fixe
 - *Pages* (typiquement 4ko, dépend de l'archi)
- Division de l'espace virtuel en blocs de taille fixe
 - *Cadres (frames)* de la taille des pages
- Chaque cadre virtuel nécessite l'allocation d'une page physique

Pagination (2/8) Implémentation

- Association entre cadres et pages gérée par l'OS
 - Table de pages
- Pas de contrainte sur le choix des pages
 - Espaces virtuels dispersés dans l'espace physique
- Aucune adresse virtuelle ne pointe sur les pages des autres processus
 - Protection intrinsèque

Pagination (3/8)

Traduction des @virt

- @virt = numéro cadre virtuel + décalage
- Numéro = index dans table de page
 - Permet de récupérer numéro de page physique
 - PFN = Page Frame Number
- @phys = PFN + décalage

- Une PTE (*Page Table Entry*) pour chaque cadre virtuel

Pagination (4/8)

Exemple de traduction

- Si @virt 32bits, @phys 24bits, page 4ko
- Décalage dans page 4ko = 12bits
 - Il reste 12bits pour décrire la page physique
- @virt = 20bits de numéro de cadre + 12bits de décalage
- Traduction numéro 20bits -> PFN 12bits
 - 2^{20} PTE par processus

Pagination (5/8)

Protection

- Chaque cadre virtuel dispose de bits de protection
 - Stocké dans le PTE
- Bit « valide ou non »
 - Pour les segmentation fault
- Bit « présent ou non »
 - Pour les défauts de page, le swap, ...
- Bits *Read-only*, *Read-write*, *Execute*, ...
 - Selon l'architecture

Pagination (6/8)

Les Page Table Entries (PTE)

- PFN représentant la page physique
- Bits de validité
- Bits de protection
 - R/O, R/W, X, ...
- Bits d'état
 - Modifié (*Dirty*), ...
- Bits de référence
 - Combien d'accès ? Quand ? ...
- Stockage architecture et/ou OS dépendant

Pagination (7/8) Avantages

- Allocation de pages faciles
 - Listes de pages libres
 - Pas de fragmentation externe
- Swap de morceaux de programmes
 - Tous les cadres virtuels ont la même taille
 - Bit de présence pour savoir si swappé ou non
 - Adapter la taille de pages aux blocs disque
- Partage et protection de pages faciles

Pagination (8/8) Inconvénients

- Un peu de fragmentation interne
 - De l'ordre de la taille d'une page
- Coûteux
 - 2 références par accès mémoire
 - Table de pages puis adresse destination
 - Impose support matériel (TLB)
- Consommation mémoire importante
 - Plusieurs octets par PTE
 - 1/1000 de l'espace mémoire de chaque processus
 - 4Mo par processus en 32bits !
 - Tables hiérarchiques, swappable, ...

Segmentation

- Partitionnement mémoire en unités logiques
 - Code, tas, pile, ...
 - Taille dynamiquement ajustable
- Espace d'adressage = Ensemble de segments indépendants
 - @virt = sélecteur de segment
 - + décalage dans ce segment
 - Table associant sélecteur et base+limite

Segmentation (2/3) Avantages

- Protection par segment
 - Bits de validité, protection, ...
 - Factorisation de beaucoup d'informations identiques
 - Ex : Code en r-x, tas en rw-, ...
- Partage à forte granularité facile
 - Ex : Bibliothèques

Segmentation (3/3) Inconvénients

- Allocation difficile
 - Doit être physiquement contigu
 - Problème de partitionnement mémoire (avec partitions de taille variable), compaction, ...
- Swap plus lourd

Approches hybrides : Segments paginés

- Segments pour distinguer unités logiques
 - Code, tas, pile
- Pagination interne à chaque segment
 - Allocation, swap à granularité fine
- Traduction @virt en @phys en 2 temps
 - Décalage d'index dans la table de pages via la table des segments
 - Adresse linéaire
 - Traduction @lin en @phys via la table de pages

Approches hybrides : Multiples tailles de page

- 8-64ko sur Alpha
- 4ko, 2-4Mo sur x86 et 1Go sur Opterons, 16Go sur Power
 - *Huge pages*
- Simultanément plusieurs tailles
- Moins de pression sur la TLB
- Plus difficile à allouer
 - Doit être physiquement contigu et correctement aligné

Partage mémoire

- Plusieurs zones virtuelles peuvent pointer sur les mêmes pages physiques
 - Avec des protections différentes
 - Bibliothèques mappées plusieurs fois
 - r-x pour code exécutable non modifiable
 - rw- pour données globales modifiables non exécutable
- Compteur de références dans les pages
 - La page est libre quand la dernière référence est relâchée
 - On peut la réutiliser
 - Après l'avoir sauvee sur le disque si nécessaire

Tables de pages

- Ensemble d'associations entre cadres virtuels et validité+localisation+protection
 - Accédé quasiment uniquement par adresse virtuelle
 - Stocké naturellement comme un grand tableau
 - Il faut une case par page virtuelle potentielle
 - On a besoin de savoir si une page est valide ou non
- La table des pages complète prend beaucoup de place en mémoire
 - 4MB pour un espace 32bits avec pages de 4ko
 - La table est forcément entièrement allouée !

Tables de pages (2/3)

- Une grande part de l'espace d'adressage n'est pas utilisé
 - Pas valide du tout
 - Voire valide mais inutilisé actuellement
- Doit-on vraiment décrire toutes ces pages virtuelles inutiles individuellement ?
 - Liste d'intervalles valides ?
 - Parcours de liste mauvais en complexité

Tables de pages (3/3)

- Arbre binaire d'intervalles valides ?
 - Support matériel difficile
 - Modifications pas atomiques ?
- Solutions plus simples ?
 - Tables hiérarchiques à plusieurs niveaux
 - Le matériel aime bien les puissances de 2
 - Mais on gaspille un peu d'espace
 - Tables hachées
 - Le matériel aime bien les choses simples
 - Mais le temps de recherche n'est pas constant

Tables de pages à plusieurs niveaux

- Une table normale = un seul niveau
- Table à plusieurs niveaux = Tableaux d'indirections vers des sous-tableaux
 - Un tableau de niveau 1 avec N pointeurs vers niveau 2
 - N tableaux de niveau 2 avec M pointeurs vers niveau 3... etc.

Tables de pages à plusieurs niveaux (2/5)

- Permet de supprimer les tableaux inutiles
 - Au lieu d'allouer un tableau d'entrées invalides, on marque le pointeur du père comme invalide
 - Ca compense largement l'espace « gaspillé » par les niveaux de pointeurs intermédiaires
 - Contrairement à la table linéaire à 1 seul niveau qui est forcément **entièrement** allouée

Tables de pages à plusieurs niveaux (3/5)

- Si un processus n'utilise qu'un seul cadre virtuel ?
 - On alloue une seule page **par niveau**
 - Très petit par rapport à table linéaire à 1 niveau
- Si un processus utilise toute la mémoire virtuelle ?
 - Le dernier niveau fait exactement la même taille qu'une table linéaire à 1 seul niveau
 - Le surcoût des niveaux supérieurs est négligeable
 - De l'ordre de 1 / nombre de pointeurs par niveau
 - Et ce cas le pire est extrêmement rare !

Tables de pages à plusieurs niveaux (4/5)

- Les processeurs supportent 2 à 4 niveaux (et bientôt 5)
- L'OS les utilise comme il veut
 - Il peut ignorer certains niveaux matériels
 - Il ne peut pas en mettre plus
 - Le matériel ne saurait pas les parcourir
 - 5 niveaux logiciels dans Linux depuis 4.12
 - Certains peuvent être « vides » pour s'adapter au matériel

Tables de pages à plusieurs niveaux (5/5)

- @virt = agrégation d'index pour chaque niveau
 - premiers bits = entrée dans table du premier niveau
 - suivants = entrée dans la sous-table du niveau suivant
 - derniers = décalage dans la page finale
 - Tous les bits pas forcément utilisés!
- Comment l'OS choisit-il le nombre de bits?
 - Simplifier la gestion en faisant tenir la table et chaque sous-table dans une page exactement
- Traduction des @virt en @phys
 - Parcours de la table de haut en bas

2018-2019

Brice Goglin

221

Les tables de pages à 4 niveaux dans Linux sur x86_64

- Architecture 64bits
 - 4 niveaux de table de pages matériels
 - pages de 4ko = 12bits de décalage
- 8 octet par PTE
 - Numéro de pages physiques
 - Bits de validité/présence/protection
 - 512 PTE dans une page
 - 9bits de dernier niveau de table
- 8 octet par pointeurs
 - 512 pointeur dans une page
 - 9bits pour les niveaux supérieurs

2018-2019

Brice Goglin

222

Les tables de pages à 4 niveaux dans Linux sur x86_64 (2/3)

- @virt = 48bits
 - 9bits de PGD (*Page Global Directory*)
 - 9bits de PUD (*Page Under Directory*)
 - 9bits de PMD (*Page Middle Directory*)
 - 9bits de PTE (*Page Table Entry*)
 - 12bits de décalage dans la page
- 64bits disponibles matériellement mais pas tous utilisés
 - Taille réelle des @virt décidée par le processeur et l'OS
 - Mais forcément stocké dans pointeur 64bits (8 octets)
- 256To de mémoire virtuelle par processus

2018-2019

Brice Goglin

223

Les tables de pages à 4 niveaux dans Linux sur x86_64 (3/3)

- 1 tableau de niveau 1 avec 512 pointeurs
- 512 tableaux de niveaux 2 avec 512 ptrs
- 512^2 tableaux de niveaux 3 avec 512 ptrs
- 512^3 tableaux de niveaux 4 avec 512 PTEs
- 512^4 PTEs en tout
 - Couvrent $512^4 * 4096 = 256\text{TB}$ de mémoire virtuelle
 - mais n'occupent que $512^4 * 8 = 512\text{GB}$ (maximum)
 - Et quelques broutilles pour les tables intermédiaires
- Mais tout ces tableaux ne sont pas forcément alloués

2018-2019

Brice Goglin

224

Tables de pages et segments sur processeur Alpha 21064

- Architecture 64bits
 - 3 niveaux de table de pages matériels
 - Pages de 8ko, PTE de 8 octets
- 10 bits par niveau + 13 bits de décalage
 - @virt 43bits (21bits non-utilisés par l'OS)
- 2bits de segments au début de l'@virt
 - 10 = Noyau (protégé)
 - 11 = Pile utilisateur
 - 0x = Code utilisateur

Tentative de résumé

- Si segmentation
 - Sélecteur de segment
 - Dans @virt ou registre spécial
- Puis, si pagination (tout le temps ?)
 - Numéro PGD au début @virt
 - Eventuellement PUD et PMD ensuite
 - Si matériel supporte > 2 niveaux
 - Numéro PTE à la fin @virt

Tables de pages hachées

- Hachage de l'@virt puis parcours d'une liste pour localiser la PTE
 - Ajouter le décalage (comme d'habitude)
- Efficace pour les espaces d'adressage très dispersés
- Lent pour les espaces remplis
 - Liste de hachage très longues
- Disponible matériellement sur IA64, PowerPC, ...

Bilan sur les tables de pages

- Table linéaire originale
 - Très simple, bon temps d'accès, TRES mauvais en espace mémoire
- Table hiérarchique
 - Assez simple, bon temps d'accès, très bon en espace mémoire
- Table hachée
 - Relativement simple, temps d'accès bon en moyenne, très très bon en espace mémoire

Traduction @phys → @virt

- A qui appartient la page que je veux swapper ?
 - Très coûteux avec tables de pages
 - Parcourir toutes les entrées ?
- Table de pages inversée
 - Tableau indexé par numéro de page physique
 - Donne les adresses virtuelles qui la référence
 - Comment stocker les pages partagées?
 - Listes chaînées?
 - Ne permet pas de traduire @virt → @phys !
 - Un complément aux tables hiérarchiques ou hachées
- Implémenté logiciellement dans l'OS

Système d'exploitation

Gestion mémoire

Pagination à la demande et défauts de page

Pagination fainéante

- L'OS peut souvent être fainéant
 - Ne pas charger une page en mémoire tant qu'elle n'est pas accédée
 - Ne pas allouer une page non plus
 - Ne pas dupliquer une page mémoire tant qu'elle n'est pas modifiée
- Défaut de page lors du premier accès
 - Allocation et remplissage
 - Reprise sur erreur

Pagination à la demande

- *Demand Paging*
- La mémoire n'est qu'un cache des données manipulées par le processus
 - Initialement tout est sur le disque
 - A la fin aussi
 - Entre temps, on charge ce qui est accédé quand c'est nécessaire
 - Et on peut évincer quand ca n'est plus nécessaire
 - Mouvement de page entre disque et mémoire
 - Sans que l'application ne s'en rende compte

Défauts de page

- Un défaut de page est un accès qui ne peut être réalisé par le matériel
 - Le processeur ne peut pas faire
 - Il demande l'aide de l'OS
 - Il envoie une exception
- Exemples :
 - Adresse virtuelle invalide
 - Adresse valide mais page absente en mémoire physique
 - Adresse valide mais écriture interdite

Défauts de page (2/2)

- L'OS a défini un traitant pour l'exception « défaut de page »
- Le traitant identifie le problème
 - Peut-il être réparé ?
 - Exemple : Chargement d'une page depuis le disque
 - Si oui, le processus redémarre la même instruction
 - Si non, le processus ne pas continuer
 - Il est tué !

Un défaut de page n'est pas (toujours) une erreur

- C'est une exception pendant la traduction
 - L'accès n'est pas possible avec la table des pages actuelle
 - Ca n'implique pas que l'application a fait une erreur, c'est souvent la faute de l'OS
- Le défaut peut souvent être réparé
 - Page chargée du disque en mémoire, Copy-on-write, malloc pas encore utilisé, ...
- Parfois c'est une erreur
 - Segfault

Allocation de pages

- L'OS dispose d'une liste de pages libres
 - ou une liste par noeud NUMA
 - Les pages ne sont finalement pas si équivalente que ça...
- Si la liste est vide, il faut libérer une page
 - Si on échoue, l'allocation échoue

Libération de pages et swap

- Choisir une page peu utilisée
- Si elle est *Dirty*, il faut la sauver sur le disque
 - Trouver où l'envoyer sur le disque
 - Bloc de swap disponible
 - Ou blocs de fichier correspondant
 - Soumettre l'I/O d'écriture sur disque
 - Libérer la page à la terminaison de l'I/O

Chargement de page lors d'un défaut

- On a récupéré une page libre
- Localiser les données à charger
 - Bloc de swap
 - ou blocs de fichier correspondant
 - Sinon remplir la page de zéros
- Soumettre l'I/O de lecture depuis le disque
- Lorsque l'I/O termine, adapter le statut de la page et reprendre le processus

Intérêts de la pagination à la demande

- Localité temporelle
 - Une page accédée récemment le sera sûrement à nouveau dans le futur proche
- Localité spatiale
 - Les zones mémoire voisines seront sûrement accédées dans le futur proche

Intérêts de la pagination à la demande (2/2)

- Une page chargée va être réutilisée plusieurs fois
- En moyenne, on utilise beaucoup les choses déjà chargées
 - En dehors de la période d'initialisation
 - Où tout doit être chargé
- Ne pas charger trop de choses inutilement
 - Surtout si tout ne tient pas en mémoire physique !

Limites de la pagination à la demande

- Certaines applications ne respectent pas vraiment la localité spatiale ou temporelle
 - Mauvaise pratique du programmeur ?
- Les politiques de chargement et remplacement doivent être bonnes
 - Heuristiques d'anticipation

Projection de fichiers

- Tout est fichier
 - Le code du programme et des bibliothèques
 - Leurs variables globales
 - Et même la pile et le tas
 - On pourrait considérer que c'est grosso-modo le *fichier de swap* qui est chargé
- L'OS ne fait que manipuler que des pages de fichiers
 - Une page chargée en mémoire est la projection du fichier sous-jacent
 - *Mapping* mémoire

Projection publique

- Les modifications effectuées en mémoire sont répercutées dans le fichier
 - Visible par les autres processus
- mmap avec MAP_SHARED

Projection privée

- Copie privée du contenu d'un fichier
 - Les modifications sont sauvées dans le swap
 - Pas dans le vrai fichier initial
 - Et sont perdues à la fin du processus
- mmap avec MAP_PRIVATE

Projection anonyme

- Projection sans fichier de support
 - Initialisé à zéro
- mmap avec MAP_ANONYMOUS
- La pile ou le tas
- Utilisé par malloc pour les allocations en dehors du tas
 - Si le tas ne peut plus augmenter
 - Si on doit allouer beaucoup
- Peut être privé ou public

Copy-on-Write

- Les copies mémoire coûtent cher
 - Éviter de dupliquer inutilement
 - Pages partagées et privées mais non modifiées
 - Et ça économise de l'espace mémoire
- Retarder la duplication au maximum
 - Jusqu'à la première modification concurrente
 - Jamais de duplication si jamais de modification
 - Fork puis Exec
 - Tout sauf le bas de la pile juste après Fork
 - Allocation facile de pages remplies de zéros

Copy-on-Write (2/2)

- Page mise en lecture seule dans le matériel
 - Forcer un défaut de page pour détecter une modification concurrente
 - Même si on écrit exactement ce qui est déjà en mémoire !
 - Exception lors de la traduction, avant l'écriture réelle
- Le traitant alloue une page, copie le contenu dedans, la donne au processus
 - Protection ajustée sur l'original et la copie
 - La copie devient privée
 - L'original reste Copy-on-Write s'il reste plusieurs utilisateurs privés

Exemples de Copy-on-Write

- On effectue deux fork() successifs
 - Combien de processus obtient-on ?
 - Zones privées sont marquées CoW avec 4 références
- Chaque processus modifie la même zone privée
 - 1ère modification : récupère une copie RW
 - La page partagée reste CoW avec 3 références
 - 2ème : 1 autre copie RW + CoW à 2 réfs
 - 3ème : 1 autre copie RW
 - Seul le 4ème utilise désormais la page partagée
 - 4ème : La page n'est plus CoW, accès normal

Copy-on-Write peut rester trop cher

- Copy-on-Write est utilisé en permanence mais ne suffit pas
 - Fork doit dupliquer la table de pages et mettre plein de pages en Copy-on-Write
 - Cher si le processus est gros
 - Bête en cas de Fork puis Exec
 - Exec va tout libérer juste après

vfork()

- vfork() ne duplique rien, rajoute simplement une référence sur la table de pages
 - Un appel à Exec ou Exit va libérer la référence rapidement
- Tout est partagé entre père et fils
 - Aucun mécanisme de détection d'accès concurrents
 - Pas de *Copy-on-Write* sur les pages
 - On doit bloquer le père en attendant Exec ou Exit
- Similaire à clone() avec CLONE_VM

Description de l'espace d'adressage

- Pour chaque page virtuelle, il faut savoir
 - Quel fichier et quel décalage dedans
 - Quel type de projection (privé ou publique ?)
 - Utilisé par plusieurs processus ou non
 - Compteur de références pour les pages physique
 - Quelle protection mémoire
 - Présent en mémoire ou non

Description de l'espace d'adressage (2/3)

- Il faut l'état actuel + l'état possible
 - Différence entre
 - Etat voulu par l'application
 - ex : dupliqué par fork
 - Etat utilisé par le système/matériel
 - ex : partagé par Copy-on-Write
 - Permet la gestion retardée
 - Allocation fainéante met page valide et absente au lieu de valide et présente

Description de l'espace d'adressage (3/3)

- Le stockage de l'état des pages est très gourmand en mémoire
 - Regrouper les pages consécutives et similaires en segments
 - Les VMA (Virtual Memory Area) de Linux
 - Cf `/proc/<pid>/maps`
 - Segments utilisés pour décrire projection, partage, ...
 - Table de pages utilisées pour protection bas-niveau

Gestion des défauts de page

- Protection actuelle des pages stockée dans le matériel
 - Permet de déclencher des défauts « virtuels » pour la gestion fainéante
 - Allocation ou duplication au dernier moment
- Exception déclenchée au premier accès
- Le traitant trouve le descripteur de la zone dans l'OS et la PTE
 - Et en déduit quoi faire

Gestion des défauts de page (2/2)

- Si l'accès est invalide
 - Tuer le processus
 - Segmentation Fault
 - Si l'accès est valide et la page absente
 - Charger depuis le disque
 - Le descripteur de page dit depuis où
 - Si l'accès est écriture et la page privée
 - Créer une copie et la donner au processus
 - Mapping privé, Copy-on-Write, ...
- Voir `handle_mm_fault()` et `handle_pte_fault()` dans `mm/memory.c`

Défauts inattendus

- Une page valide déjà chargée peut provoquer un défaut
 - Copy on Write après `fork()`
 - Accès en écriture sur protection matérielle R/O
 - Mapping privé pas encore modifié
 - Accès en écriture sur protection matérielle R/O
 - Page chargée par autre processus, mappée mais pas encore accédée
 - La table des pages ne pointe pas encore dessus

Défauts majeurs et mineurs

- Chargement de page depuis disque ou swap très coûteux
 - Défaut majeur
- Tous les autres coûtent « peu »
 - Défaut mineur

- Cf /usr/bin/time echo toto

Le Swap

- Extension de la mémoire physique
 - Plus lent
 - Pas besoin d'être déréférencable par le processeur
- Stocké sur dispositif physique quelconque
 - Disque dur, mémoire distante, ...
 - Grand tableau de bloc de la taille d'une page
- Entièrement géré par l'OS

Le Swap (2/4)

- Historiquement on swappait des processus entiers
 - Suspension complète de l'exécution
 - Impose de vérifier qu'aucune I/O n'est en cours, ...
- Pagination permet granularité de la page
 - Un processus gigantesque peut continuer à s'exécuter en résidant essentiellement dans le swap
 - Du moment que le code et les données actuellement utilisés sont en mémoire

Le Swap (3/4)

- Les PTE valides mais non-présentes peuvent pointer vers un bloc du swap
 - Un accès provoque un défaut de page et le chargement du bloc en mémoire
- L'OS décide de quelles pages envoyer sur le disque
 - Selon le principe de localité

Le Swap (4/4)

- Le *Swap* sert uniquement quand il faut libérer de la mémoire physique
 - Quand on doit allouer d'autres pages virtuelles en mémoire physique
- Seules les pages privées et/ou anonymes vont dans le swap
 - La libération des pages de mappings publics les renvoie sur le disque contenant le fichier correspondant
 - Si pas modifiées, rien à sauver !

Système d'exploitation

Gestion mémoire

Heuristiques de gestion mémoire

Qui et quand charger ?

- *Demand Paging*
- Les pages quand elles sont accédées
 - Lors du défaut de page
 - PTE valides mais non-présentes
- *Prepaging et Prefaulting*
 - Chargement à l'avance des pages suivantes
 - Principe de localité

Où charger ?

- Dans n'importe quelle page physique en général
 - La première disponible
- Important sur NUMA
 - Les tâches préfèrent utiliser de la mémoire proche
- Maintenir une liste de pages libres par noeud NUMA
 - Allouer des pages de préférences dans la liste locale

Quand sauver les modifications ?

- Il faut sauver les pages modifiées
 - Dans des fichiers correspondants ou dans le swap
- Pages mappées difficiles à gérer
 - Pas d'appel explicite à l'OS lors des modifs
 - Contrairement à Write()
 - La matériel ajoute un bit *Dirty*
 - L'OS l'enlève quand il sauve la page
- On sauve uniquement les pages *Dirty*
 - Au pire, régulièrement en tâche de fond

2018-2019

Brice Goglin

265

Qui swapper ?

- Les pages non-utilisées à l'avenir
 - Optimal = Algorithme de Belady
 - Choisir la page non-utilisée pendant le plus longtemps à l'avenir
 - Comment prédire le futur?
 - Principe de localité?
- Politique LRU (Least Recently Used) ou LFU (Least Frequently Used)
 - Assistance du matériel nécessaire pour maintenir des statistiques d'accès aux pages

2018-2019

Brice Goglin

266

Quand swapper ?

- Attendre le dernier moment?
 - Allocation mémoire impossible, impose de libérer une page
 - Bloque l'application pendant longtemps
- A l'avance ?
 - Quand la mémoire disponible passe en dessous d'un seuil
 - Démon *Swapper* qui fait du ménage en tâche de fond de temps en temps

2018-2019

Brice Goglin

267

Anomalies du swap

- Assez facile de mettre en défaut LRU
 - « Effet trou »
 - Allouer N+1 pages virtuelles dans N pages physiques et les utiliser en tourniquet
- Ajouter RAM ne réduit pas forcément le nombre de défauts de pages
 - « Anomalie de Belady »
 - Dépend des accès des application

2018-2019

Brice Goglin

268

Que conserver en mémoire ?

- La mémoire va plus vite, autant l'utiliser
 - Toujours conserver le plus possible en mémoire
 - Au cas où ça serve plus tard
 - Principe de localité
- Qu'est-ce qui peut servir plus tard?
 - Dans le même processus
 - N'importe quelle page
 - Dans les autres processus
 - Les mappings publics

Que conserver en mémoire ? (2/2)

- Sauvegarde régulière des modifications
 - Sans forcément libérer les pages
 - Uniquement libérer quand on veut allouer
- En régime permanent, un OS utilise toute la mémoire, au cas où
 - Pages publiques gardées en mémoire si un processus les réutilise
 - « cached » dans la sortie de la commande free
 - Pages privées gardées en mémoire jusqu'à la fin du processus courant
 - « used » et « shared » dans la commande free

Gestion des famines

- Si une allocation mémoire échoue?
 - Aucune page swappable, swap plein, ...
- Le système doit survivre
 - Il se garde un peu de mémoire pour lui
- L'utilisateur doit pouvoir continuer à utiliser le système
 - Tuer un processus pour libérer de la mémoire
 - Soit le processus dont l'allocation échoue
 - Soit un autre pour que l'allocation n'échoue pas
 - L'*Out-of-Memory Killer* de Linux

Gestion des famines (2/2)

- Qui tuer?
 - Celui qui essaie d'allouer?
 - Il n'a pas forcément alloué beaucoup avant
 - Celui qui alloue beaucoup?
 - Il n'a pas forcément alloué beaucoup récemment
 - Il y avait peut-être beaucoup de mémoire disponible quand il a alloué ses pages
 - Pas les processus importants ?
 - ex : Serveur X qui gère l'affichage des autres processus

Système d'exploitation

Gestion mémoire Support matériel

Gestion des adresses virtuelles dans le matériel

- Le code manipule uniquement des @virt
 - Le processeur doit les traduire en @phys
- Il faut que la table des pages soit accessible au matériel
 - Soit directement
 - Par une MMU
 - Soit indirectement
 - Par l'OS via une exception

La MMU

- *Memory Management Unit*
- Circuit dédié du processeur pour traduire les @virt
 - Obtient le numéro de page physique associée au cadre virtuel
 - Ajoute le décalage dans la page

Le TLB

- La traduction des @virt peut être lente
 - Un accès mémoire par niveau de table
 - Plusieurs accès mémoire matériels lors d'un seul accès demandé par le programme !
- Nécessité de *cache* les traductions
 - *Translation Lookaside Buffer*
 - Cache dans le processeur (dans la MMU)
 - Le TLB contient les dernières traductions
 - Sans le décalage, évidemment
 - Principe de localité

Le TLB (2/2)

- Tableau de numéros de cadres virtuels et de leur PTE associée
- Cache souvent totalement associatif
 - Toutes les entrées sont testées en parallèle
 - Rapide
- Très rapide mais cher donc petit
 - Quelques centaines d'entrées maximum
 - Couvre quelques megaoctets au maximum
 - Il est important de sortir du TLB le moins possible
- Parfois un deuxième cache plus gros mais plus lent

Défauts de TLB

- En cas de TLB miss
 - Il faut récupérer la traduction dans la table de pages
- Géré par la MMU si elle existe
 - par l'OS sinon

Défauts de TLB, par MMU

- Si support matériel pour les défauts de TLB
 - Ex : Processeurs x86
 - PGD stocké dans registre spécial CR3
 - La MMU parcourt la table pour trouver PTE et mettre dans TLB
 - Table de pages stockée en mémoire de manière lisible par le matériel
 - Contraint taille/structure des PTE pour l'OS
 - Limite la quantité d'information stockable dans les PTE

Défauts de TLB, logiquement

- Si TLB géré logiquement (*no MMU*)
 - Ex: Processeurs MIPS
 - Le processeur déclenche une exception
 - Le traitant de l'OS parcourt la table de pages, vérifie la protection, ...
 - L'entrée manquante est insérée dans le TLB
 - Le même travail qu'une MMU, mais logiquement
- Table de pages stockée de manière quelconque par l'OS
- Qui choisit l'entrée à remplacer ?
 - L'OS ou le matériel ? Algorithme LRU ?

Changement de contexte

- Lorsque l'ordonnanceur passe la main à un autre processus
 - L'espace d'adressage change
- Si le matériel sait lire la table de pages
 - Il faut lui donner la nouvelle
 - Changer le pointeur CR3 sur x86
- Il faut vider le TLB

Maintien du TLB à jour

- Vider le TLB lors des changements de contextes
 - Le matériel le fait parfois automatiquement
 - Sur x86, quand CR3 est modifié
- L'OS assure la cohérence entre TLB et tables des pages
 - Invalidation des entrées TLB quand on change des PTE

Traduction d'adresses virtuelles dans le processeur - Résumé

- Le programme passe une adresse virtuelle dans une instruction
- Le TLB est consulté pour traduire rapidement
 - Si TLB cache miss
 - Si MMU, la MMU parcourt la table des pages
 - Si pas de MMU, exception gérée par l'OS
 - Le TLB contient le PTE et valide la protection
- La mémoire physique est enfin accédée

Qui traduit des adresses en pratique ?

- TLB/MMU du processeur dans la plupart des cas
 - Beaucoup d'instructions manipulent des adresses mémoire
 - Il faut les traduire, le processeur le fait à la volée
- Lors d'une I/O, l'application donne une adresse virtuelle, le périphérique veut une adresse physique
 - L'I/O n'est pas une instruction traitée par le processeur, TLB/MMU ne peuvent pas aider

Traduction manuelle par l'OS

- Ca arrive notamment pendant les appels-système faisant des I/O physiques
- L'OS doit traduire avant de passer la requête au périphérique
 - Parcours « manuel » de la table de pages
 - Lecture PGD, puis PUD, ... puis PTE
 - Beaucoup plus lent que TLB/MMU
 - Certains processeurs ont instructions dédiées à cette traduction explicite
 - Ex : tpa sur ia64

MMU, TLB et multicoeurs

- Chaque cœur exécute une tâche différente
 - Souvent dans espaces virtuels différents
 - Pouvoir préciser la table de pages de chaque cœur
- Le TLB peut être partagé
 - Bien si tous les cœurs exécutent des threads d'un même processus
 - Sinon il faut marquer l'espace virtuel correspondant à chaque entrée
 - Et ça doit rester rapide !
 - Donc proche des cœurs, et peu de contentions
 - Donc plutôt un TLB par cœur

Quid de la mémoire cache ?

- Petite zone de mémoire rapide entre le processeur et la mémoire centrale
 - Plusieurs niveaux plus ou moins petits et rapides
 - Accès mémoire vérifient d'abord si la donnée est dans le cache
- Essentiellement géré par le matériel
 - En particulier pour la cohérence entre différents caches sur machines multicoeur
 - L'OS doit parfois invalider
 - Changement de contexte ou de mapping

Système d'exploitation

Gestion mémoire

Mémoire du noyau

Espaces d'adressage utilisés par le noyau

- Espace virtuel spécifique au noyau
 - Accessible uniquement en mode privilégié
 - Directement déréférençable depuis le noyau
 - Non modifié lors des changements de contexte
- Espace virtuel utilisateur du processus courant
 - Pendant un appel système ou une interruption
 - Pas toujours déréférençable par le noyau
 - Ca dépend s'il est dans la table de pages noyau
 - Modifié lors des changements de contexte
 - Aucun si thread noyau

2018-2019

Brice Goglin

289

A quoi le noyau peut-il accéder ?

- Le noyau est un contexte d'exécution
 - A un espace virtuel courant et une table des pages
 - Aussi vaste que quand en mode utilisateur
- Le noyau a potentiellement le droit accéder à tout
 - Toute la mémoire physique
 - Elle ne tient pas toujours intégralement en mémoire virtuelle
 - La mémoire virtuelle de tous les processus ?
- Comment y accéder ?
 - Le processeur veut une @virt dans table des pages actuelle

2018-2019

Brice Goglin

290

De quoi le noyau a-t-il vraiment besoin ?

- Son code
 - Binaire autosuffisant
 - Pas besoin de bibliothèques
 - Les fonctions de bases sont réimplémentées sans dépendre de la libc
 - strlen, strcmp, printf, memcpy, listes, ...
- Ses données propres
 - Structures allouées dynamiquement
 - Descripteurs de périphériques, fichiers, sockets, VMA, tables de pages,
- C'est tout petit (10-100 Mo) !

2018-2019

Brice Goglin

291

De quoi le noyau a-t-il vraiment besoin ? (2/2)

- La plupart des données en mémoire ne sont jamais déréférencées par le noyau
 - Ex: Code et données utilisateur vont/viennent vers/depuis les périphériques uniquement
 - Jamais besoin de les mapper en espace noyau !
- Le noyau ne mappe rien par défaut à part le strict minimum
 - Le reste peut-être mappé temporairement si nécessaire

2018-2019

Brice Goglin

292

Accès temporaire à n'importe quelle partie de la mémoire

- Besoin de mapper des pages quelconques de temps en temps
 - Ex: Copie mémoire pendant I/O logique
- Mapping temporaire
 - Ajout à la table des pages avant accès
 - Retrait après accès
 - Mise à jour du TLB des processeurs impliqués
 - On peut se limiter à un seul cœur dans les cas simples
 - Empêcher le swap de ces pages!

Accès à l'espace utilisateur depuis le noyau

- Certains OS mappent l'espace utilisateur courant dans le noyau, aux mêmes adresses
 - Ex: Linux
 - Accès normal à la mémoire utilisateur, par les mêmes pointeurs
- Certains OS ne mappent pas l'espace utilisateur dans le noyau
 - Ex: MacOS X, Windows, ...
 - Espace utilisateur pas directement déréférençable
 - Impose mapping temporaire dans l'espace noyau
 - Ex: pendant un appel-système

Espace virtuel dans Linux

- Divisé entre espaces virtuels noyau et du processus utilisateur courant
 - Suffisamment pour que les deux travaillent...
 - Espace noyau virtuellement partagé entre tous les processus
 - Une seule fois en mémoire physique
 - Mais dans toutes les tables de pages
 - Partage de portion de tables de pages ?
 - Egalement partagé avec les threads noyau
 - Table de pages avec uniquement l'espace noyau

Espace virtuel dans Linux (2/3)

- Ex: en 32bits
 - Espace utilisateur entre 0 et 3Go
 - La pile commence à 3Go (vers le bas)
 - Un processus ne peut utiliser que 3Go de mémoire virtuelle !
 - Espace spécifique du noyau entre 3 et 4Go
- En 64bits, il y a largement assez de place pour tout le monde...
 - Actuellement 128To chacun sur x86_64

Espace virtuel dans Linux (3/3)

- Le noyau peut déréférencer directement les zones mémoire noyau et utilisateur
 - Du processus courant uniquement
 - Ex: Appels système peuvent accéder à mémoire utilisateur
 - Ex: write dans un tube
- Le processus utilisateur n'a accès qu'à sa mémoire utilisateur
 - Restriction de l'espace accessible via le matériel
 - Même table de pages mais accès restreint
 - Mais la faille Meltdown a changé ça...

Espace virtuel noyau dans Linux

- Large mapping **linéaire** de la mémoire physique dans l'espace virtuel du noyau
 - Evite le mapping temporaire pour les données souvent utilisées par le noyau
 - Le code et les données du noyau
- Un peu d'espace virtuel pour mapping temporaire des autres pages physiques
- Un peu d'espace virtuel réservé pour des zones mémoire spéciales (vmalloc)
 - Ex: Mémoire des périphériques, parfois très gros

Système d'exploitation

Concurrence et synchronisation

Pourquoi de la concurrence ?

- Plusieurs files d'exécution simultanées
 - Machines multi-processeur/coeur
 - Voir intervention des périphériques (interruptions)
- Instructions non-atomiques
 - Modification de structures complexes
 - Listes, ...
- Prémption
- Aucune garantie que quelqu'un d'autre n'est pas en train de modifier la même ressource

Concurrence logique ou physique ?

- Accès concurrent physique impose exécution simultanée (matériellement) de différentes tâches
 - Impossible si un seul processeur
 - Un vrai accès concurrent au matériel (la mémoire)

Concurrence logique ou physique ? (2/2)

- La concurrence logique suffit à causer des problèmes
 - Accès interrompu par préemption
 - Autre tâche prend la main et accède à la même ressource alors qu'on n'a pas terminé
 - Ou par une interruption matérielle
 - Le traitant d'interruption modifie une ressource qu'on était en train d'utiliser avant l'interruption
 - Ex : la carte réseau donne un paquet à l'OS pendant que l'application lit le même socket

Compétition vs coopération

- Si on sait parfaitement ce que font les autres, ça peut être facile
 - On peut éviter la concurrence
 - Ils peuvent nous prévenir de leurs accès
- Sinon, il faut imaginer le pire
 - En pratique, tous les cas horribles finissent par arriver
 - Accès pile-poil quand il ne fallait pas, ...
 - S'en prémunir, même si ça peut paraître exagéré

Points communs entre le noyau et l'espace utilisateur

- Mêmes concepts et problèmes
 - Accès concurrents physiques et logiques
 - Exclusion mutuelle
 - Deadlocks
 - Famines

Différences entre le noyau et l'espace utilisateur

- Le noyau maîtrise la préemption et les interruptions
 - Il peut effectivement empêcher concurrence logique
 - En empêchant les autres acteurs ou événements de s'exécuter
- L'espace utilisateur n'a aucune maîtrise

Exclusion mutuelle (Mutex) et verrous

- Verrous à attente active
 - *Spinlocks*
 - Tentative de prise de ressource en boucle
 - Réactif mais consomme du CPU
 - Préféré pour sections critiques courtes
- Verrous à attente passive
 - Sommeil jusqu'à relâchement
 - Réveillé par l'autre tâche via l'ordonnanceur
 - Peu réactif mais consomme peu de CPU
 - Préféré pour sections critiques longues
- Protection contre famine ou deadlock dépendant de l'implémentation

Verrous en espace utilisateur

- L'ordonnanceur noyau n'a pas connaissance des verrous pris
 - Préemption peut rendre section critique très longue
 - Préemption peut passer la main à tâche qui va essayer de prendre le même verrou
 - Deadlock ?
 - Pas infini car tâche bloquée va se faire repréempter
 - Mais pas bien si attente active
- Attente passive très préférable en espace utilisateur

Verrous dans le noyau Linux

- Le coût de synchronisation est critique
 - **Beaucoup** de variantes dans Linux
 - Notamment les variantes à attente active
- Préemption désactivée quand un verrou à attente active est pris
 - Garantie d'avoir section critique courte

Optimisation des verrous actifs

- Si beaucoup de lecteurs et peu d'écrivains
 - *Read/Write Locks*
 - Plusieurs lecteurs en même temps
 - Un seul écrivain, défavorisé
 - Risque de famine ?
 - Le compteur de lecteur peut devenir un goulet d'étranglement...

Instructions atomiques

- Modification atomique simple
 - Entiers, pointeurs, ...
- Maintien de compteur
 - `atomic_inc`
- Tentative de prise seul d'une ressource
 - `test_and_set`
 - `cmpxchg`
 - Utilisé pour les spinlocks
 - Beaucoup plus efficace que modification entre prise et relâchement de verrous

Interruptions

- Le processeur peut être interrompu **n'importe quand**
 - Périphériques, horloge (préemption), ...
 - Y compris au milieu d'une section critique
 - Même s'il tient un verrou
- Si un code manipule une structure modifiable lors d'une interruption ?
 - Désactiver les interruptions avant de le manipuler
 - En plus d'un verrou pour se protéger des accès concurrents hors interruption (autres tâches)

Désactivation des interruptions

- Instructions dédiées pour désactiver les interruptions vers ce processeur
 - Ou vers tous les processeurs si vraiment nécessaire
 - Une seule ou toutes les interruptions à la fois
- Ne pas les désactiver trop longtemps
 - Ca nuit à la réactivité
 - On ne peut plus traiter les autres événements

Sémaphore

- Compteur (atomique) limité de ressources
 - Initialisé au nombre de ressources disponibles
- Equivalent à un verrou si une seule ressource
- Si trop d'accès, on bloque en attendant que quelqu'un relâche une ressource
 - `atomic_dec_and_test`
 - Sommeil si raté
 - Désordonnement
 - Possible avec attente active mais peu utile
 - L'attente est souvent longue

2018-2019

Brice Goglin

313

Attendre un événement avec un sémaphore

- Sémaphore initialisé à 0
 - Signifie « Événement non arrivé »
- Tentative de prise du sémaphore pour attendre l'événement
 - Bloque tant que personne ne relâche
- Arrivée de l'événement relâche le sémaphore
 - Réveille le processus endormi
- Utilisé pour attendre la fin des I/O
 - Le traitant d'interruption relâche le sémaphore et cela réveille l'application en attente

2018-2019

Brice Goglin

314

Variables spécifiques au processeur

- Il y a toujours une seule tâche par processeur
 - Pas de concurrence d'accès aux variables « locales » du processeur
 - Sauf en cas de préemption
 - Désactiver les interruptions
- Utile quand une tâche est assurée de rester sur le même processeur
 - Threads noyau
- Ex: pour des statistiques

2018-2019

Brice Goglin

315

Optimisations des verrous : *Read-Copy-Update*

- Certaines modifications peuvent garder la cohérence des objets
 - Lecture possible pendant modification
 - Verrouillage uniquement pour modifications concurrentes
 - Nécessite ordre/cohérence des modifications
- Ne pas désallouer avant que lecteurs ait terminé
 - On ne connaît pas les lecteurs !
 - Assistance de l'ordonnanceur pour savoir quand ils sont garantis d'avoir suffisamment progressé
- Très utile si applicable (ex: pile réseau Linux)

2018-2019

Brice Goglin

316

Gestion du temps

Intérêt

- Notion de temps critique pour l'exécution globale du système
 - Système à la fois *event-driven* et *time-driven*
- Nécessité de savoir quand exécuter des tâches
 - Travaux régulier
 - Préempter et équilibrer les listes d'ordonnement
 - Rafraîchir l'écran
 - Travaux déferés
 - Entrées-sorties disque

Intérêt (2/2)

- Fournir des indications à l'utilisateur
 - Date courante
 - Dates de modifications/accès aux fichiers
 - Fonctions avec timeout
 - Alarmes
 - Statistiques d'utilisation
 - Aussi utile pour les algorithmes d'ordonnement
 - Logs d'événements
 - ...

Comment savoir l'heure ?

- Ticks d'horloge
 - *Programmable Interval Timer (PIT)*
 - Les *jiffies* de Linux
 - Léger et régulier, mais grossier
 - 250Hz sous Linux x86 récent
 - 4 ms de précision absolue
 - Délai de traitement d'interruption
 - Augmenter la fréquence pour augmenter la précision ?
 - Coût prohibitif des interruptions

Comment savoir l'heure, plus précisément ?

- Compteur de cycles
 - Dans la plupart des processeurs modernes
 - Instruction rdtsc sur x86
 - Très précis à court terme, pas coûteux
- Difficile à relier à l'heure actuelle
 - Changements de fréquence
 - Desynchronisation entre processeurs
 - Voire entre coeurs d'un même processeur

Comment savoir l'heure absolue ?

- Ticks et TSC = heure relative
- Heure absolue donnée par l'horloge
 - RTC = *Real Time Clock*
 - Fonctionne même si l'ordinateur est éteint
 - Tant que la pile de la carte mère est en vie
 - Granularité élevée (seconde)
 - Cher à programmer et à consulter

Combiner les sources

- RTC lue au démarrage
 - Voire de temps en temps
- Sert de base aux interruptions d'horloge pour garder une heure logicielle
 - Granularité moyenne (ms)
- Autres fonctionnalités spécifiques au matériel pour augmenter la précision
 - TSC (de moins en moins souvent)

Supprimer les ticks d'horloge ?

- Les ticks d'horloge peuvent être inutiles
 - Système *idle*
 - Rien à faire dans le traitant d'interruption
 - Processus unique exécutant des calculs sans faire d'entrées-sorties
- Les OS suppriment des ticks en programmant uniquement un pour le prochain événement si nécessaire
 - Souci d'économie d'énergie

Mesurer la durée d'ordonnancement

- À chaque tick d'horloge, attribuer la durée d'ordonnancement au processus courant
- Pas très précis
 - Autres interruptions intermédiaires mal comptabilisées
 - Prémptions intermédiaires ignorées
- Suffisant pour l'ordonnanceur
 - *Timeslice* souvent au moins de l'ordre de 10ms
 - Et pas très cher

Programmer une alarme

- Enregistrer une fonction à exécuter après un certain délai
 - Pour réveiller un processus (sleep, timeout, ...)
 - Ou envoyer un SIGALRM, etc.
- Comment l'exécuter ?
 - Il faut que le noyau ait la main au bon moment pour l'exécuter
 - Même problème que pour la préemption
- Vérifier périodiquement
 - Si le délai est dépassé, exécuter

Programmer une alarme (2/2)

- Attente active jusqu'à l'expiration du délai
 - Gaspillage de temps CPU
- Vérifier si le délai expiré à chaque interruption d'horloge
 - Pas très précis
- Les processeurs modernes peuvent programmer des interruptions plus précisément (ex : HPET sur x86)
 - Ex : `ualarm(8100 µs)` si `tick = 4 ms` ?
 - Après 2 ticks d'interruption normale, programmer une alarme HPET dans 0,1 ms

Système d'exploitation

Systemes de fichiers

Intérêts des systèmes de fichiers

- Structurer et organiser les données sauveées sur le disque
 - Pour y accéder facilement
 - Que ce soit l'OS ou l'utilisateur
 - Pour y accéder après un reboot (voire crash)
 - L'OS et l'utilisateur ont oublié où étaient stockées les données sur le disque
- Séparer les données en entités logiques indépendantes
- Cacher le stockage physique sous-jacent

Les problèmes à gérer

- Intégrité des données
- Performance
- Comment stocker les données?
 - Et les metadonnées?
 - Attributs, taille, droits, répertoires, ...
- Comment réparer après un crash?
- Comment gérer les accès concurrents des différents processus
 - Pas forcément sur la même machine

Et si on utilise pas de système de fichiers ?

- On peut ouvrir et écrire directement dans le disque
 - Ex : manipuler /dev/sda5 comme un seul gros fichier de taille fixe
- Disque/partition réservé à une seule application
 - Ou les applications doivent se mettre d'accord sur qui utilise quelle(s) partie(s)

Système d'exploitation

Systèmes de fichiers

Organisation

Pile d'accès au stockage

- Interface de haut niveau
 - fopen, fread, fwrite, ...
- Appels systèmes
 - open, read, write, unlink, mkdir, stat, ...
- Couche de virtualisation des fichiers
 - *Virtual File System* dans Linux
 - Différents « pilotes » de système de fichiers
 - Ext4, vfat, NTFS, NFS, ZFS, ...

Pile d'accès au stockage (2/2)

- Couche de virtualisation des blocs
 - *Block Device Layer* de Linux
 - Différents types de système de stockage
 - IDE (P-ATA), SCSI (S-ATA), Network Block Device, ...
 - Possibilité d'agrégation RAID, LVM, cryptage, ...
- Drivers de chipsets et périphériques
- Périphériques
 - Stockage linéaire de blocs

Descripteurs de fichiers

- Chemin et *File descriptor* en espace utilisateur
 - Avec un offset
- Numéro d'i-noeud dans le VFS
 - Avec un offset ou un offset dans une page
- Numéro de bloc dans périphérique virtuel
 - Puis dans périphérique physique

Structure des systèmes de fichiers

- Arborescence
 - Lien entre noeuds étiquetés par des noms
 - Feuilles = fichiers de données
 - Le reste = Répertoires
- Lien symbolique ?
 - Fichier contenant un chemin
- Lien physique ?
 - Rupture de la structure d'arbre
 - Deux branches arrivent sur le même noeud

Structure des fichiers

- Fichier de données
 - Suite d'octets sans structure pour l'OS
 - Encodage par l'application
 - Ou par les bibliothèques systèmes pour les sections dans les binaires
- Répertoire
 - Suite de descripteurs de fils
 - Nom et attributs
 - Souvent non-ordonné

Montage

- Disque divisible en partitions
- Chaque partition contient une arborescence de fichiers
- Montage = Insérer une arborescence d'une partition dans l'arborescence d'une autre
 - Un morceau de l'ancienne est masqué pendant la durée du montage

Path Lookup

- Trouver un noeud/fichier à partir d'un chemin (chaîne de caractères)
 - Géré par le VFS selon les montages de partition
- Commencer en haut de la partition racine
 - Si chemin absolu
- Diviser le chemin en entités
 - Séparées par des slashes sous Unix
- Noms spéciaux pour faire référence au père (..) ou à soit même (.)

Path Lookup (2/2)

- A chaque étape, extraire la première entité
 - Trouver le fils correspondant dans le répertoire courant
 - Utiliser en priorité la racine du montage le plus récent s'il existe
 - Vérifier les permissions
 - Recommencer depuis cet endroit

Pourquoi des descripteurs de fichiers ?

- Le *Path Lookup* est très lent
 - Multiple accès mémoire, verrous, accès disques potentiels, ...
- Le système « cache » le résultat et l'application prend une référence dessus par l'appel système open
 - Les appels suivants utilisent directement le descripteur
- « Pointeur » dans le fichier avance selon les read/write

Protection

- Il faut pouvoir définir un ensemble de droits des utilisateurs sur les fichiers
 - Soit une liste de droits par utilisateur
 - *Capability*
 - Facile à transférer ou partager
 - Soit une liste de droits par objet
 - *Access Control List (ACL)*
 - Facile à gérer pour l'OS
 - ACLs de chaque fichier sont stockées avec attributs
 - Lourd si plein d'utilisateurs avec droits différents
 - Définir des groupes d'utilisateurs

Objets dans les systèmes de fichiers

- I-noeuds (*Inode*)
 - Objet de base décrivant un fichier ou répertoire
 - Contenu et droits associés
- Dentry (*Directory entry*)
 - Entrée stockée dans un répertoire
 - Décrire un lien vers un i-noeud fils
 - Contient un nom

Objets dans les systèmes de fichiers Linux (2/2)

- File
 - Descripteur de fichier ouvert
 - Pointe vers un Dentry et décrit la position actuelle
- Superblock
 - Descripteur de système de fichiers
 - Premiers blocs d'une partition
 - Donne le type du FS
 - Pointe vers l'i-noeud racine

Méthodes des fichiers

- Le *Virtual File System* est programmé sur le modèle des objets
 - Objets spécifiques à chaque système de fichiers, hérités des objets génériques du VFS
 - Méthodes génériques au VFS ou spécifiques au système de fichiers
 - Routines de lecture/écriture des données et métadonnées
 - read, write, stat, mkdir, rmdir, link, chmod, ...

Méthodes spécifiques aux systèmes de fichiers

- Fonctions de gestion des blocs où les données sont stockées
 - Allocation, libération, recherche, ...
 - Pour les répertoires et pour les fichiers
 - Lecture, écriture
 - Utilisé explicitement par read/write/...
 - Utilisé implicitement après mmap
- Les méthodes du FS utilisent ces routines pour gérer le stockage réel

Fichiers spéciaux

- Chaque méthode peut avoir un comportement spécial dépendant du fichier
 - /dev/null, /dev/zero, /dev/full, /dev/mem, ...
- On peut créer des fichiers spéciaux avec un comportement quelconque en lecture/écriture/ioctl

Système d'exploitation

Systèmes de fichiers

Stockage

Plusieurs problèmes

- Répartir et lier les différents blocs des fichiers/répertoires au sein du disque
- Trouver et gérer les blocs libres
- Stocker/organiser le contenu des répertoires par dessus ces blocs

Descriptions des fichiers

- Un fichier est une suite de bloc
 - Structuré (répertoire) ou non (fichier)
 - Fichier = Structuré par l'application
 - Taille variable dynamiquement
- Et des métadonnées
 - Eventuellement de taille variable
 - ACLs, ...

Besoins des fichiers

- Trouver le bloc correspondant à un morceau de fichiers
 - Pour le lire ou le réécrire
 - Accès séquentiels ou aléatoires
- Allouer un bloc
 - Pour écrire dans le fichier
 - À la fin ou dans un trou
- Libérer un bloc
 - Pour supprimer ou tronquer un fichier

Besoins des fichiers (2/2)

- Le plus important est la recherche
 - Doit être rapide pour accès aléatoires ou séquentiels
- Allocation/libération moins critiques
- Comment organiser ces blocs?
 - Les blocs contenant un fichier sont stockés dans la structure inode correspondante
 - Avec ou sans indirection

Allocation des blocs des fichiers

- Allocation contigue de tous les blocs?
 - Recherche facile
 - Accès disque minimisé (un seul seek)
 - Allocation difficile (comme le partitionnement)
 - First Fit ou Best Fit?
 - Fragmentation externe
 - La taille des fichiers peut varier
 - Souvent imprévisible
 - Agrandissement progressif pendant remplissage
 - Utilisé pour les CD-ROM
 - Préalcul de l'allocation avant gravure

Allocation par liste

- Liste chaînée de blocs pour chaque fichier
 - Éléments de liste stockés dans blocs réservés
 - Pour ne pas réduire l'espace dans les blocs réels
 - Simple, mais peu efficace pour accès non-séquentiels
- FAT (*File Allocation Table*) de DOS et Windows
 - Blocs groupés par « cluster »
 - Moins d'éléments de liste à traverser
 - FAT = Tableau statique de pointeurs vers le cluster suivant (suite du fichier courant)

Allocation par groupe de blocs

- Un nombre fixé contigu est alloué initialement
 - On rajoute d'autres groupes si nécessaire
- Accès disque optimisé
 - si la taille des groupes est grande
- Fragmentation interne
 - si la taille des groupes est grande

Allocation indexée

- Arbre de pointeurs vers les blocs
 - Accès non-séquentiels efficaces
 - Stocké dans un B-tree (arbre trié)
 - Un peu coûteux en gestion mais rapide
 - Recherche, parcours séquentiel, insertion, suppression en temps logarithmique
 - Econome en espace
 - Nombre variable mais faible de blocs d'indirections
 - Incomplet et un peu déséquilibré quand nécessaire
 - Combiner avec groupes contigus pour améliorer accès séquentiels

Occupation disque

- Stocker les données
 - Et les metadonnées décrivant les données
 - ex: Arbre d'indirection vers les blocs
- Écrire dans premier bloc pas équivalent à écrire dans 10⁹ème bloc
 - Allocation blocs d'indirection intermédiaires
 - L'occupation disque d'un fichier peut être plus grande que les données réellement contenus !

Fichiers à trous

- Seuls les blocs utiles sont alloués
 - ex: Si on écrit à la fin d'un fichier sans jamais toucher au début
- Taille fichier != Occupation disque
 - L'occupation disque d'un fichier peut être beaucoup plus petite que sa taille !
- Pour s'en rendre compte, jouer avec `dd count=...,seek=...`

Gestion de l'espace libre

- Comment trouver un ou des blocs pour les allouer à un fichier ?
- Prendre le premier bloc libre ?
 - Facile avec une liste
- Prendre un bloc libre proche d'un autre bloc ?
 - Utile pour performance des accès séquentiels sur un disque dur traditionnel
 - Arbre d'indirection des blocs libres vers les suivants

Gestion des répertoires

- Un répertoire est un fichier a contenu structuré
 - La structure dépend du système de fichier
 - La structure est stockée dans les blocs de l'inoeud
 - Eux-mêmes déjà organisés en arbre dans le stockage physique (cf avant)
- Opérations de gestion de ce contenu
 - Allocation, libération, recherche et parcours des entrées de répertoire
 - Par dessus les opérations de gestion des blocs qui contiennent ces entrées

Gestion des répertoires (2/3)

- Besoin principal = Recherche rapide !
 - Trouver le fils pour un nom donné
 - Pendant le *Path Lookup* pour tous les appels-système qui manipulent un chemin
 - Sans optimisation, on parcourerait la moitié des entrées du répertoire en moyenne
- L'allocation (créer un fichier ou répertoire), la libération (effacer) ou le parcours (*ls*, *readdir*) sont beaucoup moins utilisés

Stockage des noms

- Tableau d'entrées de taille fixe ?
 - Embêtant pour les noms de fichiers longs
- Tableau d'indirections vers des noms
 - Allocation d'espace variable pour stocker les noms dans le « contenu » du répertoire
 - Dans les blocs suivants

Gestion des répertoires (3/3)

- Comment rechercher rapidement?
 - C'est l'opération la plus courante sur les répertoires!
 - Table de hashage au lieu du tableau?
 - S'il y a peu d'entrées dans le répertoire
 - En général, un B-tree, à nouveau
 - Un B-tree logique pour les entrées de répertoire, stocké dans un B-tree de blocs physiques de l'inoeud

Système d'exploitation

Systèmes de fichiers

Fonctionnalités diverses

Accès concurrents et consistance

- Quand un processus modifie un fichier, que voient les autres et quand ?
- Sémantique de partage de fichiers UNIX
 - Modifications visibles par les autres immédiatement
- Sémantique NFS « Close-Open »
 - Modifications enregistrées lors de la fermeture et visible aux futurs open
- Partage du pointeur possible par dup/fork
 - Mais certaines fonctions n'utilisent pas ce pointeur (pread/pwrite/...)

2018-2019

Brice Goglin

365

Verrouillage des fichiers

- Verrouillage collaboratif
 - Global sur un descripteur de fichier
 - flock()
 - Sur un segment de descripteur de fichier
 - fcntl() avec argument F_SETLK, ...
- Verrouillage réel par le système
 - Vérification des droits lors des accès en lecture ou écriture
 - *System V Mandatory Locks*
 - Activé par les droits g+s-x sur le fichier

2018-2019

Brice Goglin

366

Journalisation

- Tester/corriger un FS est très long
 - Vérifier les arbres de blocs
 - Vérifier la structure des répertoires
 - Nécessaire en cas de crash avant démontage
 - Modifications pas forcément toutes déjà sauvées
- Sauver les modifications dans un journal séparé avant de les appliquer
 - Restauration à l'original si crash pendant application de la modification
- Conserver la cohérence du FS

2018-2019

Brice Goglin

367

Le Buffer-Cache

- Éviter les accès aux disques incessants
 - Regrouper les lectures/écritures
 - Éviter le seek-time des disques et profiter du débit
 - Garder les lectures en mémoire
 - Ne pas relire plusieurs fois depuis le disque
 - Principe de localité
- Ensemble de pages de fichiers gardées en mémoire du système
 - Remplies en mémoire quand c'est nécessaire
 - Sauvées sur le disque de temps en temps si modifiées

2018-2019

Brice Goglin

368

Accès explicite ou implicite

- Accès explicites (read, write, ...)
 - Copie entre l'espace utilisateur et le buffer-cache
 - On sait directement quels buffers ont été modifiés
- Accès implicites (mmap+déréférencement)
 - Mapping du buffer-cache dans l'espace du processus
 - Utilisation du bit *Dirty* pour savoir les buffers modifiés

Interaction avec le cache de pages

- Toutes les pages des processus sont considérées comme des pages de fichiers
 - Privé, publique, anonyme, ...
 - Dans Linux, *Buffer-Cache* = *Page-Cache*
- Optimisations génériques du système de gestion mémoire
 - Lecture des fichiers à l'avance (*Read-ahead*)
 - Sauvegarde regroupée, selon bit *Dirty*

Court-circuiter le *Buffer-Cache*

- *Buffer-Cache* gênant dans certains cas
 - Le système doit allouer des pages pour le *Buffer-Cache*
 - Les écritures ne sont pas synchrones sur le disque
 - Ex : Calcul *Out-of-core*
- On peut demander des accès non-cachés
 - Paramètre `O_DIRECT` de `open()`
 - DMA entre le disque et la mémoire utilisateur
 - Tampons mémoire verrouillés, tailles alignées

Système d'exploitation

Entrées-sorties

Objectifs

- Les I/O sont le maillon faible du système
 - Périphériques lents
 - Têtes de lecture des disques
- Maximiser l'utilisation pour ne pas réduire l'utilisation processeur
 - Mais plus de processus pour exploiter les processeurs impose plus I/O pour le swap
- Organiser la vision des périphériques
 - Vision uniforme, même si hiérarchique

Types de périphériques

- Entrées
 - Clavier, souris, scanner, micro, capteurs, ...
- Sorties
 - Ecran, imprimante, ...
- Entrées-sorties
 - Stockage, réseau, ...
- La mémoire n'est pas considérée comme un périphérique
 - Elle est déjà impliquée dans la plupart des instructions...

I/O logiques ou physiques ?

- I/O physiques vers les périphériques
 - Uniquement par l'OS
 - Sauf cas particuliers
- I/O logiques depuis l'application
- Structuration, protection et abstraction assurée par l'OS entre les deux

Système d'exploitation

Entrées-sorties

Entrées-sorties
physiques et matérielles

Structure matérielle

- Chipset(s) d'I/O relié(s) au bus mémoire
- Un ou plusieurs bus d'I/O par chipset
 - Des ponts (bridges) entre bus
- Arborescence de bus entre l'hôte et les périphériques
 - Voir la sortie de lspci -vt
 - Les feuilles sont des périphériques

Ressources matérielles

- Ressources exportées par périphériques vers hôte
 - Zones de mémoire ou de registres
 - Exportées par le « protocole » PCI
 - Mappées en mémoire physique par le BIOS et l'OS
 - Voir la sortie de lspci -vvxxx
 - Connu et utilisé par le pilote
- Ressources internes
 - Processeurs, mémoire, registres, ...
- Ressources de l'hôte exportées aux I/O
 - Mémoire accessible par DMA

Caractéristiques des I/O physiques

- Périphériques à caractéristiques variables
 - Débit, latence, protocole, granularité, erreurs, ...
- Gestion des I/O autonome ou non
 - Processeur qui contrôle tout
 - Périphérique qui exécute des commandes
 - Périphérique qui interrompt le processeur
 - Périphérique qui accède à la mémoire centrale
 - Périphérique programmable

Programmed I/O

- Entrées-sorties traitées par le périphérique selon des commandes du processeur
 - Ecriture de requête par le processeur
 - Lecture de réponse par le processeur
 - Accès direct aux registres du périphérique
- Attente de terminaison impose une attente active
 - while (!terminé);
- *Port I/O ou Memory Mapped I/O*

Port I/O

- *Programmed I/O* où les registres des périphériques sont accédés par instructions spéciales
 - Ils sont « mappés » dans un espace de ports
 - Voir `/proc/ioports`
 - Lus et écrits par bloc de 1, 2, 4 ou 8 octets
 - Ex : Instructions `inb`, `outb`, `ins`, `outs`, `inl`, `outl`, ...
- Sert essentiellement pour envoyer des commandes ou lire des réponses

Memory Mapped I/O

- *Programmed I/O* par accès naturel aux ressources des périphériques
 - Comme de la mémoire classique
 - Par déréférencement de pointeur
 - Un peu plus lent
 - Et pas caché en général (effets de bord, ...)
 - Permet d'utiliser la même interface pour les accès à la mémoire centrale et aux périphériques
- Peut servir pour envoyer des commandes, lire des réponses, ou transférer des données

Attente de terminaison

- Comment attendre la fin d'un traitement dans un périphérique ?
 - *Programmed I/O* impose attente active
 - Le processeur teste sans arrêt une valeur statut
 - « Scrutation »
 - Intéressant pour les petites requêtes
 - Réactivité très bonne
 - Mauvais pour les requêtes lentes
 - Monopolisation du processeur qui ne peut rien faire d'autre

Attente de terminaison (2/2)

- Scrutation de temps en temps?
 - Impose changement de contexte entre scrutation et l'application qui essaie de s'exécuter pendant ce temps
 - Mauvais pour performance et pollution du cache
- Il faut notification explicite du périphérique quand il a terminé
 - *Interrupt-driven I/O*
 - Le processeur peut recouvrir le traitement dans le périphérique par l'exécution d'une autre tâche

Interruptions

- *Interrupt Request (IRQ)*
- « Message » d'un périphérique pour prévenir le processeur d'un événement
 - Le processeur stoppe son travail actuel
 - Déroutement, comme appel système ou exception
 - Il traite le message du périphérique
 - Il retourne à son travail précédent
- Nécessite support matériel
 - Et l'OS doit fournir un traitant que le processeur exécutera en cas d'interruption

2018-2019

Brice Goglin

385

Qui a envoyé l'interruption ?

- Il y a différentes interruptions
 - Les « Lignes » d'interruption peuvent être partagées entre périphériques
 - Voir `/proc/interrupts`
- Le numéro d'interruption donne une petite liste de périphériques possibles
 - Leurs drivers savent leur demander si cela vient d'eux
 - Le driver fournit une fonction de traitant
 - PIO ou MMIO dans un registre de statut du périphérique pour savoir si l'interruption vient de lui

2018-2019

Brice Goglin

386

Que faire en cas d'interruption ?

- Traitement similaire à exception/appele-système
 - Déroutement de l'OS, traitement puis retour
 - Pas de contexte d'exécution spécifique requis
- Le driver sait quoi faire
 - Le traitant d'interruption s'en charge
 - PIO ou MMIO dans d'autres registres pour savoir les commandes à exécuter
 - Lecture de données en mémoire à traiter
 - L'état est spécifique au driver et matériel

2018-2019

Brice Goglin

387

Réactivité vs. Disponibilité

- L'exécution du traitant est immédiate
 - Trop d'interruptions peuvent empêcher tout autre travail
- Le périphérique évite de trop en envoyer
 - Il peut attendre un peu et interrompre une seule fois pour plusieurs événements
 - *Interrupt Coalescing*
 - Moins de charge processeur
 - Moins de réactivité

2018-2019

Brice Goglin

388

Traitements lourds dans les traitants d'interruption ?

- Les traitants doivent être courts
 - Ne pas nuire à la réactivité du système
 - Éviter de générer d'autres IRQ, exceptions, ...
 - Ex : Éviter les allocations mémoire
- Comment effectuer un gros traitement interruption ?
 - Ex : réception réseau, copie, CRC, ...
 - Faire le strict minimum dans le vrai traitant
 - Programmer le reste pour plus tard

Direct Memory Access

- Envoyer ou lire des données par PIO est très coûteux
 - Encore plus coûteux en CPU que copie mémoire
 - Et ca interdit le recouvrement
- Les périphériques modernes savent lire ou écrire directement en mémoire centrale
 - DMA (*Direct Memory Access*)
 - Un moteur DMA dans le chipset mémoire et un dans les périphériques
 - Processeur central pas du tout utilisé
 - Recouvrement total par autre exécution

Direct Memory Access (2/2)

- Les périphériques manipulent @bus
 - Equivalent de @phys vu des bus et périphériques d'entrées-sorties
 - Pas de notion de mémoire virtuelle, de défauts de pages, etc.
- L'OS traduit les @virt ou @phys en @bus
 - Puis les donne au périphérique
 - En paramètre de la commande
 - Et il verrouille les pages virtuelles pendant l'I/O
 - La traduction d'adresse doit être valide

Quelles stratégies utiliser ?

- C'est le pilote décide
 - selon les capacités du matériel
- Envoi de la commande
 - **PIO Write** par le processeur (en général)
 - Périphérique peut pas deviner quand on va lui envoyer une commande

Quelles stratégies utiliser ? (2/3)

- Notification de terminaison
 - Si traitement potentiellement long, **Interruption** par le périphérique
 - Le traitant du pilote est appelé
 - Si traitement court, le pilote peut faire attente active par **PIO Read** sur le processeur
 - Le pilote n'avait pas rendu la main après l'envoi de la commande, il traite aussitôt le résultat
- Les tâches qui attendaient cette I/O sont réveillées
 - Quand le pilote a terminé
 - A la fin du traitant d'interruption
 - Ou à la fin des PIO Read

Quelles stratégies utiliser ? (3/3)

- Transfert de données en entrée (hôte vers I/O)
 - Si peu de données, **PIO Write** avec la commande
 - Sinon (souvent), **DMA Read** par le périphérique
- Transferts de données en sortie (I/O vers hôte)
 - Si peu de données, le pilote fait un **PIO Read** par le processeur juste après la notification
 - Sinon **DMA Write** par le périphérique juste avant la notification

Exemples

- Liens sur la page du cours
 - Tout sauf MMIO dans le driver 8139too
 - MMIO dans le driver myri10ge

Système d'exploitation

Entrées-sorties

Entrées-sorties logiques

Pourquoi des entrées-sorties logiques ?

- Masquer la complexité du matériel
 - Virtualiser les ressources
- Structurer les ressources
 - Systèmes de fichiers, connexions réseau, ...
- Partager entre plusieurs utilisateurs
 - Avec protection

Exemples

- Pour le stockage
 - I/O logiques = lecture d'un fichier
 - I/O physiques = commandes IDE ou SCSI
- Pour le réseau
 - Logiques = accès aux sockets et protocoles
 - Physiques = envoi/réception de paquets
- Pour le clavier
 - Logique = terminal
 - Physique = lecture de caractères

Exemples avec le stockage

- Accès depuis un processus
 - I/O logique (dans la libc ou le noyau)
 - Conversion chemin d'accès en inoead
 - Conversion segment de fichier en blocs
 - I/O physique (dans le noyau ou les drivers)
 - Accès aux blocs sur un disque
 - Commandes IDE ou SCSI au controleur

Blocs ou caractères ?

- Certaines I/O sont des flux
 - Entrant et/ou sortant
 - Clavier, souris, cartes réseau
 - Les données sont lues/écrites une seule fois
- Certaines I/O ont une granularité matérielle
 - Stockage par blocs sur les disques
 - Les données sont lues par gros bloc
- On peut aussi mixer les deux
 - Fichier habituel : ni flux, ni blocs
 - Possibilité de flux de blocs, ...

Système d'exploitation

Entrées-sorties

Entrées-sorties logiques en mode caractère (flux)

Entrées-sorties par caractère

- Flux de données entrant
 - Une donnée lue est perdue
 - Entrée à vitesse différente de l'application ?
 - Mettre données supplémentaires dans une file
 - Bloquer l'application si pas assez de données
 - Lecture à l'avance
 - Remplir la file tant que possible
 - Réduire/regrouper les accès au matériel
 - Accélérer les accès logiques suivants
 - Pas besoin d'attendre le matériel

Entrées-sorties par caractère (2/2)

- Flux de données sortant
 - Une donnée écrite ne peut plus être remplacée
 - Sortie à vitesse différente du matériel ?
 - Mettre l'application en attente si elle écrit trop
 - Et les données dans une file pour le matériel
 - Regrouper les accès physiques
 - Réduire le temps apparent d'émission en réduisant les accès au matériel

Entrées-sorties partagées

- Pas de notion de cache en mode caractère
 - Chaque caractère est lu et écrit une seule fois
- Files/anneaux de caractères/paquets partagés
 - Multiplexage entre applications dans même flux logique
 - ex: Plusieurs applications partageant un tube
 - Multiplexage entre flux logiques dans même canal matériel
 - ex: Plusieurs connexions réseau

Entrées-sorties non-bloquantes

- Application bloque indéfiniment si rien à lire ou trop à écrire
 - Attente passive
 - Impossible de travailler en attendant
- Essayer une I/O non-bloquante, et abandonner s'il faudrait attendre
 - Voir O_NONBLOCK pour socket

Entrées-sorties non-bloquantes multiplexées

- Comment attendre sur plusieurs sources d'événements en même temps ?
 - L'attente sur une source empêche de récupérer les événements des autres
 - Attendre un peu sur chaque source ?
 - Couteux et ne passe pas à l'échelle
 - Ex: Serveur www avec 1000 connexions

poll/select

- Le processus devrait se mettre sur plusieurs files d'attente en même temps
 - Le premier événement d'une source le réveille
- Passer un tableau de descripteurs de fichiers au noyau
 - poll() ou select()
 - Attention au passage à l'échelle
 - Copier des milliers de descripteurs à chaque attente ?
 - Et si la source n'est pas un fichier ?

Système d'exploitation

Entrées-sorties
Entrées-sorties logiques
en mode bloc

Entrées-sorties par blocs

- Ensemble de blocs qu'on peut lire/écrire plusieurs fois
 - Pas de notion de flux
 - Accès aléatoire à n'importe quel bloc
 - Pas si simple en pratique...
- Possibilité de regrouper les accès
 - Cacher les données lues dans le système
 - *Page-Cache* pour les fichiers
 - Regrouper les écritures
 - Cachées dans le *Page-Cache* également

2018-2019

Brice Goglin

409

Entrées-sorties *bufferisées*

- Lire à l'avance ou retarder les écritures
 - Un seul accès physique pour plusieurs accès logiques répétitifs et/ou consécutifs
- Masquer la granularité matérielle
 - Blocs sur le disque
 - Comment lire un seul caractère ?
 - Pages en mémoire
 - Mémoire virtuelle non consécutive en mémoire physique
- Partage du cache entre applications

2018-2019

Brice Goglin

410

Entrées-sorties bloquantes

- I/O physiques souvent *Interrupt-driven*
 - Traitement en arrière plan par le périphérique puis interruption
 - Le processeur peut faire autre chose
- I/O logiques sont souvent bloquantes
 - Application attend terminaison I/O physique
 - Ne peut pas utiliser le processeur
 - Sauf avec des threads
 - Nécessite plus de tâches pour exploiter le processeur
- Sémantique de O **NONBLOCK** inapplicable

2018-2019

Brice Goglin

411

Écritures (moins) bloquantes

- Faire croire à l'application que l'I/O physique est terminée
 - Copier les données à écrire dans le cache et retourner un succès
- L'application peut continuer à travailler pendant I/O physique en arrière plan dans le système
- Et en cas d'erreur dans l'I/O physique?
 - L'application a continué à travailler en pensant qu'il n'y avait pas d'erreur?

2018-2019

Brice Goglin

412

Lectures (moins) bloquantes

- L'I/O physique doit avoir lieu avant
- Prédire les I/O pour les anticiper
 - Les I/O logiques seront immédiates
- Lecture disque à l'avance
 - Page-Cache des fichiers
 - Selon heuristiques d'accès aux fichiers

Entrées-sorties asynchrones

- Exposer l'asynchronisme matériel aux applications
 - Ne pas bloquer l'application pendant I/O physique
 - Notifier la terminaison de l'I/O plus tard
- Permet recouvrement des I/O
 - Sans rajouter des tâches pour exploiter les processeurs

Entrées-sorties asynchrones (2/2)

- Complexe à utiliser?
 - Impose de prévoir nos I/O à l'avance
 - Les soumettre bien-avant d'avoir besoin du résultat
 - Pas forcément plus complexe que les threads
 - Threads imposent d'exposer du parallélisme
- Voir la manpage de `io_submit()`
 - Ou `aio_read()` pour des fausses I/O asynchrones utilisant des threads

Ordonnancement des I/O

- Pas ou peu d'ordonnancement pour I/O en mode caractère
 - Premier arrivé, premier servi
 - Qualité de service réseau?
- Les I/O par blocs n'ont pas besoin d'ordonnancement ?
 - Ca dépend du matériel
 - Tous les accès ne sont pas équivalents en performance

Ordonnancement des accès disques

- Le déplacement de la tête de lecture est très lent (5 ms en moyenne)
 - Minimiser les déplacements
 - Trier par algorithme de l'ascenseur (*Elevator*)
 - Regrouper les requêtes proches, même si les fichiers n'ont rien à voir
- Optimisations
 - *Anticipatory Scheduler*
 - Attendre un peu en cas d'autres accès consécutifs
 - *Deadline*
 - Privilégier la lecture, l'écriture peut attendre

Système d'exploitation

Entrées-sorties

Entrées-sorties configurables ou dédiées

I/O logiques dédiées

- Toutes les I/O logiques ne correspondent pas à une I/O physiques en dessous
 - Tube, Socket locale, ...
 - Appels système dédiés
 - Difficile (et dangereux) d'en ajouter
 - Fichiers caractère spéciaux
 - `/dev/null`, `/dev/zero`, `/dev/random`
 - cf TD5
 - Mode caractère ou bloc selon les besoins
 - Manipulés par appels-système `read/write` habituels
 - Via méthodes spécifique à chaque fichier

Au delà de `read/write`

- On peut abuser de `read/write`
 - On définit le comportement des fichiers comme on le souhaite
 - Ex : fichier spécial pointant vers une webcam
 - `Read` pour lire la prochaine image
 - `Write` à offset spécial pour changer la résolution
 - `Write` à autre offset pour changer le zoom
 - Autant de cas particuliers qu'on veut
 - Quid des opération qui ne ressemblent ni à `read` ni à `write` ?
 - Ex : opération avec données en entrée et en sortie

Les IOCTL

- *Input/Output Control*
- `ioctl()` est un appel système configurable pour chaque fichier
 - Numéros de commandes spécifiques
 - Fonctions spécifiques au fichier pour traiter ces commandes
 - Argument pouvant être un pointeur
 - Peut pointer vers multiples données, entrées et/ou sorties, tableaux, structures, ...
 - Il suffit de définir la convention
- `man ioctl` et cf `sock_ioctl()` dans `net/socket.c`

Système d'exploitation

Virtualisation

Intérêts de la virtualisation

- Exécuter plusieurs OS en même temps
 - Jouer sous Windows et travailler sous Linux
- Placer plusieurs serveurs sur une même machine sans risquer que les pannes d'un ne gênent les autres
 - Rebooter les sous-serveurs indépendamment
- Développement et débogage
 - Débugger ou rebooter un OS depuis un autre

Intérêts de la virtualisation (2/2)

- Facilité de déploiement pour les utilisateurs
 - Création d'image système spéciale pour s'exécuter dans une VM sans casser l'installation existante
- Permettre la gestion transparente de nouveaux périphériques
 - Il suffit qu'un OS d'une VM le supporte et l'exporte aux autres

Terminologie

- OS hôte = OS principal tournant sur machine physique
- VM = *Virtual Machine*
- OS invité = OS tournant dans VM

En pratique

- On lance une nouvelle fenêtre qui affiche la sortie de la machine virtuelle
 - Elle utilise la fenêtre comme écran !
 - Plein de périphériques virtuels
 - Carte graphique, réseau, son
 - Contrôleur disque = émule disque par un fichier
 - Mémoire = mémoire virtuelle du processus hôte qui exécute la machine virtuelle
 - Processeur = processeur qui exécute le processus hôte de la machine virtuelle
 - Un gestionnaire fait le lien entre ces périphériques virtuels et ceux de l'OS hôte

Différents niveaux de privilèges

- Utilisateur dans VM
 - Mode non-privilegié habituel
- Noyau dans VM
 - Privilèges noyau habituels
 - Gérés matériellement ou non, voir plus loin
 - Ne peut pas sortir de sa VM
 - Pas toucher aux périphériques, en général
- Noyau dans l'hôte
 - Privilèges intégraux
 - Peut manipuler les VM

Virtualisation totale (qemu)

- Exposition de matériel virtuel
 - Processeur, mémoire, disque, réseau, ...
- OS invité = Processus exécuté par l'OS hôte
 - Chaque instruction de la VM est émulée par l'hôte
 - Le processus de gestion de VM s'en charge
 - Très lent
 - Privilèges vérifiés à la volée pendant émulation
- N'importe quel OS invité sur n'importe quel matériel
 - Si l'OS hôte supporte le matériel réel
 - Si l'OS invité supporte le matériel virtuel

Virtualisation totale accélérée (kqemu, VirtualBox, ...)

- Les instructions « normales » n'ont pas besoin d'être émulées !
 - Exécution native beaucoup plus rapide
 - Mais seule l'architecture native est supportée
- Détecter instruction privilégiées ?
 - En mode non-privilégié, elles déclenchent une exception dans le processeur natif
 - Traitée par le pilote du gestionnaire de VM dans OS hôte
 - Instruction privilégiées vérifiées et émulées

Para-Virtualisation (Xen)

- Un *hyperviseur* (OS très simple)
 - Partage le matériel entre plusieurs OS invités
- Le premier OS invité est le maître
 - L'hyperviseur lui confie les périphériques
 - Périphériques virtuels exportés aux autres
 - Pilotes *para-virtualisés* simples, optimisés pour performance
 - Partage de pages avec le maître
- Processeur partagé pour de vrai
 - Ordonnancement simple par l'hyperviseur

Para-Virtualisation (2/2)

- OS invité appelle l'hyperviseur quand nécessaire
 - Ex : Création de VM, accès au matériel, ...
- Tous les OS invités doivent être modifiés pour parler correctement à l'hyperviseur (Hypercall)
 - Utilisation du Ring 1 intermédiaire
 - Sinon tous les appels-système doivent passer par l'hyperviseur
 - C'est pour cela que Windows ne fonctionnait pas sur Xen

Support matériel pour la virtualisation

- Supporter nativement la gestion des VMs
 - Pas d'exception dans l'hôte en cas d'instruction privilégiée invitée
 - Intégrer l'hyperviseur dans le matériel
- Nouvelles instructions de gestion de VM
 - Intel VT-x, AMD-V, ...
 - Registres pour tables de pages hôte et invité, ...
 - Pas besoin de Ring 1
 - Le matériel gère alternance entre mode VM (restreint) et mode normal
- Utilisé par toutes les solutions modernes de virtualisation

Support matériel pour la virtualisation (2/3)

- Lancement de VM sur x86 VT-x
 - L'hôte décrit un OS invité dans structure dédiée en mémoire (VMCS)
 - VMLaunch (ou VMResume) passe la main à OS invité spécifié dans VMCS
 - Exécution de l'invité « pendant » VMLaunch ou VMResume
 - En fait, le processeur fait un super-changement de contexte VM en dessous
 - Et il se rappelle dans quel mode il est

2018-2019

Brice Goglin

433

Support matériel pour la virtualisation (3/3)

- Gestion des VM sur x86 VT-x
 - En cas d'instruction privilégiée (au sens des VM)
 - Exemple : Accès au VMCS (seul l'hôte peut)
 - Le processeur vérifie si la VM courante a les privilèges nécessaires
 - Si non (OS invité), le processeur sort de l'invité et retourne dans l'hôte (*VM Exit*)
 - L'OS hôte traite la requête puis relance l'invité
 - Equivalent d'un hypercall si l'accès était valide
 - Plus besoin d'hypercalls
 - Plus besoin de modifier l'OS invité !

2018-2019

Brice Goglin

434

Virtualisation des périphériques

- Seul le processeur est vraiment partagé entre les OS para-virtualisés
 - L'accès aux périphériques reste assez lent
 - Seul le maître y a un accès direct
- Si le périphérique n'est pas partagé, l'hyperviseur peut le confier directement à un autre OS (*PCI Pass-Through*)
 - Problème de protection
 - Ex: DMA vers la mémoire des autres OS?
 - Solution: support matériel dans le chipset PCI
 - Intel VT-d

2018-2019

Brice Goglin

435

Et les containers ?

- LXC, Docker, ...
- Isolation purement logicielle
 - Les processus sont répartis dans différents containers qui disposent de différentes ressources et différents droits
 - chroot, control group, namespaces, ...
- Peu de surcout
- Moins de protection ?
 - Plus facile de s'échapper d'un container que d'une VM ?

2018-2019

Brice Goglin

436