Programmation Système

Yves Caniou <yves.caniou@ens-lyon.fr>

Module M2OP1 – Master 2 SIR de l'UCBL

2005-2006

(version du 1er février 2006)

Présentation du module

Contenu et objectifs du module

- Un cours assez général sur les Systèmes d'Exploitation (OS)
  - Des concepts Unix
  - ... détaillés pour Linux
- Dans les TPs
  - Simulateurs : QEMU, BOCHS... ?
  - Modules noyau Linux
  - ... avec gestion de la mémoire, des périphériques
  - ... et des threads, des interruptions

Prérequis : C pour les TPs

Évaluation

- Des rapports de TPs... .. ..concis et précis
- Un examen sur table (date et modalités à préciser)

Quelques références pour faire ce cours

Livres

- William Stallings : Operating Systems, Internals and Design Principles, *International Edition* (5eédition)
- Maurice J. Bach : The Design of the UNIX Operating System, *Prentice Hall*
- Rubini & Corbet : Linux Device Drivers, *O'Reilly*, (2eédition)
  → accessible à l'URL `http://www.xml.com/ldd/chapter/book/index.html`

Les docs du noyau Linux et les URL

- Dans `/usr/src/linux/Documentation`
- Pour des infos sur Linux
  - Linux Kernel Module Programming Guide `http://www.tldp.org/LDP/lkmpg/`
  - `http://www.kernelnewbies.org`
  - `http://lxr.linux.no`

Quelques références pour faire ce cours

Des cours en ligne

- Supports de Brice Goglin sur lesquels repose ce cours (merci !)
  `http://perso.ens-lyon.fr/brice.goglin/`
- Cours de Martin Quinson, à qui j'ai emprunté le style et qqs slides (merci)
  `http://www.loria.fr/~quinson`

Les magazines : Linux Mag' (les articles sur la conception d'un OS)

URL et les différentes informations relatives au cours

- `http://graal.ens-lyon.fr/~ycaniou/teaching/0506.html`

Bibliographie

Livres

- Tanenbaum : Modern Operating Systems, *Prentice Hall*, (2eédition)
- Silbershcatz et Gavin : Principes des systèmes d'exploitation, *Addison-Wesley*,
- Cegielski : Conception des systèmes d'exploitation : le cas Linux, *Eyrolles*
- Bovet & Cesati : Understanding the Linux Kernel, *O'Reilly*, (2eédition)
- Robert Love, Linux Kernel Development, *Paperback*

Des cours en lignes

- page de S. Krakowiak à l'URL `http://sardes.inrialpes.fr/~krakowia/`

Informations plus générales

- Le site `http://commentcamarche.net`

Plan du cours :

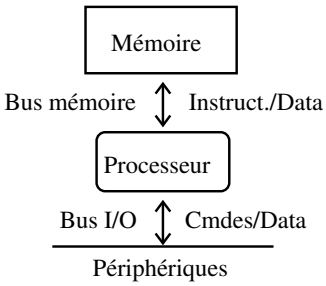
Première partie

Matériel

- Architecture générale des ordinateurs
- Généralités sur les processeurs et l'exécution de code
  - Les processeurs
  - Exécution
- Interruptions
- Hiérarchie mémoire
- Communication avec les périphériques
- Architecture complexes

Description générale

- Processeur : traite les instructions
  → Cherche les instructions en mémoire et les exécute
  ∼ opérations élém. sur données ou adresses mémoire (registres)
- Mémoire : stocke les données et les programmes
- Bus mémoire : gère les accès mémoire
- Bus I/O : gère les communications avec les périphériques
- Périphériques : Entrées/Sorties (stockage, interaction)
- BIOS : système minimal pour *booter*



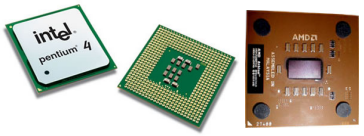
Première partie

Matériel

- Architecture générale des ordinateurs
- Généralités sur les processeurs et l'exécution de code
  - Les processeurs
  - Exécution
- Interruptions
- Hiérarchie mémoire
- Communication avec les périphériques
- Architecture complexes

Les processeurs

- Idées des années 40
- Révolution du transistor et des semi-conducteurs
- Loi de Moore : « La puissance des ordi. double tous les 18 mois. »
- Actuellement : 4 GHz, 0.06 μm + *hyperthreading*
- Dissipation de chaleur devient trop contraignante
  - Multiplication des files d'exécution
- Limite quantique ?



Première partie

Matériel

- Architecture générale des ordinateurs
- Généralités sur les processeurs et l'exécution de code
  - Les processeurs
  - Exécution
- Interruptions
- Hiérarchie mémoire
- Communication avec les périphériques
- Architecture complexes

Registres

Registres utilisateur

- Données
- Adresses (segment, pile, ...)

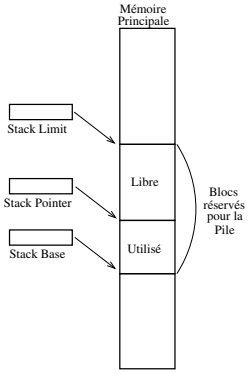
Registres de statut et de contrôle

- *Program Counter* (PC)
- Status arithmétique

Pile et appel de procédure - 1/2

Pile

- Bloc de données réservées en mémoire
- Une base : *Stack Base*
- Une limite : *Stack Limit*
- Pointeur Courant : *Stack Pointer*
- Liste LIFO : Push, Pop
- La Pile grandit des adresses hautes vers les adresses basses



Pile et appel de procédure - 2/2

Appels de procédures

- Conserver l'adresse de retour et la frame précédente
- Paramètres
  → Utilisation des registres, mémoire ou empiler les données locales
- Registres de pile pour la procédure courante
  - Stack Pointer
  - Frame Pointer

Exécution

- Code opération + localisation des données
- Opération de type entier, flottant ou mémoire
- CISC (*Complex Instruction Set Computer*) ou RISC (*Reduced Instruction Set*)
- Contrôle par horloge
- Traitement multiple SISD, SIMD, MISD, MIMD
- Différents modes d'exécution
  - Protégé
  - Réel..

Pipeline

Plus grande vitesse d'exécution des instructions en parallélisant des étapes

- Augmentation de la fréquence par découpage du traitement des instructions
  - Chargement, décodage, exécution
- Pipeline de plus en plus long
- Branches cassent le pipeline
  - Optimisation par prédiction, exploration

Première partie

Matériel

Interruptions

Interruptions

- Programme : générée par *overflow* arithmétique, division par zéro, tentative d'exécuter une instruction illégale, ou encore d'atteinte d'unespace mémoire en dehors de la zone permise pour l'utilisateur
- Timer : générée par un timer dans le processeur dans le but de lui permettre d'effectuer des tâches de façon pseudo-périodique
- I/O : générée par un contrôleur I/O, dans le but de signaler la terminaison d'un évènement ou différentes anomalies/erreurs
- Matérielle : générée par la panne (courant ou erreur de parité mémoire)

Interruptions et exécution

- Requêtes d'un périphérique vers le processeur
  - Un numéro permet de repérer le périphérique
- Suspension du programme en cours
- Déroutement vers un code de traitement
  - *handler* fixé par le système d'exploitation
- Retour au code initial

Exceptions

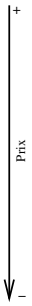
- En cas de problème
  - Erreur arithmétique, accès erroné à la mémoire
- Déroutement vers code de traitement
- Exception spéciale pour appels système
  - Changement de mode d'exécution

Première partie

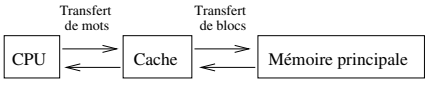
Matériel

Hiérarchie mémoire

- Registres du processeur
- Mémoire centrale
  - Stockage volatile (disparaît au reboot)
  - Adressage en 32 ou 64 bits
  - Relativement lent
- Disque, bande, ...
  - Stockage persistant
  - Très lent



Cache



- Zone de stockage intermédiaire
  - Plus petite mais plus rapide !
  - Répond au problème de la différence de célérité processeur et mémoire
- Conserve zones récemment accédées
- Précharge zones proches qui pourraient être accédées bientôt
- Algorithme de remplacement LRU (*Least recently used*)

Exemple de caches

- Mémoire cache jusqu'à 3 niveau
  - Placée entre processeur et mémoire
  - Très rapide
  - Utilisée de manière transparente
  - Invalidation logicielle possible
- Cache dans les contrôleurs disques
- Espace *swap* géré par OS
- Cache disque géré par OS

Première partie

Matériel

Entrées/Sorties - 1/2

- Mapping de registre des périphériques
- Écriture de commandes
- Traitement dans le périphérique
- Lecture de résultats
- *Programmed I/O*
  - Attente active du bon statut

Entrées/Sorties - 2/2

- *Interrupt-driven I/O* (modèle asynchrone)
  - Recouvrement traitement périphérique par autre chose dans le processeur
  - Périphérique envoie IRQ quand terminé
  - Le *handler* du processeur récupère les informations du périphérique puis retourne à l'exécution normale
- *Direct Memory Access* (DMA)
  - Transfert de données sans le processeur

Ordres de grandeur

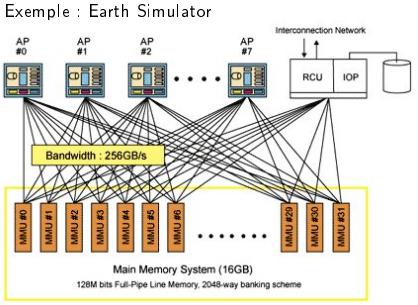
- Processeur : GHz
- Mémoire cache : Mo, ns
- Mémoire : Go, 10 ns
- Bus mémoire : Go/s
- Bus PCI : 500 Mo/s, dizaine de cycles
- Interruption : 10 μs
- Disque : 100 Go, ms, 50 Mo/s
- Réseau : variable

Première partie

Matériel

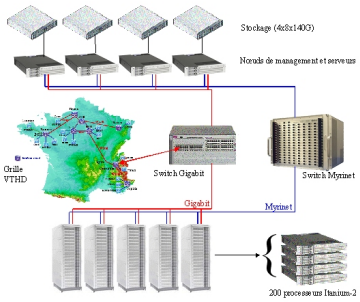
SMP, Numa et systèmes distribués - 1/2

- *Symmetric Multi-Processing*
  - Plusieurs processeurs, une seule mémoire
  - Maintenir cohérence entre différents caches
- *Non-Uniform Memory Access*
  - Plusieurs nœuds avec plusieurs processeurs
  - Un banc mémoire par nœud, accessible par tous
  - Exemple : Earth Simulator



SMP, Numa et systèmes distribués - 2/2

- *Distributed Systems*
  - Un banc mémoire par nœud, le seul accessible
  - Exemple : icluster2



Deuxième partie

Concepts Généraux des systèmes d'exploitation

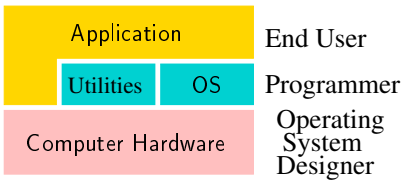
Objectifs des systèmes d'exploitation

- Rendre le système plus facile à utiliser
  - Abstraction des périphériques
- Améliorer l'efficacité du système
  - Mettre les ressources à disposition efficacement
- Facilité d'évolution
- Protection des différents programmes
- Sécurité vis-à-vis des autres utilisateurs

Interface entre utilisateurs et machines - 1/2

- Interface uniforme d'accès aux périphériques
  - Détails techniques cachés
- Exécution de programmes
- Accès contrôlés aux fichiers
  - Montrer le contenu structuré des périphériques de stockage
  - Fournir des fonctions d'édition, de développement, ...

Les couches d'un OS et d'où les voir



Interface entre utilisateurs et machines - 2/2

- Gérer les erreurs proprement
  - Signaler aux programmes les erreurs matérielles
  - Réagir en cas d'erreur logicielle
- Fournir des statistiques d'utilisation et de fonctionnement
  - Permet à l'utilisateur de mieux adapter la configuration du système
  - De mettre au point la configuration du système
  - Sur Système multi-utilsateurs, permet de charger en fonction des ressources consommées

Évolution - 0

Serial Processing : une machine, un utilisateur, un logiciel

- Pas d'OS !
- Utilisateur dialogue avec le hardware
- Code asm lu sur carte perforée
- Exemples : premières machines (1950-1955)

Problèmes

- Ordonnancement
  - Machines réservées sur slot de temps
  - Tâches qui terminent plus tôt ⇒ temps de perdu !
  - Tâches qui terminent plus tard ⇒ résultat perdu !
- Initialisation
  - L'exécution d'une tâche peut demander le chargements de plusieurs autres prog. en mémoire (langage, compilateur, linkeur)
  - Éventuellement montage, démontage de bandes, etc.
  - → Perte de temps, et d'argent

→ Complexité conceptuelle réduite ...
... mais complexité technique accrue (on refait tout à chaque fois) ⇒ cher

Évolution - 1

Simple Bach Systems : une machine, un utilisateur, un logiciel

- 1er Batch OS en 1955
- Un système Batch : l'utilisateur n'a plus accès à la machine
- Une pile séquentielle de programmes
- Chaque prog. « rend la main » quand il quitte
- Tentative de ne pas avoir d'*idle time*

Remarques

- Batch OS doit assurer la protection mémoire
- Prévenir qu'une tâche monopolise le système (timer)
- Instruction privilégées (seul OS peut exécuter, comme opérations I/O)

→ Notions de *user mode* et *kernel mode* pour accéder à zone mémoire protégée par exemple

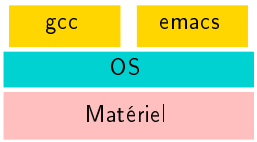
→ L'OS prend de la place mémoire et du temps machine

→ Améliore l'utilisation de la machine

Évolution - 2

Multiprogrammed Batch Systems : une machine, plusieurs logiciels

- Problème fondamental de la précédente solution :
  Si le programme est bloqué (disque, réseau), si le programme fait bcp d'I/O (proc + rapide !), la machine est inutilisée ⇒ idle time !
- Solution : **plusieurs processus**. Si un est bloqué, on exécute un autre
  ⇒ Demande espace libre en mémoire en plus de OS + tâches actuelles



Remarques

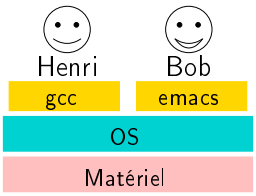
- Fonctionnalités supplémentaires : I/O interrupts, DMA
- Gestion de mémoire accrue : plusieurs programmes
- Ordonnancement

→ Améliore encore l'utilisation de la machine au prix d'une sophistication de l'OS

Évolution - 3

Time Sharing : une machine, plusieurs logiciels, plusieurs utilisateurs

- La solution précédente est chère (en 1970) : il faut une machine par utilisateur, et on a besoin de réactivité
- Solution : **partage de la machine entre utilisateurs**
  (les utilisateurs tapent moins vite que l'ordinateur compile)



- Mais que se passe-t-il si les utilisateurs sont :
  - Trop gourmands ou trop nombreux
  - Mal intentionnés
- Le système doit protéger et gérer les ressources

Processus

- File d'exécution
  - État des registres
- Contexte mémoire propre
  - Exécution indépendante des autres processus
- Données privées au système d'exploitation
  - Pour manipuler le processus
- Exécution concurrente pour maximiser l'utilisation des ressources matérielles

Gestion de la mémoire

- Isolation des processus
  - Pas de collision entre leurs mémoires
    → données et instructions propres
- Allocation et gestion transparente
  - Programmes chargés dynamiquement en mémoire
  - Pas de contraintes pour le programmeur
- Programmation modulaire
- Partage de mémoire avec protection
- Stockage en mémoire persistante

Mémoire virtuelle

- Virtualisation des adresses mémoire manipulées par les processus
  → illusion d'être le seul processus
- Découpage en pages (et/ou segments)
- Stockage par défaut sur tout le disque
  - Rappel en mémoire des pages nécessaires
- Partage de pages aisé
- Support matériel

Protection des données et sécurité

- Disponibilité
- Confidentialité : contrôle des autorisations
  - Accès aux données critiques du système
  - Accès et modification des données des utilisateurs
- Authentification des utilisateurs
- Survie à un problème technique

Ordonnancement et gestion des ressources

- Accès équitable aux ressources
  - En particulier pour les travaux de même « demande »
- Différentiation de classes de travaux
- Contrôle dynamique
- Efficacité
  - Optimisation de l'utilisation du matériel
  - du temps de réponse des tâches
  - et de sa réactivité

Ordonnancement

- *Round Robin*
- Priorités
- Priorités dynamiques
- Contraintes dates d'échéance : *deadline*
- Contraintes matérielles : *elevator* et *batch*
- Anticipation

Deuxième partie

Concepts Généraux des systèmes d'exploitation

Généralités

- Noyau
  - Cœur
  - Pilotes de périphériques
  - Interface utilisateur
- Bibliothèque utilisateur de bas niveau
  - Appels système
- Gestion de la mémoire au cœur du système
  - Processus
  - Systèmes de fichiers
  - Réseau
- Systèmes de fichiers et stockage
- Ordonnancement
- Communication entre processus
- Réseau

Deuxième partie

Concepts Généraux des systèmes d'exploitation

Noyaux modulaires

- Chargement/Déchargement dynamique de code
  - Pilote de périphériques
  - Fonctionnalités spécifiques
- Réduction de la taille du noyau initial
- Possibilité d'évolution dynamique : pas de redémarrage !

Micronoyaux : idées - 1/3

- Noyaux monolithiques trop gros
  - Pas organisés, même avec des couches
  - Sécurité difficile à assurer car trop d'interactions
  - Difficiles à maintenir et à faire évoluer
- Micronoyaux
  - Embarque uniquement le strict nécessaire
  - Tout le reste dans des processus serveurs dédiés

Micronoyaux : design - 2/3

- Un serveur pour chaque tâche
  - Processus, gestion mémoire, ordonnancement, réseau, systèmes de fichiers
  - Passage de messages entre applications et serveurs, validés par le micronoyau
- Interfaces simples et uniformes
- Extensible, flexible, portable
- Fiable

Micronoyaux : performance - 3/3

- Passage de messages plus lent qu'appel direct de procédure au noyau ?
- Trop d'interactions critiques entre les différentes parties du système d'exploitation ?
  - Par exemple : mémoire et systèmes de fichiers
- Difficile de comparer avec noyaux monolithiques
  - Pas de vrais OS fonctionnel fondé sur un micronoyau

Deuxième partie

Concepts Généraux des systèmes d'exploitation

Multithreading

- Exécution simultanée de plusieurs *threads* dans le même processus
  - Travailler pendant qu'un *thread* bloque sur une I/O
- Ressources d'exécution propres
  - Contexte processeur (PC et pointeur de pile)
  - Son propre espace mémoire pour la pile (gestion appel de sous-routine)
- Espace mémoire partagé

SMP et NUMA

- Nouvelles contraintes sur l'ordonnancement
- Plusieurs processeurs
  - ...qui peuvent effectuer les mêmes opérations
  - Accès concurrents (mémoire, I/O...)
  - Affinité pour un processeur (cache, TLB, ...)
  - Affinité des threads d'un même processus
- Contraintes NUMA
  - Affinité pour certaines zones mémoire

Entrées/Sorties asynchrones

- Matériel naturellement asynchrone
  - Soumission de commandes
  - Attente active de statut ou passive d'IRQ
- Remontée de l'asynchronisme jusqu'à l'application
  - Pas besoin de threads pour recouvrir les I/O

Systèmes distribués

- Clusters et grappes d'ordinateurs
- Systèmes à image unique
- Coopération de différents nœuds
  - Avec ou sans maître
- Mémoire partagée distribuée
  - Ping-Pong de pages
- Problème de synchronisation

Deuxième partie

Concepts Généraux des systèmes d'exploitation

Windows - 1/2

- Un seul utilisateurs, plusieurs tâches
- Pseudo-micronoyau
  - Beaucoup d'autres fonctions du système en mode noyau
    → Performance en évitant les chgs de modes, mémoire add...
- Très modulaire
  - *Executive* : base de l'OS, ensemble de *gestionnaires*
    - *I/O, Cache, Object, PnP, Power, Security, VM, Process, ...*
  - Noyau qui contient l'ordo, gestion interrupt. et except...
    → N'est pas en thread, donc pas préemptible, ni « pageable »
  - HAL : Couche d'abstraction des périphériques, des controllers
    - DMA, Interrupt. et except. controller, system timer..
  - Pilotes de périphériques
  - Fenêtrage et graphisme

Windows - 2/2

- *Local Procedure Calls*
  - Échange de messages entre applications et managers
  - Modèle Client-Server
- Support SMP, threads et communications entre processus
- Design orienté objet
  - Polymorphisme

Deuxième partie

Concepts Généraux des systèmes d'exploitation

Unix - 1/3

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

Unix - 2/3

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

Unix - 3/3

- *System V Release 4*
  - Académique et commercial (AT&T et Sun)
- Solaris
  - Distribution commerciale (Sun) fondée sur SVR4
  - Maintenant sous GPL !
- *Berkeley Software Distribution*
  - Très répandu dans le mode académique
  - Base de MacOS X (bien modifié depuis)
- Beaucoup d'autres...

Deuxième partie

Concepts Généraux des systèmes d'exploitation

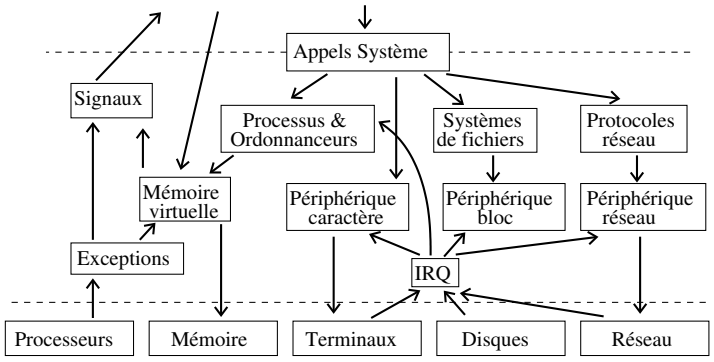
Linux - 1/4

- Apparu en 1991
- Fondé sur *Minix* de Tanenbaum
  - Pas trop cher pour usage personnel
- Libre ! + appuis de FSF (et des progs...)
- Développement collaboratif via Internet
- Supporte de nombreuses architectures
- Supporte de nombreux périphériques matériel

Linux - 2/4

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

Linux - 3/4



Linux - 4/4

- `module_init`, `module_exit`
- `module_param`
- `insmod`, `rmmod`, `modprobe`
- `EXPORT_SYMBOL`

Troisième partie

Processus

Buts

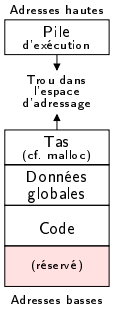
- Ressources disponibles pour de multiples applications
- Toutes les applications semblent progresser simulanément
  - Les processeurs physiques les exécutent en alternance
- Le processeur et les périphériques sont utilisés efficacement

Définitions du *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
  - Un ensemble de ressources système
- 2 éléments : le code du programme, le jeu de données

Processus UNIX

- Processus = exécution d'un programme
  - Commande (du langage de commande)
  - Application
- Un processus comprend :
  - Une mémoire qui lui est propre (mémoire virtuelle)
  - Contexte d'exécution (pile, registres du processeur)
- Les processus sont identifiés par leur pid
  - Commande ps : liste des processus
  - Commande top : montre l'activitÃ© du processeur
  - Primitive getpid() : renvoie le pid du processus courant



Troisième partie

Processus

Caractérisation du processus

- À l'initialisation
  - Code du programme
  - Ensemble de données
- Durant l'exécution
  - Un identifiant
  - Un état (running...)
  - Une priorité
  - PC
  - Pointeurs mémoire
  - Contexte de donnée
  - I/O status information
  - Accounting information

  → Bloc de contrôle géré par l'OS

État des processus

- *Running*
- *Ready*
- *Blocked*
- *New*
- *Exit*
- *Suspend*

Ressources système

- Tables mémoire
- Tables de périphérique
- Tables des fichiers
- Tables des processus

Ressources des processus

- Localisation mémoire
  - Données utilisateur
  - Programme
  - Pile système
  - Bloc de contrôle
- Attributs
  - Identifiant
  - Informations d'état du processus

    → PC, code condition, status info, stack pointers
  - Données de contrôle

Identification d'un processus

- Identifiants généralement numériques
  - Processus cible
  - Parent
  - Utilisateur

Données de contrôle d'un processus

- État et ordonnancement
- Gestion de la mémoire
- Ressources attribuées et/ou utilisées
- Communications inter-processus
- Privilèges
- Données de structure

Troisième partie

Processus

État du processeur

- Registres utilisateur
- Registres de statut et contrôle
- Pointeurs de pile

Modes d'exécution - 1/2

- Plusieurs niveaux de privilèges dans le processeur
  - Accès aux registres de contrôle (comme PSW)
  - Instructions d'entrées/sorties de bas niveau
  - Gestion mémoire
    → Tables de pages
  - Accès à certaines zones mémoire
    → Limitées par les structures pointées par certains registres

Modes d'exécution - 2/2

- Le noyau peut tout faire (mode *réel*)
- L'utilisateur est limité (mode *protégé*)
  - Pas d'accès à la mémoire noyau
    → Non visible dans les tables de pages
  - Pas d'accès bas niveau au matériel
- Autre mode généralement pas/peu utilisés
- Interruption/exception provoque passage en mode privilégié

Troisième partie

Processus

Création d'un processus - 1/2

- Obtenir un identifiant disponible
- Allouer de l'espace pour le processus
- Initialiser le bloc de contrôle
- Initialiser les structures de données

Création d'un processus - 2/2

- Duplication (**fork**)
  - Partages des structures principales du processus courant
    → Espace d'adressage, fichiers, signaux, ...
  - Création d'un copie des structures uniques
- 2 processus identiques pour l'utilisateur
  - Valeur de retour différente

Transformation d'un processus

- Exécution (**exec**)
  - Remplacement du programme courant
  - Lancement d'un nouveau programme
  - Espace d'adressage complètement réinitialisé

Copy-on-write

- Partage des données le plus longtemps possible
- Création d'une copie à la première modification par un des processus
- Duplication rapide (**vfork**)
  - Copie du strict minimum
  - Père bloqué tant que le fils ne s'est pas transformé

Terminaison d'un processus

- Relâchement des ressources
- Passage d'un code de retour au père
- Père peut attendre terminaison d'un fils (**wait**)
- Que faire des orphelins ?

Troisième partie

Processus

Changement de contexte - 1/3

- Nécessaire si
  - Blocage sur une I/O
  - Interruption (timer)
  - Erreur d'accès mémoire
  - → avec une `trap`, OS sait si fatal
  - Appel superviseur par un prog. user
    → pour l'ouverture d'un fichier par exple
- Recommandé pour
  - Équitabilité
  - Réactivité

Changement de contexte - 2/3

- Vérifications régulières que le processus courant n'abuse pas du processeur
  - Interruptions d'horloge
  - Retour de traitement d'interruption
  - Retour d'appel système

Changement de contexte - 3/3

Fonctionnement

- différent de Mode Switching
  - À chaque cycle d'instruction, test si signal en attente
    → Oui ⇒ @handler in PC + switch user mode en **kernelmode**
  - Le context du processus est sauvé dans control block
- Déroulement
  - Sauvegarde du contexte du processeur (dont PC et registres)
  - Mise à jour du bloc de contrôle du processus 1
    - Changement de son état en particulier
  - Mettre le bloc de contrôle dans la queue appropriée (Ready, blocked..)
  - Choisir un nouveau processus 2
  - Mise à jour du bloc de contrôle de 2
    → Changement de son état en Running entre autre
  - Mise à jour des structures d'accès à la mémoire
  - Restauration du contexte du processeur avant le switch du 2

Troisième partie

Processus

Exécution du noyau

OS est logiciel

- Doit avoir le contrôle de tout le reste
- Dépend du processeur pour lui redonner ce contrôle

Plusieurs approches

- Approche traditionnelle : fonctions kernel extérieures aux processus
  - Passage des processus en mode *réel* (noyau)
    - Traitement de leurs appels système
    - Traitement des interruptions
  - Changement de contexte
    - Changement des privilèges
    - Utilisation d'une pile spéciale
    - Sauvegarde et restauration des registres

Micronoyaux

- Appels système minimaux
  - Passage de messages
- Traitement réel du système d'exploitation en dehors du noyau
  - Dans les processus des serveurs dédiés

Threads noyau

- Noyaux monolithiques ont besoin de traitements supplémentaires
  - Appels système imprévisibles et pas adaptés
  - *Handlers* d'interruption trop limités
- Utilisation des *threads* dédiés (**kernel thread**)
  - Toujours en mode noyau
  - Traitent les travaux périodiques ou programmés
    → Traitement lourd post-interruption

Troisième partie

Processus

Processus et *threads*

2 concepts indépendants

- Proprio de la ressource : processus ou tâche
  - processus inclue virtual address pour le process image (prog, data, stack, attributs définis dans bloc de contrôle)
- L'exécution : le thread
  - la trace le long de un ou plusieurs progs en particulier

Pourquoi ?

- Plusieurs processus partageant des ressources
  - Utilisation mémoire partagée
  - Initialisation complexe
  - Pas forcément suffisant
- Plusieurs tâches dans un même processus
  - Recouvrement des blocages sur I/O

Définitions

Processus

- Unité d'allocation ressource et de protection
  → Adresse virtuelle pour l'image process
  → Accès protégés aux processeurs, les autres proces (comm)
  mais aussi fichiers, I/O...

Thread

- Un état
- Un contexte sauvegardé quand `NOT RUNNING`
  → Une sorte de PC indép dans le processus
- Piles et contextes d'exécution distincts
- Stockage statique par thread pour variable
- Accès à la mém. et ress. partagées avec autres threads du même processus
  - Espace d'adressage
  - Fichiers ouverts
  - Signaux
  - Identifiants de processus
- Clônage du thread courant (**clone**)

Threads en espace utilisateur

- Ordonnancement des threads par l'application
  - Changement de contexte, création, ... rapides
- Noyau n'a aucune connaissance des threads
  - Un seul processus noyau, une seule file d'exécution
- Besoin d'assistance du système d'exploitation
  - Blocage de tous les threads si un thread bloque
    → *Scheduler Activation*
  - Temps processeur pas équitable

Modèle 1-on-1

- Un processus système par thread
  - Le noyau maintient les infos contexte du process + infos pour thread du process
- Opérations quasi-normales
  - Création, terminaison, changement de contexte
  - Même principe que les processus
    → Sauf l'espace d'adressage
  - Lent
- **pthread** sur Linux 2.4

Modèle M-on-N

- N processus système pour M threads utilisateur
- Avantages des deux stratégies
  - Ordonnancement rapide en moyenne
  - Assistance du noyau pour les cas difficiles
- NPTL sur Linux 2.6
- Migration

LightWeight Processes

- 1 ou plusieurs LWP par processus
- 1 thread noyau par LWP
- Threads utilisateurs liés à 1 ou plusieurs LWP
- Permet aux applications de contrôler leur degré de parallélisme
- **thr** sur Solaris

Troisième partie

Processus

Architectures parallèles

- SIMD (*Single Instruction Multiple Data Stream*)
- MIMD (*Multiple Instruction Multiple Data Stream*)
  - Mémoire partagée (fortement couplée)
    - Modèle maître/esclave
    - SMP (*Symmetric MultiProcessing*)
      → Homogène ou non
  - Mémoire distribuée (faiblement couplée)
    → grappes et *clusters*

SMP et NUMA, migration

- Exécution simultanée de plusieurs tâches
  - Plusieurs processus
  - Plusieurs threads d'un même processus
- Contraintes matérielles sur leur placement
  - Éviter le partage de structures
    → Défauts de lignes de cache
- Migration des tâches entre nœuds
  - Avec leurs données

Troisième partie

Processus

Hiérarchie

- **init**
  - Threads noyaux
  - Services
    → *Dæmons*
  - Concoles texte
  - Gestionnaire de sessions graphiques
    - Applications graphiques
      → Terminaux

Routines principales (2.4 et 2.6)

- **do_fork(flag,stack,regs,...)**
- **kernel_thread(func,arg,flags)**
- **do_exit(code)**
- **do_execve(file,args,envs,...)**
- **sys_wait4(pid,&stat,options,...)**
  - **wait_task_zombie(pid,&stat,...)**

Structures noyau

- File d'exécution (**struct task_struct**)
- Espace d'adressage (**struct mm_struct**)
  - **struct vm_area_struct** et leur méthodes
  - `/proc/<pid>/maps`
- PID et table de hashage

Descripteur

- **struct task_struct**
  - 4 ou 8 ko
  - Ressources attribuées
  - PID et TGID
  - Liens vers père et fils
    → vers `init` pour les orphelins
  - Pile noyau

État des processus

- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE

Ordonnancement et évènements

- **schedule**
  - **schedule_timeout**
- **set_current_state**
- **wait_queue_t** et **wait_head_t**
  - **add/remove_wait_queue**
- **struct completion**
  - **wait_for_completion**
  - **complete/complete_and_exit**

Quatrième partie

Concurrence

- Principes de la concurrence
- Détails de Linux

Principe de l'exclusion mutuelle

Exemple : deux banques modifient un compte en mêeme temps

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 1000
3. update_account (1867A, nouveau)
```

Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 1000
3. update_account(1867A, neue)
```

- variables locales + exéecutions parallèles entremêelées ⇒ différents réesultats :
  - N.1 ; N.2 ; N.3 ; K.1 ; K.2 ; K.3 →
  - N.1 ; K.1 ; N.2 ; K.2 ; N.3 ; K.3 →
  - K.1 ; N.1 ; K.2 ; N.2 ; K.3 ; N.3 →

C'est une **condition de compétition** (*race condition*)

- Solution : opérations atomiques ; pas d'exécutions entremêlées
- On dit que cette opération est une section critique
- Elle doit s'exécuter en exclusion mutuelle
  → la protéger par la prise d'un verrou

Coopération

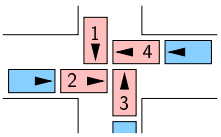
- Coopération par partage
  - Pas de connaissance *explicite* des autres
  - Modifications doivent conserver l'intégrité
    → Sections critiques
  - Accès en lecture non problématiques
- Coopération par communication
  - Connaissance explicite des autres
  - Synchronisation via les communications

Compétition

- Accès concurrents à une même ressource
- Aucune connaissance des autres
- Support nécessaire dans le système d'exploitation
  - Attribution de ressources à un acteur unique
  - Les autres acteurs attendent qu'il les relâche
- Nécessité de rendre la ressource dans un état intègre

Interblocage (*deadlock*)

Exemple : carrefour lyonnais un vendredi à 18h



Conditions d'apparition

- Exclusion mutuelle
- Un acteur possède une ressource et en attend une autre
- Attente circulaire
- Pas de préemption

Comment prévenir l'interblocage ?

Solution 1 : réservation globale

- Demandes en bloc de toutes les ressources nécessaires
- Inconvénient : réduit les possibilités de parallélisme
- Analogie du carrefour : mettre des feux tricolores (et les respecter)

Solution 2 : requêtes ordonnées

- Tous les processus demandent les ressources dans le même ordre
- Interblocage alors impossible
- Analogie du carrefour : construire un rond-point ici, avec verroux exclusifs

```
v-excl (f1)
v-excl (f2)
accès à  f1 et f2
dev (f2)
dev (f1)
```

```
v-excl (f1)
v-excl (f2)
accès à  f1 et f2
dev (f2)
dev (f1)
```

Solution 3 : modification de l'algorithme

- Modifier code utilisateur pour rendre impossible l'interblocage
- Analogie du carrefour : on ne s'engage que si on peut sortir du carrefour

Famine (*starvation*)

- Exclusion mutuelle
  - Acteurs en attente d'une ressource
- Privation d'un acteur au profit des autres
  - Respecter l'ordre d'arrivée des acteurs en attente

Support matériel

- Désactivation des interruptions
- Instructions spéciales pour sérialiser les accès mémoire
- Instructions atomiques
  - `test_and_set`
  - `exchange`
- Attente active
- Pas de protections contre interblocage et famine

Solution logicielle

Algo de Dijkstra en 1965

- Pour une machine avec n proc.
- Shared memory
- Avant les sémaphores
- Attente active

→ Présentation algo au tableau

Sémaphores

- Compteur initialisé strict. positif
- `SemWait` décrémente le compteur et bloque s'il devient négatif
- `SemSignal` incrémente le compteur et réveille les `SemWait` bloqués
- Exclusion mutuelle si initialisé à 1
- Attente d'évènement si initialisé à 0

Moniteurs

- Objet dont des données locales sont accessibles uniquement par ses méthodes
- Entrées dans le moniteur par une méthode
- Un seul acteur à la fois
- `CondWait` pour suspendre l'acteur dans le moniteur en le relâchant, jusqu'à une condition
- `CondSignal` pour reprendre l'exécution d'un acteur dans le moniteur

Schémas de synchronisation

Situations usuelles se retrouvant lors de coopérations inter-processus

- Exclusion mutuelle : ressource accessible par une seule entitée à la fois
  - Compte bancaire ; Carte son
- Problème de cohorte : ressource partagée par au plus N utilisateurs
  - Un parking souterrain peut accueillir 500 voitures (pas une de plus)
  - Un serveur doom peut accueillir 2000 joueurs
- Rendez-vous : des processus collaborant doivent s'attendre mutuellement
  - Roméo et Juliette ne peuvent se prendre la main que s'ils se rencontrent
  - Le GIGN doit entrer en même temps par le toit, la porte et la fenêtre
  - Processus devant échanger des informations entre les étapes de l'algorithme
- Producteurs/Consommateurs : un processus doit attendre la fin d'un autre
  - Une Formule 1 ne repart que quand tous les mécaniciens ont le bras levé
  - Réception de données sur le réseau *puis* traitement
- Lecteurs/Rédacteurs : notion d'accès exclusif entre *catégories* d'utilisateurs
  - Sur une section de voie unique, tous les trains doivent rouler dans le même sens
  - Un fichier pouvant être lu par plusieurs, si personne ne le modifie
  - Tâches de maintenance (défragmentation) quand pas de tâches interactives

Comment résoudre ces problèmes avec les sémaphores ?

Une instance du problème des lecteurs/rédacteurs

- Plusieurs lecteurs simultanément
- Un seul rédacteur
- Éviter les famines pour les écrivains
- Éviter les ping-pong entre caches de différents processeurs

Passage de messages

- Synchronisation naturelle car réception après envoi
- Adressage du destinataire et/ou de l'émetteur
  - `Broadcast`, `Reduce`
- Ordonnancement FIFO ou avec priorités
- Exclusion mutuelle par l'envoi d'un message à l'unique acteur autorisé à entrer en section critique

Mécanismes de concurrence dans Unix

- Communication entre processus
  - Tubes
  - Messages
  - Mémoire partagée
- Déclenchement selon actions des autres processus
  - Sémaphores
  - Signaux

Quatrième partie

Concurrence

Concurrence et synchronisation dans le noyau Linux

- Communication et synchronisation entre processus utilisateur
  - Mécanismes IPC (*InterProcess Communication*)
  - Mutex et sémaphores pour la synchronisation des pthreads
- Mémoire explicitement partagée dans le noyau
- Beaucoup de stratégies pour la synchronisation dans le noyau

Synchronisation dans le noyau Linux

- Performance (notamment si contention)
- Support SMP (avec passage à l'échelle)
- Accès plutôt en lecture ou plutôt en écriture
- Minimisation des contraintes sur les différentes parties du système
  - Ne pas trop désactiver les interruptions
  - Ne pas trop faire attendre les tâches déferrées

Opérations atomiques

- `atomic_t` pour opérations entières
  - `void atomic_add(int i, atomic_t *v)` ajoute i à v
  - `atomic_inc_and_test(int i, atomic_t *v)` teste en + si result 0
- `atomic_t` pour opérations sur bits
  - `void set_bit(int nr, void* addr)` set bit nr in bitmap pointed by addr
- `cmpxchg`
- Implantation en assembleur
  - Utilisation directe des instructions atomiques quand elles sont disponibles
  - Instructions `CISC` suffisent à peu près
  - Émulation via plusieurs instructions sur les `RISC`

Spinlocks - 1/2

- Attente active (*spin*)
- Utilisation d'un `spinlock_t`
  - `void spin_lock_init(spinlock_t *lock)`
  - `void spin_lock_irq(spinlock_t *lock)`
  - Exemple

```
spin_lock(& lock)
/* Section Critique */
spin_unlock(& lock)
```

- `rw_lock_t` pour autoriser plusieurs lecteurs
- *Big Reader Lock* (`br_lock_t`) pour éviter ping-pong antre les tâches
  - Partage des parties nécessaires à l'accès en lecture

Spinlocks - 2/2

- Ne pas dormir en tenant en lock
  - Désactivation de la préemption
  - Pas de fonction bloquante

Sémaphores

- Sommeil jusqu'à disponibilité d'une ressource
- `struct semaphore`
  - `void sema_init(struct semaphore *sem, int count)`
  - `up`, `down`
  - `down_interruptible`
  - `down_trylock`
- `struct rw_semaphore` pour autoriser plusieurs lecteurs

Conditions

- Sommeil dans une queue par `wait_event`
  - Variantes interruptibles par des signaux, avec un timeout, ...
  - `sleep_on` risqué car ne vérifie pas la condition avant de dormir
- Réveil d'une queue par `wake_up`
  - Variantes pour réveiller plusieurs tâches

Big Kernel Lock

- Verrou global (spinlock)
- De moins en moins utilisé
  - Restreint fortement la concurrence
- `lock_kernel`, `unlock_kernel`

Réactivité et préemption

- Ne pas bloquer les autres en gardant un verrou
  - Ne pas perdre la main en tenant un verrou
- Les spinlocks désactivent la préemption
- Désactivation des interruptions
  - `local_irq_save/restore`
  - `spin_lock_irqsave`
  - `disable_irq`

Cinquième partie

Mémoire

Gestion de la mémoire

- Mémoire système et mémoire utilisateur
- Division de la mémoire utilisateur entre les différents programmes
- Maximiser le nombre de processus pour maximiser l'utilisation du matériel

Cinquième partie

Mémoire

Relocalisation - 1/2

- Pas de contraintes sur le programmeur
  - Localisation quelconque du programme en mémoire
- Ne pas dépendre des autres programmes résidants en mémoire
- Pouvoir être évincé sur le disque puis ramené en mémoire à un endroit différent

Relocalisation - 2/2

- Ajuster les références internes au programme
  - Données
  - Branchement
- Ajuster les références dans le système
  - Facile si le système gère lui-même la relocalisation

Protection

- Pas d'accès aux données des autres processus ou du système
  - Intentionnel ou par erreur
- Détection à l'avance difficile
  - Relocalisation imprévisible
  - Langages autorisent calcul dynamique d'adresses
- Détection dynamique coûteuse logiciellement
  - Support matériel

Partage

- Protection configurable
  - Partage de zones mémoire possible
- Pas de duplication de code
- Partage explicite de données
- Support logiciel trop coûteux
  - Support matériel

Organisation logique

- Mémoire linéaire
- Programmes modulaires
  - Protections différentes dans différents modules
  - Écriture et compilation indépendantes des différents modules
    → Références ajustées dynamiquement
  - Partage de modules parmi différents programmes
- Segmentation

Organisation physique

- Structure hiérarchique de la mémoire
  - Mémoire principale assez rapide, mais limitée et volatile
  - Mémoire secondaire lente mais grande et persistante
- Ne pas contraindre le programmeur
  - Quels modules seront en mémoire principale ?
  - Si plein d'autres programmes ?
- Responsabilité du système d'exploitation

Cinquième partie

Mémoire

Partitionnement fixe - 1/3

- Mémoire physique divisée statiquement en partitions fixes
- Programme chargé dans une zone suffisamment grande
- Facile à implanter, peu coûteux
- Gaspillage mémoire (*fragmentation mémoire*)
- Nombre limité de programmes

Partitionnement mémoire : taille des partitions - 2/

- Tailles identiques
  - Taille du programme ou nombre très limité de programmes
  - Gros gaspillage pour les petits programmes
- Tailles croissantes
  - Moins de fragmentation interne
  - Quand même relativement limité

Partitionnement mémoire : algo de placement - 3/3

- Placement dans la partition la plus petite qui peut contenir le programme
- Utiliser une plus grande si aucune petite partition disponible
- Suspendre les processus su aucune partition disponible

Partitionnement dynamique - 1/2

- Création de partition de taille optimale à la volée
- Fragmentation externe augmente
- Compactage pour réduire fragmentation
  - Gaspillage de temps processeur

Partitionnement dynamique : algo de placement - 2

- Premier espace disponible (*fisrt-fit*)
  - Simple et rapide
  - Un peu de fragmentation
- Meilleur espace disponible (*best-fit*)
  - Coûteux
  - Beaucoup de petites fragmentations
- Nouveau placement quand programme revient du disque vers la mémoire principale

Buddy System

- Avantages partitionnements fixes et dynamiques
  - Pas trop coûteux
  - Peu de fragmentation
- Listes de blocs libres de taille 1 à 2^n
- Allocation d'une partition dans le plus petit bloc adapté
- La partie gaspillée est convertie en petits blocs
- Fusion des petits blocs à la désallocation

Pagination

- Découpage en petits *cadres* physiques (*frames*) de taille fixe
  - Chaque *page* du processus va dans un *cadre*
- Pas de fragmentation externe
- Fragmentation interne faible
- Table des pages pour chaque processus
  - Associations page-cadre
- Adresses logiques transparentes page+offset

Segmentation

- Similaire à partitionnement dynamique
- Plusieurs segments par processus
  - Fragmentation externe moindre
- Pagination non transparente
  - Utilisé pour séparer zones de types différents
    → Code, données, ...
- Table des segments, avec leurs longueurs

Cinquième partie

Mémoire

Chargement des programmes

- Assemblage des modules et bibliothèques
- Calcul des différentes adresses mémoire
  - Chargement absolu
    → Par le programmeur, à la compilation ou à l'assemblage
  - Chargement relocalisable
    → Adresses relatives au début du module
  - Chargement dynamique
    → Adresses relatives calculées au moment de l'exécution

Édition de liens

- Utilisation de symboles pour référencer les adresses
- Adresses relatives au début du module
- Références non résolues provoquent chargement de modules supplémentaires
  - Mise à jour des modules sans changer les applications
  - Partage de code facile

Détails de Linux

- Le compilateur précise l'interpréteur
- Le noyau lancer l'interprêteur du programme (`ls.so`)
- Emplcement des bibliothèques : `ld.so.conf` et `ldconfig`
- Emplacement des bibliothèques alternatives : `LD_LIBRARY_PATH`
- Bibliothèque de surcharge : `LD_PRELOAD`

Cinquième partie

Mémoire

Intérêts

- Pagination et segmentation avantageux
  - Usage combiné dans les systèmes modernes
  - Ne pas les imposer au programmeur
- Mémoire virtuelle complexe
  - Relation entre support matériel et logiciel
  - Besoins logiciels très grands
    → Sécurité et protection
    → Flexibilité
    → Efficacité

Pagination et segmentation

- Les références mémoire d'un processus sont logiques
  - Traduire à la volée en adresses physiques
  - Support de la relocation après évinçage
- Les processus peuvent être découpés en parties non contiguës en mémoire physique
  - Pages ou segments

Résidence mémoire

- Un processus peut s'exécuter sans que toutes ses parties soient en mémoire
- Le système charge les parties nécessaires au fur et à mesure
  - Exception dans le processeur en cas de défaut de page
  - I/O pour charger les pages en mémoire
    → Processus en attente, un autre peut s'exécuter pdt I/O
    → Processus en Ready State qd I/O terminée
- Localité mémoire pour le code et les données
  - Bénéfices attirants
  - Éviter le *Trashing*
  - Faisabilité de la solution théorique ?

Autres avantages

- Les processus utilisent moins de mémoire physique
- Plus de processus peuvent être chargés simultanément
  - Meilleure utilisation du processeur
- Un processus peut utiliser plus de mémoire virtuelle qu'il n'y a de mémoire physique
  - Pas de contraintes sur le programmeur

Cinquième partie

Mémoire

Table des pages - 1/2

- Ensemble d'associations page + cadre physique
  - *Page Table Entry*
  - Ensemble de bits décrivant les propriétés
    → *Present*, *Modified*, ...
- Début de la table dans un registre spécial
- Table stockée en mémoire
  - Virtuelle pour ne pas monopoliser trop de physique

Table des pages - 2/2

- Traduction par parcours de différents niveaux (entre 1 et 4)
- Exemple avec adresse virtuelle 32 bits
  - 10 bits pour 1erniveau de la table
  - 10 bits pour 2^{e}niveau
  - 12 bits d'offset

Table des pages inversée

- Table des pages normale trop grande
- Une entrée par cadre physique
  - Identifiant du processus propriétaire
  - Page du processus
  - Bits de contrôle
  - Données de chaînage
    → Pour les pages partagées

Translation Lookaside Buffer

- Une référence mém. virtuelle peut causer deux accès mém. physique
  - Au moins un pour lire l'entrée de la table de pages
  - Un pour lire la donnée
- Cache matériel pour l'éviter
- Mapping associatif pour recherche rapide
- TLB dans la *Memory Management Unit*
  - MMU dans le processeur
  - Placés entre caches de niveau 1 et 2

Taille des pages - 1/2

- Si trop petit
  - Trop de pages, tables trop grandes
  - Moins de localité
    → Plus de défauts de page
  - TLB moins efficace
    → Taille matérielle limitée
- Si trop grand
  - Trop de fragmentation
  - Moins d'impact de la localité
    → Plus de défaut de page

Taille des pages - 2/2

- Taille idéale dépend
  - Mémoire physique disponible
  - Taille des programmes
  - Techniques modernes de programmation
    → Orienté objet, multithreadé, ...
- Tailles multiples
  - Grandes zones contiguës peuvent être mappées avec petit nb de grandes pages
  - Supporté par beaucoup d'architectures
    → Peu de support dans les systèmes d'exploitation

Segmentation

- Partitions de taille variable et dynamique
- Table des segments
  - Adresse de base dans un registre
  - Recherche associative tenant compte de la longueur
  - Un seul niveau
- Bits similaires à ceux des tables de pages
- Segments visibles pour le programmeur

Combiner Paging et Segmentation

- Ensemble de segments découpés en pages
- Une table des segments et plusieurs tables des pages
- Adresse logique segment + page + offset
- Support par la plupart des architectures

Protection et partage

- Protection par les segments
  - Accès invalide si en dehors
- Partage explicite par partage de segments entre processus
- Partage de pages invisible pour le programmeur
- Utilisation d'anneaux privilégiés pour définir les autorisations d'accès à certaines zones

Cinquième partie

Mémoire

Clés du design

- La gestion de la mémoire dans un système d'exploitation dépend de
  - L'utilisation de la mémoire virtuelle
    → Support matériel nécessaire
  - L'utilisation de la pagination et/ou de la segmentation
    → Support matériel nécessaire
  - Les différents algorithmes utilisés
    → Prendre des décisions en faveur de l'efficacité du système...
    (très difficile)

Politique de chargement

- Quand charger une page en mémoire ?
- À la demande (*demand paging*)
  - Beaucoup de défauts de pages au début
  - Localité réduit les défauts de pages par la suite
- À l'avance (*pre-paging*)
  - Exploitation du matériel (disque)
    → Lecture de plusieurs blocs contigus plus efficace
  - Efficace au début du programme
  - Peut-être nuisible par la suite

Politique de placement

- Important pour la segmentation
- Inutile si pagination
- Critique sur NUMA
  - Placer les pages à côté des processeurs qui en ont besoin
  - Problème si allocation et utilisation à des endroits différents
    → Allocation fainéante

Politique de remplacement - 1/2

- Si la mémoire est pleine
  - Libérer des pages pour en charger des nouvelles
- Gestion des pages résidentes
  - Combien de pages de chaque processus peuvent résider en mémoire ?
  - Supprimer n'importe quelle page ? Ou bien une du processus qui veut charger une page ?

Politique de remplacement - 2/2

- Quelle page supprimer ?
  - Celle qui a le moins de chances d'être utilisée dans un futur proche ?
  - Localité tend à ce que ce soit la page la moins récemment utilisée
  - Support matériel nécessaire
- Verrouillage
  - Pages importantes ne peuvent pas être évincées

Algorithmes pour le remplacement - 1/2

- Optimal
  - Évincer la page qui sera utilisée le plus tardivement
  - Impossible car nécessite de connaître l'avenir
- *Last Recently Used*
  - Algorithme naturel d'après le principe de localité
  - Semble bon d'après expérience
  - Difficile à implanter dans le matériel
    → Marquer les pages

Algorithmes pour le remplacement - 2/2

- *First-In-First-Out*
  - Pages évincées en *Round-Robin* dans un tampon circulaire
  - Simple à implanter dans le matériel
  - Pas efficace
- Politique de l'horloge
  - FIFO
  - Semble bon d'après expérience
  - Difficile à implanter dans le matériel
    → Marquer les pages

Page Buffering

- Les pages non-utilisées sont placées dans une liste spéciale mais pas effacées
  - Liste des pages modifiées ou pas modifiées
- En cas de défaut de page, on regarde d'abord si la page est dans la liste
- Optimisations en lien avec le cache
  - Éviter les défauts de cache quand une page est évincée

Politique de nettoyage

- Quand écrire sur le disque une page modifiée ?
- À la demande (*Demand Cleaning*)
  - Quand elle est évincée
  - Défaut de page plus lent
- À l'avance (*Pre-cleaning*)
  - Permet de regrouper les écriture (*Batch*)
  - Dommage si modifiée juste après
- Combinaison avec *Page Buffering*
  - Écriture régulière et passage en état non-modifié

Degré de multiprogrammation

- Si peu de processus
  - Matériel pas bien utilisé
- Si trop de processus
  - Trop de défaut de pages
- Adapter la fréquence des défauts à leur coûts
- Suspendre des programmes
  - Ceux dont la probabilité de faute est trop grande

Cinquième partie

Mémoire

Pagination dans SVR4 et Solaris

- Pagination pour les processus et blocs disque
- Pages décrites par
  - Table de pages
  - Descripteurs de bloc disque
  - Table des cadres physiques
  - Table d'utilisation du *swap*
- Remplacement par horloge

Kernel Memory Allocator de SVR4 et Solaris

- Allocation des petites zones
  - Moins d'une page
  - Toutes les structures internes au noyau
- *Lazy Buddy Algorithm*
  - Fusion des blocs pas immédiate
    → Dépassement d'un seuil
    → Quand vraiment nécessaire
- Remplacement par horloge

Cinquième partie

Mémoire

Adressage dans Windows

- Espace utilisateur privé pour chaque processus
  - Espace paginé
  - Entre 0 et 2 Go
    → Zones spéciales autour pour détecter les débordements
- Espace système partagé
  - Entre 2 et 4 Go
    → Micronoyau
    → *Executive*
    → Pilotes de périphériques
- Remplacement par horloge

Cinquième partie

Mémoire

Généralités sur la mémoire dabs dans Linux

- Surtout de la pagination
- Un peu de segmentation quand l'architecture le nécessite
  - Toute la mémoire pour le noyau
  - La mémoire utilisateur pour les processus
  - Quelques segments spéciaux

Espace d'adressage

- Espace utilisateur privé
  - Entre **0** et **PAGE_OFFSET** (3 Go sur IA32)
  - Divisé en différentes zones
  - *swappable*
- Espace noyau partagé
  - Au dessus de **PAGE_OFFSET**
  - Accessible uniquement en mode réel (privilégié)
  - Pas *swappable*

Structures mémoire

- Espace d'adressage (`struct mm_struct`)
  - Zones utilisateur (`struct vm_area_struct`)
    → Sauf pour les threads noyaux
  - Table de pages (`pgd_t`,`pmd_t`,`pte_t`)
- Cadre physique (`struct page`)

Espace utilisateur - 1/2

- Espace divisé en différentes zones
  - `struct vm_area_struct`
  - Caractérisée par localisation, protection, verrouillage
  - Méthodes décrivant comment les manipuler
    → Par exemple en cas de défaut de page
- Détaillé dans `/proc/<pid>/maps`
- Code en lecture seule
- Données en lecture et écriture

Espace utilisateur - 2/2

- Code placé en bas
- Tas juste au dessus
  - Données globales
  - `malloc()` utilise `sbrk()` pour l'agrandir
    → Modification de la VMA du tas dans `do_brk()`
- Pile tout en haut
  - S'agrandit automatiquement
- Grand espace libre entre les deux

Mapping en espace utilisateur

- VMA correspond au mapping d'un fichier
  - Mapping *anonyme* pour les données normales
    → Pile et tas
  - Mapping du programme
    → En lecture seule pour le code
    → En lecture et écriture pour les données
- Mapping des bibliothèques
  → En lecture seule pour le code
  → En lecture et écriture pour les données

Modifications de l'espace utilisateur

- Mapping explicite d'un fichier par l'utilisateur
  - Appels système `mmap()`, `mremap()`, `munremap()`
- Routines noyau pour manipuler les VMA
  - `do_mmap()`, `do_mremap()`, `do_munmap()`
  - Fichiers mappés sous la pile
    → Avec les bibliothèques
- Utilisé par `malloc()` pour les grandes allocations
- Permet de partager de la mémoire
  - Les mapping peuvent être privés ou partagés

Accès aux données utilisateur

- Espace d'adressage réellement divisé entre parties utilisateur et noyau
  - Cas particuliers 4Go/4Go
- Pas accès à la mémoire noyau en mode utilisateur
- Accès à la mémoire utilisateur en mode noyau
  - Adresses non déférençables
  - `get/put_user(addr,lentgh)`
  - `copy_from/to_user(to,from,length)`
  - Mapping des pages cibles

Cadres physiques dans Linux

- Grand tableau linéaire de `struct page` décrivant tous les cadres physiques
  - Indexé par *Page Frame Number*
  - Stockés après `mem_map`
- Pas de pointeurs vers la page virtuelle dans la plupart des cas
  - Car le cadre peut être partagé
- Bits de protections
- Pointés par la table de pages

Tables de pages - 1/2

- 3 (ou 4) niveaux selon les noyaux
  - Niveaux réels dépendant de l'architecture
- *Page Global Directory*
  - `pgd=pgd_offset(mm,address)`
- *Page Middle Directory*
  - `pmd=pmd_offset(pgd,address)`
- Page Table Entry
  - `pte=pte_offset(pmd,address)`
- Cadre physique
  - `page=pte_page(pte)`, `pfn=pte_pfn(pte)`

Tables de pages - 2/2

- `pgd=pgd_offset(mm,address)`
- `pmd=pmd_offset(pgd,address)`
- `pte=pte_offset(pmd,address)`
  - `pte_offset_map(pmd,address)`
- `page=pte_page(pte)`
- Vérification des différentes étapes
  - Existence : `pgd/pmd/pte_none()`
  - Présence en mémoire : `pgd/pmd/pte_present()`
  - Validité : `pgd/pmd_bad()`

Espace mémoire du noyau

- Mapping linéraire de la mémoire physique
  - `virt_to_phys()` et `phys_to_virt()` valides
    → Translation de **PAGE_OFFSET**
  - Physiquement contigu
  - Limité à 896 Mo sur IA32
    → Toute la mémoire physique n'y est pas
- Espace *vmalloc*
- Mapping fixes
  - Mémoire des périphériques et *High Memory*

High Memory

- Cadres physiques non mappés dans l'espace linéraire du noyau
- N'importe quel cadre physique peut être mappé temporairement
  - `addr=kmap(page)` et `kunmap(page)`
  - `addr=kmap_atomic(page,type)` et `kunmap_atomic(page,type)`
    → Visible sur un seul processeur
    → Ne doit pas durer longtemp
  - Nombre illimité

Mapping de pages utilisateur dans le noyau

- Manipulation intense de pages utilisateur dans le noyau
- `get_user_pages()` donne les cadres physiques (`struct page`)
- Mapping dans le noyau par `kmap`
- Possibilité de déréférencer comme n'importe quelle structure noyau
- Démapper (`kunmap`) puis relâcher les pages
  - `page_cache_release(page)`

Mapping linéaire de la mémoire

- Code tout en bas
- Ensemble de pages disponibles
  - Peuvent être allouées dans le noyau
    → `alloc_pages` et `free_pages`
  - Peuvent être utilisées pour les espaces utilisateur
- Ensemble de *Slab Caches* utilisant pages du mapping linéraire
  - Ensemble de zones de tailles fixes préallouées
  - *Buddy Algorithm*

Allocation mémoire dans le noyau - 1/2

- Allocation d'une structure de type précis
  - `kmem_cache_alloc(cache,flags)` et
  - `kmem_cache_free(cache,ptr)`
- Création d'un cache
  - `kmem_cache_create` et `kmem_cache_destroy`
- Allocation d'une structure de taille précise
  - `kmalloc(taille,flags)` et `kfree(ptr)`
  - Implanté par un cache de taille 2^n supérieure
    → de 32 octets à 128 ko

Allocation mémoire dans le noyau - 2/2

- Différents *flags* pour décrire ce que le *Slab Allocator* peut tenter pour allouer la mémoire
  - Exemples de flags internes
    → `__GFP_WAIT` si on peut bloquer
    → `__GFP_IO` si on peut faire des entrées/sorties
    → `__GFS_HIGHMEM` si on peut prendre des pages *HighMem*
  - Exemples de flags manipulés par le programmeurs
    → `__GFP_KERNEL` pour les allocations normales
    → `__GFP_ATOMIC` dans un contexte risqué
    → `__GFS_NOFS` dans un contexte d'accès aux fichiers

Zones de mémoires non-contiguës

- *Slab Allocator* limité aux zones assez petites
- Zones physiquement contiguës pas toujours utiles
- `ptr=vmalloc(size)` et `vfree(ptr)`
  - Alloue des pages quelconques
    → `alloc_pages` et `__GFP_HIGHMEM`
  - Les mappe contigument dans l'espace *Vmalloc*
    → Dans l'espace du noyau
  - Utilisé pour charger les modules

Le *Swap - 1/2*

- Pages transférées sur une partition de swap
  - Zones anonymes des processus
  - Mapping privés dans les processus
- Partition divisée en slots de la taille d'une page
- Mapping partagés et cache de fichiers écrits sur leur partition d'origine
- Libération des pages non récemment utilisées
  - Listes de pages actives et inactives

Le *Swap - 2/2*

- Quand swapper ?
  - Quand une allocation est impossible
  - `kswapd` vérifie régulièrement qu'il y a suffisamment de pages libres
- `try_to_free_pages()`
- `shrink_caches()`
  - Réduire les *Slab Caches*

Laziness

- Ne rien faire avant que ce ne soit nécessaire
  - Allouer réellement les pages
    → Attente d'un défaut de page
  - Libérer les pages inutiles
    → Attente qu'il n'y ait plus de mémoire libre
    → Linux utilise souvent toute la mémoire disponible
  - Dupliquer les pages partagées
    → Attente de la première modification concurrente
- Moins d'utilisation processeur et mémoire

Exceptions et défauts de page

- `handle_mm_fault(mm,vma,addr,wr)`
  - `handle_pte_fault()`
- Page pas encore réellement allouée
  - `do_no_page()`
- Page d'un fichier pas encore réellement mappée
  - `do_file_page()`
- Page swappées
  - `do_swap_page()`
- Page partagée mais encore privée
  - `do_wp_page()`

Copy-on-Write

- Pages dupliquées au dernier moment
  - À la première modification
- Permet un partage de code facile
  - Bibliothèques système
  - Après un `fork()`
- Utilisé pour allouer des pages initialisées à 0

Sixième partie

Entrées/Sorties

Types de périphériques

- Interaction avec l'homme
  - Entrée : clavier, souris, scanner, etc.
  - Sortie : écran, imprimante, retour de force, etc.
- Interaction interne à la machine
  - Stockage, senseurs, contrôleurs, etc.
- Interaction entre machines
  - Carte réseau, modem, etc.

Caractéristiques des périphériques

- Taux de transfert
- Application
- Complexité du contrôle
- Unité de transfert
- Représentation des données
- Gestion des erreurs

Taux de transfert

- Clavier et souris : 100 bits/s
- Modem : 56 kb/s
- Disquette, imprimante, scanner : 1 Mb/s
- Disque optique, Ethernet : 10 Mb/s
- Mémoire : 10 Gb/s

Sixième partie

Entrées/Sorties

- Périphériques
- Fonctionnalités pour les entrées/sorties
- Conception des entrées/sorties dans un système d'exploitation
- Exemples
  - SVR4
  - Windows
  - Détails de Linux

Évolution des entrées/sorties

- Le processeur contrôle tout
- Les contrôleurs de périphériques peuvent traiter les commandes
- Les contrôleurs peuvent interrompre le processeur
- Les contrôleurs peuvent accéder directement à la mémoire centrale
- Les contrôleurs deviennent programmables

Programmed I/O

- Processeur émet une commande à un périphérique
- Le périphérique traite la requête
- Le processeur attend de manière active le changement d'un registre de statut du périphérique
- Le périphérique change le registre quand il a terminé et indique le succès ou une erreur

Interrupt-driven I/O

- Processeur émet une commande
- Le périphérique traite la requête
- Le processeur continue son exécution
- Le périphérique émet une interruption quand il a terminé
- Le processeur s'interrompt et interroge le périphérique pour savoir si la requête a été traitée avec succès

Direct Memory Access - 1/2

- Un moteur DMA gère la transfert de données entre la mémoire principale et un périphérique
- Le processeur envoie une requête de transfert à un moteur DMA
- Le moteur DMA interrompt le processeur lorsque le transfert est terminé

Direct Memory Access - 2/2

- Requête
  - Lecture ou écriture ?
  - Adresse du périphérique
  - Adresse en mémoire centrale
  - Longueur
- Moteurs DMA placés sur les périphériques ou sur le bus d'entrées/sorties

Quelle stratégie utiliser ?

- Si le traitement par le périphérique est long
  - Interruption
    → Pas d'attente active
- Si la quantité de données à transférer est grande
  - DMA
    → Pas de gaspillage du temps processeur
- Si la requête est simple à traiter
  - PIO

Sixième partie

Entrées/Sorties

- Périphériques
- Fonctionnalités pour les entrées/sorties
- Conception des entrées/sorties dans un système d'exploitation
- Exemples
  - SVR4
  - Windows
  - Détails de Linux

Objectifs

- Efficacité
  - Les I/O sont le maillon faible
  - Multiprocessus pour maximiser l'utilisation
  - Swap pour augmenter le nombre processus actifs
    → C'est une I/O de plus
- Généralité
  - Voir les périphériques de manière uniforme
  - Organisation hiérarchique et modulaire

Structure logique - 1/2

- I/O logiques
  - Périphériques logiques (virtuels)
    → Systèmes de fichiers
    → Protocoles réseau
    → Terminaux
  - Accessibles aux processus utilisateur
- I/O physique
  - Périphériques physiques
  - Accessibles surtout à travers des I/O logiques
    → Structure organisée exposée par les I/O logiques

Structure logique, exemple - 2/2

- Accès aux fichiers depuis un processus
- I/O logique
  - Conversion du chemin d'accès en identifiant de fichier
  - Conversion du contenu du fichier en blocs d'un périphérique
- I/O physique
  - Conversion en blocs physiques sur un disque
  - Passage de commandes IDE de bas de niveau

Entrées/sorties *bufferisées* 1/3

- Pages mises en jeu ne peuvent pas être évincées
- Zones mises en jeu par forcément alignées
- I/O réelles effectuées par le système
  - Copie temporaire dans/depuis le système
  - Éviter des I/O en gardant les données dans le système
  - Regrouper les I/O dans le système
  - Réduire le blocage des applications
    → Seul le système attend à la fin de l'I/O réelle

Entrées/sorties *bufferisées* 2/3

- I/O orientées *bloc*
  - Disque dur, CDCROM, bande, etc.
  - Le système manipule des blocs
  - L'application utilise une granularité quelconque
- I/O orientées *caractère*
  - Terminaux, réseau, imprimante, souris, clavier, etc.
  - Le système stocke les données en attente dans le flux

Entrées/sorties *bufferisées* 3/3

- *Buffer* unique
  - Un buffer par I/O demandée par l'application
- *Buffer* multiple ou circulaire
  - Un buffer en copié pendant que l'autre subit I/O
- Cache général
  - Partagé entre les applications
  - Multiplexage des flux par caractère

Entrées/Sorties asynchrones

- Utilisé dans le système si *Interrupt-driven*
- Nécessaire dans le système pour multiplexage
  - Ne pas bloquer le système pendant une I/O
- Peu habituel dans les applications
  - Interface standard bloquante
- Interfaces modernes deviennent asynchrones
  - Recouvrir les I/O sans utiliser des threads
  - Soumission de requêtes
  - Récupération de notification de complétion plus tard

Cache disque

- Ensemble de buffers système partagés entre les applications
- Pages disque conservées en mémoire
  - Évincées par algorithme de type LRU
    → Gérées comme les pages des processus
- Préchargement des pages suivantes
  - Principe de localité
- Écritures disque déferrées et regroupées

Sixième partie

Entrées/Sorties

Entrées/sorties dans Unix SVR4

- Chaque périphérique est un fichier spécial
  - Accès uniforme aux périphériques et aux fichiers
- *Buffer Cache*
  - Organisé en table de hachage
- *Character Queue*
  - Modèle producteur/consommateur
    → Un seul lecteur possible
- I/O non bufferisées possibles

Sixième partie

Entrées/Sorties

Entrées/sorties dans Windows

- *I/O manager*
  - *Cache manager*
  - Pilotes de systèmes de fichiers
  - Pilotes réseau
  - Pilotes de périphériques matériel
- I/O synchrones ou asynchrones
  - 4 techniques de notification

Sixième partie

Entrées/Sorties

Accès utilisateur aux périph. en mode char - 1/2

- Fichier spécial
  - `mknod /dev/mychr c <major> <minor>`
  - Correspond à un périphérique réel ou non
    → Peut être un simple point d'entrée dans le noyau
- Manipulé comme n'importe quel fichier
- `register_chrdev(major,name, fops)`
  - `major` identifie le fonctionnement du périphérique
    → `/proc/devices`

Accès utilisateur aux périph. en mode char - 2/2

- Comportement du fichier spécial configurable par ses `struct file_operations`
  - `open`, `release`, `read`, `write`, `mmap`, `fsync`, etc.
- Fichier stocké dans un `struct inode`
  - Champ `i_rdev` donne les `major` et `minor`
    → `minor` est un paramètre
- Instance du fichier ouvert dans un `struct file`
  - Champ `private_data` pour stocker des données

Accès utilisateur aux périph. en mode bloc - 1/2

- Fichier spécial
  - `mknod /dev/myblk b <major> <minor>`
  - Correspond à un disque réel ou virtuel
- `register_blkdev(major,name)`
  - `/proc/devices`
- Comportement normal du fichier spécial
  - Comme tous les disques

Accès utilisateur aux périph. en mode bloc - 2/2

- Chaque disque est décrit par `struct gen_disk`
  - Enregistré par `add_disk()`
- Quelques opérations spécifiques configurables
  - `struct block_device_operations`
- Déclaration d'une fonction de traitement des requêtes d'entrées/sorties
  - `blk_init_queue(request_queue_t *queue,request_fn_proc *request)`

Les *IOCTL* : Input Output ConTroL

- Permet de définir un ensemble de commandes
  - Spécifique à un périphérique (logique ou physique)
  - Évite de définir de nouveaux appels système
  - Appel système ioctl(fd,cmd,arg)
  - Champ ioctl des `file_operations` et des `block_device_operations`
- Échange de données avec l'espace utilisateur
  - Utilisation du champ `arg` pour passer un pointeur

Détection des périphériques PCI

- Reconnus par `struct pci_device_id`
  - `vendor`, `device`, etc.
    → Peuvent être `PCI_ANY_ID`
  - `class` et `class_mask`
  - Données spécifique au pilote
- Dans `struct pci_driver`
  - Liste des périphériques gérés par le driver
  - Avec le nom du pilote
  - Et les fonctions d'initialisation (`probe`) et arrêt (`remove`) du périphérique

Initialisation des périphériques PCI

- `int pci_register_driver(struct pci_driver *drv)`
  - Initialise tous les périphériques reconnus
    → Sauf ceux déjà gérés par un pilote
  - Crée un `struct pci_dev` décrivant le périphérique
    → Ressources disponibles, ligne d'interruption, etc.
  - Appel de la fonction `probe` du pilote
- `pci_enable_device` active le périphérique
- `pci_read_config_byte` donne des infos
  - `PCI_REVISION_ID`

Accès aux ressources des périphériques PCI

- `struct pci_dev` listées par lspci -vv
  - ou `/proc/pci`
- Ressources spécifiques disponibles
  - `pci_ressource_flags` donne le type
    → Mémoire ou ports I/O, avec des caractéristiques
  - `pci_ressource_start/end/len` donnent les adresses
  - `pci_request_region` pour réserver ces ressources
- Même principe pour les autres types de bus

Accès aux ports I/O

- Ensemble de registres de contrôle et statut rendus (CSR) accessibles par les périphériques
  - Organisés par le **BIOS**
  - `/proc/ioports`
- Utilisés pour les PIO
  - Lecture par `int/inw/inl(port)`
  - Écriture par `outb/outw/outl(val,port)`
  - Suffixe `_P` pour forcer des pauses
- Mapping pas forcément linéraire

Accès à la mémoire des périphériques - 1/2

- Ensemble de zones mémoire rendues disponibles par les périphériques
  - Organisées par le **BIOS**
  - `/proc/iomem`
- Espace mémoire I/O pas toujours identique à l'espace mémoire normal du processeur
  - Remapper la mémoire I/O dans la table des pages du processeur
- Mapping pas forcément linéraire

Accès à la mémoire des périphériques - 2/2

- `ioremap/ioremap_nocache(addr,size)`
  - Similaire à `vmalloc`
- Pas forcément déférençable
  - Lecture par `readb/readw/readl(addr)`
  - Écriture par `writeb/writew/writel(val,addr)`
- Barrières mémoires pour éviter réordonnancement `mb/rmb/wmb()`
- `iounmap(addr)`

Accès utilisateur à la mémoire des périphériques

- Mapping d'un chrdev spécifique
  - Configurations de son opération `mmap`
- Remapping de PTE dans une autre VMA
  - `remap_page_range(vma,vaddr,paddr,size,prot)`
    → `remap_pfn_range(vma,vaddr,pfn,size,prot)`
- Utilisation des PTE créées par `ioremap`

DMA et adresses de bus

- Pas forcément les mêmes adresses pour les périphériques
  - Adresses de bus `dma_addr_t`
- Adresses ISA identiques aux adresses physiques
- Adresses PCI pas nécessairement identiques
  - `IOMMU` sur certaines architectures
  - `virt_to_bus/bus_to_virt` dans la mapping linéraire du noyau, sur certaines architectures
  - Mapping PCI spécifique des pages concernées

DMA et mapping PCI

- `pci_dma_supported(pci_dev,mask)`
  - `pci_set_dma_mask(pci_dev,mask)`
- Mapping persistant (consistent)
  - `dma=pci_map_single(pd_dev,vaddr,size,direction)`
    → `PCI_DMA_BIDIRECTIONAL`, `PCI_DMA_TODEVICE`, etc.
  - `pci_map_page` et `pci_map_sg`

Programmation d'un traitant d'interruption

- Déclaration de l'interface
  - `request_irq(irq,func,flags,name,data)`
  - Numéro de la ligne d'interruption
  - Fonction chargée de traiter l'interruption
    → `irqreturn_t func(irq,data,regs)`
  - `SA_SHIRQ` pour accepter de partager la ligne avec d'autres périphériques
  - Nom du périphérique
  - Donnée spécifique du pilote
- `/proc/interrupts`

Traitement des interruptions - 1/1

- Le noyau appelle le traitant programmé pour chaque périphérique de la ligne d'interruption
- Vérification du statut du périphérique
  - `IRQ_NONE` si le périphérique n'est pas concerné
- Traitement de l'interruption
  - Stockage des informations
  - Programmation du traitement lourd
- Mise à jour du statut du périphérique
- Renvoi de `IRQ_HANDLED`

Traitement des interruptions - 2/2

- Interruptions traitées dans contexte très spécial
  - Ne s'exécute pas dans le contexte d'un processus
    → Ne peut accéder à l'espace utilisateur
  - Ne peut pas passer la main
    → Ne pas dormir (`sleep_on()` ou `wait_event()`)
    → `kmalloc` avec `GFP_ATOMIC` uniquement
    → Pas d'accès aux pages swappables
  - Certaines interruptions sont désactivées
    → Ne pas nuire à la réactivité trop longtemps

Travaux déferrés

- Le traitant d'interruption est très limité (*Top Half*)
- Le vrai travail est programmé et effectué plus tard dans un contexte normal (*Bottom Half*)
  - S'exécute avec les interruptions activées
    → Meilleure réactivité du système
    → Latence légèrement supérieure en moyenne
  - Moins de blocage en cas d'interruptions en rafale
  - Implantatable dans un thread noyau
  - Interface noyau dédiée

Bottom Halves

- Plusieurs stratégies disponibles dans le noyau
  - *BH* et *Task Queues* supprimés depuis 2.5
  - *Soft IRQ*
    → Réservé aux tâches critiques
  - *Tasklet*
    → Plus souple
- Pas exécuté dans le contexte d'un processus
- Ne peut pas dormir
- Ne peut appeler `schedule()`

Soft IRQ

- 32 alloués statiquement à la compilation
  - Classés par priorité
- Initialisé pas `open_sofirq`
- Exécuté régulièrement par `do_softirq`
- Un thread noyau dédié par processeur `ksoftirqd/<cpu>`
  - Exécution concurrente

Tasklets

- Création et destruction dynamiques
- Implantation au dessus d'un Soft IRQ spécial
- Pas d'exécution concurrente d'un même Tasklet
- `DECLARE_TASKLET(name,func,data)`
- `tasklet_schedule`
  - Exécution dans un futur proche
- `tasklet_disable/enable`

Synchro. des diff. contextes en jeu dans les I/O

- Traitant d'interruption contre *Bottom Half*
  - Verrou et interdiction d'être interrompu par un traitant
    → `spin_lock_irqsave/spin_unlock_irq restore`
- *Bottom Half* contre contexte de processus
  - Verrou
    → `spin_lock/spin_unlock`
- Interdiction d'être préempté par un *Bottom Half*
  - `local_bh_disable/enable`

Work queues

- Travail déferré qui peut dormir
  - S'exécute dans un thread noyau `evnts/<cpu>`
    → Ou dans un thread noyau spécifique
  - Allocation mémoire, sémaphore, etc.
- `DECLARE_WORK(name,func,data)`
- `schedule_work`
- `flush_scheduled_work`
- `schedule/cancel_delayed_work`

Attente d'évènement avec un fichier en mode carac

- Les appels systèmes `poll` et `select` attendent sur un ensemble de descripteurs
- Processus placé dans les `wait_queue` par l'opération `poll` des descripteurs
  - `unsigned int my_poll(file,poll_table)`
    → Utilise souvent `pdl_wait`
    → Renvoie les évènements
- Les pilotes concernés réveillent la `wait_queue` en cas d'évènement

Rqs : Interruptions

- 16 interruptions matérielles avec *Programmable Interrupt Controller* (PIC)
- Jusqu'à 256 avec l'*APIC*

Maître	Esclave	Utilisation
IRQ0		Timer
IRQ1		Clavier
IRQ2		Connexion avec esclave
	IRQ9	Réservé
	IRQ10	Réservé
	IRQ11	Réservé
	IRQ12	Réservé
	IRQ13	Copro arithm.
	IRQ14	Contrôl. disque
	IRQ15	Réservé
IRQ3		Port série 2
IRQ4		Port série 1
IRQ5		Port // 2
IRQ6		Contrôl. D7
IRQ7		Port // 1

Septième partie

Ordonnancement

Types d'ordonnancement

- À long terme
  - Quels processus vont être exécutés ?
- À moyen terme
  - Quels processus sont résidents en mémoire ?
- À court terme
  - Quel processus est exécuté par le processeur ?
- Entrées/sorties
  - Quelles I/O de quel processus sont traitées par un périphérique disponible ?

Ordonnancement à long terme

- Degré de multiprogrammation avec des batchs
  - Maximiser l'utilisation du matériel
    → Processeurs et périphériques
  - Éviter trop de défauts de page
- Peut utiliser différents critères
  - Priorités, temps d'exécution prévu, besoins en I/O
  - Optimiser l'utilisation des différentes ressources
- Exécuté au lancement des processus
- Doit être contourné pour les processus interactifs

Ordonnancement à moyen terme

- Consiste essentiellement à évincer ou rappeler en mémoire des processus
- Également lié au degré de multiprogrammation
- Exécuté assez régulièrement

Ordonnancement à court terme

- Exécuté très régulièrement
- Quand un évènement risque de bloquer un processus
  - I/O
- Quand il est bon de favoriser un autre processus
  - Interruption d'horloge
  - Signaux

Septième partie

Ordonnancement

Critères pour l'utilisateur

- Critères relatifs à la performance
  - Rapidité d'exécution des processus
  - Temps de réponse
  - *Deadlines*
- Autres critères
  - Prédictabilité

Critères pour le système

- Critères relatifs à la performance
  - Quantité de travail effectué
  - Utilisation des processeurs
    → et des périphériques
- Autres critères
  - Equité
  - Respect des priorités
  - Gestion des ressources

Préemption

- Un processus ne rend jamais la main
  - Sauf en cas d'I/O
- Le système prend la main de force
  - Léger surcoût
  - Bien meilleur temps de réponse
    → Interactivité
  - Quand préempter ?

Algorithmes classiques

- FIFO
- *Round-Robin*
  - Exécution pendant de petites périodes (*timeslices*)
- *Short Process Next* et *Short Remaining Next*
  - Demande de prévoir la durée d'exécution
- *Feedback*
  - Pénalité pour les processus trop gourmands
    → Ajuste la priorité initiale des processus

Multiprocesseurs - 1/2

- *Load Sharing*
  - File générale de processus
- *Gang Scheduling*
  - Threads reliés ordonnancés simultanément sur un ensemble de processeurs
- Assignation d'un processeur dédié
  - Processeur inaccessible jusqu'à la fin d'exécution
- Ordonnancement dynamique

Multiprocesseurs - 2/2

- Concurrence réelle entre les files d'exécution sur différents processeurs
  - Pas seulement en cas de préemption
  - Synchronisation critique
- Effets de cache entre threads
- Localisation mémoire sur NUMA

Besoins du temps réel

- Déterminisme
- Réactivité
- Fiabilité
- Contrôle utilisateur

Caractéristiques des ordonnanceurs temps réel

- Changement de contexte rapide
- Taille et fonctionnalités réduites
- Réaction rapide aux interruptions
  - Minimisation des désactivations
- Ordonnancement préemptif fondé sur priorités
- Primitives de retardement et pause de tâches
- Alarmes et *timeouts* dédiés

Septième partie

Ordonnancement

Accès disque - 1/2

- Temps d'accès
  - Recherche du cylindre + rotation jusqu'au secteur
  - De l'ordre de 10 ms
- Débit
  - De l'ordre de 50 Mo/s
- Optimiser les accès pour augmenter le débit réel
  - Regrouper les accès proches

Accès disque - 2/2

- FIFO, avec priorité
- Élevator
  - Réduire les déplacements de la tête de lecture
- *Deadline*
  - Lectures privilégiées
- *Anticipatory*
  - Petite pause pour attendre des requêtes consécutives
- *Completely Fair Queuing*
  - Accès aux périphériques par *Timeslice*

Accès réseau

- *Quality of Service*
- *Queuing Disciplines*

Septième partie

Ordonnancement

Ordonnancement traditionnel dans Unix

- Priorité de base dynamique
  - *Feedback*
- Facteur ajustable par l'utilisateur (*nice*)
- Swapping très prioritaire
- Priorité croissante des applications aux périphériques réels

Septième partie

Ordonnancement

Le *Scheduler* de Linux

- Ordonnanceur Unix traditionnel
- Bonne interactivité et équité
- Priorité effective ajustée selon bonus/pénalités
  - Crédit d'interactivité quand la tâche dort
- Beaucoup d'optimisations locales
  - Tâche créée sans `CLONE_VM` préempte père
    → Évite coût du *Copy-on-Write*

Les *Timeslices*

- Temps divisé en cycles d'ordonnancement
  - Chaque processus prêt reçoit un *timeslice* par cycle
- *Timeslice* entre 5 et 800 ms (100 par défaut)
  - Priorité de 19 à -20 (0 par défaut)
- Peut-être consommé en plusieurs fois
  - Si préempté pendant
- Partage du *Timeslice* entre père et fils

Le *O(1)-Scheduler* - 1/2

- Supporte beaucoup de processus et de processeurs
  - Optimisation des structures pour performance
- *runqueues* indépendantes sur chaque processeur
- Chaque processus `RUNNING` placé dans une des *runqueues*
  - Tableau de priorités
    → Actif ou expiré
- Processus non prêts dans files d'attente externes

Le *O(1)-Scheduler* - 2/2

- Permutation des tableaux expirés et actifs à la fin de chaque cycle
- Prochain *timeslice* calculé à l'expiration du *timeslice* précédent
- Affinités pour les processeurs
  - Champs `cpus_allowed` de `task_struct`
    → `sched_set/getaffinity()`
  - Migration uniquement en cas de déséquilibre
    → *Load Balancer*

Préemption

- Si un processus prioritaire arrive
  - Bit `NEED_RESCHED` de la tâche courante
  - Vérifié régulièrement par le *scheduler*
- Préemption possible lors du retour en espace utilisateur
  - Fin d'appel système
  - Fin d'interruption
    → Interruption d'horloge

Préemption dans le noyau

- Préemption dans le noyau depuis 2.6
- Compteur `preempt_count` dans `task_struct`
- Augmenté pendant les opérations où la préemption doit être désactivée
  - Tenue d'un `spin_lock`
  - Possession d'un mapping atomique (`kmap_atomic`)
  - À la main `preempt_disable/enable()`
- `need_resched` testé quand compteur nul

Qui préempte qui ? - 1/2

- Les interruptions sont prioritaires
  - Sauf si désactivées
  - Exécution dans un contexte spécial
    → Un contexte dédié par processeur (`irq_ctx hard_irq`)
  - Interruptions peuvent être désactivées pendant traitant
- *Bottom Halves* au retour du traitant d'interruption
  - Exécution dans un contexte spécial
    → Un contexte dédié par processeur (`irq_ctx soft_irq`)
  - Peut être préempté par une interruption

Qui préempte qui ? - 2/2

- Les threads noyaux sont préemptés par
  - Les interruptions puis les *Bottom Halves*
  - Les tâches prioritaires à certains endroits
    → sauf si `preempt_count` > 0
- Les processus utilisateur sont préemptés dans le noyau comme les threads noyaux
  - et avant leur retour en espace utilisateur
    → Interruptions (horloge), signaux et appels-système

Changement de contexte

- `schedule()` appelé par un processus
  - Effectue lui-même les tests du *scheduler*
- `switch_mm()` permute les espace d'adressage
- `switch_to()` permute les contextes d'exécution

`sched_yield`

- Demande explicite de passage de main
  - Déplace la tâche dans le tableau expiré
  - Le *timeslice* est perdu !
- Comportement différent entre 2.4 et 2.6
  - Le *timeslice* n'était pas perdu dans le 2.4
  - Incompatibilités entre applications pour 2.4 et 2.6

Temps réel

- Politiques plus prioritaires que `SCHED_OTHER`
- `SCHED_FIFO`
  - Garde le processeur jusqu'à le rendre explicitement
- `SCHED_RR`
  - *Timeslice* prédéfini
- Pas de garantie sur l'efficacité
  - Satisfaisant dans les cas simples
- Gros impact sur les autres tâches