Programmation Système

Cours 6

Entrées-sorties

Brice Goglin

Copyright

Plan

- Périphériques
- Fonctionnalités pour les I/O
- Conception des I/O dans les OS
- Exemples
- Détails de Linux

Périphériques

Types de périphériques

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

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 bit/s
- Modem : 56 kb/s
- Disquette, imprimante, scanner : 1 Mb/s
- Disque optique, Ethernet : 10 Mb/s
- Disque dur : 100 Mb/s
- Carte graphique, GigaEthernet : 1 Gb/s
- Mémoire : 10 Gb/s

Fonctionnalités pour les entrées-sorties

Évolution des entrées-sorties

- Le processeur contrôle tout
- Les controleurs de périphériques peuvent traiter des commandes
- Les controleurs peuvent interrompre le processeur
- Les controleurs peuvent accéder directement à la mémoire centrale
- Les controleurs deviennent programmable

Programmed I/O

- Processeur émet une commande à un périphérique
- Le périphérique traite la requète
- 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

- Le processeur émet une commande
- Le périphérique traite la requète
- 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

- Un moteur DMA gère le 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 d'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

Conception des entrées-sorties dans un système d'exploitation

Objectifs

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

Structure logique

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

Structure logique (2/2) : exemple

- 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 niveau

Entrées-sorties *bufferisées*

- 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
  - Eviter 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 la fin de l'I/O réelle

Entrées-sorties *bufferisées* (2/3)

- I/O orientées *bloc*
  - Disque dur, CDROM, bande, ...
  - Le système manipule des blocs
  - L'application utilise une granularité quelconque
- I/O orientées *caractère*
  - Terminaux, réseau, imprimante, souris, clavier, ...
  - 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
  - Evincées par algorithme du type LRU
    - Gérées comme les pages des processus
- Préchargement des pages suivantes
  - Principe de localité
- Ecritures disque déferrées et regroupées

Exemples

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

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

Détails de Linux

Accès utilisateur aux périphériques en mode caractère

- 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 périphériques en mode caractère (2/2)

- Comportement du fichier spécial configurable par ses struct file_operations
  - open, release, read, write, mmap, fsync, ...
- 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
    - Pointeur vers structure décrivant le périphérique

Accès utilisateur aux périphériques en mode bloc

- 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 périphériques 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 requête d'entrées-sorties
  - blk_init_queue(func, &lock)

Les *IOCTL*

- Appel système ioctl(fd, cmd, arg)
- Champ ioctl des file_operations et des block_device_operations
- Permet de définir un ensemble de commandes
  - Spécifique à un périphérique (logique ou physique)
  - Evite de définir de nouveaux appels système
- Echange 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, ...
    - Peuvent être PCI_ANY_ID
  - class et class_mask
  - Donnée spécifique au pilote
- Table placée dans struct pci_driver
  - Avec le nom du pilote
  - Et les fonctions d'initialisation (probe) et arrêt (remove) du périphérique

Initialisation des périphériques PCI

- pci_register_driver utilise le pci_driver
  - 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, ...
  - 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_resource_flags donne le type
    - Mémoire ou ports I/O, avec des caractéristiques
  - pci_resource_start/end/len donnent les adresses
  - pci_request_regions pour réserver ces resources
- 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 inb/inw/inl(port)
  - Ecriture par outb/outw/outl(val, port)
  - Suffixe _p pour forcer des pauses
- Mapping pas forcément linéaire

Accès à la mémoire des périphériques

- 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

Accès à la mémoire des périphériques (2/2)

- ioremap/ioremap_nocache(addr, size)
  - Même similaire à vmalloc
- Pas forcément déréférençable
  - Lecture par readb/readw/readl(addr)
  - Ecriture 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
  - Configuration 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 le mapping linéaire 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*)
  - kaddr = pci_alloc_consistent(pci_dev, baddr, &dma)
- Mapping temporaire (*streaming*)
  - dma = pci_map_single(pci_dev, vaddr, size, direction)
    - PCI_DMA_BIRECTIONAL, PCI_DMA_TODEVICE, ...
  - pci_map_page et pci_map_sg

Programmation d'un traitant d'interruption

- 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

- 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 pas 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
  - Implémentable 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

Soft IRQ

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

Tasklets

- Création et destruction dynamiques
- Implémenté 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`

Synchronisation des différents 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_irqrestore
- *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 events/<cpu>
    - Ou dans un thread noyau spécifique
  - Allocation mémoire, sémaphores, ...
- DECLARE_WORK(name, func, data)
- schedule_work
- flush_scheduled_work
- schedule/cancel_delayed_work

Attente d'événement avec un fichier en mode caractère

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