Chapitre 9

Processus et Threads

Introduction

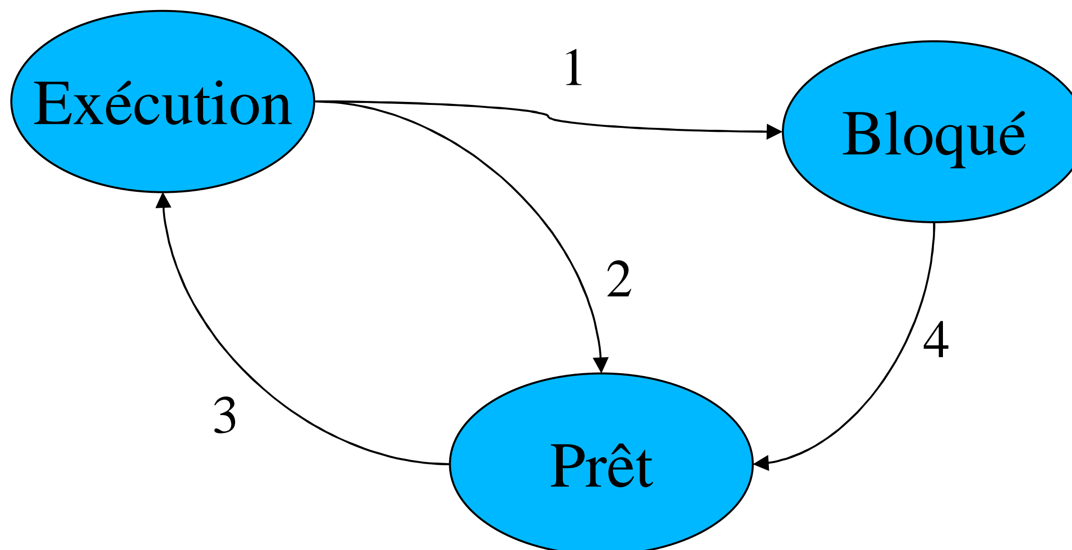
- Les ordinateurs donnent l'impression de faire plusieurs choses à la fois
- Mais à un instant donné, un seul programme est en exécution
  - Pseudo parallélisme
- Sauf sur un multiprocesseur/multicore
- Un processus est un programme en cours d'exécution
  - Compteur d'instruction en cours
  - Registres
  - Variables
- Chacun des ces processus peut avoir le cpu pour une durée limitée

Création et terminaison de processus

- Un processus est crée dans 4 situations
  - Démarrage du système
  - Appel système de création par un processus
  - Demande de création par un utilisateur
  - *Batch job* (soumission d'un travail à un cluster…)
- Terminaison
  - Fin normale
  - Fin anormale
  - Erreur
  - Tué par un autre processus (*kill*)

États

- Les processus peuvent se trouver dans plusieurs états
- En exécution
  - Utilise actuellement le CPU
- Prêt
  - En attente du CPU
- Bloqué
  - En attente d'un évènement extérieur



1. Attente d'entrées
2. Scheduler choisit autre processus
3. Scheduler choisit ce processus
4. Entrées disponibles

Threads

- Un processus regroupe 2 concepts
  - Des ressources (données, fichiers…)
  - Une exécution
- Un *thread* est une abstraction de l'exécution
- Chaque *thread* a son propre compteur de programme qui maintient l'historique des instructions
- Plusieurs *threads* partagent le même espace mémoire
- Les processus sont une abstraction pour grouper des ressources
- Les *threads* sont les entités exécutées par le CPU
  - Elles permettent à plusieurs exécutions d'avoir lieu dans le même processus
  - Il n'y a pas de protections entre les *threads* d'un même processus
- Les *threads* sont appelés processus légers

Données et méthodes

- Données par processus
  - Espace d'adressage
  - Variables globales
  - Fichiers ouverts
  - Processus fils
  - Alarmes en attente
  - Signaux et gestionnaires de signaux
  - Informations de comptabilité
- Données par *thread*
  - Compteur d'instruction
  - Registres
  - Pile
  - État
- Méthodes
  - *thread_create*
  - *thread_exit*
  - *thread_wait* : attente d'un autre *thread*
  - *thread_yield* : abandon du cpu pour laisser un autre *thread*

Intérêt des *threads*

- Dans une application, plusieurs activités s'exécutent en parallèle
  - Les *threads* sont un bon modèle de programmation
  - Comme les processus ☺
- Les *threads* partagent le même espace mémoire
  - Indispensable pour certaines applications
- Les *threads* ont peut d'information propre
  - Très faciles à créer/détruire
  - En général, 100 fois plus rapide à créer qu'un processus
- Permettent de recouvrir le calcul et les I/Os
  - Si les *threads* font que du calcul, peu d'intérêt
- Très pratiques sur les systèmes multi cpu

Implémentation des *threads* en User Space

- Les *threads* sont implémentées par un package en User Space
- Permet de fonctionner sur des OS qui ne connaissent pas les *threads*
  - L'OS manipule toujours des processus mono-threads
- Chaque processus a une table de *threads*
  - Contient les informations pour les *threads* du processus
  - Mise à jour quand changement d'état
- Si un *thread* va faire une opération potentiellement bloquante
  - Il appelle une méthode spéciale
  - Cette méthode vérifie si le *thread* doit être mis en attente
  - Si oui, modification de la table, recherche d'un autre *thread*, chargement du PC et registres…

Implémentation des *threads* en User Space

- *Thread switching* très rapide
  - Pas de passage en kernel space
- Permet à un programme d'avoir son propre *scheduler*
- Mais problème avec les appels systèmes bloquants
  - Ex: lire une touche d'un clavier, peut être bloquant ou pas
  - Si bloquant, *tous* les *threads* sont bloqués
  - Solution : wrapper les appels système pour les rendre non bloquants
- Et les page fault aussi ☹
- Les *threads* doivent collaborer
  - *thread_yield*
- En pratique beaucoup de boulot/contraintes pour gérer le blocage et le scheduling
  - Alors que c'est l'intérêt des *threads!*

Implémentation des *threads* en kernel space

- Le kernel gère les *threads*
  - Il a une table des *threads* dans le système
- La création et la destruction sont effectués par le kernel à travers des appels système
  - Plus coûteux qu'en user space
  - Utilisation de pools de *threads*
- Quand un *thread* bloque
  - Le kernel peut choisir un autre *thread* du même processus
  - Ou pas

Implémentation hybride

- Combiner les avantages des 2 approches
- Existence de *threads* kernel
- Et des *threads* utilisateur sont multiplexées dessus
- Les *threads* kernel sont gérées par le kernel

Scheduling

- Le *scheduler* est le mécanisme de l'OS chargé de décider quel processus exécuter
- Tous les processus ne sont pas égaux
  - Interactivité est importante
  - Il vaut mieux délayer les opérations non visibles
- Passer d'un processus à un autre est coûteux
  - Sauvegarde de l'état du processus
  - Chargement du nouvel état (registres, MMU…)
  - Démarrage du nouveau processus
  - Invalidation probable du cache mémoire
- Les processus alternent souvent des phases de calcul avec des phases d'IO
  - Si phases de calcul très importantes : *CPU-Bound*
  - Si I/O : *I/O Bound*

Quand scheduler

- Les décisions de scheduling sont prises dans plusieurs circonstances
- Fork
  - Qui choisir entre le père et le fils
- Fin d'un processus
  - En trouver un autre
  - Si aucun, un processus spécial est exécuté (*idle* ou *Processus inactif du système*)
- Quand un processus devient bloqué
  - Attente d'une I/O ou d'une condition
  - Peut influer sur le prochain à exécuter
- Interruption I/O
  - Si indique fin de traitement, rescheduler le processus qui attendait
- Si horloge 50Hz ou 60Hz
  - Décision de scheduling à chaque tic
- Scheduling non préemptif
  - Un processus est choisi et a le CPU jusqu'à ce qu'il bloque
- Scheduling préemptif
  - Un processus n'a le cpu que pour une durée maximale

Familles de schedulers

- Des systèmes ont des propriétés et besoins différents
- Il faut des schedulers adaptés
- Batch
  - Pas d'utilisateurs devant des écrans
  - Schedulers non préemptifs
  - Process switch réduits
- Interactive
  - Préemption pour maintenir la réactivité
- Temps réel
  - Préemption
  - Mais pas forcément nécessaire car ces systèmes ne font tourner que des programmes prévus pour

Objectifs

- Les objets des algorithmes de scheduling dépendent du système considéré
  - Mais certaines propriétés sont communes
- Commun
  - Équité : chaque processus doit avoir accès au cpu de manière équitable
  - Respect des règles : permettre à certains processus d'avoir un scheduling particulier
  - Équilibre : Maximiser l'utilisation du systèmes (alterner des I/O et du cpu)
- Batch
  - Débit : maximiser le nombre de jobs par heure
  - Temps à complétion : minimiser le temps entre la soumission et la complétion
  - CPU : maximiser l'utilisation du cpu

Objectifs

- Interactif
  - Temps de réponse : répondre rapidement aux demandes
  - Proportionnalité
    - Les utilisateurs ont souvent une idée du temps que va prendre une opération (click == rapide…)
    - Le scheduler doit choisir les processus pour être conforme à leurs attente
- Temps réel
  - Respecter les contraintes temporelles
  - Prédictibilité : le scheduler doit être prévisible
- Certains de ces objectifs peuvent ne pas être maximisés en même temps
  - Débit et temps à complétion par exemple…

First Come First Served

- Scheduling dans les batch
- Unique queue des processus prêts
- Le premier processus a le CPU, jusqu'à ce qu'il devienne bloqué
- Le suivant est mis à l'exécution
- Facile à comprendre et à programmer
- Mais pas du tout optimal
  - Un processus utilise le cpu 1 seconde, puis fait une i/o
  - Les autres processus doivent faire 1000 I/O
    - Ils prendront 1000 secondes pour finir
    - Avec d'autres algorithmes on aurait un meilleur résultat
    -

Shortest Job First

- On suppose que le temps d'exécution des jobs est connu à l'avance
- Pas irréaliste
  - Réservations sur un cluster
  - Batch de jobs identiques
- Le scheduler prend le job qui a le temps d'exécution le plus court
- Temps moyen à complétion optimal quand tous les jobs sont disponibles
  - Pas de pause en attente de job
- Une variante préemptive existe
  - Shortest Remaining Time Next

Round-Robin Scheduling

- Pour les systèmes intéractifs
- À chaque processus est associé un *quantum*
  - Durée maximale durant laquelle il peut s'exécuter
  - Si il atteint son quantum, il est préempté
  - Si il est bloqué avant, un autre processus est mis
- Une unique liste de processus prêts est nécessaire
- La difficulté est de choisir la bonne durée pour le *quantum*
  - Trop petit, mauvaise utilisation du cpu (coût des context switch)
  - Trop grand, mauvaise réactivité
  - En général, entre 20 et 50ms (42?!)

Scheduling à priorité

- Le Round Robin fait une hypothèse forte : tous les processus sont aussi importants
- Pas vrai en réalité
  - Un processus de vidéo est plus important qu'un processus qui affiche l'heure
- On associe à chaque processus une priorité
  - Le processus de plus haute priorité est autorisé à s'exécuter
- Éviter la famine
  - A chaque tic d'horloge la priorité du processus en exécution est réduite
  - Si inférieure à un autre processus, context switch
  - Mélange avec les quanta
- Optimiser les performances
  - Un processus qui fait beaucoup d'I/O devrait avoir une grande priorité car il consomme peu de CPU
  - Simple : la priorité est une fraction du quantum effectivement utilisé
    - Un processus qui tourne 1ms sur 50 a une priorité de 50
    - Un processus qui tourne 50ms a une priorité de 1

Shortest Process Next

- Inspiré du Shortest Job First du batch scheduling
- Idée
  - Donner le cpu au processus qui aura le temps d'exécution le plus cours
- Comment savoir?
  - Utilisation des performances passées
- Somme pondérée
  - Soit T_0 et T_1 les 2 dernières durée du processus sur le cpu
  - On estime le temps suivant comme $a*T_0 + (1-a)*T_1$
  - Nécessaire de choisir un « bon » a
  - Même principe que le vieillissement dans la gestion de pages

Scheduling de threads

- 2 niveaux de parallélisme
  - Processus
  - *Threads*
- User level threads
  - Le kernel ne connaît que les processus
  - Pas de moyen de préempter les threads d'un même processus
    - Collaboration nécessaire
  - Le scheduler peut être adapté à l'application
    - Connaissance des fonctions des threads
    - Très performant
- Kerne level threads
  - Le kernel choisit les threads
  - Rien ne l'oblige à tenir compte du processus parent
  - Mais peut nécessiter un context switch complet
    - Couteux
  - Le scheduler peut décider de privilégier les threads du même processus

Étude de cas : Windows XP

Concepts

- WinXP supporte les processus traditionnels qui peuvent communiquer et se synchroniser
- Chaque processus contient au moins un *thread*
- Chaque *thread* contient au moins une fibre (*fiber* ou *lightweight thread*)
- Les processus peuvent être regroupés en Jobs
- Job
  - Un ou plusieurs processus gérés comme un ensemble
  - Gestion de quotas (nombre de processus, temps CPU individuel et combiné….)
  - Limitation des droits
- Processus
  - Similaires aux processus Unix
  - 4GB d'adressage mémoire, 2GB pour l'utilisateur (3GB pour les versions server)

- Processus (suite)
  - Possède une ID, un ou plusieurs *threads*, une liste de *handle* (référence vers des objets) et un jeton d'accès
  - Les processus sont crées par un appel Win32 avec comme paramètre le nom de l'exécutable
- Threads
  - Unité de scheduling
  - Chaque thread a un état (prêt…) contrairement aux processus
  - Peuvent être crées par un appel Win32
  - Possède un ID (différent du processus)

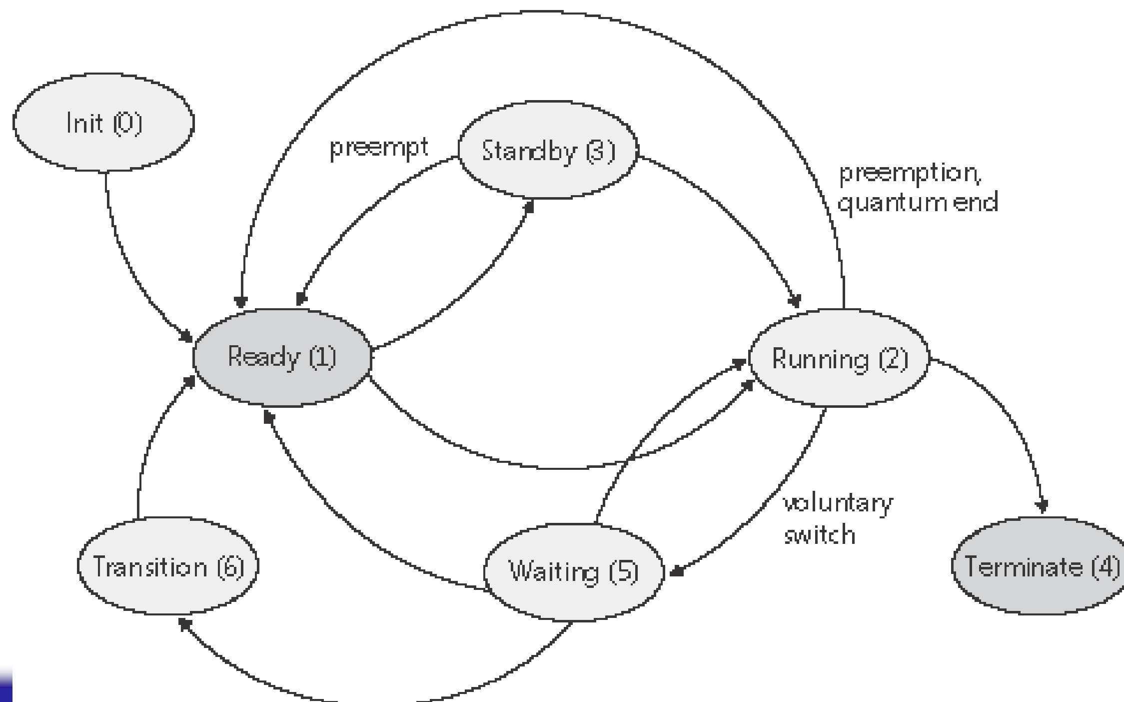
- Threads (suite)
  - Un thread s'exécute en User Mode mais peut passer en Kernel Mode à la suite d'un appel système
    - Il continue son exécution avec le même ID et les même restrictions
  - Chaque thread possède 2 piles, une pour User mode et l'autre pour Kernel mode
  - Peut posséder son propre jeton d'accès
    - Remplace celui du processus parent
  - Quand le dernier thread termine, le processus s'arrête
  - Les threads peuvent accéder à n'importe quelle ressource de leur processus parent
    - Y compris les ressources crées par d'autres threads

Concepts

- Switcher de *thread* est coûteux
  - Implique un passage de User en Kernel
- Introduction des *fibers*
  - Thread léger
  - Schedulés en User Space par le programme qui les a crées
  - Le kernel ne connaît pas les fibres
  - Création par des appels Win32

États des threads

- *Ready*
- *Standby* : le thread est été choisi pour être exécuté. Un seul *thread* dans cet état par processeur. Ne peut être préempté avant d'avoir commencé une exécution
- *Running*
- *Waiting*
- Transition Un *thread* est dans cet état si il est prêt mais sa pile kernel est dans le swap
- *Terminated*
- *Initialized* Utilisé en interne lors de la création d'un thread



Scheduling

- WinXP n'a pas de scheduler centralisé
- Quand un thread ne peut plus continuer à s'exécuter, il passe en kernel mode et exécuter lui-même le scheduler
- Conditions amenant à l'exécution du scheduler par le thread courant
  - Blocage sur un sémaphore, I/O…
    - Thread en mode kernel, exécution du sémaphore
  - Signal sur un objet (libération d'un sémaphore…)
    - Le thread pourrait continuer, mais doit vérifier qu'un thread de plus haute priorité n'a pas été réveillé
  - Fin de son quantum de temps
    - Basculement en mode kernel
    - Le thread peut obtenir un nouveau quantum et continuer

Scheduling

- Autres conditions
  - Fin d'une opération I/O
  - Timeout d'un *wait*
- Indication de priorité
  - Un processus a 2 façons d'influer sur la priorité
  - *SetPriorityClass*
    - Classe de priorité de tous les threads (*realtime, high, above normal, normal, below normal, idle*)
  - *SetThreadPriority*
    - Priorité relative d'un thread par rapport aux autres du même processus
    - *time critical, highest, above normal, normal, below normal, lowest, idle*)
  - 42 Combinaisons possibles

Scheduling

- Le système a 32 priorités (0-31)
  - Les 42 priorités des threads sont mappées sur les 32 priorités système
  - Permet de déterminer la priorité de base du thread (*thread base priority*)
- Chaque thread possède également une priorité courante (*current priority*)
  - Peut être supérieur à la priorité de base mais jamais inférieur

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Scheduling

- Le système maintient une table de 32 listes de *threads* prêts
- L'algorithme de scheduling parcourt la table en ordre inverse
  - Si un *thread* est trouvé, il est mis sur le CPU pour un quantum
  - Quand le quantum expire, le thread est replacé en fin de la liste de sa priorité
- Les priorités sont découpées en groupes
  - -1 : idle
  - 0 : utilisé pour effacer les cadres non utilisés (sécurité)
  - 1-15 : priorités dynamiques
  - 16-31 : priorités temps réel (pas vraiment)
    - Réservées aux threads du système ou données explicitement par l'administrateur

Augmentation de priorité

- Si un thread a une priorité dynamique, le système peut temporairement l'augmenter
  - Mais ne sera jamais augmentée jusqu'au niveau temps réel
- Conditions pour augmentation de priorité
  - Un thread qui attendait sur une I/O
    - Lui donne une chance de redémarrer rapidement pour refaire une I/O
    - Augmentation dépend du périphérique (1 pour disque, 6 pour clavier…)
  - Fin d'attente sur un sémaphore, mutex…
    - +2 si application au premier plan
    - +1 sinon
  - Un thread appartient à une application qui vient de passer en premier plan
    - La priorité devient supérieur à la priorité des threads en arrière plan
  - Un thread appartient à une fenêtre qui vient de recevoir un évenement

Augmentation de priorité

- Un thread qui atteint la fin de son quantum a sa priorité courante diminuée de 1
  - Jusqu'à atteindre la priorité de base
- Un thread qui n'a pas eu le CPU depuis longtemps passe en priorité 15 pour 2 quanta
  - Retour à son ancienne priorité courante ensuite
- Quantum
  - 20ms pour un XP Pro
  - 120ms pour un XP Serveur

Inversion de priorité

- On considère 3 threads
  - T1 haute priorité
  - T2 basse priorité, en section critique
  - T3 priorité moyenne, toujours prêt
- Si T1 attend une ressource verrouillée par T2
  - T1 ne peut pas s'exécuter
  - T3 ne peut pas s'exécuter (trop basse priorité)
  - T2 s'exécute
- T1 ne pourra jamais s'exécuter malgré sa haute priorité
- Solution : Inversion de priorité
  - Le scheduler augmente aléatoirement la priorité d'un thread prêt

Multiprocesseur

- Machine SMP
- Scheduling similaire au cas mono processeur
  - Des précautions dans le noyau pour la gestion des structures critiques
- Vraie concurrence entre les threads
- *Thread Affinity*
  - Force le scheduler a mettre certains threads sur un processeur donné
  - Peut nuire aux performances (ou pas)
- *Thread Ideal Processor*
  - Indique au scheduler un ensemble de processeurs à utiliser pour un thread
  - Juste une indication

Pour aller plus loin

Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000

By Mark E. Russinovich, David A. Solomon