Faculté des Sciences

UMONS CHARLEROI

Université de Mons

Faculté des Sciences

US-MC-INFO60-014-C — Systèmes d'exploitation

Synthèse:
Systèmes d'exploitation

Auteur:
Baptiste Grosjean

Directeur:
Seweryn Dynerowicz

Numéro d'étudiant:
232732

Résumé

Juin 2024

Sommaire

1 Introduction

1.1 Vue d'ensemble

Un système d'exploitation est constitué d'un noyau auquel vient s'ajouter le code d'une distribution. Le développeur d'un système d'exploitation doit abstraire le fonctionnement des périphériques afin d'offre un interface de programmation au programmeur. La gestion des ressources et les communications avec les différents périphériques est gérée par le système d'exploitation.

1.2 Abstractions de programmation

Un système d'exploitation peut être perçu comme exposant une interface de programmation système aux programmes qui tournent sur une machine.

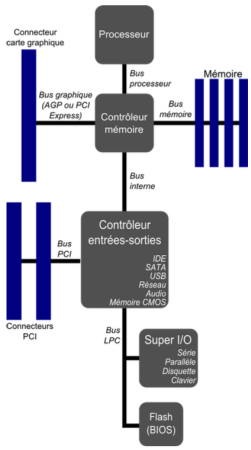

Figure 1: View

1.3 Processeur

Le **processeur** exécute les instructions et réalise les traitements. Il est caractérisé par son jeu d'instructions (Instruction Set Architecture), qui définit les instructions exécutables et leur format d'encodage (opcode et opérandes). Les instructions sont classées en plusieurs types :

- Arithmétique (e.g. `add`, `div`)
- Logique (e.g. `and`)
- Accès mémoire (e.g. `lw`, `sw`)
- Instructions système

Les processeurs modernes utilisent une structure de pipeline, permettant à différentes instructions d'occuper simultanément différentes parties du pipeline. Deux types principaux d'architecture de jeu d'instructions existent :

- RISC (Reduced Instruction Set Computer)
- CISC (Complex Instruction Set Computer)

Les responsabilités de gestion du processeur en collaboration avec le système d'exploitation incluent :

- Surveillance de l'exécution des instructions
- Répartition du temps processeur entre différents programmes

1.3.1 Exécution de programme

La première responsabilité de gestion consiste à surveiller l'exécution des instructions d'un programme utilisant le processeur. Le système d'exploitation et les programmes se partagent le processeur, nécessitant la suspension de l'exécution en cours lors d'un évènement particulier. Le processeur peut réagir de différentes manières :

- **Mauvais comportement** : déroutement vers une fonction du système d'exploitation pour traiter des erreurs telles qu'une division par zéro, un accès mémoire invalide ou une tentative d'exécuter une instruction privilégiée.
- **Anomalie non fatale** : correction des circonstances ayant provoqué l'anomalie et reprise de l'exécution.
- **Appel système** : branchement vers une fonction du système d'exploitation pour des demandes comme l'obtention de mémoire ou l'écriture dans un fichier.
- **Interruption externe** : traitement d'une interruption provenant d'un périphérique externe. Par exemple, une interruption du clavier nécessite de communiquer avec le contrôleur du clavier pour identifier la touche concernée et le programme destinataire.

1.3.2 Ordonnancement de programmes

La seconde responsabilité du système d'exploitation est de répartir le temps processeur entre les programmes. L'ordonnancement sélectionne le prochain programme à exécuter lorsque le processeur est libre. Différentes stratégies sont utilisées selon le type de système et la charge de travail. Les métriques d'évaluation incluent la minimisation du temps d'exécution moyen et l'adéquation entre les besoins du programme et le temps processeur alloué.

1.4 Mémoire

La mémoire stocke les données de travail des programmes et est organisée en une hiérarchie :

- **Caches** (e.g. L1, L2) : Intermédiaires entre les registres et la mémoire principale, stockant temporairement les données fréquemment accédées (principe de localité).
- **Mémoire principale** (e.g. RAM, GPU) : Stocke les données de travail (e.g. tableaux, variables, stack) lorsqu'il n'y a pas de registres disponibles.
- **Mémoire secondaire** (e.g. HDD, SSD) : Utilisée comme espace temporaire (swap) lorsque la mémoire principale est pleine.

L'allocation dynamique permet de gérer la mémoire de manière flexible. La mémoire virtuelle charge/décharge les données non actives pour pallier les limitations de la mémoire physique (e.g. adressage 32 bits, 0x00000000 à 0xffffffff).

1.4.1 Protection de la mémoire

Le système d'exploitation contrôle les accès mémoire pour garantir qu'un programme n'accède qu'à sa propre mémoire, évitant ainsi la lecture ou la modification de données sensibles (e.g. clefs de chiffrement). Le processeur vérifie les accès mémoire selon les descriptions des espaces mémoire accessibles à chaque programme, mises à jour par le système d'exploitation.

- **Adresse valide** : Accessible au programme, l'accès est accordé (e.g. `lw`, `sw`).
- **Adresse indisponible** : Accessible mais non présente en mémoire principale, nécessite un chargement depuis la mémoire secondaire.
- **Adresse invalide** : Hors de l'espace accessible, entraîne l'arrêt du programme.

1.4.2 Répartition de la mémoire

La seconde responsabilité de gestion est la répartition de la mémoire principale entre les différents programmes. Cela inclut la représentation et le suivi des portions de mémoire (libres ou non) et l'identification d'une portion adéquate pour satisfaire une demande d'allocation dynamique.

1.5 Stockage

Le stockage préserve les données produites par un programme au-delà de son exécution. Une fois le programme terminé, sa mémoire est libérée et peut être allouée à un autre programme. La notion de fichier assure la pérennité des données, malgré la diversité des supports physiques adaptés à différents scénarios d'utilisation :

- **HDD** : Disque dur, bon marché, accès lent, organisé en plateaux, cylindres, têtes, et secteurs.
- **SSD** : Disque à état solide, rapide, coût plus élevé, organisé en blocs et pages.
- **Bande magnétique** : Grande capacité, fiable, utilisé pour les backups.

Un système de fichiers met en place une abstraction pour accéder au stockage, en représentant les fichiers et répertoires et leurs méta-données (e.g. taille, dates, permissions). Il garantit également la cohérence des écritures et réduit la possibilité de corruption des données.

1.6 Typologie des systèmes

Les types de systèmes influencent la conception des systèmes d'exploitation :

- **Mainframe** : Exécute des jobs réguliers avec des E/S intensives (e.g. calcul des fiches de paie). Supporte un grand volume de transferts et offre une gestion de timesharing pour plusieurs utilisateurs.
- **Serveur** : Fournit des services aux utilisateurs distants (e.g. serveur d'impression), gère les ressources et la facturation.
- **Personal Computer** : Interaction humaine via interface graphique, exécution simultanée de plusieurs programmes, interactivité centrale (mesurée par la latence entre l'action et la réaction).
- **Système mobile** (e.g. tablette, smartphone) : Contraintes de ressources, économie d'énergie cruciale, processeur et mémoire limités, alimentation par batterie.
- **Système embarqué** (e.g. TV, voiture) : Logiciel prédéterminé pour une machine spécifique, pas de flexibilité pour l'installation d'applications supplémentaires.
- **Système de senseurs** : Surveille des paramètres environnementaux (e.g. température, humidité), transmet les données à un central, optimisé pour faible consommation d'énergie.
- **Système temps-réel** : Respect des échéances crucial, priorisation des tâches, gestion précise du temps pour aligner les traitements avec l'environnement.

1.7 Architecture

Un système d'exploitation inclut le code, les données, et la stack, similaires à un programme classique. Il utilise des instructions système nécessitant des niveaux de privilège spécifiques, et le noyau gère les ressources.

- **Noyau monolithique** : Code dans un fichier unique, exécuté au plus haut niveau de privilège. Exemple : système avec configuration matérielle fixe.
- **Noyau modulaire** : Base toujours en mémoire, modules chargés/déchargés selon les besoins. Exemple : Linux.
- **Microkernel** : Composantes exécutées à des niveaux de privilège réduits pour éviter les compromissions. Un serveur de ré-incarnation gère les erreurs fatales. Exemple : MINIX3.
- **Exokernel** : Gestion des ressources laissée aux programmes, le noyau assure uniquement la protection et l'isolation. Exemple : machines virtuelles.

2 Introduction2

2.1 Vue d'ensemble

Le système d'exploitation (SE) est le logiciel de base qui gère les ressources matérielles et logicielles d'un ordinateur, offrant des services essentiels pour l'exécution des programmes. Un SE permet aux utilisateurs et aux applications d'interagir avec le matériel sans nécessiter de connaître les détails techniques sous-jacents.

2.2 Rôles et fonctions du système d'exploitation

Les principales fonctions d'un SE incluent :

- **Gestion des processus** : Supervise la création, l'exécution, la suspension et la terminaison des processus. Il gère également les threads et les mécanismes de synchronisation.
- **Gestion de la mémoire** : Alloue et libère de la mémoire, gère la mémoire virtuelle, et s'assure que chaque processus a un espace mémoire isolé et protégé.
- **Gestion des entrées/sorties (I/O)** : Coordonne et contrôle les opérations d'entrée/sortie des périphériques, utilisant des pilotes pour assurer la communication entre le matériel et les logiciels.
- **Gestion des fichiers** : Gère les fichiers sur les différents systèmes de stockage, offrant des opérations de lecture, écriture, création, et suppression de fichiers.
- **Sécurité et protection** : Protège les données et les ressources contre les accès non autorisés et les défaillances, assurant l'intégrité et la confidentialité des informations.

2.3 Typologie et architecture des systèmes d'exploitation

Il existe différents types de SE, chacun adapté à des environnements et des usages spécifiques :

- **Systèmes batch** : Traitent les lots de tâches sans interaction utilisateur pendant l'exécution.
- **Systèmes temps partagé** : Permettent à plusieurs utilisateurs d'utiliser le système simultanément en partageant le temps processeur.
- **Systèmes temps réel** : Offrent des réponses immédiates aux événements externes, essentiels pour les applications critiques.
- **Systèmes embarqués** : Intégrés dans des appareils pour des tâches spécifiques, optimisés pour des contraintes de ressources limitées.

2.4 Défis de la conception des systèmes d'exploitation

La conception d'un SE implique de nombreux défis, notamment :

- **Complexité du matériel** : La gestion de divers composants matériels nécessite des abstractions et des interfaces standardisées.
- **Multiplicité des utilisateurs et des tâches** : Assurer une utilisation équitable et efficace des ressources tout en isolant les utilisateurs et les tâches pour éviter les interférences.
- **Sécurité et fiabilité** : Protéger le système contre les attaques et les défaillances tout en maintenant une performance stable.
- **Évolutivité** : Adapter le SE aux nouveaux matériels et aux besoins croissants des utilisateurs sans compromettre les performances.

2.5 Conclusion

Les systèmes d'exploitation sont des composants essentiels des systèmes informatiques, assurant la gestion et la coordination des ressources matérielles et logicielles. Une compréhension approfondie de leurs fonctions, types et défis est cruciale pour les étudiants en sciences informatiques, permettant de développer des applications efficaces et sécurisées.

3 Evénements

3.1 Composition et exécution de programme

Un programme est composé de trois parties principales :

- **Code machine** : Instructions représentant la logique des traitements.
- **Espace de données** : Contient les données manipulées par le programme (variables globales, tableaux, variables dynamiques obtenues via `malloc`).
- **Stack** : Héberge les données de travail courantes (variables locales) et les informations d'appels de fonctions imbriquées.

Ces composantes sont liées par des adresses, utilisées pour les branchements, appels et retours de fonction, accès aux cases d'un tableau, et manipulations de la stack. Pour s'exécuter, elles doivent être placées en mémoire principale.

Le processeur suit un cycle *fetch-decode-execute* :

1. **Fetch** : Récupération de l'instruction à exécuter depuis la mémoire.
2. **Decode** : Décodage des détails de l'instruction pour déterminer le traitement.
3. **Execute** : Exécution des opérations requises par l'instruction
   (a) Mise à jour des registres, de la stack, des données.
   (b) Mise à jour (incrément) du Pointeur d'Instruction

L'*instruction pointer* (ou *program counter*) contient l'adresse de l'instruction courante et est mis à jour après chaque instruction (soit par incrémentation, soit par branchement). L'état du programme à un moment donné est déterminé par :

- Les valeurs des registres
- Le contenu de la stack
- Le contenu de l'espace de données
- L'*instruction pointer*

Une sauvegarde de ces éléments représente un instantané de l'état du programme, permettant de reprendre l'exécution là où elle s'est arrêtée.

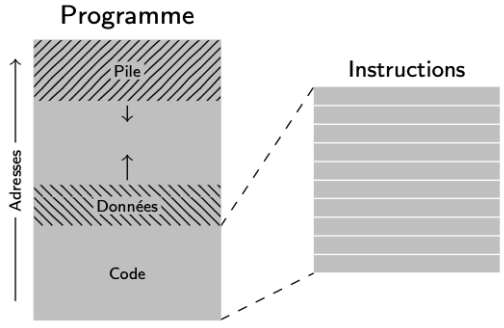

Figure 2: Program view

3.2 Concept de base – évènement

Lorsqu'un programme est exécuté par le processeur, deux questions se posent :

- **Erreur d'instruction** : Doit-on redémarrer la machine ou gérer l'anomalie pour limiter les dégâts ?
- **Activité de périphérique externe** (e.g. frappe au clavier) : Peut-on suspendre temporairement le programme en cours pour traiter l'activité ?

Ces situations sont résolues par la gestion des **évènements**, mécanismes permettant au processeur de brancher vers du code spécifique en réponse à un évènement.

Les exceptions et interruptions sont des mécanismes implémentés par le processeur permettant d'effectuer un branchement en réaction à un événement.

Les différents types d'événements auxquels un OS peut être confronté sont définis par l'architecture du processeur qui les exécute.

Le système d'exploitation dispose d'un gestionnaire d'événements qui prend le relais en cas d'interruption ou d'exception lors de l'exécution d'un programme.

3.3 Privilèges

La vérification du niveau de privilèges est nécessaire pour :

1. Modifier la configuration des interruptions/exceptions
2. Modifier la configuration de la mémoire
3. Modifier le niveau de privilège courant

Dans une architecture x86, il ya 4 anneaux de protection

1. Ring 0: Noyau du os
2. Ring 1 & 2 : Services de l'OS
3. Ring 3 : Programmes

Dans une architecture MIPS, il y a 2 modes de fonctionnement:

- Kernel Mode : os
- User Mode : programmes

3.4 Typologie

3.4.1 Exceptions (évènements synchrones)

Origine interne, résultant de l'exécution d'une instruction problématique :

- **Code opératoire invalide** : Opcode non reconnu par le processeur.
- **Division par zéro** : Nécessité d'un dénominateur non-nul.
- **Instruction privilégiée** : Niveau de privilège insuffisant pour exécuter l'instruction.
- **Instruction non-chargée en mémoire** : Le système d'exploitation charge la partie manquante en mémoire.

3.4.2 Interruptions (évènements asynchrones)

Origine externe, provoquées par un signal de périphérique :

- **Frappe de clavier** : Fermeture d'un circuit électronique, générant une interruption.
- **Fichier prêt pour lecture** : Le contrôleur de disque génère une interruption après lecture des données.
- **Réception d'un paquet réseau** : La carte réseau génère une interruption après réception complète d'un paquet.
- **Erreur du matériel** : Anomalies détectées (e.g. température excessive).
- **Signal d'horloge** : Interruption générée 32.768 fois par seconde pour suivre le temps.

3.4.3 Traitements

Multiplicité des événements (interruptions) => priorité Evenements pendant le traitement d'évenements (exceptions) => Réantrance et préemption

3.5 Conséquences possibles pour le programme

Le système d'exploitation doit gérer les évènements survenus pendant l'exécution d'un programme. Selon la nature de l'évènement, deux décisions peuvent être prises :

- **Évènement récupérable** : Après traitement, il est possible de continuer l'exécution du programme.
- **Évènement irrécupérable** : La faute ne peut être corrigée, et il est nécessaire de terminer le programme.

3.5.1 Evenement récupérable

Lorsqu'un évènement récupérable survient, il est crucial de préserver l'état du processeur. Le gestionnaire d'évènements doit sauvegarder le contexte du programme suspendu (registres et stack) avant de traiter l'évènement. Cette sauvegarde est similaire à celle d'un appel de fonction.

1. Sauvegarde
2. Traitement
   - Reprise
     (a) Identifier
     (b) Charger
   - Poursuite
     (a) Récupérer le code
     (b) Notifier
     (c) Transfert du code vers le programme
3. Restauration
4. La continuation peut être :
   - **Reprise** : Retenter l'exécution de l'instruction qui a causé l'évènement. A l'adresse de l'instruction corrélée à cet événement.
   - **Poursuite** : Passer à l'instruction suivante. A l'adresse de l'instruction suivante.

Les interruptions sont généralement des évènements récupérables avec poursuite, car elles n'affectent pas l'instruction en cours. Les exceptions comme les appels systèmes (e.g. `syscall`, `int`) sont également des évènements récupérables avec poursuite.

Par exemple, une interruption causée par une frappe au clavier est traitée en lisant le code de la touche depuis un port d'entrée/sortie, le traduisant en caractère, puis le transférant au programme destinataire, souvent la fenêtre active dans une interface graphique.

3.5.2 Evenement irrécupérable

Lorsqu'un évènement irrécupérable survient, le programme ne peut plus continuer. Une sauvegarde complète du contexte d'exécution est effectuée pour le débogage.

Le programme acquiert diverses ressources (mémoire, descripteurs de fichiers, sockets) au cours de son exécution. Ces ressources doivent être libérées lorsque le programme est terminé.

1. Sauvegarde
2. Libération des ressources
3. Traitement Les étapes de gestion d'un évènement irrécupérable incluent :
   - Affichage d'une notification à l'utilisateur décrivant le problème rencontré.
   - Sauvegarde du contexte dans un fichier (e.g. `coredump`) ou envoi par mail à un serveur de rapports de crash.
   - Libération du processeur, permettant la continuation d'un autre programme via un *context switch*.
4. Abandon

3.6 Mécanismes de déroutement

Chaque architecture de processeur spécifie et implémente son mécanisme de déroutement pour la gestion des évènements. Deux techniques principales sont utilisées :

- **Déroutement avec registre de cause** : Un registre indique la cause de l'évènement, permettant au gestionnaire d'évènements de déterminer l'action appropriée.
- **Vectorisation** : Différents types d'évènements sont associés à des adresses spécifiques dans une table de vecteurs, chaque adresse pointant vers le code de gestion correspondant.

La technique utilisée influence l'organisation et les responsabilités du code du gestionnaire d'évènements.

3.6.1 Registre de cause

Le registre de cause encode les évènements survenus lors du cycle d'exécution courant, mettant à jour les bits pour les exceptions et interruptions. Si au moins un bit est positionné, le processeur réalise un déroutement vers une adresse précise.

Le gestionnaire d'évènements doit :

- Lire le contenu du registre de cause.
- Tester et traiter les bits des interruptions en attente (bits à 1).
- Traiter les exceptions stipulées par le code d'exception.

Cette technique permet au système d'exploitation de déterminer l'ordre de priorité des traitements des différents évènements. La fonction principale du gestionnaire d'évènements doit être placée à l'adresse de déroutement définie par le processeur au démarrage du système.

Pour permettre la continuation du programme, l'adresse de l'indicateur d'instruction courante est sauvegardée dans un registre dédié (e.g. Exception Program Counter en MIPS32).

3.6.2 Vectorisation

Dans la vectorisation, chaque type d'évènement est associé à un numéro unique (e.g. interrupt vector). Ce numéro identifie une entrée dans une table de descripteurs (e.g. interrupt descriptor table, IDT) qui référence la fonction de traitement spécifique à invoquer (interrupt service routine, ISR).

À la fin d'un cycle d'exécution, le processeur :

- Consulte les numéros d'évènements survenus.
- Invoque les fonctions de gestion des évènements correspondantes dans un ordre déterminé.

Le code du gestionnaire d'évènements se concentre uniquement sur l'évènement correspondant. Au démarrage du système, l'OS initialise le contenu de la table des descripteurs et la référence au processeur via une instruction privilégiée (e.g. `lidt` en x86).

Lorsqu'un évènement survient :

- Le numéro d'évènement sert d'indice dans l'IDT.
- Le descripteur correspondant (interrupt gate) contient l'adresse de la fonction de gestion vers laquelle brancher (offset).

3.6.3 Niveaux de privilège

Le jeu d'instructions d'un processeur inclut toutes les instructions reconnaissables par son architecture, comprenant les instructions arithmétiques, logiques et systèmes. Les instructions systèmes modifient la configuration du processeur et doivent être exécutées uniquement par le système d'exploitation.

Un niveau de privilège contrôle quelles instructions peuvent être exécutées et quand. La spécification du processeur décrit la dynamique de changement de ce niveau de privilège et son utilisation pour les vérifications.

3.6.4 Basculement de stack

Lorsqu'un évènement survient, le processeur déroute vers une fonction du système d'exploitation, modifiant le contenu des registres et utilisant une stack pour stocker les valeurs temporaires. Il est crucial de décider si cette stack sera celle du programme suspendu ou une stack dédiée du système d'exploitation.

Le processeur peut basculer vers une stack du système d'exploitation, en sauvegardant les informations nécessaires pour explorer la stack du programme suspendu (e.g. stack pointer). Par exemple, lors d'un appel système, les arguments peuvent être passés via la stack, et le système d'exploitation doit pouvoir les localiser.

Après le traitement de l'évènement, la stack courante redevient celle du programme. La figure 1.11 illustre ce processus :

- Avant le déroutement : `sp(pgm)`
- Après le déroutement : `sp(os)`, contenant :
  - Sauvegarde du pointeur de stack précédent `sp(pgm)`
  - Indicateur d'instruction courante `ip(pgm)`
  - Éventuel code d'erreur décrivant l'évènement

Ces informations permettent de revenir au programme pour une éventuelle continuation.

3.7 Multiplicité des occurences

Il est possible que plusieurs évènements se produisent pendant un même cycle d'exécution du processeur. Les règles d'agrégatioons entre événements sont les suivantes :

- **Reprise** l'emporte sur une **poursuite**.
- **Abandon** l'emporte sur une **reprise**.

Lors de deux évènements similaires (deux poursuites, deux reprises, ou deux abandons), la conséquence est respectivement une poursuite, une reprise, ou un abandon.

Pour des évènements de types différents :

- **Reprise et poursuite** : La reprise l'emporte, les deux traitements doivent être réalisés. Si l'évènement avec poursuite est une interruption, il peut être traité en priorité.
- **Poursuite et abandon** : L'abandon l'emporte. Si l'évènement avec poursuite est une exception, son traitement n'est pas nécessaire. Si c'est une interruption, il doit être traité.
- **Reprise et abandon** : L'abandon l'emporte. Le traitement de l'évènement avec reprise n'est pas nécessaire car le programme ne continuera plus.

3.8 Gestion ré-entrante des évènements

Un évènement peut survenir pendant n'importe quel cycle d'exécution, y compris pendant la gestion d'un autre évènement (exception ou interruption). Le gestionnaire d'évènements doit être adapté pour gérer cette imbrication correctement.

Un gestionnaire ré-entrant permet de gérer de nouveaux évènements survenant pendant l'exécution du gestionnaire actuel. Les considérations sont similaires à la gestion des appels de fonctions récursives.

Un gestionnaire d' évènements réentrant permet aux évènements survenant pendant certaines phases de son exécution de provoquer un déroutement.

=> Focntion récursive

1. Sauvegarde
   (a) Désactivation des interruptions
   (b) Empilement du contexte
   (c) Empilement de l'adresse corrélée
   (d) Réactivation des interruptions
2. Restauration
   (a) Désactivation des interruptions
   (b) Dépilement de l'adresse corrélée
   (c) Dépilement du contexte
3. Continuation
   (a) Préparation de l'adresse de retour
   (b) Retour & réactivation des interruptions

Pour garantir une sauvegarde cohérente des contextes imbriqués :

- Utiliser la stack pour sauvegarder le contexte du gestionnaire d'évènements suspendu.
- Désactiver les interruptions pendant les phases critiques (sauvegarde du contexte, restauration du contexte, continuation), mettant ainsi en attente les interruptions.
- Préserver l'adresse de l'instruction corrélée à chaque niveau d'imbrication.

3.9 Priorités et masques d'interruptions

Dans un système typique, il est crucial de respecter l'ordre de priorité des interruptions. Un gestionnaire ré-entrant seul ne suffit pas pour éviter l'inversion de priorité, où une interruption de basse priorité pourrait interrompre le traitement d'une interruption de haute priorité.

Un gestionnaire d'évènement préemptif garantit qu'une interruption de haute priorité est traitée avant toute interruption du plus basse priorité en cours ou en attente de traitement.

La solution consiste à utiliser un masque d'interruptions, permettant une granularité dans l'activation et la désactivation des interruptions. Ce masque contrôle quelles interruptions sont actives (ou inactives) en utilisant un bit par type d'interruption. Le masque est combiné avec les bits des interruptions en attente à la fin de chaque cycle d'exécution.

=> l'un à la suite de l'autre Lors du traitement d'une interruption, le gestionnaire :

- Positionne les bits du masque pour autoriser uniquement les interruptions plus prioritaires.
- Après traitement, restaure l'état du masque à son état précédent.

4 Mémoire

Un programme c'est donc 3 composantes essentielles:

1. Code machine: instructions représentant la logique de traitement

2. Espace de données (RAM): contient les données manipulées par le programme , incluant variables statiques (globales, tableaux) et dynamiques (via malloc)

3. Stack (espace de travail dédié ecrire et effacer): héberge les données de travail courantes telles que les variables locales et les inforamtions d'appel de fonctions imbriquées pour gérer les retours

Ces éléments sont liés par des adresses représentant la logique du programme (branchements, appels et retours de fonctions, accès aux tableaux et manipulation de stack).

Cablage selon une logique interne

1. branchement

2. Appel retour de fonction

3. Référence à des tableaux

4. Manipulation de la stack

Le programme est transformé en mots binaires représentant les instructions et les données via adresses absolues et relatives.

Lors de son exécution, le programme est en mémoire principale et un système de projection des adresses permet de maintenir son éxécution correcte.

Gestion de la mémoire :

- Projection : transformer le adresses du programme en adresses physiques disponibles

- Protection : restreindre l'accès mémoire pour protéger les données.

- Gestion de la mémoire physique : représenter et suivre l'état de la mémoire en fonction des besoins des programmes

4.1 Projection et protection

4.1.1 Adressage absolu

Une adresse absolue désigne un emplacement spécifique dans la mémoire principale, identifiant une instruction ou une donnée précise.

4.1.2 Adressage relatif

Une adresse relative représente un décalage par rapport à un point de référence dans la mémoire principale.

4.1.3 Segmentation

4.1.4 Pagination

4.2 Gestion de la mémoire physique

4.2.1 Bitmpas

4.2.2 Listes chaînées

5 Mémoire2

5.1 Problématique

La gestion de la mémoire est essentielle pour assurer l'efficacité et la stabilité des systèmes informatiques. Elle implique la répartition de la mémoire entre les différents processus et la protection de l'espace mémoire alloué. Les défis incluent la gestion de la fragmentation, la protection de la mémoire et l'allocation dynamique.

5.2 Hiérarchie de la mémoire

La mémoire d'un système informatique est organisée en une hiérarchie :

- **Registres** : Mémoire la plus rapide, utilisée pour stocker les données temporaires.
- **Caches (L1, L2)** : Stockent les données fréquemment utilisées pour réduire les temps d'accès.
- **Mémoire principale (RAM)** : Stocke les données et les programmes en cours d'exécution.
- **Mémoire secondaire (HDD, SSD)** : Utilisée pour le stockage persistant des données.

5.3 Allocation de la mémoire

L'allocation de la mémoire peut se faire de différentes manières :

- **Allocation fixe** : Les blocs de mémoire ont une taille fixe, ce qui peut conduire à une fragmentation interne.
- **Allocation dynamique** : La mémoire est allouée et libérée au besoin, ce qui nécessite une gestion plus complexe mais réduit la fragmentation.

5.4 Mémoire virtuelle

La mémoire virtuelle permet à un système d'exécuter des programmes qui nécessitent plus de mémoire que celle disponible physiquement. Elle utilise des mécanismes de pagination et de segmentation pour gérer l'allocation et la protection de la mémoire. La mémoire virtuelle permet de charger et décharger les portions de mémoire selon les besoins du programme.

5.5 Protection de la mémoire

La protection de la mémoire est cruciale pour empêcher un programme d'accéder à la mémoire allouée à un autre programme ou au système d'exploitation lui-même. Les techniques incluent :

- **Protection par segmentation** : Divise la mémoire en segments distincts avec des permissions d'accès spécifiques.
- **Protection par pagination** : Utilise des pages de mémoire avec des permissions d'accès contrôlées par des tables de pages.

5.6 Gestion de la fragmentation

La fragmentation de la mémoire peut être interne ou externe :

- **Fragmentation interne** : Se produit lorsque des blocs de mémoire ont une taille fixe et que les allocations ne remplissent pas complètement ces blocs.
- **Fragmentation externe** : Se produit lorsque des blocs de mémoire de tailles variables sont alloués et libérés, créant des espaces de mémoire inutilisables.

5.7 Stratégies de gestion de la mémoire

Plusieurs stratégies peuvent être utilisées pour gérer efficacement la mémoire :

- **Bitmaps** : Utilisent une série de bits pour suivre l'état de chaque unité d'allocation (libre ou allouée).
- **Listes chaînées** : Maintiennent des listes chaînées des blocs de mémoire libres et alloués pour faciliter la gestion dynamique.

5.8 Conclusion

La gestion de la mémoire est un aspect fondamental des systèmes d'exploitation, impactant directement la performance et la stabilité du système. Une compréhension approfondie des techniques d'allocation, de la hiérarchie de la mémoire et des mécanismes de protection est essentielle pour optimiser l'utilisation de la mémoire dans un environnement informatique.

6 Processus

6.1 Constitution

6.1.1 Table des processus

6.1.2 Cycle de vie

6.2 Ordonnancement

6.2.1 Systèmes batch

Les systèmes batch exécutent des lots de travaux sans interaction avec avec l'utilisateur pendant l'exécution.

Algorithmes

- First-Come, First-Served (FCFS): Les processus sont exécutés dans l'ordre de leur arrivée.
- Shortest Job Next (SJN): Les processus avec le plus court temps d'exécution sont prioritaires

6.2.2 Systèmes interactifs

Les systèmes interactifs permettent une interaction continue avec l'utilisateur, nécessitant des temps de réponse courts.

Algorithmes:

- Round Robin: chaque processus reçoit un temps de CPU fixe, appelé quantum, et est ensuite placé à la fin de la file d'attente.
- Priority scheduling: les processus sont exécutés en fonction de leur priorité.

6.2.3 Systèmes en temps réel

Les systèmes en temps réels doivent répondre à des contraintes temporelles strictes, souvent utilisés dans des applications critiques.

Algorithmes:

- Rate Monotonic Scheduling (RMS): les processus avec les périodes les plus courtes ont la plus haute priorité
- Earliest Deadline First (EDF) : Les processus sont exécutés en fonction de leur échéance la plus proche

6.3 Multi-threading

7 Processus2

7.1 Problématique

Un processus est une instance d'un programme en cours d'exécution, comprenant son code, ses données et ses ressources système allouées. La gestion des processus est essentielle pour assurer l'exécution efficace et sécurisée des programmes sur un système informatique.

7.2 Constitution d'un processus

Un processus se compose de plusieurs éléments :

- **Espace d'adressage** : Comprend le code du programme, les données statiques, la pile et le tas.
- **Descripteur de processus** : Contient l'identifiant unique (PID), l'état, les registres, le pointeur de pile, et les descripteurs de fichiers ouverts.

7.3 Cycle de vie d'un processus

Le cycle de vie d'un processus inclut plusieurs états :

- **Création** : Un processus parent crée un processus enfant via un appel système tel que fork().
- **Exécution** : Le processus utilise le processeur pour exécuter les instructions.
- **Prêt** : Le processus est en attente de l'allocation du processeur.
- **Bloqué** : Le processus attend qu'une condition soit remplie (e.g., fin d'I/O).
- **Terminaison** : Le processus libère ses ressources et passe à l'état zombi jusqu'à ce que son descripteur soit nettoyé par le processus parent.

7.4 Ordonnancement des processus

L'ordonnancement est la méthode utilisée par le système d'exploitation pour attribuer du temps processeur aux processus prêts à s'exécuter. Les principales politiques d'ordonnancement incluent :

- **First-Come, First-Served (FCFS)** : Les processus sont exécutés dans l'ordre de leur arrivée.
- **Round-Robin** : Chaque processus reçoit un quantum de temps fixe pour s'exécuter tour à tour.
- **Shortest Job Next (SJN)** : Le processus avec le temps d'exécution le plus court est exécuté en premier.

7.5 Multi-threading

Le multi-threading permet à un processus de contenir plusieurs threads d'exécution, ce qui améliore l'efficacité et la réactivité. Les threads partagent l'espace d'adressage du processus mais possèdent des contextes d'exécution indépendants.

7.6 Gestion des événements et des interruptions

Les interruptions permettent de gérer des événements asynchrones (e.g., I/O). Lorsqu'une interruption se produit, le système d'exploitation sauvegarde le contexte du processus courant et exécute le gestionnaire d'interruption approprié.

7.7 Conclusion

La gestion des processus est fondamentale pour le fonctionnement d'un système d'exploitation, impactant directement l'efficacité et la stabilité des systèmes informatiques. Une bonne compréhension de la constitution des processus, de leur cycle de vie, des stratégies d'ordonnancement et du multi-threading est essentielle pour développer des applications performantes et robustes.

8 Concurrence

8.1 Problématique

La concurrence en informatique permet d'exécuter plusieurs processus ou threads simultanément, augmentant ainsi la réactivité et l'efficacité des systèmes. Toutefois, cette parallélisation introduit de nouveaux défis, notamment les conditions de course, l'exclusion mutuelle et les deadlocks.

8.2 Mécanismes de communication

Les processus et threads peuvent communiquer via plusieurs mécanismes :

- **Mémoire partagée** : Utilisation d'une mémoire commune pour échanger des données.
- **Passage de messages** : Envoi de messages explicites entre processus ou threads.
- **Fichiers** : Utilisation de fichiers communs pour la lecture et l'écriture de données.

8.3 Conditions de course

Les conditions de course surviennent lorsque plusieurs threads accèdent simultanément à des données partagées sans synchronisation adéquate, entraînant des résultats imprévisibles. Pour les éviter, des mécanismes de synchronisation tels que les verrous (mutex) et les sémaphores sont utilisés.

8.4 Sections critiques

Une section critique est une portion de code qui doit être exécutée par un seul thread à la fois pour éviter les conditions de course. Les solutions incluent :

- **Attente active** : Les threads vérifient continuellement une condition jusqu'à ce qu'elle soit vraie, ce qui peut être inefficace.
- **Blocage** : Les threads se mettent en attente jusqu'à ce qu'ils puissent entrer dans la section critique, ce qui est plus efficace que l'attente active.

8.5 Problèmes classiques

Les problèmes classiques de la concurrence comprennent :

- **Producteur-consommateur** : Synchronisation entre un producteur qui crée des données et un consommateur qui les utilise.
- **Dîner des philosophes** : Un problème de synchronisation qui illustre les deadlocks et les solutions potentielles, comme l'utilisation de mutex.

8.6 Deadlocks

Un deadlock survient lorsque plusieurs threads se bloquent mutuellement en attente de ressources. Les conditions nécessaires pour un deadlock sont l'exclusion mutuelle, la possession et l'attente, l'absence de préemption, et la formation d'un cycle d'attente.

8.7 Stratégies pour éviter les deadlocks

Pour éviter les deadlocks, plusieurs stratégies peuvent être utilisées :

- **Prévention** : Empêcher les conditions nécessaires au deadlock.

- **Évitement** : Utiliser des algorithmes pour éviter d'entrer dans des états de deadlock, comme l'algorithme du banquier.
- **Détection et récupération** : Détecter les deadlocks et prendre des mesures pour les résoudre.

8.8 Conclusion

La gestion de la concurrence est essentielle pour tirer parti du parallélisme et améliorer les performances des systèmes informatiques. Une compréhension approfondie des mécanismes de communication, des conditions de course, des sections critiques et des stratégies de prévention des deadlocks est cruciale pour développer des systèmes robustes