Cours de Système d'Exploitation

Mamadou M. Kanté

Université Blaise Pascal - LIMOS, CNRS

Bibliographie: Tanenbaum, Cours de SE suivi en 2004 à Univ Bordeaux 1, Cours de F. Pellegrini (LaBRI) et autres.

Plan

Plan

Introduction

Pourquoi SE ? voir Cours 1

Rappel Objectifs Cours

1. Programmation Systèmes (appels systèmes).
2. Programmation noyau
    1. Systèmes de fichiers
    2. Processus
    3. Noyau
3. Exemples de Systèmes: Linux et Windows

Principales Fonctionnalités d'un système

1. Proposer des services pour accéder au matériel: appels systèmes
   - read, open, fork, dup, etc.
2. Traiter les erreurs matérielles des processus
   - division par zero, seg fault, etc.
3. Traiter les interruptions matérielles
   - erreur de lecture disque, ecran, souris, clavier, etc.
4. Entretien global: accés au processeur, allocation de mémoire, etc.

Noyau?

Machine virtuelle

1. Vue uniforme des E/S
2. Gestion de la mémoire et des processus, réseau
3. Système de fichiers

Gestionnaire de ressources

1. Fonctionnement des ressources (processeur, délais, ...)
2. Contrôle d'accés aux ressources (Allocation CPU, disque, mémoire, canal de communication réseau, ...)
3. Gestion des erreurs
4. Gestion des conflits

Problématiques

1. Comment interdire certaines instructions?
2. Accéder à toute la mémoire?
3. Dialoguer directement avec les périphériques?

Problématiques

1. Comment interdire certaines instructions?
2. Accéder à toute la mémoire?
3. Dialoguer directement avec les périphériques?

Mécanisme de mémoire virtuelle et 2 modes d'exécution: utilisateur et noyau

Mémoire virtuelle

1. Les adresses mémoire des programmes ne peuvent référencer les adresses physiques.
2. Les processus ont des espaces d'adressage virtuel
3. Lors du chargement les adresses virtuelles sont traduites en adresses physiques (changement de contexte)
   - Un circuit Memory Management Unit fait la conversion à l'aide de registres
   - Une table de conversion pour chaque processus

Modes d'Exécution

Utilisateur

1. Processus peut accéder uniquement à son espace d'adressage et à un sous-ensemble du jeu d'instructions.

   $\Longrightarrow$ pas de corruption du système
2. L'accés à l'espace noyau est protégé et on y accède par une instruction protégée.

Noyau

1. Accès à tous les espaces : noyau et utilisateur
   - Code et données du SE accessible seulement en mode noyau: les segments mémoire sont inclus seulement lors du passage en mode mémoire.
2. Accés à toutes les instructions protégées (qui ne peuvent exécutées qu'en mode noyau)
   - Instructions de modification segments de mémoire: un processus ne peut pas modifier ses droits d'accés à la mémoire.
   - Accés aux périphériques: E/S, réseaux, allocation mémoire, etc.

Modes d'exécution

Mode noyau $\neq$ mode root

1. Mode noyau = gestion par le matériel via des interruptions (matérielle et logicielle)
2. Mode root = gestion logicielle (par le code du SE) et est souvent en mode utilisateur.

Mode noyau par le matériel

1. non connaissance lors de la compilation des segments de mémoire où se trouvent les fonctions systèmes.
2. Raisons: maintenabilité et portabillité du SE

Passage en Mode Noyau sous Linux

1. Lors de l'initialisation on installe les codes des appels systèmes dans une table Interrupt handler et à chaque fonction système on associe un numéro interrupt
2. Dans le code de l'appel système on a une instruction de passage en mode noyau (sous Linux: int) qui prend en arguments le numéro de la fonction système et les différents arguments de la fonction.
3. Depuis le mode noyau
    1. On appelle le gestionnaire d'exception trap handler: sauvegarde du contexte et transfert des données vers espace noyau.
    2. Ce dernier à son tour appelle la vraie fonction système (indexée par son numéro)
    3. Après calcul, transmission valeur de retour au trap
    4. transmission de la valeur de retour et des données et retour en mode utilisateur (encore instruction protégée) après restauration du contexte

Plan

Standard POSIX: Portable Operating System Interface

1. Est une interface de programmation système.
   - Un ensemble de fonctions disponibles sur tous les SE *IX et pratiquement implémentées par tous.
   - Un ensemble de types : time_t, size_t, dev_t, ...
2. Beaucoup de fonctions libc sont des wrappers:ne font en fait que faire appel à la fonction système (ex: time, E/S, etc.)
   - Stocker les arguments dans les bons registres
   - Invoquer l'appel système
   - Interpréter la valeur de retour et si possible positionner la variable `errno`.

Gestion des erreurs

1. Une variable globale `errno` dans errno.h qui permet de transmettre les erreurs des fonctions systèmes aux codes utilisateurs
2. Un appel système qui réussit et alors le retour de la wrapper est un entier ≥ 0
3. Un appel système qui échoue et alors le retour de la wrapper est un entier < 0 et un positionnement de la variable `errno` : numéro de l'erreur

   Les fonctions `strerror(int)` et `perror(string)` pour avoir/afficher le texte associé à l'erreur : perror("ouverture") affiche

   "ouverture:"+message associé à `errno`

   man 2 intro pour avoir la liste des valeurs possibles pour `errno`

Quelques Définitions

Utilisateur. Quiconque autorisé à utiliser les services du SE. Un utilisateur est identifié par un numéro uid. Les utilisateurs sont regroupés en **groupes** identifiés aussi par un numéro gid.

Périphérique. Regroupe toutes les ressources physiques (disques, imprimantes, souris, ...) et logicielles (machine virtuelle, système de fenêtrage, ...).

Fichiers. Ce terme recouvre:

1. Fichiers ordinaires: vecteurs d'octets et permettant de stocker les données
2. Fichiers spéciaux FIFO: files d'octets ne pouvant être lus qu'une fois
3. Fichiers spéciaux type bloc et caractère: interface avec les périphériques (pour les E/S). Le type caractère dépend en général de l'implémentation, et le type bloc on a un accés direct.
4. Répertoires: des références à d'autres fichiers.

Quelques Définitions

Système de fichiers. Collection organisée de fichiers. Un périphérique d'E/S a un seul système de fichiers.

Matricule de fichier. Numéro identifiant un fichier dans un système de fichiers.

Descripteur de fichier. Numéro identifiant donné à un fichier lors de la manipulation par un processus. Ceci correspond à un index dans une table des fichiers ouverts et permet d'accés aux informations du fichier: méta-données et contenu.

Lien. C'est une paire composée d'un nom de fichier et d'une référence à un fichier. La référence peut être le matricule d'un fichier. Un répertoire possède en général 2 liens: un vers son parent et un autre vers lui-même.

Permissions. droits de lecture (R), écriture (W) et exécution (X) d'un fichier pour: uid (RUSR, etc) , gid (WGRP, etc.) et autres (XOTH, etc.).

Terminal. Un fichier spécial de type caractère permettant des E/S asynchrones. Il fonctionne en mode full duplex: E et S en parallèle.

Quelques Définitions

2 types d'appels système

Appel bloquant. Le processus appelant ne pourra continuer son travail que lorsque l'appel système a terminé (lorsque les données demandées sont prêtes par exemple). Ex: appels système: open, read, write

Appel non bloquant. On fixe un délai δ. La main est redonnée automatiquement au processus appelant si au bout de δ temps l'appel système n'a pas terminé. Ex: read, write. Il existe des fonctions pour passer d'un mode bloquant à un mode non bloquant ou inversement.

Attente active. Le processus simule lui-même un mode bloquant sur un appel non bloquant. Ex: while (1) { r= read (...); if (r≥ 0) break;}

Plan

Services Attendus

1. Stockage des données (grandes ou petites) de façon pérenne
2. Fichier= objet nommé résidant en dehors de l'espace d'adressage des processus (aka mémoire vive) et disposant d'interface pour la lecture et l'écriture.
3. Système (de gestion) de fichiers = espace des fichiers + organisation interne.
4. Un programme ne voit pas un fichier tel qu'il est stocké, mais a une vue abstraite donnée par le SE (détails de l'implémentation cachée).

Services Attendus

1. Indépendance vis-a-vis du périphérique de stockage
2. Création, suppression, modification de fichiers
3. Gestion des droits d'accés (et protection)
4. Ouverture (et gestion accés concurrent) et fermeture
5. Accés à n'importe quelle position du fichier
6. Troncature et extension de la taille
7. ...

Structuration des Fichiers

1. Suites d'octets: pas de structure visible et interprétation par les processus utilisateurs. Ex: *IX, DOS, ...
2. Suites d'enregistrements: le fichier est constitué d'enregistrements ou blocs de taille fixe et on lit/écrit enregistrement en totalité. Ex: bases de données, ...
3. Arbre d'enregistrements: les enregistrements sont de taille variable, indexés par des clés et organisés sous forme d'un arbre. Le type d'arbre et les primitives dépendront de l'implémentation (on peut par exemple éclater des blocs lors de l'ajout ou regrouper des blocs lors de la suppression. Ex: bases de données indexées, ...

Types de Fichiers

1. Fichiers ordinaires: vecteurs d'octets et permettant de stocker les données
2. Fichiers spéciaux type bloc: périphérique E/S travaillant par blocs. Ex: disques, disquettes, ...
3. Fichiers spéciaux type caractère: périphériques E/S travaillant par caractères. Ex: terminaux, fichiers FIFO, imprimantes, sockets, ...
4. Répertoires: des références à d'autres fichiers.

Fichiers Ordinaires

Organisés en fonction de leur nature

1. La structure interne dépend de son type.
   - Simple: séquence de lignes séparées par des CR et lisible sans codage particulier.
   - Complexe: organisation interne comme les exécutables et lisibles que par des outils spécifiques
2. Identification du type selon plusieurs méthodes
   - L'extension est importante. Ex: sous DOS seuls .bin, .com ou .exe peuvent être exécutés
   - Lecture du fichier pour identifier son type

Répertoires

1. Fichier interprété par le système
2. Contient des références à d'autres fichiers ou répertoires $\Longrightarrow$ vue hiérarchique (Parler de la représentation en DAGS et des liens)
3. Une entrée d'un fichier peut être par exemple ses méta-données + organisation physique dans le disque (ou une information permettant de la récupérer)
   - méta-données: numéro périphérique+ numéro fichier + type de fichier+taille+ dates de création + droits d'accés

Répertoires

Quelques opérations souhaitées

1. Création d'une entrée
2. Suppression d'une entrée (avec suppression physique si dernier à la référencer)
3. Renommage: en général c'est création d'une nouvelle entrée (avec copie des données de l'entrée à renommer) + suppression de l'ancienne entrée
4. Accés séquentiel aux entrées au moyen d'un itérateur.

Stockage en Disque des fichiers

Rappel au tableau représentation logique des blocs d'un périphérique de stockage

Stockage en Disque des fichiers

Rappel au tableau représentation logique des blocs d'un périphérique de stockage

Allocation contigue.

1. Les blocs constituant le fichier sont logiquement contigus (blocs adjacents)

   A. Accés rapide

   I. Il faut connaître la taille du fichier (augmentation peut nécessiter l'allocation de nouveaux fichiers de grande taille). Il faut fournir des outils de réorganisation (compactage/fusion).

Stockage en Disque des fichiers

Allocation chaînée.

1. Les blocs sont stockés dans une liste chaînée (on peut séparer le chainage des blocs eux-mêmes pour des accés non séquentiels: une seule liste pour tous les fichiers)
2. On ne garde que le premier bloc (début de liste)

   A. Traitement et sauvegarde des informations de chaînage facilitées, évolution dynamique des fichiers facilitée, ...

   I. mise à jour des pointeurs couteux en temps lors d'insertion et mode séquentiel. Ex: pour 4Go et des blocs de 4ko: on a une table d'1million d'entrées.

Stockage en Disque des fichiers

Allocation Indexée (I-nodes).

1. On gère un tableau de blocs (de dimension variable) contenant la liste des blocs d'un fichier

   A. pas besoin de dispersion des pointeurs; augmentation dynamique taille fichier facile (allocation d'un nouveau bloc enregistré dans le tableau)

   P. Comment ne pas créer de surcoût pour les petits fichiers tout en gérant efficacement les gros ? table à plusieurs niveaux

Stockage en Disque des fichiers

Allocation Indexée (I-nodes).

1. On gère un tableau de blocs (de dimension variable) contenant la liste des blocs d'un fichier

   A. pas besoin de dispersion des pointeurs; augmentation dynamique taille fichier facile (allocation d'un nouveau bloc enregistré dans le tableau)

   P. Comment ne pas créer de surcoût pour les petits fichiers tout en gérant efficacement les gros ? table à plusieurs niveaux

   Au tableau le schéma de représentation des blocs sous *IX

Stockage en Disque des fichiers

Allocation Indexée (I-nodes).

1. On gère un tableau de blocs (de dimension variable) contenant la liste des blocs d'un fichier

   A. pas besoin de dispersion des pointeurs; augmentation dynamique taille fichier facile (allocation d'un nouveau bloc enregistré dans le tableau)

   P. Comment ne pas créer de surcoût pour les petits fichiers tout en gérant efficacement les gros ? table à plusieurs niveaux

   Au tableau le schéma de représentation des blocs sous *IX

   A. On accède aux blocs en 3 redirections; implémentation de gros fichiers

Stockage en Disque des fichiers

Allocation Indexée (I-nodes).

1. On gère un tableau de blocs (de dimension variable) contenant la liste des blocs d'un fichier

   A. pas besoin de dispersion des pointeurs; augmentation dynamique taille fichier facile (allocation d'un nouveau bloc enregistré dans le tableau)

   P. Comment ne pas créer de surcoût pour les petits fichiers tout en gérant efficacement les gros ? table à plusieurs niveaux

   Au tableau le schéma de représentation des blocs sous *IX

   A. On accède aux blocs en 3 redirections; implémentation de gros fichiers

Répertoires?

Il suffit par exemple de garder dans les liens vers les entrées leurs numéros pour pouvoir les traiter/parcourir.

Fichiers Spéciaux?

1. Ils sont associés aux pilotes qui disent comment y accéder (caractère, bloc ou réseau)
2. Sont dans l'arborescence du système de fichiers (/dev/tty1, /dev/sda1, /dev/eth0, ...)
3. On y accède avec l'appel système `ioctl`
4. Cas spécial des tubes: ci-dessous avec la commande/fonction `mkfifo`

Espace Disque (suite)

Au tableau : structuration physique d'un disque (Pellegrini)

1. Le choix de la taille des blocs est important: trop gros on perd de l'espace, trop petit on augmente les accés disques
2. Gestion des blocs endommagés: matériel ou logiciel (comme les blocs libres)
3. Système de cache et donc cohérence en cas de crash du système : en dire plus plus tard.

Principes des E/S sous POSIX

1. Transferts processus/périphériques et processus/processus
2. Dans les 2 cas: on initialise un canal de communication (un descripteur vers un fichier bloc/caractère/spécial/...)
3. Du point de vue programmeur peu de différence sur le principe entre les 2 types de communication (dépendant de l'implémentation: paramétrage sur certains, contraintes sur d'autres : séquentiel par exemple dans le cas type caractère)
4. La communication processus/processus se fait souvent à travers les fichiers spéciaux FIFO (de type caractère)
    1. Ouverture en lecture par P1 (lecteur ou consommateur) et en écriture par P2 (producteur ou rédacteur)
    2. On peut faire une synchronisation entre les 2 processus
    3. C'est un mécanisme de file: P2 écrit et P1 lit
    4. Si l'un ferme (ou fichier plein): blocage et un mécanisme de déblocage existe: si tampon plein rédacteur bloqué et si tampon vide lecteur bloqué
    5. On peut utiliser le mécanisme de tube anonyme avec `pipe`
5. Utiliser `O_NONBLOCK` pour un mode non blocquant (si supporté)

Quelques Fonctions système E/S

3 tables : 2 pour le noyau et 1 pour le processus.

Noyau.

1. Table des i-noeuds mémoire: dans le i-noeud mémoire on a les informations correspondant à l'i-noeud + quelques informations comme le périphérique depuis lequel il a été chargé (structure `stat` contenant les méta-données)
2. Table des fichiers ouverts: un i-noeud mémoire + quelques informations: position courante, état (R/W/X) et mode (APPEND,*BLOCK,*SYNC)

Processus.

1. Table des descripteurs de fichiers: un pointeur (index) vers la table des fichiers ouverts.

Quelques Fonctions système E/S

3 tables : 2 pour le noyau et 1 pour le processus.

Noyau.

1. Table des i-noeuds mémoire: dans le i-noeud mémoire on a les informations correspondant à l'i-noeud + quelques informations comme le périphérique depuis lequel il a été chargé (structure `stat` contenant les méta-données)
2. Table des fichiers ouverts: un i-noeud mémoire + quelques informations: position courante, état (R/W/X) et mode (APPEND,*BLOCK,*SYNC)

Processus.

1. Table des descripteurs de fichiers: un pointeur (index) vers la table des fichiers ouverts.

Quelques exemples de fonctions au tableau (Celui de J. Croboczek)

Tubes

1. `mkfifo(chemin, mode)` pour créer un fichier spécial FIF0 qui sera nommé.
2. Bloquante par défaut: si pas de lecture et ouverture en écriture -> bloquant et si pas d'écriture et ouverture en lecture -> bloquant.
3. Ouverture en lecture/ecriture -> pas bloquant (pas de sens en fait)
   - Si on veut lire n caractères et il y a $p \geq 1$ caractères, read retourne $\min\{p, \}$ caractères
   - Si $p = 0$ et pas de producteur => fermeture du tube et read retourne 0.
   - Sinon dépend si mode bloquant ou non.
   - Comportement similaire en écriture.
4. `pipe(int tab[2])` pour avoir un tube anonyme: tab[0] = lecture et tab[1] = écriture

   ceci permet par exemple d'initialiser une redirection entre processus père et fils

Système de Cache

Eviter les accés importants aux disques car lents.

1. Lecture et écriture se font autant que possible dans le cache.
2. On peut forcer l'écriture avec `fsync`: vidage des tampons en écriture du fichier.

Système de Cache

Eviter les accés importants aux disques car lents.

1. Lecture et écriture se font autant que possible dans le cache.
2. On peut forcer l'écriture avec `fsync`: vidage des tampons en écriture du fichier.

Tableau: Cours de R. Namyst suivi

Plan

C'est Quoi un Processus?

Processus.

1. C'est un programme qui tourne en machine (le chargement d'un code en mémoire)
   - Ensemble d'instructions et de données: code + données statiques + données allouées dynamiquement
2. Et une structure allouée par le système pour le contrôler = Environnement
   - Une partie pour la gestion du processus (appartient au noyau)
   - Une partie constituant le paramétrage du processus: arguments, variables d'environnement, etc.

Etats d'un Processus

On est dans un environnement multi-processus.

1. Le processus utilise un laps de temps très bref à chaque fois le(s) processeur(s).
2. Particulièrement vrai pour un processus en attente d'une ressource (éviter de surcharger le système)
3. Ces différentes étapes d'un processus sont appelés Etats.
4. On a les états: Prêt pour l'exécution, Actif (utilise le processeur), bloquer/endormi (attend une ressource), suspendu (un utilisateur le désactive), zombie (réside en mémoire, mais ne peut être réactivé: par exemple pas de contrôleur de tâches pour le supprimer de la liste des processus)
5. Un processus peut demander intentionnelement à passer à l'état suspendu avec la fonction système sleep(unsigned int)

Voir schéma de R. Namyst

Création d'un Processus

1. Allocation d'un nouveau processus par clonage (appel système fork())
   - le fils hérite l'environnement du père (environnement d'exécution, variables système)
   - C'est une copie et non un partage : faire attention aux effets de bords (partage de tampon par exemple), descripteurs de fichiers dupliqués (partagent le même décalage par exemple).
   - Le fils reçoit un identifiant (numéro): `pid` différent de celui de son père `ppid`
   - Si un processus devient orphelin, il est adopté par le processus initial (le 1)
2. Remplacement du code père par un autre si besoin (exec*(chemin,arg,...))
   - Par le passé recopier tout, d'où le remplacement
   - Aujourd'hui: environnement seulement est copié et on récupère au fur et à mesure ce qui est important
3. Exemple au tableau

Création d'un Processus

1. Allocation d'un nouveau processus par clonage (appel système fork())
   - le fils hérite l'environnement du père (environnement d'exécution, variables système)
   - C'est une copie et non un partage : faire attention aux effets de bords (partage de tampon par exemple), descripteurs de fichiers dupliqués (partagent le même décalage par exemple).
   - Le fils reçoit un identifiant (numéro): `pid` différent de celui de son père `ppid`
   - Si un processus devient orphelin, il est adopté par le processus initial (le 1)
2. Remplacement du code père par un autre si besoin (exec*(chemin,arg,...))
   - Par le passé recopier tout, d'où le remplacement
   - Aujourd'hui: environnement seulement est copié et on récupère au fur et à mesure ce qui est important
3. Exemple au tableau
4. Synchronisation parfois nécessaire

Création d'un Processus

Les différences (après duplication) entre père et fils sous POSIX

1. `pid` différents de tous les autres `pids` et `pgid`
2. `ppid fils` = `pid père`
3. Mesures de temps consommés initialisé à 0 [normal car pas encore consommé du processus]
4. Les verrous posés par le parent ne sont pas hérités [un verrou est relatif à un `pid`]
   - Un fichier verrouillé ne sera pas dupliqué
5. Minuterie désactivée
6. Ensemble des signaux pendants du père au moment de la duplication est initialisé à $\emptyset$.

Chargement d'un exécutable

1. Construction d'un nouveau code à partir d'un exécutable (précédent supprimé)
2. Initialisation de la liste d'arguments à transmettre au main
3. Initialisation de la variable globale `environ` (peut être une copie du processus)
4. Placement dans le CO l'adresse du main (on appelle le main en gros)
5. Six fonctions dans POSIX: `execl`, `execvl`, `execle`, `execve`, `execlp`, `execvp`
   - Descripteurs de fichiers verrouillés fermés (au niveau descripteur)
   - répertoires ouverts fermés
   - On ignore les signaux ignorés et le reste on prend le comportement par défaut
   - Les éléments d'environnement restants sont inchangés (répertoire courant, répertoire racine, masque, signaux pendants, etc.)
   - Si bit de positionnement (le fameux s à la place du x), alors on change `uid` avec celui de l'exécutable (de même pour `gid`).

Terminaison (1)

1. `exit(int)` (vidage tampons, flots ouverts fermés, etc.) ou `_exit(int etat)` (plus bas niveau)
2. L'état d'un processus arrêté peut être retourné à son parent si gestionnaire de tâches existe.
3. Utiliser les fonctions `wait` (bloquant) et `waitpid` (bloquant ou non et on peut demander une notification). Dans ce dernier cas on peut demander des infos d'arrêt sur plusieurs processus.
4. Faire attention aux processus fils zombis (en particulier il faut vraiment gérer les processus qui ne sont pas vraiment morst, mais juste par exemple stoppés).

Terminaison (2)

Le père est averti de la terminaison du fils (un signal `SIGCHLD`)

1. Cas 1: Le père récupère le code de retour du fils (donné à return ou exit) et appelle wait pour l'obtenir afin de libérer les ressources associées au fils

```
int status; pid_t pid_fils;
pid_fils = wait (&status);
if (pid_fils == -1) {
perror ("wait");
exit (EXIT_FAILURE);
}
if (WIFEXITED(status)) { /* terminaison normale */
printf ("%d = %d\n",pid_fils,WEXITSTATUS(status));}
```

2. Cas 2: le père ne récupère pas le code de retour, le fils devient un zombie: Lorsque le père meurt, les processus sont définitivement tués.
3. Cas 3: le père meurt avant le fils, le fils orphelin est adopté par `init`

Terminaison (3): Double fork

Que faire pour éviter les zombis ? =
Déléguer l'attente à un processus

1. On crée un processus fils
2. Le processus fils crée le processus que l'on voulait créer et meurt aussitôt
3. Celui que l'on voulait créer est adopté par `init`
4. Il ne peut pas être zombi car `init` nettoie (il est l'initial)
5. C'est la seule solution desfois car on n'a pas accés direct à `init`

Multi-Processus

1. A un instant t un seul processus sur un processeur
2. Le système fait l'entrelacement entre les processus (par le mécanisme d'ordonnancement)
3. Illusion de parallélisme: attention l'entrelacement est imprévisible (si un processus gourmand le système peut décider de le stopper)
4. Il y a un système de priorité et un ordonnanceur

Gestion des Processus

Une table des processus (d'où les numéros) contenant

1. Valeurs compteur ordinal, registres, pointeur de pile,
2. numéro de processus, état, priorité, tableau d'interruptions: ce dont le système a besoin
3. mémoire, tableau des descripteurs de fichiers, etc.
4. En GROS: ce dont il a besoin pour être à l'état ACTIF
5. Une partie doit toujours être en mémoire (et une partie peut être stockée en disque et être chargé lorsqu'il est en état ACTIF): en général 2 structures: une pour le système (à garder en mémoire) et l'autre que l'on peut jeter de la mémoire (enregistrer dans le disque).

Communication Inter-Processus

1. Processus non forcément isolés: discussion à travers les fichiers, attente exécution d'un autre processus, attente ressource, interblocage, incohérences, etc.
2. Ceci suppose une certaine synchronisation entre les processus pour garder des états cohérents
3. Exemple pool d'impression gérée par des variables

Communication Inter-Processus

1. Processus non forcément isolés: discussion à travers les fichiers, attente exécution d'un autre processus, attente ressource, interblocage, incohérences, etc.
2. Ceci suppose une certaine synchronisation entre les processus pour garder des états cohérents
3. Exemple pool d'impression gérée par des variables
4. Le système doit proposer un moyen d'éviter ces incohérences:

Communication Inter-Processus

1. Processus non forcément isolés: discussion à travers les fichiers, attente exécution d'un autre processus, attente ressource, interblocage, incohérences, etc.
2. Ceci suppose une certaine synchronisation entre les processus pour garder des états cohérents
3. Exemple pool d'impression gérée par des variables
4. Le système doit proposer un moyen d'éviter ces incohérences:
   - Un moyen rudimentaire de discussion : les tubes (nommés ou non)
   - Mémoire partagé, sémaphores: gestion de l'accés concurrent à une variable en général

Section Critique

1. On dispose d'une zone mémoire partagée et on ne veut pas que 2 processus y accèdent en même temps.
2. Une exclusion mutuelle sur une partie du code à travers section critique
3. **section critique** = partie du code exécuté exclusivement par le nombre de processus désignés à partager en même temps (1, 2, ...)
4. Quelques conditions:
   - Pas 2 en même temps (dans le cas de l'exclusion mutuelle)
   - Indépendant des vitesses des processus et # processeurs
   - Seul un processus en section critique bloque les autres (si bloqué pour une autre partie du code, il ne devrait pas interférer sur l'accés à la section critique)
   - Pour que ça marche: on ne doit pas attendre indéfiniment pour accéder à la section critique (sinon pas d'égalité)

Section Critique

1. On dispose d'une zone mémoire partagée et on ne veut pas que 2 processus y accèdent en même temps.
2. Une exclusion mutuelle sur une partie du code à travers section critique
3. **section critique** = partie du code exécuté exclusivement par le nombre de processus désignés à partager en même temps (1, 2, ...)
4. Quelques conditions:
   - Pas 2 en même temps (dans le cas de l'exclusion mutuelle)
   - Indépendant des vitesses des processus et # processeurs
   - Seul un processus en section critique bloque les autres (si bloqué pour une autre partie du code, il ne devrait pas interférer sur l'accés à la section critique)
   - Pour que ça marche: on ne doit pas attendre indéfiniment pour accéder à la section critique (sinon pas d'égalité)

Mécanismes

Masquage des interruptions, variables de verrouillage, et sémaphores

Rappel: Gestion des interruptions

1. Il existe plusieurs types d'interruption. Ex: 0 -> horloge, 1-> disque, 4 -> trap (appels système)
2. Une table des interruptions qui appelle la routine associée à l'interruption (écrite en assembleur par ex)
3. En général voici les différentes étapes de la gestion de l'interruption:
    1. Sauvegarde CO, détermine le type d'interruption, passe en mode noyau et donc charge la routine correspondante
    2. Cette routine en général stocke le contexte du processus: CO, pile, etc.
    3. On appelle le vrai code de la routine (qui est donc par ex écrite en C): ex: sata_read,
    4. Au retour de la procédure précédente, on restaure le contexte et on continue avec le processus si peut continuer, sinon on charge le contexte de celui qui doit continuer.

Masquage des interruptions

On veut pouvoir exécuter une partie du code en mode atomique

1. Une instruction spéciale pour bloquer les interruptions
2. On l'invoque avant d'entrer en section critique
3. On réactive les interruptions à la sortie de la section critique

Masquage des interruptions

On veut pouvoir exécuter une partie du code en mode atomique

1. Une instruction spéciale pour bloquer les interruptions
2. On l'invoque avant d'entrer en section critique
3. On réactive les interruptions à la sortie de la section critique

Pas bon: un processus peut accaparer tout le processeur. Mais OK pour le SE dans un système mono-processeur.

Variables de Verrouillage

Permettre plusieurs sections critiques indépendantes

1. Une variable par section critique.
2. Comment faire?

```
while (verrou == 1);
verrou = 1;
section_critique();
verrou = 0;
```

Variables de Verrouillage

Permettre plusieurs sections critiques indépendantes

1. Une variable par section critique.
2. Comment faire?

```
while (verrou == 1);
verrou = 1;
section_critique();
verrou = 0;
```

Pas bon: si stoppé avant d'écrire dans verrou?

Variables de Verrouillage

Solution 1.

masquer les interruptions avant de modifier la variable (ie entrer en mode atomique)

Solution 2: variables d'alternance

1. Définir son tour (qui a le droit d'entrer en section critique?)

```
while (tour != p); section_critique(); tour = 1-p;
```

   - Si $p = 0$, processus 1 (qui a tour=1) ne pourra pas s'exécuter.
   - Le plus rapide attend le plus lent.

2. Variable de Peterson.

```
int drapeau[2]={F,F};
int tour=0;
entrer_sc(i){                          sortir_sc(i) {
   drapeau[i]=T;                              drapeau[j]=F;}
   tour=i;
   while(drapeau[(i+1)%2]=F or tour=(i+1)%2);
}
```

   - OK pour 2 processus, mais délicat pour n processus.

Variables de Verrouillage

Variables d'alternance pour n processus et m processeurs

1. Il faut une instruction matérielle atomique: Test,Set,Lock
2. Charger le contenu d'ume mémoire
3. Mettre une valeur non nulle à cette zone mémoire
4. Pour garantir l'atomiticité: on verrouille le bus de données.

```
 TSL (&verrou) { old = verrou; verrou = 1; return old}
entrer_sc() {
while (TSL(&verrou));}    sortir_sc() { verrou=0; }
```

Variables de Verrouillage

Variables d'alternance pour n processus et m processeurs

1. Il faut une instruction matérielle atomique: Test,Set,Lock
2. Charger le contenu d'ume mémoire
3. Mettre une valeur non nulle à cette zone mémoire
4. Pour garantir l'atomiticité: on verrouille le bus de données.

```
 TSL (&verrou) { old = verrou; verrou = 1; return old}
entrer_sc() {
while (TSL(&verrou));}    sortir_sc() { verrou=0; }
```

Problèmes de ces approches

1. Attente (boucle) => consommation processeur et bus mémoire pourquoi?

Variables de Verrouillage

Variables d'alternance pour n processus et m processeurs

1. Il faut une instruction matérielle atomique: Test,Set,Lock
2. Charger le contenu d'ume mémoire
3. Mettre une valeur non nulle à cette zone mémoire
4. Pour garantir l'atomiticité: on verrouille le bus de données.

```
 TSL (&verrou) { old = verrou; verrou = 1; return old}
entrer_sc() {
while (TSL(&verrou));}    sortir_sc() { verrou=0; }
```

Problèmes de ces approches

1. Attente (boucle) => consommation processeur et bus mémoire pourquoi?
2. Priorité: si H prioritaire sur B (en section critique), alors H s'accapare le processeur, B ne pourra pas s'exécuter et donc inter-blocage.

Variables de Verrouillage

Variables d'alternance pour n processus et m processeurs

1. Il faut une instruction matérielle atomique: Test,Set,Lock
2. Charger le contenu d'ume mémoire
3. Mettre une valeur non nulle à cette zone mémoire
4. Pour garantir l'atomiticité: on verrouille le bus de données.

```
 TSL (&verrou) { old = verrou; verrou = 1; return old}
entrer_sc() {
while (TSL(&verrou));}    sortir_sc() { verrou=0; }
```

Problèmes de ces approches

1. Attente (boucle) => consommation processeur et bus mémoire pourquoi?
2. Priorité: si H prioritaire sur B (en section critique), alors H s'accapare le processeur, B ne pourra pas s'exécuter et donc inter-blocage.
3. Implémenter au niveau SE un mécanisme qui permet de sortir ces processus des processus `prêts`.

Sémaphores: Par Dijkstra

1. sémaphore = un compteur pour stocker le nombre de processus à réveiller
2. Cette variable est encapsulée dans des appels système
   - `P()`: down ou décrémenter le sémaphore
   - `V()`: up ou incrémenter le sémaphore
   - Une valeur initiale. Ex: 0.
   - Atomocité => masquage des interruptions (mais OK car sous contrôle SE).

```
P (&verrou) { if (verrou >= 0) verrou--;  else wait();}

V(&verrou) { verrou++; if (verrou >=0) wake-up(1 au hasard);
```

Sémaphores: Par Dijkstra

1. sémaphore = un compteur pour stocker le nombre de processus à réveiller
2. Cette variable est encapsulée dans des appels système
   - `P()`: down ou décrémenter le sémaphore
   - `V()`: up ou incrémenter le sémaphore
   - Une valeur initiale. Ex: 0.
   - Atomocité => masquage des interruptions (mais OK car sous contrôle SE).

```
P (&verrou) { if (verrou >= 0) verrou--;  else wait();}

V(&verrou) { verrou++; if (verrou >=0) wake-up(1 au hasard);
```

Attention: peut être dangereux si mal utilisé, Ex: inter-blocage

Exclusion Mutuelle avec Sémaphores

Programmeur.

```
mutex = 1;

entrer_sc () {P(&mutex);}

sortir_sc () { V(&mutex);}
```

Compilateur = moniteur

1. Un module qui gère un ensemble de primitives
2. le compilateur gère l'accès sous forme de conditions.
3. A un instant donné, un seul processus est actif dans le moniteur (plusieurs peuvent accéder au moniteur en même temps).

Barrière de Synchronisation

1. La barrière de synchronisation consiste à demander l'attente de tous les processus
2. Le dernier devra réveiller tous les processus.
3. A l'aide des sémaphores.
4. Exemple au tableau (Cours de Namyst).
5. Exemple aussi des producteurs/consommateurs

Ordonnancement des Processus

1. On virtualise le processeur: impressions de parallélisme
2. Mais à un instant t: un seul processus s'exécute
3. L'ordonnanceur choisit le processus qui s'exécute à l'instant t en maintenant ces structures de données:
   - Le processus qui s'exécute (Ready)
   - La liste des processus prêts à s'exécuter (r)
   - Pour chaque événement, l'ensemble des processus en entente de cet événement (Wait): appel bloquant, attente de terminaison fils, etc.
   - l'ensemble des processus Zombies
4. Rappel automates des états: w->r<=>R->W ou R->Z

Ordonnancement des Processus

1. FIFO (premier r = premier R) ou file de priorité (on gère les priorités et on exécute ceux de grande priorité)
2. Tourniquet (ou anneau) : Chacun à son tour suivant l'anneau et chacun a 1 temps fixé (quantum) nécessaire pour changer de contexte
3. Le plus court d'abord, ensuite le plus court et le 2eme plus court et ainsi de suite
4. Système intertactif
5. Priorité + système interactif
6. Voir Pellegrini et Cours Namyst pour explication.

Plan