Système d'exploitation **FEUILLE DE NOTES: INF2610 (intra) - H2020**

- Gère le matériel d'un ordinateur
- Fournit un ensemble d'appels systèmes (en librairies) permettant une abstraction sur la manière de développer un programme.
- POSIX (normes IEEE de standard. des interf. de programm. système basées sur Unix) a pour objectif d'interfacer le OS lui-même!

Mémoire virtuelle:

- Chaque processus a sa propre mémoire virtuelle. On peut ensuite charger les blocs présents dans la mémoire virtuelle jusqu'à la mémoire physique.
- La MV est un concept abstrait, en fait tout ce qui est dans la M.V. se trouve dans la mémoire persistante (qui persiste même après le redémarrage, contrairement à la RAM).
- Si la RAM est déjà utilisée à pleine capacité, on peut partitionner l'information entre la RAM et le disque.
- La taille de l'espace d'adressage peut être plus grande que celle de la mémoire physique.

i-node:

- Sous UNIX/Linux, un fichier est représenté par une structure de données appelée i-nœud (i-node)
- Contient :
  - type et mode du fichier,
  - le nombre de liens vers le fichier,
  - l'identifiant du propriétaire,
  - l'identifiant du groupe du propriétaire,
  - la taille du fichier,
  - le périphérique représenté par ce fichier (si le
  - fichier représente un périphérique,

Contrôleur d'un composant : assure son fonctionnement et les interactions avec les autres composants.
Pilote : procure une interface au noyau pour communiquer avec un périphérique via des fonctions simples (open, read, write, close, etc.)

Gestion d'E/S: (les E/S sont dirigées par des interruptions)

- PAR SCRUTATION: Le système vérifie régulièrement l'état des périphériques afin de détecter un changement d'état et d'agir en conséquence.
- PAR INTERRUPTIONS: Le périphérique a la responsabilité de signaler une interruption lors de son changement d'état. Le système reçoit cette interruption et agit en conséquence.

Interruptions:

Interruptions **matérielles** générées par des horloges (interruptions après un certain délai), périphériques (signalées lors du début ou de la fin d'une E/S), etc.

Interruptions **logicielles** générées par des erreurs arithmétiques (division par 0), données non disponibles en mémoire (défaut de page), appels système.

Modes de fonctionnement:

- mode noyau réservé au OS, où toutes les instructions sont autorisées
- mode utilisateur imposé aux programmes et utilitaires, où certaines instructions sont défendues, à des fins de protection du OS

Interactions utilisateur/système:

- Un appel système consiste en une interruption logicielle qui a pour rôle d'activer le système d'exploitation :
  - changer le mode d'exécution pour passer du mode utilisateur au mode noyau (exemple, sous Linux: syscall);
  - retourner au programme appelant avec retour au mode utilisateur.
  - récupérer les paramètres et vérifier la validité de l'appel;
  - exécuter la fonction demandée;
  - récupérer la (les) valeur(s) de retour;
- Les appels systèmes permettent de: créer, attendre et terminer des processus, établir une communication interprocessus (segments de données,partagés, fichiers, tubes, etc.) et synchroniser des processus.
- Pour lister les appels système : man syscalls

Évolution des OS:

- Traitement par lots (1955-1965): file de travaux exécutés un à un (FIFO), ravaux terminées vont dans une file de tâches terminées

Multiprogrammation (1965-1980):

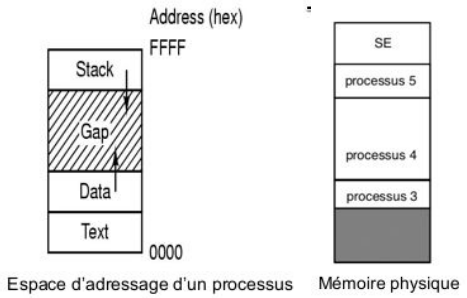
- Séparation de la mémoire en plusieurs partitions
- Chaque partition contient un travail
- Partage de processeur et des périphériques
- Si un travail tombe en attente d'E/S, le processeur est alloué à un autre travail (on fait du swapping entre mém. et disque)
- Après l'E/S, le système traite l'interruption

Partage de temps (1965-1980):

- Le SE attribue un temps donné à chaque tˆache
- Au bout du temps, si la tˆache n'est pas terminée, elle retourne dans la file des travaux prˆets
- Pseudo-parallélisme
- PROBLÈME: La taille de l'espace d'adressage d'un travail est limitée par celle de la mémoire physique

Types de noyaux:

- Monolithique modulaire (Linux, BSD, Solaris): ensemble de modules qui peuvent être chargés dynamiquement en mémoire principale et qui s'exéc. en mode noyau
- Micronoyau (MINIX): contient un minimum de fonctionnalités dans le mode noyau et déplace un grand nombre de services appartenant normalement au mode noyau vers le mode utilisateur. Comme la plupart des services sont dans l'espace utilisateur, cela permet de facilement modifier/ajouter des services
- Noyau hybride (Windows, Mac OS X): entre les deux précédents: possède donc l'efficacité du noyau monolithique modulaire (chargement de services dynamiquement en mémoire) et la modularité du micronoyau



Machines virtuelles

- fichiers (images) qui définissent un ordinateur (avec un système d'exploitation, des disques, de la mémoire, etc.)
- Hyperviseur: permet de lancer plusieurs machines virtuelles simultanément

Processus: programme en cours exéc. (espace d'adressage, état, compteur ordinal, etc.) Exécute un code (pouvant être chargé avec exec) et est représenté par un PCB (Process Control Block) et identifié par un PID

Contenu d'un PCB:

- Compteur ordinal: adresse de la prochaine instru. à exéc.
- Registres : espace réservé à la sauvegarde des contenus des registres lors de la suspension de l'exécution du proc.
- État : prêt, en exécution, bloqué, etc.
- Priorité : sert à l'ordonnancement des processus (choisir le plus prioritaire).
- Espace d'adressage : regroupe le code, les données, la pile d'exécution, etc. du processus.
- Fichiers ouverts : accessibles au processus. etc.

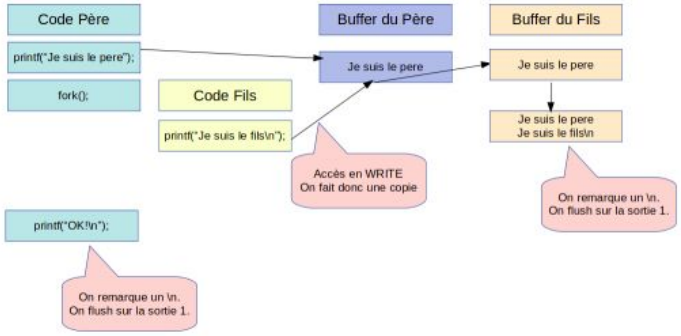
Famille de processus: Windows vs. Linux: Chaque processus connaît son parent (commun aux 2), mais lorsque le parent se termine le processus fils, il n'est pas adopté par le proc. grand-père sur Windows, alors qu'il l'est sur Linux, peu importe quel parent est encore en vie

fork(): crée un processus fils du processus appelant. Duplique avec le principe de COW. Il s'agit donc d'un clone du processus père (espaces d'adressage identiques, tables de descripteurs de fichiers identiques, etc). Retourne 0 au processus fils créé, le PID du processus fils créé au père ou −1 en cas d'erreur.

Copy On Write (COW): Le processus utilise les mêmes données que le père tant que l'accès se fait en lecture. Il doit faire une copie des données dans son propre espace d'adressage s'il fait soudainement un accès en écriture.

Gestion de processus: appels système

- getpid(): retourne le PID du processus appelant
- getppid(): retourne le PID du parent du processus appelant
- fork(): crée le processus fils du processus appelant
- _exit(value)
  - termine le processus appelant, value est la valeur de retour du processus (codée sur 8 bits)
  - la terminaison peut aussi être forcée par un signal (ex. SIGKILL), mais il s'agirait là d'une terminaison dite anormale
- wait(&status): met le processu appelant en attente de la terminaison d'un processus fils avec possibilité de récupérer l'état de terminaison de ce dernier



- execl(chemin nom, params): remplace le code exécutable du processus appelant par celui d'un autre programme

Terminaison d'un processus: avec _exit(value) ou un signal (ex. SIGKILL). Lorsqu'un processus se termine, son état de terminaison est enregistré dans le PCB du processus. Le processus bascule ensuite vers l'état zombie (exéc. terminée, en attente du père).

Remplacement du code d'un processus

- Se fait avec un appel système de la famille exec: execl, execlp, execle, execv, execvp et execve.
  - "l" (resp. "v") indique que les mots composant la ligne de commande du nouveau code exécutable sont passés séparément l'un à la suite de l'autre (resp. sont passés via un vecteur de mots): (nom, chemin,
  - "p": indique que le chemin d'accès au nouveau code exécutable est sauvegardé dans la variable PATH (echo $PATH è ensemble de chemins d'accès) .
  - "e" signifie qu'il est possible d'ajouter un vecteur de variables d'environnement (accessibles à partir du nouveau code).

Descripteur de fichier: outil qui nous permet de manipuler un fichier (ordinaire ou spécial) selon les accès accordés lors de l'ouverture ex: O WRONLY — O XXXX — XXXX.

Threads: concept qui permet de lancer en concurrence plusieurs parties d'un même processus

- Lors de sa création, un processus contient un seul thread principal qui exécute la fonction main. Ce thread principal peut créer d'autres threads qui, à leur tour, peuvent en créer d'autres (mais pas de relation hiérarchique).
- Tous les threads ainsi créés sont rattachés au processus et partagent les ressources du processus (espace d'adressage, table des descripteurs de fichiers, table des gestionnaires de signaux, etc.).
- Chaque thread a: une pile, un compteur ordinal, des registres, un état et surtout **une priorité**

pthread: librairie qui permet de créer, terminer, se mettre en attente de la fin, synchroniser, forcer la terminaison, etc., de threads

Fonctions importantes pthread

- pthread_create crée un thread:
  - int pthread_create(pthread_t* tid, const pthread_attr_t* attr, void* func, void* arg) où :
    - tid sert à récupérer l'identifiant du thread créé,
    - attr spécifie les attributs de création du thread (NULL pour les attributs par défaut),
    - func est le nom de la fonction exécutée par le thread (le prototype de la fonction func est : void* func (void* arg)), et arg est l'argument à passer à la fonction func.

    - Cet appel retourne la valeur 0 en cas de succès et une valeur différente de 0 en cas d'échec.
    - La fonction exécutée par un thread doit obligatoirement avoir un seul paramètre! Si vous voulez que le thread exécute une fonction ayant plusieurs paramètres, vous devez les regrouper dans une struct et créer une fonction wrapper qui appellera la fonction ayant plusieurs paramètres avec les attributs de la struct.
- pthread_exit() termine le thread appelant: void pthread_exit(void* retval), où retval est un pointeur.
- p_thread_cancel() termine le thread ciblé en paramètre:
    - int pthread_cancel(pthread_t tid)
    - Cette fonction retourne 0, en cas de succès et une autre valeur, en cas d'erreur. L'état de terminaison PTHREAD_CANCELED pourrait être récupéré par un autre thread via la fonction pthread_join.
- pthread_join() met le thread appelant en attente de la fin d'un thread:
    - int pthread_join(pthread_t tid, void** pstatus), où :
    - tid est l'identifiant du thread à attendre (qu'il se termine), pstatus contient l'adresse de la variable où récupérer les informations de termin. du thread tid. Le double pointeur pstatus est égale à : &pval, si le thread tid s'est terminé avec pthread_exit(pval), et PTHREAD_CANCELED, si le thread tid s'est terminé anorm. (suite à un appel à pthread_cancel).
    - Cet appel retourne 0 en cas de succès et une valeur non nulle en cas d'échec.

Fonctions de nettoyage pthread
- Chaque thread dispose d'une pile de nettoyage dans laquelle le thread peut empiler les fonctions à exéc. lors de sa terminaison :
- void pthread_cleanup_push(void func, void* args): ajoute la fonction func dans la pile de nettoyage.
- void pthread_cleanup_pop(int execute)
    - dépile la fonction en tête de la pile de nettoyage et exécute celle-ci si le paramètre execute n'est pas 0.
    - Lors de la termn. d'un thread, les fonctions figurant dans sa pile de nettoyage sont dépilées et exéc. séquentiellement.

Envoi de signaux:
- La fonction pthread_kill() permet d'envoyer un signal à un thread : int pthread_kill(pthread_t tid, int sig)
- Elle retourne 0 en cas de succès, une valeur non nulle en cas d'erreur.
- Chaque thread a un masque de signaux privé (man pthread_sigmask). La fonction kill() envoie un signal à un proc. Ce signal peut être traité par n'importe quel thread du processus qui n'a pas bloqué ce signal, utilisation: int kill(pid_t pid, int numsignal)

Traitement de signaux:
- Pour traiter un signal reçu, un processus : suspend tempo. son traitement en cours, réalise celui associé au signal (adresse du début du code est dans TGS) & reprend le traitement suspendu ou se termine (si le traitement du signal force sa termin.).
- On peut aussi faire attendre un signal avec: pause(), sleep(secs) (pause() mais avec délai max) et sigsuspend(set), qui remplace temporairement le masque des signaux par set puis se met en attente d'un signal

Cas de Linux:
- Linux ne fait pas de distinction entre un processus et un thread. Ils sont appelés communément tâche et représentés par la structure de données task_struct.
- Une tâche Linux peut être créée via les appels systèmes fork, vfork mais aussi via l'appel système clone.
- L'appel système clone (propre à Linux) permet de créer une tâche en spécifiant les ressources à partager entre la tâche créatrice et la tâche créée (espace d'adressage, table des descripteurs de fichiers, table des gestionnaires de signaux, etc.)

Threads niveau noyau vs niveau utilisateur
- TODO: ...

```c
void sigHandler(int num) {
    printf("handling signal: %d\n",↩
        num);
}
int main() {
    signal(SIGINT, sigHandler);
    return 0;
}
```

Communication: signaux: ex. signal(
- Peuvent être envoyés entre processus ou entre threads
- Chaque thread noyau a un masque de signaux. Ce dernier permet de bloquer ou non des signaux pour un processus. On ne peut pas ignorer certains signaux (ex: SIGKILL)
- On peut définir un handler pour un signal afin d'exécuter une certaine fonction quand un processus (ou un thread) reçoit le signal:

Communication: tube
- Tubes anonymes permettent une communication unidirectionnelle entre le créateur du tube et ses descendants (parenté obligatoire) ⇒ Deux tubes reliés de manière circulaire permettrait une comm. bidirectionnelle
- Tubes nommés permettent une communication entre des processus d'une même machine (parenté facultative).

Communication: tube anonyme
- Avec l'opérateur shell (|):
    - La sortie standard (1) du programme ls est redirigée dans le bout d'écriture du tube
    - L'entrée standard (0) du programme cat est redirigée vers le bout de lecture du tube.
- Avec l'appel système pipe():
    - pipefd[0] contient le bout en lecture
    - pipefd[1] contient le bout en écriture

```
ls | cat
```

```c
int pipefd[2];
pipe(pipefd);
if (!fork()) {
    close(fd[0]);
    // ecrire...
    close(fd[1]);
    exit(0);
}
close(fd[1]);
// lire...
close(fd[0]);
wait(NULL);
exit(0);
```

```c
mkfifo("/chemin/vers/tube", S_IRUSR | S_IWUSR↩
    |                       S_IRGRP | ↩
    S_IWGRP);
int tubeLecture = open("/chemin/vers/tube", ↩
    O_RDONLY);
int tubeEcriture = open("/chemin/vers/tube", ↩
    O_WRONLY);
```

Communication: tube nommé:
- Avec la fonction mkfifo, qui crée le tube sans l'ouvrir. Pour ouvrir le tube, on utilise open():

Communication: segments de données partagés
- Chaque proc. a un espace d'adressage privé et accède à la mém. phys. via son espace d'adressage.

- Cependant, un proc. peut créer des segments de données et spécifier qui ont le droit d'y accéder (segm. de données non privés).
- Pour pouvoir accéder à un segment de données créé, un processus droit d'abord l'attacher à son espace d'adressage (mapper).
- Tous les processus qui ont ce segment attaché à leurs espaces d'adressage pourront accéder au segment.
- Les appels système POSIX pour les segments de données partagés sont regroupés
- dans la librairie <sys/mman.h> (man 7 shm_overview).
    - L'appel système shm_open permet de créer ou de retrouver un segment de données. Il retourne un descripteur de fichier ou -1 en cas d'erreur.
    - L'appel système mmap permet d'attacher un segment de données à l'espace d'adressage d'un processus. Il retourne l'adresse du début du segment ou -1.
    - L'appel système munmap permet de détacher un segment de données de l'espace d'adressage d'un processus.
    - L'appel système shm_unlink permet de supprimer le segment

Redirection des descripteurs de fichiers:

- Soit par le terminal:

./a > t.txt	Redirige le descripteur à l'index 1 vers le fichier t.txt
./a < t.txt	Redirige le descripteur à l'index 0 vers le fichier t.txt
./a 2 > t.txt	Redirige le descripteur à l'index 2 vers le fichier t.txt
./a 2>&1	Le descripteur à l'entrée 2 devient synonyme de celui à l'entrée 1
./a |./b	STDOUT de a va dans le bout d'écriture du tube anonyme et STDIN de b va dans le bout de lecture du tube anonyme

- Soit avec dup2:
    - Utilisation: dup2(modèle, copie)
    - Après cette instruction, à chaque fois que le processus (ou un fils), utilise copie, il utilisera modèle
    - ATTENTION! Il ne s'agit pas de remplacement de références, mais bien simplement d'entrées dans la TDF

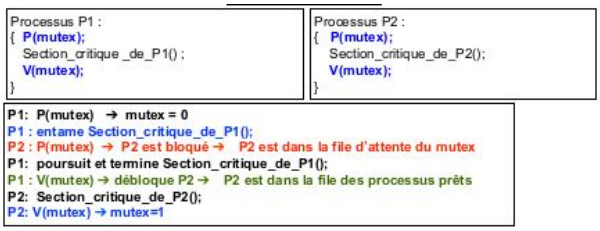
0: stdin, 1: stdout, 2: stderr

Conditions de concurrence:

- Dans un système d'exploitation multiprogrammé en temps partagé, plusieurs processus/threads s'exécutent en concurrence, ce qui peut conduire à des résultats incohérents.
- On doit s'assurer que les sections critiques (suite d'instructions qui accèdent en lecture et écriture à un ou plusieurs objets partagés) des processus qui opèrent sur des objets communs s'exécutent en exclusion mutuelle.
- 4 conditions de l'exclusion mutuelle:
    - Deux processus ne peuvent pas être, en même temps, dans leurs sections critiques.
    - Aucune hypothèse ne doit être posée sur les vitesses relatives des processus ni sur le nombre de processeurs.
    - Aucun proc. suspendu en dehors de sa section critique ne doit bloquer les autres processus d'entrer dans leurs sect. crit.
    - Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).

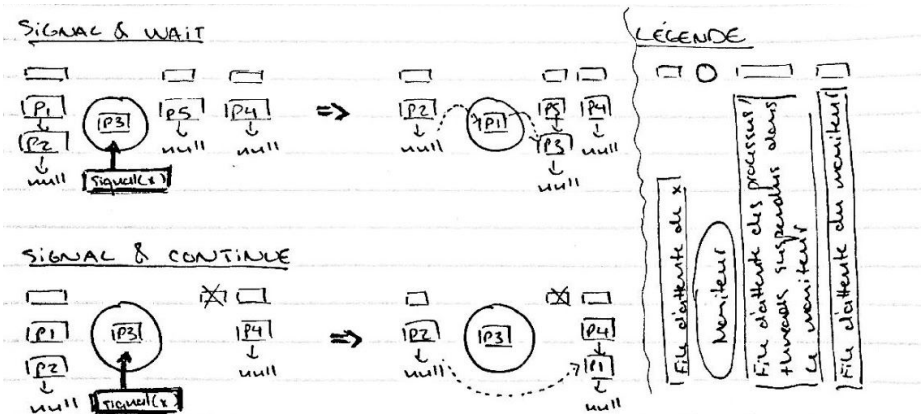
Sémaphores

- Compteur entier dont la valeur représente le nombre d'accès à une certaine section critique pouvant ˆetre fait:
    - Chaque fois qu'un processus/thread entre dans la section critique, le compteur est décrémenté
    - Chaque fois qu'un processus/thread sort de la section critique, le compteur est augmenté
    - Si un processus/thread essaye de décrémenter le sémaphore, mais que celui-ci est à 0, alors le processus/thread va dans la file d'attente du sémaphore.
- Les sémaphores sont manipulés au moyen d'opérations atomiques :
    - P() (down ou wait) pour demander un jeton et
    - V() (up ou signal) pour libérer un jeton.



Mutex: Sémaphore binaire, leur valeur maximale est donc 1. Ils permettent d'assurer l'exclusion mutuelle à des objets partagés en bornant à 1 les accès simultanés à ces objets:

Moniteurs: Gestion des threads niveau utilisateur avec des files d'attentes des threads en FIFO. Utilisation de variables de condition afin d'assurer le contrôle des sections critiques



Synchronisation des moniteurs: voir →

Problème du producteur-consommateur

- TODO

Problème des lecteurs-rédacteurs

- TODO

Problème des philosophes

- ..TODO