# Système d'Exploitation - L3, S6
# TP 3 - Processus 2 : Gestion des Processus

L'objectif de ce TP est d'implémenter un système de gestion des processus, une gestion simplifiée empruntée aux simulateurs nachos et moss. Dans ce TP on va s'intéresser aussi bien à l'ordonnancement qu'à la communication entre processus. Pour simplifier, un programme sera vu comme un tableau de mots (non fixé a priori), chaque mot étant composé de deux champs : le premier indique si l'instruction est valide (ou autorisée) et le deuxième si elle est bloquante ou pas. Lorsqu'une instruction est bloquante, le processus devrait être mis dans la liste des processus en attente. Lorsqu'un processus est en attente, il faudra faire la différence entre l'attente bloquante d'une ressource (on n'a pas pour le moment à gérer les ressources) ou l'attente de la terminaison d'un processus. On gérera le réveil des processus en attente dans les questions suivantes.

## 1 Définition d'un Processus

Un processus sera identifié par son programme, un numéro dans la table des processus, le numéro de son père, la liste des numéros de ses fils. On incluera également un entier `pt_pile` indiquant la première instruction à exécuter lorsque c'est le processus courant. Pour l'instant, nous ne gérons pas la mémoire et on peut alors supposer que le programme est stocké dans l'objet processus. Ainsi, le `pt_pile` servira pour stocker le contexte d'un processus (réduit pour le moment à l'index où commencer l'exécution lorsqu'un processus est réveillé). Il faudra aussi gérer l'état d'un processus.

**Exercice 3.1** *Module processus*
Dans le module **processus**, nous gérons : la création d'un processus, la sauvegarde du contexte d'un processus, le placement d'un processus dans la liste des processus en attente, des processus zombies, des processus prêts à s'exécuter ou en exécution. Vous devrez également gérer dans ce module la fin d'un processus (suite à une instruction autorisée ou pas), ainsi que les fonctions d'attente de processus. On supposera que l'on dispose toujours d'un processus 0 qui est l'ancêtre de tous les processus (et qui hérite en particulier des processus orphelins). Pour faire simple, lors de la création d'un processus, on spécifie en paramètre le numéro de son père.

1. Implémentez le module **processus**.
2. Testez votre module dans un fichier séparé (nommé par exemple **test-processus.c**).

## 2 Ordonnancement

On va dans cette section s'intéresser à l'exécution des processus et à l'ordonnancement. Commençons d'abord par gérer les interruptions. On va supposer qu'il y a deux types d'interruption : une interruption signalant la fin d'un processus, une interruption signalant la fin d'un appel bloquant à une ressource. A chaque instant on dispose d'une liste d'interruptions

et à chaque cycle d'horloge (dans notre cas à chaque tour de boucle dans l'exécution d'une instruction d'un programme) il faudra regarder cette liste d'interruptions et décider quoi faire. Pour la gestion des interruptions, on va disposer d'une fonction **gestion-interrupt** qui décide de la stratégie. Pour commencer, on choisit la stratégie suivante :

1. Si l'interruption concerne le processus en cours, on la traite.
2. Si l'interruption concerne un autre processus :
   (a) Si c'est l'attente de la fin d'un processus fils, on réveille le processus (s'il n'attendait que ce dernier) et on place le processus dans la liste des prêts à s'exécuter.
   (b) Si c'est l'attente bloquante d'une ressource (dans notre modèle il n'y en a qu'une seule par processus), on place le processus concerné dans la liste des prêts à s'exécuter et on informe sur le fait que l'instruction courante n'est plus bloquante. Pour éviter lors de la mise en exécution de ce processus que l'opération pour laquelle il était en attente soit à nouveau considéré bloquant, on peut supposer que l'on dispose d'un registre dans la machine qui informe de ce fait (il faudra alors modifier la structure processus pour enregistrer ce registre lors du changement de contexte).

**Exercice 3.2** *Interruptions*
Ecrire le module **interrupt** gérant les interruptions. Pour la gestion des interruptions, on appliquera la règle FIFO.

On s'intéresse maintenant à l'exécution et à l'ordonnancement. Ces deux fonctions seront intégrées aux modules **machine** et **ordonnanceur** respectivement. Pour le moment, une machine est réduite à un compteur ordinal, au registre expliqué plus haut et à un pointeur sur un programme. Pour une première implémentation de l'exécution d'un processus, à chaque cycle d'horloge, on exécute les instructions suivantes :

1. Si le temps CPU imparti au processus courant est terminé, on met le processus courant dans l'état prêt à s'exécuter et on exécute l'ordonnanceur. Ce changement de contexte ne doit se faire que lorsqu'il existe un autre processus (autre que le 0).
2. On lance 1 dé (Pile ou face) et si Pile on génère aléatoirement une interruption correspondant à une attente bloquante d'une ressource, et on l'enregistre.
3. On traite une interruption parmi celles présentes (pour le moment on supposera que cela ne coûte pas de temps CPU).
4. On exécute l'instruction courante du processus courant.
   — Si c'est une instruction autorisée, on fait un affichage « instruction autorisée. »
   — Si c'est une instruction non autorisée, on termine le processus (avec tous les évènements qui en découlent) et on affiche un message décrivant la fin du processus.
   — Si c'est une instruction bloquante, on enregistre l'interruption attendue, on bloque le processus (avec tous les évènements qui en découlent).

Pour l'ordonnancement, on applique les règles suivantes :
— Lorsqu'un processus est choisi pour s'exécuter, on tire aléatoirement un nombre entre $a$ et $b$ (entiers à fixer par exemple lorsque l'on lance le simulateur) qui détermine le nombre de cycles maximum donné au processus avant d'être potentiellement remplacé par un autre.
— l'implémentation d'un algorithme d'ordonnancement lors de la demande de changement de contexte.

Pour l'ordonnancement, implémentez l'algorithme FIFO, puis l'algorithme qui favorise celui qui semble nécessiter le moins de CPUs. Pour calculer le temps CPU, on peut faire un calcul basé sur le nombre d'instructions restant, le nombre d'instructions bloquantes du passé (ou leur fréquence) et le nombre d'interruptions traitées. Si vous avez encore du temps, vous pouvez implémenter d'autres algorithmes pour les comparer (et modifier l'algorithme d'exécution des processus si besoin).

**Exercice 3.3** *Ordonnancement*

1. Implémentez les modules **machine** et **ordonnanceur**.
2. Ecrire un programme **simulateur** qui permet de tester notre simulateur. On peut par exemple supposer qu'on lui donne en entrée un fichier avec la liste des processus à créer et les programmes associés.

**Exercice 3.4*** *Instructions complexes*
Modifiez le comportement pour inclure de nouvelles instructions comme par exemple créer des processus (tel fork, exec), faire des sauts, etc. Inclure également la pagination par exemple.

# 3 Ressources et Communication (Facultatif en 2017-2018, sauf semaphore)

On va dans cette section implémenter les modules **disque** et les fonctions d'entrées-sorties (a)synchrones, et les communications entre processus. On suppose que l'on a deux types de disques : les disques de type caractère (un seul caractère à la fois est lu/écrit) et de type bloc (un bloc entier est lu/écrit). Un disque est caractérisé par : le nombre de plateaux, le nombre de secteurs par plateaux, la taille d'un secteur, le temps pour passer d'un secteur à un autre (dans un plateau), le temps pour passer d'un plateau à un autre (on bouge d'un niveau, mais on reste sur le secteur parallèle), le temps de transfert. Un disque accepte trois opérations : `lire` et `ecrire`, chacune ne pouvant opérer que sur un secteur entier (pas plus, ni moins), et la possibilité de connaître le temps nécessaire pour satisfaire la demande. Un disque ne peut accepter qu'une requête à la fois. On va se restreindre à des disques asynchrones, *i.e.*, qui ne bloquent pas l'appelant, mais envoient une interruption lorsque l'opération est finie.

Une entrée-sortie asynchrone est une entrée-sortie où on n'attend pas le résultat (et on est prévenu - par une interruption - lorsque l'opération est finie). Une entrée-sortie synchrone est une opération d'entrée-sortie bloquante lorsque la donnée n'est pas prête.

**Exercice 3.5** *Disques*

1. Commencez d'abord par inclure dans le module **interrupt** la possibilité d'activer/désactiver les interruptions.
2. Implémentez les sémaphores et les mutex dans un module **semaphore**.
3. Implémentez le module **disque** qui simule l'accés à un disque asynchrone. Attention : un disque ne peut satisfaire qu'une seule demande de lecture/écriture à la fois. Lorsqu'on fait une demande de lecture/écriture, vous devez vérifier que le disque n'est pas en action et si oui, c'est une erreur de demander une opération de lecture/écriture. Lorsqu'un disque a fini de traiter une requête de lecture/écriture, il envoie une interruption pour signaler la fin de l'opération (il faudra ajouter ce type d'interruption dans

le module **interrupt**). Pour simuler le disque, vous pouvez par exemple lui associer un fichier où on stockera les données à lire/écrire.

4. Implémentez les modules **es-sync** et **es-async** d'entrées-sorties synchrones et asynchrones respectivement. On peut se restreindre aux fonctions de lecture/écriture d'un secteur. Vous pouvez utiliser des mutex et sémaphores pour gérer les entrées-sorties synchrones.
5. Proposez des fonctions systèmes qui permettent de lire/écrire une chaîne de caractères. Ces fonctions doivent s'exécuter en mode noyau. Ainsi, vous devrez modifier la définition de programme et l'exécution d'un programme qui lorsqu'il voit un appel système, crée une interruption `trap` pour éxécuter le code du système d'exploitation - ceux de la question précédente - (vous devrez gérer une table des appels systèmes). Vous implémenterez les deux types d'appel : bloquant et non bloquant.
6. (Bonus*) Si vos fonctions systèmes ne sont pas ré-entrantes, modifiez les pour qu'elles le soient (en gérant la possibilité que plusieurs processus puissent l'utiliser en même temps).
7. Modifiez l'exécution d'un programme pour inclure les fonctions d'entrée-sorties aux programmes utilisateurs. Attention, maintenant un processus peut attendre plusieurs interruptions d'entrée-sortie non bloquantes.

### Exercice 3.6 *Communication*

1. Proposez un mécanisme de communication de processus par tubes (le même mécanisme que producteur/consommateur). On peut voir les tubes comme des disques de type caractère.
2. Proposez un mécanisme de communication synchronisée de processus.