TD/TP6 - MÉMOIRE - INTERRUPTIONS

1. MÉMOIRE LINÉAIRE

Dans ce type de gestion, la totalité du programme et des données d'un processus doivent être chargés en mémoire pour pouvoir exécuter le processus. L'allocation d'un espace mémoire à un processus peut se faire suivant différentes stratégies :

- La stratégie First Fit recherche le premier espace en mémoire de taille suffisante ;
- La stratégie Best Fit recherche le plus petit espace libre pouvant contenir le processus ;
- La stratégie Worst Fit recherche le plus grand espace libre pouvant contenir le processus.

1.1.

Soit une liste des blocs libres composée dans l'ordre de blocs de 100K, 500K, 200K, 300K et 600K. Comment des processus de tailles respectives 212K, 417K, 112K, et 426K, arrivés dans cet ordre, seraient-ils placés en mémoire

- avec une stratégie Best-Fit ?

- avec une stratégie First-Fit ?

2. SEGMENTATION

La segmentation divise l'ensemble des informations nécessaires à l'exécution d'une tâche en segments. Un segment, de longueur quelconque, représente une zone de mémoire pour la tâche. Il est associé à un aspect spécifique de la représentation de la tâche en mémoire (code, données, pile, ...). Un segment est chargé entièrement dans des zones de mémoire contiguës.

On note <s : d> une adresse segmentée composée d'un numéro de segment et d'un déplacement. On considère une tâche utilisant 3 segments, dont la table des segments à un instant donné est :

	B	L	p	m	u	xrw
0		132	0			
1	7435	400	1	0	0	011
2		325	0			

où B est la base, L la longueur du segment, p, m , u et xrw respectivement les bits de présence, de modification, d'utilisation et de droits d'accès au segment. On suppose que la tâche fait un accès en écriture à la donnée située à l'adresse <1 : 260>.

2.1.

Quelle est l'adresse physique de cette donnée en mémoire ?

2.2.

Donnez les modifications effectuées dans la table lors de cet accès, en précisant si elles sont effectuées par le système ou le matériel.

2.3.

Que se passe-t-il lors d'un accès à l'instruction d'adresse <0 : 125> ?

3. INTERRUPTIONS (TP)

Les processus système sont capables de communiquer entre eux via des signaux. Lorsqu'un processus reçoit un signal, il interrompt momentanément son exécution, et exécute une portion de code associée au signal, appelée « traitant d'interruption ». Il existe 32 types de signaux dans Linux, et chacun d'entre eux possède un traitant d'interruption par défaut. Cependant, il est possible de redéfinir les traitants par défaut et ainsi modifier le comportement du processus lorsque le signal est reçu. Nous allons nous intéresser au signal SIGINT.

3.1.

Écrire un programme qui affiche dans la console la chaine de caractères « Execution... » à intervalles réguliers. Pour cela, le programme exécute l'algorithme suivant :

tant que (vrai)

  afficher « Execution... »

  attendre 1 sec

fin tant que

L 'attente du programme peut être implémentée avec la fonction `sleep` de la librairie `unistd.h` (taper `man 3 sleep` dans la console pour plus de détails).

Chaque processus qui s'exécute sur le système possède un identifiant unique appelé `pid` (pour *process identifier*). Il est possible de connaître le pid d'un processus via la commande `ps -a` de la console Linux.

3.2.

Écrire un programme qui envoie un signal SIGINT à un processus dont le pid est spécifié en paramètre. Pour cela, le programme récupère d'abord le pid donné en paramètre du programme avec argc + argv (le pid est un *entier*). Ensuite, le programme utilise la fonction `kill` de la librairie `signal.h` pour envoyer le signal au processus (taper `man 2 kill` dans la console pour plus de précisions).

3.3.

Lancer le programme écrit dans la question 3.1. Récupérer son pid via la commande `ps -a` de la console. Enfin, utiliser le programme de la question 3.2. pour envoyer un signal SIGINT à ce processus. Que se passe-t-il ?

Le comportement observé dans la question précédente est le comportement par défaut d'un processus lorsqu'il reçoit un signal SIGINT. La redéfinition des traitants d'interruption par défaut d'un processus se fait via la fonction `signal` de la librairie `signal.h` (c.f., man signal). Pour cela, il suffit de définir une fonction correspondant au nouveau traitant d'interruption (ayant la signature `void traitant (int sig)`). Puis, dans la fonction

`main` de votre programme, appeler `signal(SIGINT, traitant)` pour remplacer le traitant par défaut de SIGINT par votre traitant.

3.4.

Dans le programme écrit dans la question 3.1, redéfinir le traitant par défaut du signal SIGINT afin que sur réception du signal, le processus affiche « Reception du signal SIGINT » dans la console.

Un processus peut stopper son exécution pour attendre un signal. Pour cela, on utilisera la fonction `pause()` de la librairie `unistd.h`.

3.5.

Modifiez la boucle du programme qui reçoit le signal afin qu'il *attende* la réception d'un signal avant d'afficher « Execution... ». Modifiez le programme qui envoie le signal pour qu'il envoie un signal SIGINT à intervalles réguliers (période = 1 sec). Testez le bon fonctionnement des programmes.