PARTIEL 2017-2018 - L2 (2h00)

Architecture des Ordinateurs et Systèmes d'Exploitation

Exercice 1: Conversions (2 points)

Question 1.1: (1 point) Convertir ces nombres décimaux en binaires sur 8 bits : `164`, `-34`

164 : % 1010 0100 ($ A4) -34 : % 1101 1110 ($ DE)

Question 1.2: (1 point) Convertir ces nombres binaires (8 bits signés et non signés) en décimaux : `%1111 0101`, `%1101 1011`

% 1111 0101 : -11 ou 245 % 1101 1011 : 219 ou -37

Exercice 2: Compilation (2 points)

Question 2.1: (2 points) Expliquer succinctement les 4 étapes du pipeline de compilation. (une ou deux phrase(s) par étape)

1. Pré-Compilation : suppression des commentaires, et résolution des macros/pragma/routines de précompilation
2. Compilation : transformation du code source écrit en langage haut niveau vers un code en assembleur dédié à la plateforme cible
3. Assemblage/Assembler : transformation du code assembleur (encore lisible et compréhensible par un humain) en code objet composé écrit en langage machine (suite de 0 et 1)
4. Édition de liens/Link Edit : résolution des adresses des fonctions et variables globales, concaténation des fichiers objets en un exécutable final, et indication du point d'entrée dans le programme

Exercice 3: Architecture (4 points)

Question 3.1: (2 points) Expliquer succinctement les 5 principales étapes qu'un processeur effectue lorsqu'il lit et exécute une instruction. (une phrase par étape)

1. Instruction Fetch : récupération de l'instruction à exécuter depuis la mémoire (instruction à l'adresse indiquée par le registre PC (Program Counter), puis incrémentation de 1 de l'adresse)
2. Decode : l'instruction récupérée est analysée pour connaitre quelle opération doit etre appliquée et quelles sont les opérandes impliquées (une addition a deux opérandes, un décalage n'a qu'une opérande, ...) afin de les rassembler
3. Execute : l'opération est exécutée sur les opérandes
4. Memory : si un accès mémoire en lecture/écriture doit etre fait, celui-ci est exécuté
5. Write-Back : l'état final est propagé dans les registres modifiés (si un JUMP/CALL est fait, le registre PC est mis à jour avec la bonne valeur)

Question 3.2: (1 point) Expliquer succinctement quelles sont les fonctions/l'utilité du processeur, de la mémoire, et des périphériques dans un ordinateur.

— Processeur : c'est le composant qui exécute les instructions des programmes et réagit aux interruptions qu'il reçoit des périphériques. Il accès à la mémoire pour mettre les données qui s'y trouvent.
— Mémoire : elle stocke les données du programme de façon volatile (si l'ordinateur est éteint, la mémoire perd son état/les données). Les données peuvent être des variables, des structures, et toute forme plus complexe.
— Périphériques : ils servent à intéragir avec l'utilisateur/le monde physique (périphériques d'entrée et périphériques de sortie) et d'autres machines/périphériques (périphériques d'entrées/sorties).

Question 3.3: (1 point) Pour un même programme dont les sources sont compilées pour un processeur CISC puis pour un processeur RISC, expliquer quelles seront les différences dans les deux binaires exécutables finaux. En admettant que les deux processeurs fonctionnent exactement à la même vitesse (chaque instruction prend un temps fixe à s'exécuter sur les deux processeurs), quelles seraient les conséquences lors de l'exécution ?

Étant donné qu'un processeur CISC effectue des instructions complexes, et qu'un RISC effectue des instructions simples, pour un même code source, on obtiendra un code assembleur "généralement" plus court en CISC/plus long en RISC, et donc des exécutables probablement plus gros en RISC qu'en CISC.
Si deux processeurs CISC et RISC exécutent chacun leur version de l'exécutable avec une durée fixe pour chaque instruction, alors l'exécutable sur RISC prendra plus de temps que sur CISC.
(malgré tout, dans la pratique, les processeurs CISC sont plus lents et plus complexes, et très peu de CISC "purs" existent de nos jours... il s'agit très souvent de RISC offrant une surcouche CISC).

Exercice 4: Système d'Exploitation (12 points)

Question 4.1: (2 points) Expliquer la notion de "temps partagé". Citer le composant du système d'exploitation qui gère cette notion, le mécanisme du processeur sur lequel il s'appuie, et le composant physique permettant cela. Puis illustrer avec un exemple où le temps est partagé puis un autre exemple où le temps n'est pas partagé.

Le temps partagé est la capacité d'un système à faire fonctionner plusieurs tâches "en apparence" en même en leur donnant régulièrement à chacune un peu de temps pour s'exécuter. Le composant du système d'exploitation qui permet de partager le temps est l'ordonnanceur, il s'appuie sur les interruptions que reçoit le processeur générées par un quartz.
Exemple de temps partagé : plusieurs tâches s'exécutent sur un système, un lecteur de musiques et un navigateur web... l'utilisateur utilise son navigateur et écoute la musique en même temps, il ne perçoit pas que les deux processus ne s'exécutent que quelques microsecondes chacun en alternance. Dans l'ordonnanceur, les deux tâches sont alternées pour permettre un usage optimal du temps : le temps que le fichier contenant de la musique soit lu par le disque dur au fur et à mesure de l'écoute, le navigateur web peut afficher des pages... et lorsque le navigateur attend une réponse des serveurs sur internet, le lecteur de musique peut prendre plus de temps CPU.
Exemple de temps non partagé : les "batchs". On lance une tâche, et uniquement lorsque celle-ci se

termine, on peut en lancer une deuxième. Ce mode n'a que très peu de sens/intérêt sur nos PC, car l'intéractivité est quasi inexistante. Dans le cas de la musique et du navigateur web, si on lance la musique en premier, il faudra attendre que l'ensemble du processus de lecture soit terminé avant de lancer une autre tâche... et lorsque le navigateur web est lancé, aucune musique externe ne peut être lancée, seule la navigation est possible.

Question 4.2: (1,5 points) Expliquer la différence entre un programme et un processus, et donner les propriétés classiques d'un processus.

— Programme : c'est le code qui sera exécuté. Il est stocké dans un fichier et peut être chargé en mémoire.

— Processus : c'est l'instance d'un programme en mémoire qui est en cours d'exécution. Chaque instance est différenciée par son PID.

Les propriétés classiques d'un processus sont :
— PID : identifiant du processus
— PPID : identifiant du processus parent/père
— État : état actuel du processus (actif, dormant, zombie, ...)
— User (UID) : les droits de l'utilisateur avec lequel le processus fonctionne
— Group (PGID) : les droits du groupe avec lequel le processus fonctionne
— Command : la commande avec laquelle le processus a été lancé
— Pages Mémoire : les pages mémoires associées à cette instance précise

Question 4.3: (1 point) Expliquer la différence entre processus et threads.

Un processus dispose des propriétés précédentes, et chaque processus est isolé des autres grâce à la mémoire virtuelle. Les threads sont des *fils d'exécution/processus légers* qui partagent le code/-programme du processus auquel ils sont rattachés, ainsi que la section *tas* de l'espace d'adressage. Cependant, chaque thread dispose de sa propre pile d'appels pour pouvoir exécuter son propre "cheminement dans le code"/fil d'exécution.

Question 4.4: (2 points) Expliquer la relation entre i-node et blocs, puis expliquer la différence entre fichier et dossier et particulièrement où sont stockés les noms de fichiers.

Un i-node est une structure référençant les blocs constituant le contenu du fichier. L'i-node contient donc une liste de numéros de blocs, et l'ordre dans lequel les lire (ainsi que d'autres métadonnées propres au fichier). Les blocs, au contraire, ne contiennent "que" de la donnée.
Les blocs d'un fichier ne contiennent donc que le contenu du fichier (de la donnée brute). À l'inverse, les blocs d'un dossier contiennent la liste des fichiers présents dans le dossier, et le numéro de chaque i-node associé à chaque fichier.
Ainsi, les i-nodes ne contiennent pas les noms de chaque fichier référencé, mais on retrouve le nom de chaque fichier dans les blocs du dossier les contenant.

Question 4.5: (1,5 points) Donnez 3 raisons différentes qui peuvent empêcher cette ligne de commande de fonctionner : `./_script.sh`

- Droit exécution pas donné/Problème de droit
- Erreur dans le script (le langage n'est pas du script shell)
- Le script n'existe pas
- L'interpréteur indiqué au début n'existe pas (le `#! /bin/sh` remplacé par `#! /lol`)

Question 4.6: (2 points) Expliquer ce que fait chaque ligne du script, puis globalement ce que le script semble faire.

```
1 #! /bin/sh
2
3 OUTFILE=`mktemp fileXXXX`
4 cat streams.list | sort -t";" -k1 -o $OUTFILE
5 NB=`wc -l streams.list | cut -c 1`
6 rm -f $1
7 for i in `seq 0 $NB`; do
8   FILE=`tail -n $i $OUTFILE | head -n 1`
9   cat "${FILE}$i" >> $1
10 done
11 rm -f $OUTFILE
```

script.sh

- Ligne 1 : On appelle l'interpréteur de script shell `sh`
- Ligne 3 : On crée un fichier temporaire dont le préfixe est "file" et qui est fini par 4 caractères aléatoires, le nom sera sauvegardé dans la variable OUTFILE
- Ligne 4 : On lit le fichier "streams.list" et on envoie son contenu sur la sortie standard. Celle-ci est redirigée vers l'entrée de "sort" qui va trier selon le 1er champs, les champs sont séparés par des point virgules ";", et écrira dans le fichier temporaire désigné par la variable OUTFILE
- Ligne 5 : La variable NB va prendre comme valeur le nombre de lignes contenues dans le fichier "streams.list" (wc va compter le nombre de lignes et affichera plusieurs résultats, puis cut ne sélectionnera que le 1er caractère contenant le nombre de lignes... il faut en déduire que streams.list doit contenir moins de 10 lignes)
- Ligne 6 : Le fichier passé en premier paramètre du script sera supprimé (qu'il existe ou non : aucun message ne sera affiché)
- Ligne 7 : On va itérer avec la variable "i" depuis l'entier 0 jusqu'à l'entier contenu dans la variable "NB" (elle contient le nombre de ligne de streams.list)
- Ligne 8 : La variable FILE va prendre comme valeur chaque ligne du fichier temporaire désigné par la variable "OUTFILE" (tail va afficher les "i" dernières lignes, et head ne va garder que la 1ère ligne de ce que tail a affiché... tail va afficher de moins en moins de lignes)
- Ligne 9 : Chaque ligne contenue dans la variable "FILE" sera concaténée à la valeur de "i", et cela formera le nom d'un fichier (cat travaille sur des fichiers, echo travaille sur des phrases). On va afficher le contenu du fichier dont le nom est listé dans "streams.list" + suivi du numéro de ligne, et on va renvoyer le tout vers le fichier passé en paramètre du script SANS écraser le contenu précédent
- Ligne 11 : On supprime le fichier temporaire désigné par la variable OUTFILE

De façon générale, le script va lister des fichiers référencés dans "streams.list" et concaténer leur contenu dans le fichier passé en 1er paramètre du script.

Question 4.7: **(2 points) Écrire un script qui listera tous les fichiers commençant par le nom "transaction-" modifié durant les 24 dernières heures, puis afficher pour chaque fichier la colonne contenant le montant de chacune des lignes, et renvoyer le tout vers le fichier passé en premier paramètre du script.**

```
ID Transaction ; Date ; Montant ; Numero Carte Fidelite ; Remarques
```

Format des fichiers de transactions

```
1445;2017-12-08;42;1001;fait a 11h01
1446;2017-12-08;200;1002;fait a 11h11
1447;2017-12-08;100;1001;fait a 12h04
```

transaction-2017-12-08

```
#! /bin/sh

find . -name "transaction-*" -mtime -1 -exec cut -f 3 {} \; >> $1
```

script.sh