# Pale INF570 – Systèmes d'exploitation

2009–2010

- L'examen dure 2 heures.
- Tous documents autorisés.
- Il est impératif de commenter les programmes et de justifier vos réponses.

## Exercice 1

### Question 1

Décrivez la fonction réalisée par le programme suivant, et décrivez (succintement) le rôle de chaque instruction.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  FILE *fic;
  int i = 0;
  if (argc != 2) {
    fprintf(stderr, "Usage: %s fichier\n", argv[0]);
    exit(1);
  }
  fic = fopen(argv[1],"r");
  if (fic == NULL) {
    perror("fopen");
    exit(2);
  }
  while (getc(fic)!=EOF) i++;
  printf("%s: %d\n", argv[1], i);
  exit(0);
}
```

Quelle commande UNIX classique est implémentée par ce programme ?

### Question 2

Ce programme est très inefficace, pourquoi ? La bibliothèque standard du langage C, qui implémente la fonction `getc()`, utilise un mécanisme de tampon mémoire (buffer) pour y remédier. Décrivez informellement ce mécanisme, en illustrant son fonctionnement lors de l'exécution du programme précédent sur un gros fichier.

### Question 3

Modifiez le programme précédent pour implémenter directement ce mécanisme de tampon mémoire, sans utiliser les fonctions d'entrée-sortie de la bibliothèque C. Vous utiliserez directement les appels système pour les entrées-sorties POSIX (`open`, `read`, etc.). Prenez garde au traitement des nombreux cas d'erreur ou d'exécution incomplète des appels système. Commentez votre programme pour expliquer les principales difficultés et les choix de conception importants.

# Exercice 2

Il s'agit d'un questionnaire à choix multiple sur le C et le système. Écrivez sur votre copie le numéro de la réponse choisie.

Notation : 1 point par bonne réponse, 0 points en l'absence de réponse, $-0,5$ points par réponse fausse. La note obtenue entre $-2$ et 4 est ensuite ramenée au nombre de points accordé à l'exercice dans le barème.

1. Que déclare la ligne de C suivante ?

```
int(*p)(void*);
```

   (1) un pointeur de fonction sans arguments
   (2) rien, la syntaxe est incorrecte
   (3) une fonction retournant un pointeur d'entiers
   (4) un pointeur de fonction avec argument pointeur quelconque

2. Que vaut `x` après l'exécution des lignes suivantes ?

```
int x=1, y=2;
int *p=&x;
p=&y; *p+=1;
p=&x; *p+=y;
```

   (1) 1 (2) 2 (3) 3 (4) 4

3. Le code suivant, nommé `Bond.c` est compilé avec la commande `gcc Bond.c -o Bond` :

```
#include <stdio.h>
#define REFREF **
int main (int argc, char REFREF argv) {
  printf ("My name is %s, James %s", (*argv)+1, *(argv+1));
}
```

   Que produit l'exécution de la commande `Bond BOND` dans le shell ?
   (1) elle écrit `My name is Bond, James Bond`
   (2) elle écrit `My name is ond, James BOND`
   (3) elle écrit `My name is Cond, James BOND`
   (4) rien, la compilation donne une erreur de type

4. Que fait le code suivant :

```
#include <stdio.h>
int main (int argc, char *argv[]) {
  char buf[100];
  FILE *fd=fopen ("toto", "r+");
  read (fd, buf, 100);
}
```

   (1) il ouvre un fichier `toto` en lecture seule et lit 100 octets
   (2) il ouvre un fichier `toto` en lecture/écriture et lit 100 octets
   (3) rien, la syntaxe est incorrecte
   (4) n'importe quoi, les types des arguments sont incorrects

# Exercice 3

Le but de l'exercice consiste à écrire une commande `repete` pour exécuter un même programme plusieurs fois de suite, en utilisant des tubes (`pipe`) pour connecter la sortie de la $k$-ième exécution sur l'entrée de la $k+1$-ième. Par exemple :
`$ repete 3 programme`
est équivalent à

```
$ programme | programme | programme
```

Les exécutions des différentes instances du programme donné en argument de `repete` doivent pouvoir s'effectuer en parallèle, et le code de retour de la commande doit être le nombre de fois que le programme a réellement été exécuté.

### Question 4

Écrivez le code correspondant au cas où on souhaite n'exécuter qu'une seule fois le programme donné en argument.

### Question 5

Pour $n$ exécutions du programme, $n > 1$, on propose le schéma d'exécution suivant. Le processus $P_0$ initié par la commande `repete` crée $n-1$ tubes puis $n$ processus en affectant la sortie standard du processus $P_k$ au premier descripteur du tube $T_k$ et l'entrée standard du processus $P_{k+1}$ au deuxième descripteur du tube $T_k$. L'entrée du processus $P_1$ et la sortie du processus $P_n$ restent inchangées (celles de $P_0$). Enfin, chaque processus fils ($P_k$, pour $1 \leq k \leq n$) exécute le programme donné en deuxième argument de la commande `repete`.

Écrivez le code correspondant au cas où $n = 2$ (sans utiliser les fonctions de la bibliothèque C `popen` et `system`).

### Question 6

Écrivez le code correspondant au cas où $n > 2$.

### Question 7

Le système d'exploitation impose une limite sur le nombre de descripteurs ouverts.

Expliquez selon vous d'où vient cette limite par rapport au fonctionnement et aux structures internes du noyau, sa portée (processus ? utilisateur ? système ?) et la primitive système qui risque d'échouer.

### Question 8

Proposez un nouveau schéma d'exécution qui évite de saturer le nombre de descripteurs, sans restreindre les opportunités d'exécution parallèle des instances du programme. Expliquez comment modifier le programme pour ce nouveau schéma d'exécution (inutile de réécrire tout le programme).

### Question 9

En considérant un grand nombre d'exécutions, la table des processus risque elle même d'atteindre une limite maximale. Montrez sans écrire de code qu'il existe une solution *générale* pour conduire tout de même l'exécution à son terme (si la mémoire est suffisante), en vérifiant l'état de remplissage des tubes et en ne créant de nouveau processus que si nécessaire.

### Question 10

Modifiez le programme en ce sens.

### Question 11

Ce comportement n'est pas toujours très intéractif : que se passe-t-il en particulier lors d'une éxécution (stupide) du type

```
$ repete 10000 cat
```

lorsque l'entrée standard est un terminal (10000 est supposé supérieur au nombre maximal de processus) ?

Modifiez donc le programme pour ne chercher à conduire l'exécution à son terme en bloquant la création de nouveaux processus que lorsque l'entrée standard est un *fichier ordinaire*.

### Question 12

(Il s'agit d'une question ouverte nécessitant une étude et des explications plus approfondies que les questions ordinaires.)

Pour maximiser les performances de l'exécution sur un processeur multicoeur, il est important de veiller à l'équillibrage de la charge de calcul dévolue à chaque coeur. Ainsi, l'exécution de la commande

```
repete 4 programme
```

sur un processeur disposant de 4 coeurs de calcul est généralement loin d'être 4 fois plus rapide que sur un unique coeur, car les processus sont souvent bloqués en attente de données en entrée, ou bloqués en raison de la saturation de leur tube de sortie. Proposez une modification de l'ordonnanceur de processus du noyau, pour que celui-ci favorise les processus ayant davantage de travail en attente dans leur tube d'entrée, et pénalise les processus écrivant dans un tube proche de la saturation. Sans écrire de code, vous détaillerez les informations

nécessaires à cette adaptation dynamique de la politique d'ordonnancement des processus, le cheminement à travers les structures internes du noyau pour obtenir ces informations, et la conception des fonctions d'entrée-sortie sur les tubes pour que ces informations soient mises à jour au fur et à mesure des appels à `read()` et `write()` sur ceux-ci.