Nom :

Prénom :

Polytech Nice Sophia Antipolis
SI3 — 2015–2016

Programmation-système
Examen final : mercredi 1er juin 2016
Jean-Paul Rigault

L'examen comporte 8 pages. Les réponses doivent être directement fournies **sur ce document lui-même**. Écrivez lisiblement et proprement. **Évitez les ratures et les renvois** (il est préférable d'utiliser un crayon de papier et une gomme). Essayez de respecter la place allouée à chaque réponse. Aucun intercalaire n'est nécessaire et ne sera fourni. Dans les extraits de programme demandés, il est **inutile** de spécifier les `#include` nécessaires. Les questions principales sont indépendantes et beaucoup de sous-questions le sont aussi. L'information de notation n'est fournie qu'à titre indicatif.

Aucun document n'est autorisé sauf dérogation explicite (par exemple, un dictionnaire Français-Chinois) et **tout moyen de communication est strictement interdit** (téléphone ou ordinateur portable, sémaphore, signaux de fumée…). La durée de l'examen est de **2 heures** (2 heures 40 minutes en cas de tiers temps).

1 Questions de pages [3/20]

1.1 Exemple de remplacement

Supposons que la mémoire physique d'un ordinateur ne comporte que 4 pages (!). À un instant donné, les temps de chargement en mémoire et de dernière référence (lecture ou écriture), ainsi que le bit de référence (R) et celui de modification (M) sont indiqués dans le tableau suivant.

page	temps chargement	temps référence	R	M
0	126	200	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

Les temps sont donnés dans un unité quelconque, mais dont la valeur croit avec le « vrai » temps. On interprète donc ce tableau ainsi, par exemple pour la page 1 : elle a été chargée à l'instant $t = 230$ et a été référencée dans la dernière tranche de temps ($R = 1$) à $t = 260$ soit 30 unités de temps plus tard. Elle n'a pas été modifiée depuis qu'elle réside en mémoire ($M = 0$).

Question 1
Indiquez la première page qui sera remplacée par chacun des algorithmes de remplacement suivants : FIFO *(*first in, first out*),* LRU *(*least recently used*),* NUR *(*not used recently*).*
Réponse 1
FIFO 2
LRU 0
NUR 0

Question 2
Justifiez rapidement votre réponse
Réponse 2
FIFO choisit la page la plus ancienne (2, chargée à $t = 120$), LRU la moins récemment utilisée (0, utilisée à $t = 200$). Quant à NUR, il choisit aussi la page 0 puisqu'elle n'a pas été utilisée dans la dernière tranche de temps ($R = 0$) ni été modifiée depuis qu'elle est en mémoire ($M = 0$).

1.2 Coût des algorithmes de remplacement

Les algorithmes de remplacement de page vus en cours (RANDOM, FIFO, LRU, NUR) requièrent des informations dans la table des pages (bit M ou R, droits d'accès aux pages…) qui sont en principe gérées directement par le matériel (la MMU, *Memory Management Unit*) lors de l'exécution d'une instruction à référence mémoire. ***Ces informations de la table des pages ne sont pas concernées par cette question.***

Mais ces algorithmes requièrent aussi des informations allouées par le système d'exploitation lui-même dans son propre espace d'adressage pour gérer le remplacement (liste ou file de pages par exemple). Ces informations sont évidemment plus « chères » à manipuler que celles qui le sont par le matériel. En utilisant comme critères de coût

- la taille de ces structures de données propres au système (par rapport au nombre de pages physiques ou virtuelles),
- la fréquence à laquelle ces structures doivent être mises à jour (à chaque accès, uniquement lors du remplacement d'un page…),

veuillez répondre à la question suivante:

Question 3
Indiquez l'ordre dans lequel se classent les quatre algorithmes RANDOM, FIFO, LRU, NUR *en taille des structures de données (le moins cher en premier).*
Réponse 3
RANDOM NUR LRU/FIFO (LRU et FIFO ont des structures de données équivalentes)

Question 4
Indiquez l'ordre dans lequel se classent les quatre algorithmes RANDOM, FIFO, LRU, NUR *en fréquence de mise à jour (le moins cher en premier).*
Réponse 4
RANDOM FIFO LRU NUR (NUR et LRU ont des fréquences de mise à jour équivalentes mais NUR requiert en plus la mise à 0 du bit R à chaque tranche de temps).

Question 5
Justifiez rapidement vos deux réponses précédentes.
Réponse 5
RANDOM ne nécessite aucune structure de données, et donc aucune mise à jour. NUR a besoin d'une structure de données très légère (2 bits) mais qui doit être mise à jour à chaque accès [a]. LRU et FIFO ont des structures de données identiques, une file de toutes les pages chargées. Pour FIFO, cette file n'est mise à jour que lors du chargement d'une page, alors que pour LRU c'est à chaque accès (comme NUR) donc plus fréquemment.

a. On peut noter cependant que ces deux bits et leur gestion sont la plupart du temps fournis directement par le matériel (la MMU).

2 Questions courtes [3/20]

2.1 Une variable peut en cacher une autre…

Considérez le programme suivant :

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

main()
{
    int i = 0;

    if (fork()) i++;
    i++;
    printf("%d\n", i);
```

```
}
```

Question 6
Qu'affiche-t-il sur la sortie standard lors de son exécution ?
Réponse 6
1 suivi de 2 ou 2 suivi de 1

Question 7
Justifiez (rapidement) votre réponse.
Réponse 7
Après le `fork()`, `i` désigne deux variables indépendantes, une dans le père, l'autre dans le fils. Celle du père (où `fork()` a retourné un résultat non nul) est incrémenté deux fois, celle du fils (où `fork()` valait 0) l'est une seul fois. En outre l'ordre d'exécution du père et du fils est indéfini.

2.2 Comportement des démons

Certains « démons » (la plupart ?) des systèmes Posix débutent leur fonction `main()` de la manière suivante :

```
int main()
{
    if (fork()) return 0;
    // ... reste du code ...
}
```

Question 8
Quel processus exécute le « reste du code » : le père ou le fils ?
Réponse 8
Le **fils**, devenu orphelin. Rappelons que `if (fork())...` équivaut à `if (fork() != 0)...`

3 Fonction `popen()` [7/20]

La fonction de bibliothèque `popen()` présente le prototype suivant :

```
FILE *popen(const char *cmd, const char *mode);
```

où `cmd` est une chaîne de caractères représentant une commande au **shell**, et `mode` est soit la chaine `"r"` soit la chaine `"w"`. Cette fonction crée un « pipe », et transmet la commande `cmd` à `/bin/sh` pour exécution. Si `mode` est `"r"`, la commande `cmd` doit s'exécuter avec son entrée standard redirigée sur l'extrémité de lecture du pipe et la fonction retournera l'extrémité d'écriture du pipe à l'appelant. Si mode est `"w"`, c'est la sortie standard de la commande qui est redirigée sur l'extrémité d'écriture du pipe et c'est l'extrémité de lecture que la fonction retournera à l'appelant. En cas d'erreur, `popen()` retourne le pointeur `NULL`.

> **Remarque importante** La commande `cmd` passée en paramètre de `popen()` peut être une commande **shell** quelconque et arbitrairement complexe, contenant des redirections, des pipes, des caractères spéciaux, etc. Par exemple
>
> ```
> FILE *fp = popen("ls -l foo* | wc -l 2> bar", "r");
> ```
>
> Le traitement de ces divers mécanismes devra être fait par le **shell**. On utilisera pour cela l'option **-c** du **shell** vue en TD. Par exemple, si vous tapez au terminal la commande
>
> ```
> sh -c "ls -l foo* | wc -l > bar"
> ```
>
> vous obtenez une manière (ici un peu tordue) d'exécuter la commande
>
> ```
> ls -l foo* | wc -l > bar
> ```

> **Attention** Notez que, conformément à la description précédente, la fonction `popen()` retourne un flux de *lecture* quand `mode` est `"w"` (la commande `cmd` *écrit* dans le pipe) et d'*écriture* quand `mode` est `"r"` (la commande `cmd` *lit* depuis le pipe).

Le flux retourné par `popen()` n'est pas un « descripteur de fichier » de Posix (un simple entier) mais bien un `FILE *`, c'est-à-dire un flux standard de la bibliothèque **stdio** (quelque chose comme `stdin` ou `stdout`, que l'on peut donc utiliser avec `printf()`, `scanf()`, `fgets()`, etc.). Vous aurez donc besoin de la fonction suivante (définie dans `<stdio.h>` et qu'il n'est donc pas besoin d'écrire) :

```
FILE *fdopen(int fd, const char *fdmode);
```

où `fd` est le descripteur Posix à transformer en `FILE *` et `fdmode` est la direction de ce flux (`"r"` pour un flux de lecture, `"w"` pour un flux d'écriture).

Exemple Pour fixer les idées, voici un exemple de programme utilisant `popen()` (les directives `#include` et `#define` nécessaires ont été omises) :

```
int main()
{
    char line[MAX_LINE];

    FILE *fp = popen("ls -l", "w");     // création du pipe et lancement de la commande
    assert (fp != NULL);
    while (fgets(line, MAX_LINE, fp))   // lecture ligne par ligne
        puts(line);                     // affichage de la ligne lue
    }
}
```

Question 9

On vous demande d'écrire ci-après le code de la fonction `popen()`, mais ***uniquement dans le cas où le second argument (`mode`) est égal à `"r"`***

Réponse 9

```
FILE *popen(const char *cmd, const char *type)
{
    int p[2];
    pipe(p);
    if (strcmp(type, "r") == 0)
    {
        if (fork() != 0)
        {
            // surtout pas de wait() ici!! Il faut que le père...
            // ... puisse continuer pour écrire dans le pipe retourné
            close(p[0]); // mais pas p[1] qui va continuer à être utilisé ...
            return fdopen(p[1], "w"); // ... encapsulé dans l'objet FILE retourné
        }
        else
        {
            dup2(p[0], 0);
            close(p[0]);
            close(p[1]);
            execlp("sh", "sh", "-c", cmd, NULL);
            perror("exec read");
            exit(1);
        }
    }
    // .... reste du code de popen() ...
```

Question 10
Que pensez-vous de l'invocation suivante de `popen()` *:*
`FILE *fp = popen("ls -l > foo", "w");`
Que lira le programme sur `fp` *? Que trouvera-t-on dans le fichier* `foo` *?*
Réponse 10
La redirection de la sortie standard effectuée par la commande elle-même (`> foo`) supplantera celle effectuée (avant) par `popen()`. Donc `foo` contiendra le résultat de la sortie standard de la commande et rien n'apparaitra sur le `FILE *fp` (sinon, immédiatement, une fin de fichier).

4 Un peu de plomberie [7/20]

Considérez le programme suivant, dans le fichier `pipes.c`. Les deux fichiers `progP` et `progQ` sont des binaires exécutables dont on n'a pas besoin (ici) de savoir ce qu'ils font et qu'il est donc inutile d'essayer de définir vous-mêmes. Ces deux exécutables sont supposés résider dans le répertoire courant (`./`) : `progP` prend un seul argument de ligne de commande, un entier ; `progQ` ne prend aucun argument.

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <stdio.h>
5  #include <assert.h>
6  #include <sys/types.h>
7  #include <sys/wait.h>
8
9  #define N 4
10 int p[N][2];
11 int q[2];
12
13 void create_pipes() {
14     for (int i = 0; i < N; ++i) {
15         pipe(p[i]);
16         pipe(q);
17     }
18 }
19
20 static char buffi[5];
21 void createP(int i) {
22     if (fork() == 0) {
23         dup2(p[i][0], 0);
24         dup2(p[(i+1)%N][1], 1);
25         dup2(q[1], 2);
26         sprintf(buffi, "%d", i); //
27         execl("./progP", "progP", buffi, NULL);
28         perror("exec progP");
29         exit(1);
30     }
31 }
32
33 void createQ()
34 {
35     if (fork() == 0) {
36         dup2(q[0], 0);
37         execl("./progQ", "progQ", NULL); //
38         perror("exec progQ");
39         exit(1);
40     }
41 }
```

```
42
43  int main()
44  {
45      create_pipes();
46      for (int i = 0; i < N; ++i) createP(i);
47      createQ();
48      for (int i = 0; i < X; ++i) wait(NULL); //
49      return 0;
50  }
```

4.1 Petites questions simples

Question 11
Il est indispensable que chaque processus (père et fils) ferme tous les descripteurs de fichiers dont il n'a pas besoin. Indiquer ici le numéro des lignes du programme pécédent ***devant*** *lesquelles on doit invoquer cette fermeture et quels sont les descripteurs qu'il convient de fermer.*
Réponse 11
Devant les lignes 26, 37 et 48, fermer **tous** les descripteurs des tableaux `p` et `q`.

Question 12
Combien le programme ci-dessus crée-t-il de processus fils.? Autrement dit, quelle valeur faut-il donner au paramètre `X` à la ligne 48 ?
Réponse 12
N fils exécutant `progP` et 1 exécutant `progQ`, au total $N + 1$, soit 5.

4.2 Structure de communication

On vous demande maintenant de dessiner (sur la page suivante) la structure de processus, de tubes et de redirections que ce programme met en place. Pour cela vous vous utiliserez la *notation* de la figure 1 qui décrit, *à titre d'exemple*, la structure nécessaire à l'exécution de la (célèbre) commande au **shell** : `ls | wc -l`. c

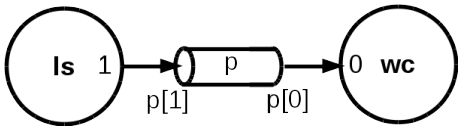


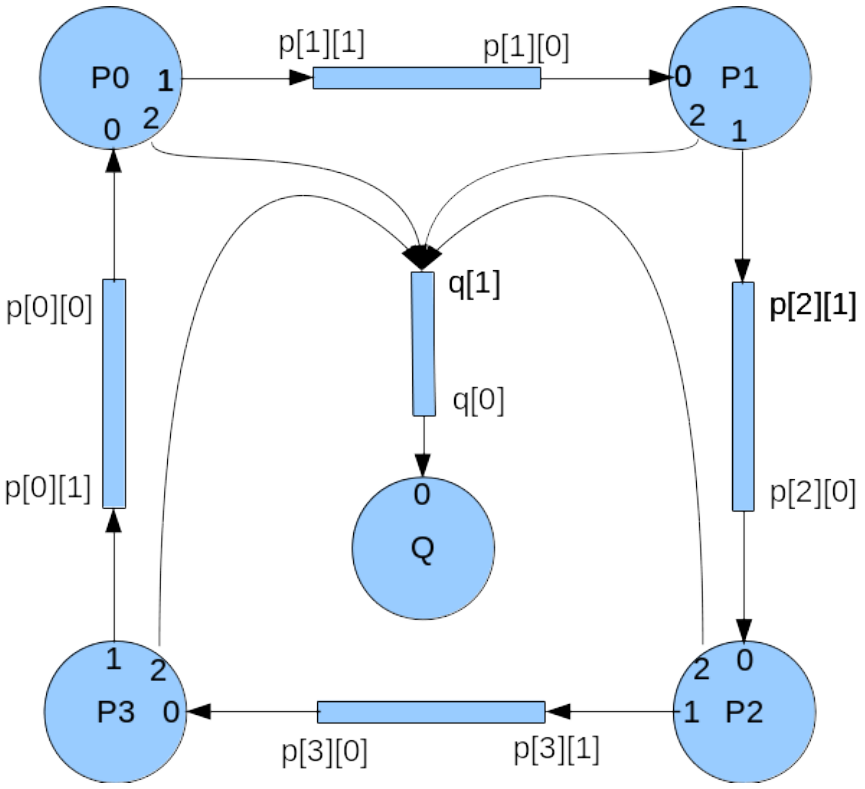
FIGURE 1 – Exemple de notation : processus, tubes et redirections. Le processus exécutant le programme **ls** a sa sortie standard (descripteur 1) redirigée sur l'extrémité d'écriture du pipe `p` (`p[1]`) ; le processus exécutant le programme **wc** a son entrée standard (descripteur 0) redirigée sur l'extrémité de lecture du pipe `p` (`p[0]`).

Question 13
Si vous souhaitez commenter ou justifier votre réponse à la question de la page suivante, faites-le ici.
Réponse 13
Les processus exécutant `progP` forment un anneau : ils sont connectés grâce aux pipes `p` par leur entrées et sorties standard. Le flux d'erreur standard de tous ces processus est connecté à l'extrémité d'écriture du pipe `q` et l'extrémité de lecture de ce pipe est connectée à l'entrée standard du processus exécutant `progQ`.

Question 14

En utilisant la notation de la figure 1, dessiner la structure de processus du programme `pipes.c` *dans ce cadre.*

Réponse 14



5 Doutes, troubles, questions… Exprimez-vous !