ROB4 - InfoSys

17 janvier 2021
Devoir sur table : 90 min
Tous documents papier autorisés. Ordinateurs, téléphones et calculatrices interdits.

NOM et PRÉNOM:

Barème – 20 points + 2 bonus

Exercice	Points	Theme
1	7	QCM
2	2	Cours
3	2.5	Pile
4	4	Fork
5	4.5 + 2 bonus	Exec, pipe, fichiers

Exercice 1 – QCM (7 points - 0.5 par question)

— Cochez la bonne réponse.
— Il y a une seule bonne réponse par question.
— Chaque question vaut 0.5 point si la bonne réponse (et elle seule) est cochée et 0 dans tous les autres cas. (Les réponses fausses ne retirent pas de points supplémentaires.)

1. En langage C, l'utilisation de *define guards* (`#ifndef FOO_H` / `#define FOO_H` / `#endif`)

   o Est indispensable pour que le programme compile
   o Ne concerne que les fichiers `.c` (et non `.h`)
   o Permet d'éviter certaines erreurs de segmentation
   o Facilite le débogage d'un programme avec `gdb`
   o Permet d'éviter des fuites mémoire
   X Aucune des propositions précédentes n'est correcte

Les *define guards* permettent d'éviter qu'un même fichier header soit inclus plusieurs fois, ce qui pourrait causer des erreurs de compilation dues à des définitions multiples.

2. Etant donné :

```
1 struct p3d { // a point in 3D space
2         float x;
3         float y;
4         float z;
5 };
6 struct p3d seg3d[2]; // a segment in 3D space
```

Indiquez laquelle de ces propositions logiques est vraie :

- o `sizeof(seg3d) == sizeof(struct p3d*)`
- o `sizeof(seg3d) == 2*sizeof(float)`
- o `sizeof(seg3d) == sizeof(float)`
- X `sizeof(seg3d) == 6*sizeof(float)`
- o `sizeof(seg3d) == 3*sizeof(float)`
- o Aucune des propositions précédentes n'est correcte

La taille d'une structure est la taille de la totalité de ses champs, et la taille d'un tableau correspond à sa longueur multipliée par la taille d'un élément.

3. Lors d'une écriture dans un tube avec `write`, si le nombre d'octets écrits est inférieur à la constante symbolique PIPE_BUF (généralement 4096 octets) l'écriture est qualifiée d'atomique. Cela signifie :

- o Que le processus lecteur du tube doit utiliser un appel système particulier (autre que `read`) pour lire ces données
- o Que ces données sont placées directement au début de la file d'attente et seront les prochaines lues par le processus lecteur
- o Que l'écriture est forcément réalisée immédiatement (`write` ne peut pas être bloquant)
- X Que l'ensemble de ces données sont écrites en une seule fois
- o Que le processus lecteur du tube est immédiatement ordonnancé après l'appel `write` et reçoit les données
- o Aucune des propositions précédentes n'est correcte

L'écriture atomique permet notamment de garantir que, quand le processus lecteur lira le tube, il pourra y trouver l'ensemble des données écrites et non juste une partie. Lorsque les écritures sont d'une taille

supérieure à PIPE_BUF, on peut avoir un changement de contexte (vers le processus lecteur, notamment) avant que tout ne soit écrit.

4. Une exécution de `make`, produit les messages d'erreur suivants :

```
15:40 endy@demandred ~/src/testc% make
gcc -c monprogramme.c
gcc monprogramme.o -o monprogramme
monprogramme.o : Dans la fonction « main » :
monprogramme.c:(.text+0x15) : référence indéfinie vers « mafonction »
collect2: error: ld returned 1 exit status
Makefile:2 : la recette pour la cible « monprogramme » a échouée
make: *** [monprogramme] Erreur 1
zsh: exit 2     make
```

Cette erreur a pu être causée par...

- o L'omission de *define guards* dans une fichier
- o L'omission d'une directive `#include` dans `monprogramme.c`
- o L'omission d'un paramètre `-I<qqchose>` dans un appel à GCC
- X L'omission d'un fichier `.o` dans la recette pour la cible `monprogramme`
- o L'omission d'un fichier `.h` dans la recette pour la cible `monprogramme.o`
- o Aucune des propositions précédentes n'est correcte

Cette erreur est une erreur d'édition de liens. La compilation (`gcc -c monprogramme.c`) se passe correctement. L'édition de liens (`gcc monprogramme.o -o monprogramme`) échoue car le linker ne trouve pas, dans aucun fichier objet ou bibliothèque fournies, le code compilé de la fonction "mafonction", dont le prototype a été donné dans `monprogramme.c` ou plus probablement dans un fichier `.h` inclus.

Ce code compilé peut se trouver soit dans un fichier `.o` manquant – c'est la réponse – soit dans une bibliothèque qui aurait été omise (paramètre `-l<nom_de_bibliothèque>` manquant). L'omission de *define guards*, de `#include` ou de paramètres `-I` ne peuvent causer que des erreurs de compilation et pas d'éditions de lien.

5. Un processus Linux *zombie* peut être détruit... :

- o en tapant CTRL+Z dans le shell pendant son exécution
- o en détruisant la partie *head* du processus
- X par son parent, en appellant `wait(NULL);`
- o par son enfant, en appellant `wait(NULL);`
- o il ne peut pas être détruit sans redémarrage du système

o aucune des propositions précédentes n'est correcte

CTRL+Z suspend temporairement le processus au premier plan dans la console ; aucun lien avec les zombies.

6. Deux processus apparentés communiquent par un tube. Le processus écrivain ferme le tube en écriture. Puis, le processus lecteur utilise `read` pour lire dans le tube.

   o Rien de particulier, la lecture fonctionne et lit les dernières données placées dans le tube.
   X L'appel à `read` renvoie 0, ce qui correspond à "fin de fichier"'
   o L'appel à `read` est bloquant jusqu'à ce que le fifo soit rouvert en écriture.
   o Le processus lecteur reçoit le signal SIGPIPE
   o L'appel à `read` échoue (il renvoie -1) et `errno` prend la valeur correspondant à "Broken pipe"
   o Aucune des propositions précédentes n'est correcte

   Tenter de lire dans un tube fermé en écriture se comporte globalement comme tenter de lire un fichier vide. Un appel à `read` est bloquant si le tube est vide, et non fermé. Quant au, SIGPIPE il est reçu par le processus écrivain dans la situation inverse (on essaye d'écrire dans un tube fermé en lecture). Ce n'est pas possible de rouvrir un tube anonyme une fois qu'il a été fermé.

7. Dans l'espace mémoire d'un processus sous GNU/Linux, la pile :

   o Est l'espace mémoire dans lequel pointent les pointeurs de fonction
   o Part d'une adresse mémoire basse et croît vers le haut
   o Contient les variables globales utilisées par le processus
   o Contient les espaces mémoire alloués par `malloc`
   o Contient les espaces mémoire alloués par `mmap`
   X Contient les arguments des fonctions en cours d'exécution
   o Aucune des propositions précédentes n'est correcte

   La pile part d'une adresse haute et croît vers le bas. Elle contient les variables locales, les valeurs de retour et les arguments des fonctions en cours d'exécution. Les pointeurs de fonctions pointent vers le code des fonctions, dans le segment `text`. Les variables globales sont dans le segment `data`. Les espaces alloués par `malloc` sont entre `&end` et le *program break*, ceux utilisés par `mmap` sont dans l'espace dit non alloué, entre le *program break* et la pile.

8. Initialiser les variables de type pointeurs à `NULL` :

- o Permet d'éviter des erreurs de segmentation
- o Ne doit jamais être fait car cela entraîne des erreurs de segmentation
- o Entraîne l'émission d'un warning par GCC
- X Est une bonne pratique qui facilite la correction des bugs de gestion mémoire
- o Est indispensable ; le programme ne peut être compiler sans
- o Aucune des propositions précédentes n'est correcte

> On ne doit déréférencer un pointeur que s'il pointe vers une adresse valide. Déréférencer un pointeur nul est incorrect et provoque toujours immédiatement une erreur de segmentation. Un pointeur non initialisé contient une valeur arbitraire, le déréférencer est tout aussi incorrect que déréférencer un pointeur nul (vu qu'il ne pointe pas vers une adresse valide) mais provoque une comportement aléatoire (selon la valeur du pointeur) ce qui est souvent l'origine d'erreurs mémoires obscures et difficiles à débugger. Par conséquent, initialiser ces pointeurs à NULL est une bonne pratique qui permet de détecter et corriger certaines erreurs mémoire dues aux pointeurs plus facilement.
>
> L'idéal est de gérer vos pointeurs de façon à ce qu'ils pointent toujours soit vers une adresse valide, soit vers NULL : les initialiser à NULL ou à une adresse valide, et si l'adresse vers laquelle ils pointent devient invalide (par exemple parce que c'est l'adresse d'une variable sur le tas qui vient d'être free()), les remettre à NULL.

9. Considérons `int main(int argc, char** argv)`. Alors `argv[argc-1]` :

- X contient le dernier argument passé au programme
- o est de type `char*[]`
- o contient le nom du programme
- o contient le PID du parent du processus exécutant ce programme
- o contient l'avant-dernier argument passé au programme
- o aucune des propositions précédentes n'est correcte

> argv est un tableau de chaines de caractères. Il contient argc chaînes de caractères, de argv[0] à argv[argc−1] (et par ailleurs argv[argc] == NULL). argv[0] est le nom du programme. argv[1] à argv[argc−1] sont les arguments ; argv[argc−1] est donc le dernier argument passé au programme.
>
> Les argv[i] sont de type **char***.

10. Un avantage des bibliothèques partagées (fichiers `.so` sous linux ou `.dll` sous windows) sur les bibliothèques statiques est :

- o De faciliter les échanges de données entre mode noyau et mode utilisateur
- o De permettre la création d'espaces mémoire partagés au sein duquel deux processus peuvent échanger des donnéees
- X De ne charger en mêmoire qu'une seule instance d'une bibliothèque utilisée par plusieurs processus
- o De créer des threads partagés entre plusieurs processus
- o De ne pas avoir à spécifier le nom de la bibliothèque avec `-l` lors de l'édition de liens
- o Aucune des propositions précédentes n'est correcte

> Voir le cours sur la mémoire virtuelle. La création d'espaces mémoire partagés est possible entre processus apparentés avec un `mmap` avec `MAP_SHARED`, ça n'a pas de rapport avec les bibliothèques partagées. Aucun lien non plus entre les échanges de données entre mode noyau et mode utilisateur. La notion de "threads partagés entre plusieurs processus" n'a pas de sens.

11. Le signal SIGSEGV

- X Entraîne par défaut l'arrêt du processus qui le reçoit
- o Termine les processus enfants du processus cible
- o Est généralement envoyé par un enfant à son parent
- o Ne peut pas être bloqué ou ignoré
- o Provoque une erreur de segmentation
- o Aucune des propositions précédentes n'est correcte

> Le signal SIGSEGV est envoyé par le noyau à un processus quand celui-ci produit une erreur de segmentation. Il ne *cause* pas une erreur de segmentation ; ce qui cause l'erreur de segmentation est un problème de gestion mémoire dans votre programme. Au contraire, c'est la *conséquence* d'une telle erreur, et la façon dont elle est gérée par le noyau. Par défaut, il entraîne l'arrêt du processus qui le reçoit : c'est pour cela qu'un programme plante quand il produit une erreur de segmentation.

12. L'appel `execvp` :

- X Ne retourne jamais en cas de succès
- o Ne peut échouer que si le programme demandé n'est pas trouvé
- o Transforme le processus en zombie s'il échoue
- o Entraîne la création d'un nouveau processus
- o Renvoie 0 en cas de succès
- o Aucune des propositions précédentes n'est correcte

L'appel execvp est une primitive de recouvrement : il remplace (recouvre) le programme en cours d'exécution par le processus par un autre chargé depuis le système de fichiers. Il ne retourne donc jamais (et ne renvoie rien) en cas de succès, puisque le code qui a exécuté execvp ne s'exécute plus ; al n'y a plus nulle part où "retourner". Il peut échouer pour différentes raisons : si le programme demandé n'est pas trouvé, mais aussi par exemple s'il n'est pas lisible par l'utilisataire en cours, ou si ce n'est pas un fichier exécutable valide. Il ne crée pas de nouveau processus mais cause une *mutation* du processus en cours. S'il échoue, il renvoie -1 et met un code d'erreur dans errno, comme de nombreuses fonctions de la bibliothèque standard C.

13. Le mot clef `typedef` :

- o Ne peut être utilisé que pour des pointeurs et des structures
- o Est utilisé pour déclarer un pointeur de fonction
- o Ne doit être employé que dans les fichiers `.h`
- o Permet de déclarer une nouvelle structure
- X Permet de donner un nouveau nom à un type de données existant
- o Aucune des propositions précédentes n'est correcte

Le mot clef **typedef** permet de donner un nouveau nom à un type de données existant. Il est souvent utilisé pour des structures ou des pointeurs, mais peut être utilisé pour absolument tous les types de données (voir par exemple le TP3 où il est utilisé pour des types numériques non signés, ui8_t, ui16_t, etc.) **typedef** en soi ne déclare ou ne définit rien du tout, à part un nouveau nom, un alias, pour un type existant. Si vous voulez l'utiliser pour une structure, celle-ci doit être définie par le mot-clef **struct** (soit préalablement, soit dans la même ligne que le **typedef**).

14. Une différence entre un fifo et un tube nommé est...

- X Aucune différence n'existe, c'est la même chose.
- o Un tube nommé n'est utilisable qu'entre processus apparentés, pas un fifo
- o Un fifo peut être fermé et rouvert, pas un tube nommé
- o Seul un fifo et pas un tube nommé peut être crée avec `mkfifo`
- o Un tube nommé est unidirectionnel, un fifo est bidirectionnel
- o Aucune des propositions précédentes n'est correcte

Un fifo et un tube nommé sont deux noms pour la même chose. Ils peuvent être utilisés pour communiquer entre tous types de processus (pas forcément apparentés) qui connaissent et ont accès à son chemin. Ils peuvent être fermés et rouverts plusieurs fois (mais les données dedans ne sont pas conservées entre les ouvertures/fermetures successives). Ils sont crées avec `mkfifo` (la commande shell ou la fonction C). Tout comme les tubes anonymes, ils sont unidirectionnels : al faut une paire de tubes pour avoir une communication bidirectionnelle.

Exercice 2 – Compréhension du cours (2 points)

Note : Je réponds en détail aux questions en corrigeant certaines erreurs que vous avez faites, ce n'était pas nécessaire de dire tout ce que je dis pour avoir les points. Chaque fois, l'essentiel est en gras.

1) Quelle est la différence entre un programme exécutable et un processus ?

Un programme exécutable est un fichier contenant du code exécutable (compréhensible par le processeur), produit typiquement par la compilation et l'édition de lien. Un processus est un programme en cours d'exécution par le système. Il est associé à une entrée dans la table des processus, et à un espace mémoire d'adressage virtuel qui contient le code du programme et son contexte d'exécution (pile, tas, fichiers ouverts, etc.).

2) Deux processus appartenant à un même utilisateur peuvent-ils, dans le cas général, lire l'espace mémoire l'un de l'autre ? Si oui, comment ? Si non, pourquoi ?

Dans le cas général de deux processus quelconques, non, à cause des mécanismes de protection fourni par la mémoire virtuelle : chaque processus travaille dans son espace d'adressage virtuel propre et ne peut pas lire ceux des autres processus ; il ne peut intéragir avec les autres processus et le matériel qu'indirectement, à travers les outils fournis par le noyau (appels système).

Pour des processus apparentés, al est possible d'obtenir un espace mémoire partagé par un mmap anonyme avec MAP_SHARED exécuté dans le parent (ou un ancêtre commun). Mais ça ne concerne pas le cas général de deux processus quelconques, et même entre processus apparentés ce n'est vrai que dans ce cas précis.

C'est exact deux processus apparentés partagent au moins initialement les mêmes cases de la mémoire physique – à cause de la copie en écriture – mais même dans ce cas on ne peut pas vraiment dire qu'ils "lisent l'espace mémoire l'un de l'autre" : ils lisent chacun leur espace mémoire propre, et simplement ces deux espaces mémoires correspondent (pour tout ou partie) aux mêmes cases de la mémoire physique. Dans tous les cas cela ne concerne que des processus apparentés et non le cas général.

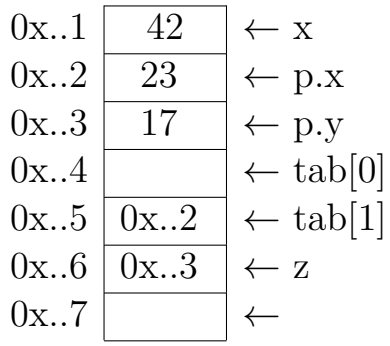
Enfin, deux *threads* au sein du même processus partagent le même espace mémoire (celui du processus en question), mais c'est hors sujet par rapport à la question qui parle de deux processus.

Exercice 3 – Pile

On considère le code suivant, qui s'exécute sur un système imaginaire sur lequel tous les types de données primitifs (int, float, etc.) et les pointeurs ont la même taille et font exactement un emplacement mémoire.

```
1  struct point {
2          int x;
3          int y;
4  };
5
6  int main(int argc, char* argv[]);
7  {
8          int x = 42
9          struct point p;
10         p.x = 23;
11         p.y = x;
12         int* tab[2];
13         tab[1] = &p.x;
14         int *z = tab[1]+1;
15         *z = 17;
16 }
```

Représentez l'état de la pile à la fin de l'exécution de ce `main`, en indiquant le contenu de chaque emplacement mémoire et l'emplacement de chaque variable, en prenant example sur la première ligne (la variable `x` est stockée à l'adresse 0x..1 et a pour valeur 42 à la fin de l'exécution.) :


0x..1
42
← x
0x..2
23
← p.x
0x..3
17
← p.y
0x..4
← tab[0]
0x..5
0x..2
← tab[1]
0x..6
0x..3
← z
0x..7
←

Exercice 4 – Fork

On considère le programme `exo_fork.c` suivant :

```c
#include <unistd.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
        for(int i = 0; i < 3; i++)
        {
                int pid = fork();
                if(pid == 0) // child
                {
                        if(i==1)
                                fork();
                        printf("child i == %d\n", i);
                        break;
                } else { // parent
                        printf("parent i == %d\n", i);
                }
        }
        while(1);
        return 0;
}
```

1) (1 point) Après compilation et édition de liens, on lance le programme dans un terminal. Dessinez le graphe des processus générés par cette exécution. (tel qu'il pourrait par exemple être affiché par les commandes `ps fx` ou `pstree -c`).

On a un parent qui génère trois enfants, et le deuxième de ces enfants génère lui-même un enfant. L'arborescence est donc (je représente aussi le shell, ce n'est pas obligatoire) :

```
bash---exo_fork-+-exo_fork
                +-exo_fork---exo_fork
                +-exo_fork
```

(Le petit-enfant peut apparaître sur la première ou la dernière ligne, c'est assez arbitraire et cela dépend de l'ordre d'exécution des enfants et de l'outil de visualisation des processus que vous utilisez. Tant que vous avez un parent, trois enfants, et un petit-enfant, c'est bon.)

2) (1 point) Donnez un exemple de sortie console possible générée par cette exécution.

Un exemple de sortie. L'ordre des lignes peut être n'importe lequel, sauf que `parent i == 0`, `parent i == 1` et `parent i == 2` doivent être dans cet ordre.

```
parent i == 0
child i == 0
parent i == 1
child i == 1
child i == 1
parent i == 2
child i == 2
```

3) (2 points) On considère maintenant ce programme :

```
1 #include <unistd.h>
2 int main(int argc, char* argv[]) {
3         while(1)
4                 fork();
5 }
```

Que fait ce programme ? Combien de processus ont été crées après 100 itération de la boucle (dans tous les processus existants) ? Pensez-vous que l'exécution de ce programme peut entraîner un effet quelconque, et si oui lequel ?

Le nombre de processus double à chaque itération. Au bout de 100 itération de la boucle dans tous les processus, on a donc 2^{100} processus, c'est à dire de l'ordre de 10^{30} processus. Aucun système existant n'est bien sûr capable de gérer autant de processus (à titre de comparaison, le volume totale de données transféré par Internet dans le monde par an est de l'ordre de "seulement" 10^{22} bits) ; le programme s'arrête donc bien avant 100 itérations.

Ce programme est une *fork bomb* : il va typiquement provoquer un plantage du système en saturant ses ressources. Contrairement à ce qu'on pourait croire, ce n'est généralement pas la mémoire qui va saturer en premier mais la table des processus du noyaux. La mémoire est relativement épargnée par le peu de ressources demandé par les processus crées et le mécanisme de la copie en écriture. La table des processus a elle une taille limitée (qu'on peut connaître avec `sysctl kernel.pid_max`). Des variantes de *fork bombs* saturant la mémoire existent également ; al suffit de faire un `malloc` avant le fork.

C'est possible de se protéger de ce type d'attaque en limitant le nombre de processus crées par utilisataire. Si cela vous intéresse, documentez vous

sur le fichier `/etc/security/limits.conf` et la commande `ulimit`.

Exercice 5 – fork/exec, pipes, fichiers

On s'intéresse dans cette question à un jeu vidéo programmé en C, où al est possible de sauvegarder la partie en cours pour la recharger plus tard. Pour réaliser cela, le jeu est capable de stocker toutes les informations nécessaires pour représenter l'état complet d'une partie dans un une variable de type **struct** game_state, une structure dont le contenu exact n'est pas détaillé ici et n'a pas d'importance pour cet exercice ; on fait juste l'hypothèse qu'elle est assez petite pour être lue en un seul appel à read.

Pour sauvegarder une partie, le jeu va représenter son état dans une variable **struct** game_state, puis écrire cette variable dans un fichier. Pour charger une partie, il va réaliser l'opération inverse : lire une variable **struct** game_state depuis un fichier puis restaurer l'état du jeu à partir des informations.

On s'intéresse dans cet exercice au code de lecture des sauvegardes. Voici le code de la fonction load_game, qui lit la sauvegarde dans le fichier filename et met les données lues dans la variable **struct** game_state pointée par le pointeur gs :

```
1  #include "loadgame.h" // Prototype, and definition of struct game_state
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <errno.h>
5  #include <unistd.h>
6
7  int load_game(const char* filename, struct game_state* gs)
8  /* Read the game state saved in file 'filename' and
9     restore it to the struct game_state 'gs' passed by pointer.
10    gs must point to a valid struct game_state, that will be filled
11    with the loaded data.
12    Returns 0 if success, 1 if error. */
13 {
14         // Open file
15         int fd = open(filename, O_RDONLY);
16         if(fd == -1) { // Error on open
17                 perror("open");
18                 return 1;
19         }
20
21         // Read the game state from the file
22         int n_read = read(fd, gs, sizeof(struct game_state));
23         if(n_read != sizeof(struct game_state))
24         {
25                 printf("Error : could not read a full game state.");
26                 return 1;
27         }
28         close(fd);
```

```
29          return 0; // all good
30  }
```

1) (1 point) Ce programme utilise les appels open, read et close pour ouvrir, lire et fermer le fichier. Un autre choix aurait été possible, lequel ? Donnez l'équivalent de la ligne 15 (celle avec open) avec cette autre possibilité.

On pourrait utiliser les accès bufferisés de la bibliothèque standard C, avec fopen, fread et fclose. L'équivalent de la ligne 15 serait alors :

```
1  FILE* f = fopen(filename, "r");
```

Attention, O_RDONLY, O_WRONLY, O_RDWR ne fonctionnent qu'avec open. Pour fopen, on utilise "r", "w", "r+", etc.

De façon à économiser l'espace disque, on veut maintenant que les fichiers de sauvegarde soient compressées avec l'algorithme gzip (le même que celui utilisé par exemple par les archives `.tar.gz`). Pour lire la sauvegarde compressée, on va créer un autre processus qui exécutera le programme de décompression `gunzip` avec l'option `-c`, qui permet de lire un fichier compressé gzip et d'envoyer les données (décompressées) sur la sortie standard. (`gunzip` se termine avec 0 en cas de succès et 1 sinon (fichier manquant, illisible, pas au format gzip, etc.)) Ce processus communiquera les données au processus parent par un tube.

On va donc avoir la séquence suivante :

— Le processus parent crée un tube, puis `fork`
— L'enfant redirige sa sortie standard vers l'entrée du tube, puis exécute `gunzip` avec execlp ou execvp
— Le parent utilise le même appel read que précédemment, sauf qu'il lit la sortie du tube et non le fichier
— Le parent vérifie que l'enfant s'est bien terminé correctement avant de se terminer lui-même

Voici une version à trous de la fonction load_game_gz pour charger un fichier de sauvegarde compressé :

```c
int load_game_gz(const char* filename, struct game_state* gs)
/* Read the game state saved in gzipped file 'filename' and
   restore it to the struct game_state 'gs' passed by pointer.
   gs must point to a valid struct game_state, that will be filled
   with the loaded data.
   Returns 0 if success, 1 if error. */
{
        // Prepare pipe
        int pipe_fd[2];
        // ===== EMPLACEMENT 1 QUESTION 3 =====

        // Fork
        int pid = fork();
        if(pid == 0) { // child
                // Manage pipe in child
                // ===== EMPLACEMENT 2 QUESTION 3 =====
                // ===== EMPLACEMENT 3 QUESTION 3 =====
                // Execute gunzip
                // == REPONSE QUESTION 2 ==
                // If we are here, exec failed
                perror("Exec failed");
                return 1; // Have the child terminate in error
        }
        /* We are in parent here, the child ended either
           with the end of gunzip or with return 1 if
           gunzip failed */

        // Manage pipe in parent
        // ===== EMPLACEMENT 4 QUESTION 3 =====
        // Read the game state from the pipe
        // ===== EMPLACEMENT 5 QUESTION 3 =====
        if(n != sizeof(struct game_state))
        {
                printf("Error : could not read a full game state.");
                return 1;
        }

        int child_status;
        wait(&child_status); // Wait for end of child

        if(child_status != 0) {
                puts("Could not load game.\n");
                return 1;
        } else { // All good
                puts("Game loaded.\n");
                return 0;
        }
}
```

2) (1 point) Donnez le code permettant au processus enfant d'exécuter le programme `gunzip` suivi de l'option `-c` et du nom du fichier stocké dans filename, à la ligne 19. Vous utiliserez execlp ou execvp au choix. (La réponse peut prendre une seule ligne ou quelques lignes.)

Avec execlp :

```
1  execlp("gunzip", "gunzip", "-c", filename, NULL);
```

Avec execvp :

```
1  char* argv_gunzip[4];
2  argv_gunzip[0] = "gunzip";
3  argv_gunzip[1] = "-c";
4  argv_gunzip[2] = filename;
5  argv_gunzip[3] = NULL;
6
7  // Ou autre possibilite en une ligne :
8  // char* argv_gunzip[] = {"gunzip", "-c", filename, NULL};
9
10 execvp("gunzip", argv_gunzip);
```

Attention à ne pas oublier le premier argument (`"gunzip"`) et le NULL à la fin !

3) (2.5 points) Utilisez les propositions suivantes pour compléter les emplacements indiqués aux lignes 10, 16, 17, 29 et 31 de façon à rendre le programme fonctionnel. Chaque emplacement correspond à **une seule** proposition ; il y a plus de cinq propositions, donc certaines sont des fausses pistes. Pour répondre, écrivez le numéro de ligne ou d'emplacement correspondant à côté de chaque proposition que vous utilisez, n'écrivez rien ou "aucun" pour celles que vous n'utilisez pas.

- pipe(pipe_fd); : **emplacement 1 (ligne 10)**
- close(pipe_fd[1]); : **emplacement 4 (ligne 29)**
- dup2(pipe_fd[0], STDIN_FILENO); : non utilisé
- **int** n = read(pipe_fd[0], gs, **sizeof**(**struct** game_state)); : **emplacement 5 (ligne 31)**
- **int** n = write(pipe_fd[1], gs, **sizeof**(**struct** game_state)); : non utilisé
- dup2(pipe_fd[1], STDOUT_FILENO); : **emplacement 3 (ligne 17)** ou éventuellement 2 (ligne 16)
- pipe_fd = pipe(); : non utilisé
- close(pipe_fd[0]); : **emplacement 2 (ligne 16)** ou éventuellement 3 (ligne 17)

4) (bonus, jusqu'à 2 points – à faire surtout si vous avez fini le reste, c'est un peu subtil) On remarque un comportement problématique dans le cas où le fichier est un fichier gzip

valide, mais de grande taille (beaucoup plus grande que **sizeof**(**struct** game_state). Dans ce cas, l'appel à la fonction reste bloqué sans jamais se terminer, et une inspection de l'arbre des processus montre que l'enfant comme le parent sont dans un état bloqué (en attente d'un évènement). Quel est l'origine de ce problème ? Comment le corriger ? (Plusieurs réponses plus ou moins simples et élégantes sont possibles. Ce n'est pas nécessaire d'écrire le code, juste d'expliquer comment on va procéder.)

Le parent ne va lire le tube qu'une fois, pour **sizeof**(**struct** game_state)), mais `gunzip` lui va écrire dans le tube tant qu'il a des données décompressées. Si le fichier est très grand, il va donc remplir le tube, et l'écriture dans le tube va devenir bloquante, ce qui va empêcher l'enfant exécutant `gunzip` de se terminer. Le parent, lui, attend avec wait que l'enfant se termine après avoir lu le tube. On a donc une situation d'interblocage : l'enfant attend que le parent le débloque en lisant le tube, le parent attend que l'enfant le débloque en se terminant.

Plusieurs solutions sont possibles pour résoudre ce problème, les plus naturelles sont :

- Fermer le tube en lecture dans le parent après le read, avant d'attendre l'enfant. Si l'enfant continue d'écrire dedans, il recevra immédiatement le SIGPIPE et se terminera ;
- Tuer l'enfant avec kill après le read. L'effet est sensiblement le même que la solution précédente. Si l'enfant est déjà mort, le kill n'aura pas d'effet.
- Rendre la lecture plus propre, avec une boucle sur read pour lire les données jusqu'à ce que le tube soit vide.
- Utiliser alarm pour mettre en place un *timeout* et faire se terminer l'enfant s'il est bloqué. Ici ce n'est pas une solution très propre par rapport aux autres, mais de façon générale c'est une bonne solution quand on exécute un programme qu'on ne connait pas ou contrôle pas et qu'on ne veut pas être bloqué s'il fonctionne de façon incorrecte.