BRONNERT
Edouard

POLYTECH LYON
3A INFO

TP5 – Mémoire partagée

Exercice 1

1. Explication du code shmid.c :

shmid : identifiant de du segment de mémoire partagée (=**SH**ared **M**emory) crée par *shmget()* ; valeur renvoyée par *shmget()*.

*shmget(*key_t clé, size_t size, int shmflg*)* : Alloue un segment de mémoire partagée associé à la valeur clé. Ce segment est de taille size est créé si clé vaut IPC_PRIVATE (c'est le cas ici). Il est aussi créé s'il n'existe aucun segment de mémoire associé à clé, et que shmflg vaut IPC_CREAT. *shmget()* renvoie shmid, ou -1 s'il échoue.
IPC_EXCL est utilisé avec IPC_CREAT (IPC_CREAT | IPC_EXCL). Ce dernier attribut garantit l'échec s'il y a déjà un segment de mémoire partagée pour clé.
0700 correspond aux permissions d'accès.

*shmat(*int shmid, const void *shmaddr, int shmflg*)* : *shmat()* effectue une opération sur la mémoire partagée : elle attache le segment de mémoire partagée shmid au segment de données du processus appelant. shmaddr est l'adresse d'attachement, si elle vaut :

- NULL (c'est le cas ici) : une adresse libre lui sera allouée pour attacher le segment.
- Non NULL + shmflg="SHM_RND" : l'attachement à lieu à l'adresse shmaddr, arrondie au multiple inférieur de SHMLBA. Sinon shmaddr doit être alignée sur une frontière de page.

shmat() renvoie donc l'adresse d'attachement du segment de mémoire partagée s'il réussit, sinon il renvoie (void *) -1.

*shmctl(*int shmid, int cmd, struct shmid_ds *buf*)* : effectue l'opération de contrôle cmd sur le segment de mémoire partagée identifiée par shmid. buf est un pointeur sur une structure **shmid_ds** définie dans <sys/shm.h> :

```
struct shmid_ds {
    struct ipc_perm  shm_perm;    /* operation permissions */
    size_t         shm_segsz;   /* size of segment in bytes */
    pid_t          shm_lpid;    /* pid of last shm op */
    pid_t          shm_cpid;    /* pid of creator */
    short          shm_nattch;  /* # of current attaches */
    time_t          shm_atime;   /* last shmat() time*/
    time_t          shm_dtime;   /* last shmdt() time */
    time_t          shm_ctime;   /* last change by shmctl() */
    void           *shm_internal; /* sysv stupidity */
};
```

cmd peut prendre de nombreuses valeurs (IPC_STAT, IPC_SET, IPC_RMID ...), nous ne les détaillerons pas toutes. Ici, cmd=IPC_RMID.
IPC_RMID : marque le segment pour le détruire. Retire le segment de mémoire partagée identifié par shmid et détruit les données associées. Il ne sera détruit qu'après le dernier

détachement (quand *shm_nattch* de la structure *shmid_ds* associée vaudra 0). L'appelant doit être le créateur du segment, le propriétaire ou être prévilégié.

rand() : génère un entier pseudo-aléatoirement, compris dans l'intervalle [0, RAND_MAX].

Pour résumer, que fait le code ?

On crée un segment de mémoire partagée d'une taille de 100*sizeof(int) (=400) octets. Avec *shmat()*, on attache le segment de mémoire partagée au segment de données du processus appelant. On enregistre cette adresse d'attachement dans buffer_ptr.
Avec *fork()*, on crée ensuite un nouveau processus.

Dans le processus père :
On génère un nombre pseudo-aléatoire indice, que l'on insère dans buffer_ptr[0]. On entre la valeur 42 dans buffer_ptr[indice].
Le processus attend que le fils se termine.
On détruit le segment de mémoire partagée avec *shmctl(shmid, IPC_RMID, NULL).*

Dans le processus fils :
On affiche le contenu de buffer_ptr[buffer_ptr[0]], soit buffer_ptr[indice]=42.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/wait.h>

#define N 100

int main (int argn, char * argv[]) {
    pid_t pid;
    /* Création d'un segment de mémoire partagée */
    int shmid = shmget(IPC_PRIVATE, N*sizeof(int), IPC_CREAT|0700);
    int* buffer_ptr = shmat(shmid, NULL, 0);
    pid=fork();

    if(shmid<0){
        printf("Erreur\n");
        exit(-1);
    }


    if(pid > 0) {
        /* Processus parent */
        int indice = (rand() + 1) % (N / sizeof(int));
        buffer_ptr[0] = indice;
        buffer_ptr[indice] = 42;

        /* J'attend que mon enfant se termine */
        wait(NULL);
        shmctl(shmid, IPC_RMID, NULL);
    } else {
        /* Processus enfant */
        printf("buffer_ptr = %d\n", buffer_ptr[buffer_ptr[0]]);
    }
    return EXIT_SUCCESS;
}
```

Si on l'exécute :

```
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % gcc ./shmid.c -o shmid
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ %
```

On peut l'exécuter des milliers de fois, on trouvera toujours la valeur 42 :

```
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ %
```

En fait, *rand( )* génère un entier pseudo-aléatoirement, comme dit plus haut. rand() va prendre la première (puis la 2ème pour le 2ème *rand()*, etc.) valeur d'une séquence de nombre prédéfinis. Si on ré-exécute la fonction, on retombera sur les mêmes valeurs. Il va donc falloir tirer une "position de départ" aléatoire dans cette séquence de valeurs. Pour cela on peut utiliser time.h :

```
srand(time(NULL)) ;
nbAleatoire = rand( ) ;
```

On doit donc paramétrer srand( ) avec une "graine" (seed), que l'on fera varier grace à time.h.
time( ) retourne le nombre de secondes depuis le 1er janvier 1970.

Le problème c'est que la probabilité de tirer un nombre dans la séquence de valeurs n'est pas uniforme, mais cela est négligeable pour des séquences de valeurs petites, ce qui est notre cas.

On teste avec srand(time(NULL)); :

```
buffer_ptr[77] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr[84] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr[91] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr[98] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr[98] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./shmid
buffer_ptr[5] = 42
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ %
```

Nos indices sont différents à chaque fois, ça fonctionne avec *time( )* !

Comment fonctionne cette ligne ? :

```
int indice = 1 + (rand() % N-2);
```

Le nombre aléatoire tiré sera dans l'intervalle [1 ; N-1]. En fait, lorsque l'on fait rand()%maxIntervalle, le nombre aléatoire tiré sera dans l'intervalle [0 ; maxIntervalle]. En ajoutant 1 on évite d'écrire sur la case "0" en évitant d'avoir un *overflow* (débordement).

On est dans une condition de « race condition » (plusieurs processus peuvent accéder à la mémoire partagée). Pour résoudre le problème, on va devoir faire de « l'attente active » :

```
if(pid > 0) {
    /* Processus parent */
    int indice = 1 + (rand() % N-2);

    //int indice = (rand()+1)%(N / sizeof(int));
    buffer_ptr[0] = indice;
    buffer_ptr[indice] = 42;
    buffer_ptr[N-1]=1;

    /* J'attend que mon enfant se termine */
    wait(NULL);
    shmctl(shmid, IPC_RMID, NULL);
} else {
    while(buffer_ptr[N-1]!=1){
        printf("Attente...[le processus parent n'a pas fini d'écrire]\n");
        sleep(1);
    }
    /* Processus enfant */
    printf("buffer_ptr[%d] = %d\n", buffer_ptr[0],buffer_ptr[buffer_ptr[0]]);
}
return EXIT_SUCCESS;
```

On peut aussi passer par un sémaphore (long et compliqué), mais il ne faut pas passer par un *sleep* car ça ne garantit rien !

Explications tirées du manuel Linux et de :
https://openclassrooms.com/fr/courses/1389636-a-la-decouverte-de-laleatoire-et-des-probabilites/1389903-laleatoire-en-c-et-c-se-servir-de-rand

2. On modifie le code précédent afin que le père stock des valeurs aléatoires dans le tableau, et que le fils les trie. Le père affichera ensuite les valeurs.
   Pour cela, on va coder des attentes actives dans chacun des processus. La première case du tableau (*buffer_ptr[0]*) sera réservée pour la synchronisation des processus.

   On initialise donc *buffer_ptr[0]* à 0. On fait un *fork( )*.

   Dans le père :
   On génère des valeurs aléatoires avec *rand( )* que l'on met dans le tableau (*ici, je génère des valeurs entre 0 et 1000)* ; on fait bien attention de commencer à la case 1 pour ne pas écrire sur notre case 0 réservée à la synchronisation des processus.
   Une fois qu'on a fini d'écrire, on met *buffer_ptr[0]* à 1 : c'est au tour du fils.

   Dans le fils :
   La première boucle while est pour l'attente active : le fils attend que la valeur de *buffer_ptr[0]* valle 1 pour commencer le tri.
   Le fils va maintenant trier le tableau. Pour cela j'ai choisi un tri à bulle, qui est simple à implémenté (vu le nombre de valeurs, nous n'avons pas besoin d'un tri plus optimisé) :

```
/* Tri à bulles */

for(int i=N; i>0; i--){
    for(int j=1; j<i; j++){
        if(buffer_ptr[j+1]<buffer_ptr[j]){
            int tampon=buffer_ptr[j+1];
            buffer_ptr[j+1]=buffer_ptr[j];
            buffer_ptr[j]=tampon;
        }
    }
}
```

J'ai aussi fait un tri à bulles optimisé pour le plaisir, les deux fonctionnent très bien :

```c
/* Tri à bulles optimisé */

int tabTrie = 0;

for(int i=N; i>0; i--){
    tabTrie=1;
    for(int j=1; j<i; j++){
        if(buffer_ptr[j+1]<buffer_ptr[j]){
            int tampon=buffer_ptr[j+1];
            buffer_ptr[j+1]=buffer_ptr[j];
            buffer_ptr[j]=tampon;
            tabTrie=0;
        }
    }
    if(tabTrie){
        break;
    }
}
```

L'implémentation de ce tri plus optimisé m'a permis de découvrir que *bool* n'existait pas en C. Cela peut paraître trivial, mais je n'ai jamais fait de C (seulement un peu de C++ en PeiP). Pour déclarer un booléen il faut donc déclarer un entier donc la valeur 0 correspond à false, et toute autre valeur correspond à true.

Après avoir fini de trier, le fils met *buffer_ptr[0]* à 2 : c'est au tour du père !

Dans le père :

On fait aussi une attente active avec un while, il attend que *buffer_ptr[0]* valle 2. On peut alors afficher le tableau avec une simple boucle for. J'ai décidé d'afficher le tableau qu'à partir de *buffer_ptr[1]* car la première case est réservée à la synchronisation et n'a donc aucun intérêt (elle contient la valeur 2 à la fin du programme).

A l'affichage on obtient bien le tableau trié avec des valeurs générées aléatoirement :

```
Last login: Tue May 26 12:15:56 on ttys000
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % gcc ./tri.c -o tri
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ % ./tri
buffer_ptr[1]= 0
buffer_ptr[2]= 14
buffer_ptr[3]= 22
buffer_ptr[4]= 35
buffer_ptr[5]= 39
buffer_ptr[6]= 47
buffer_ptr[7]= 56
buffer_ptr[8]= 80
buffer_ptr[9]= 86
buffer_ptr[10]= 96
buffer_ptr[11]= 135
buffer_ptr[12]= 155
buffer_ptr[13]= 178
buffer_ptr[14]= 183
buffer_ptr[15]= 188
buffer_ptr[16]= 196
buffer_ptr[17]= 199
buffer_ptr[18]= 285
buffer_ptr[19]= 287
buffer_ptr[20]= 287
buffer_ptr[21]= 295
buffer_ptr[22]= 307

buffer_ptr[90]= 883
buffer_ptr[91]= 886
buffer_ptr[92]= 896
buffer_ptr[93]= 917
buffer_ptr[94]= 927
buffer_ptr[95]= 944
buffer_ptr[96]= 946
buffer_ptr[97]= 957
buffer_ptr[98]= 966
buffer_ptr[99]= 979
buffer_ptr[100]= 999
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ %
```

Le code :

tri.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/wait.h>
#include <time.h>

#define N 100

int main (int argn, char * argv[]) {
    pid_t pid;
    /* Création d'un segment de mémoire partagée */
    int shmid = shmget(IPC_PRIVATE, N*sizeof(int), IPC_CREAT|0700);
    int* buffer_ptr = shmat(shmid, NULL, 0);

    if(shmid<0){
        printf("Erreur\n");
        exit(-1);
    }

    buffer_ptr[0]=0;

    pid=fork();

    /* ------------------------------------------------------------------------------------------------
    srand(time(NULL));


    /* buffer_ptr[0 va me servir de canal de communication entre le processus parent et le processus enfant */
    /* Au début, buffer_ptr[0] aura pour valeur 0.
    Une fois que le processus parent aura terminé d'écrire les valeurs aléatoires dans le tableau,
    il écrira 1 dans la case buffer_ptr[0] */
    /* Quand le processus enfant aura fini d'écrire des valeurs aléatoires dans le tableau,
    il écrira 2 dans la case buffer_ptr[0] */



    if(pid > 0) {
        /* Processus parent */

        /* Tableau de taille N, de 0 à N-1, écriture des données aléatoires */

        for(int i=1; i<N; i++){
            buffer_ptr[i] = rand() % 1000;
        }

        buffer_ptr[0]=1; // On a fini d'écrire, on peut rentrer 1


        /* J'attend que mon enfant écrive 2 dans la case 0 de buffer_ptr */
        while(buffer_ptr[0]!=2){
            printf("[parent] Attente du tri...\n");
            printf("%d\n", buffer_ptr[0]);
            sleep(1);
        }

        /* On affiche à partir de 1, car la case 0 est réservée à la synchronisation */
        for(int i=1; i<=N; i++){
            printf("buffer_ptr[%d]= %d\n",i,buffer_ptr[i]);
        }

        wait(NULL);
        shmctl(shmid, IPC_RMID, NULL);

    } else {
        /* Processus enfant */
        /* J'attends que mon parent écrive 1 dans la case 0 de buffer_ptr,
        i.e. qu'il ait fini d'écrire des données aléatoires */

        while(buffer_ptr[0]!=1){
            printf("[enfant] Attente de remplissage...\n");
            printf("%d\n", buffer_ptr[0]);
            sleep(1);
        }

        /* Tri à bulles */

        for(int i=N; i>0; i--){
            for(int j=1; j<i; j++){
                if(buffer_ptr[j+1]<buffer_ptr[j]){
                    int tampon=buffer_ptr[j+1];
                    buffer_ptr[j+1]=buffer_ptr[j];
                    buffer_ptr[j]=tampon;
                }
            }
        }


        buffer_ptr[0]=2; // le fils a fini de trier

        //printf("buffer_ptr[%d] = %d\n", buffer_ptr[0],buffer_ptr[buffer_ptr[0]]);
    }
    return EXIT_SUCCESS;
}
```

Exercice 2

On va créer un morpion avec un plateau de jeu placé dans un segment de mémoire partagée.

Question 1 : L'arbitre

On va, dans le processus père, créer un segment de mémoire partagée :

```
printf("%s\n","Les joueurs joueront avec 'X' et 'O'");
printf("%s\n","_____________________________________");

/* Initialisation de la grille */
/* On va utiliser un tableau à 13 cases (de 0 à 12) */
/* Case 9 : case jouée précédemment */
/* Case 10 : case de synchronisation (attente active) */
/* Case 11 : quel joueur a joué */
/* Case 12 : fin du jeu */
int shmid = shmget(IPC_PRIVATE, 13*sizeof(int), IPC_CREAT|0700);
int *grille = shmat(shmid, NULL, 0);
```

On crée un tableau de 13 cases :

- Les 9 premières cases sont dédiées au plateau de jeu.
- Dans la case 9, le processus en cours inscrira le numéro de la case sur laquelle il a posé un pion.
- La case 10 est réservée à la synchronisation (attente active) entre processus père et processus fils. Elle peut prendre 3 valeurs :
    - 1 => c'est au joueur1 (fils 1) de jouer
    - 2 => c'est au joueur2 (fils 2) de jouer
    - 3 => c'est au tour de l'arbitre (père)
- La case 11 sert à distinguer les processus fils. On met dedans le numéro du fils qui vient de jouer, pour savoir à qui le père donnera le relai.
- La case 12 sert pour annoncer la fin du jeu. Elle prend les valeurs suivantes :
    - 0 => La partie est toujours en cours
    - 1 => Le joueur 1 a gagné
    - 2 => Le joueur 2 a gagné
    - 3 => Ex-aequo, la grille est remplie

L'affichage de la grille se fait avec la fonction *afficheGrille( )* appelé dans chacun des joueurs. Elle affichera le caractère ' ' si la case contient 0, 'X' si elle contient 1, et 'O' si elle contient 2.

```c
void afficheGrille(int *grille){
    int k = 0;
    for(int i=0; i<3 ; i++){
        printf("%c",'|');
        for(int j=0; j<3 ; j++){
            //printf("%d", grille[i+j+k]);
            if(grille[i+j+k]==1) printf("%c",'X');
            else if(grille[i+j+k]==2) printf("%c",'O');
            else printf("%c",' ');
            printf("%c",'|');
        }
        printf("\n");
        k+=2;
    }
}
```

On initialise la grille :

```c
/* On initie la grille */
for(int i=0; i<10; i++){
    grille[i]=0;
}
grille[9] = -1; // dernier coup joué, personne n'a encore joué
grille[10] = 0; // synchronisation des processus
/* Synchronisation des processus : */
// grille[10] = 1 => fils 1
// grille[10] = 2 => fils 2
// grille[10] = 3 => père
grille[11] = 0; // contient le numéro du joueur qui a joué
grille[12] = 0; //Jeu toujours actif quand elle vaut 0
```

Le plateau de jeu est initialisé à 0. En fait, le plateau de jeu prendre 3 valeurs :

- 0 : La case est libre
- 1 : La case contient le pion du joueur 1
- 2 : La case contient le pion du joueur 2

On met dans la case 9 la valeur -1 : aucun joueur n'a encore joué.
La case 12 prend la valeur 0 : le jeu est actif (on aurait pu tout mettre dans la boucle for, mais je voulais l'expliciter).

On crée les joueurs :

```c
// création des joueurs :
int joueur1 = 0;
int joueur2 = 0;
char pion1 = 1;
char pion2 = 2;
```

joueur1 et *joueur2* peuvent prendre 4 valeurs :

- 1 => Joueur Débutant
- 2 => Joueur Malin
- 3 => Joueur Tricheur
- 4 => Joueur Polytech

Avant de créer les processus fils et de lancer la partie, on va encore initialiser les cases 10 et 11 de la grille, qui correspondent à la synchronisation :

```
grille[10] = 3; //Le père initialise la partie
grille[11] = 2; // Le joueur 1 commencera
int tour=0; // Affiche le numéro de tour.
```

Le père lance le jeu, le joueur 1 commence (on met donc la case 11 à 2, comme si le joueur 2 avait joué auparavant).

On *fork( )* dans une boucle for, sans oublier de mettre un break pour que le processus fils sorte de la boucle (et ainsi éviter la création d'un 3ème fils). On crée aussi une variable *numFils* pour distinguer les processus fils.

```
/* On crée les processus fils */
int numFils = 0;
for(numFils=1; numFils<=2; numFils++){
    pid = fork();
    if(pid==0) break;
}
```

Dans le père, tant que le jeu est actif (grille[12]=0) :

- On fait une attente active (première boucle while).
- On appelle la fonction *verifGagnant( )* qui vérifie si un joueur a gagné, ou si la grille est remplie, et qui modifie la case 12 de la grille.
- On vérifie la valeur de la case 12 de la grille (switch), si elle vaut 0, on ne fait rien, sinon on affiche le résultat.
- On passe le relai au fils. Pour cela on vérifie quel fils a joué précédemment (*grille[11]*), et on attribue la bonne valeur dans *grille[10]*.
- En dehors de la boucle while, on fait un *wait(NULL)*.

Attention à mettre le while(grille[12]=0) dans le processus père, et non pas mettre les processus père et fils dans un while( ), sinon on pourra pas gérer l'attente du père (wait(NULL)). J'ai passé 3 jours à essayer de comprendre mon erreur*  *

```c
if(pid>0){
    while(grille[12]==0){
        /* Processus père */
        /* On fait une attente active */
        while(grille[10]!=3){
            //sleep(1);
        }

        verifGagnant(grille);
        tour++;

        if(grille[12]==0){
        printf("\n");
        printf("•\n");
        printf("%s\n","_____________________________________");

        printf("                  COUP  %d\n",tour);
        printf("%s\n","_____________________________________");
        }


        switch(grille[12]){
            case 0 :
                break;
            case 1 :
                printf("Le joueur 1 à gagné ! Félicitations ! X\n");
                break;
            case 2 :
                printf("Le joueur 2 a gagné ! Bien joué ! O\n");
                break;
            default :
                printf("Ex-aequo, on veut une revanche ! XO\n");
                break;
        }

        /* C'est au tour du fils */
        if(grille[11]==2){
            grille[10]=1;
        }
        else{
            grille[10]=2;
        }


    }
    wait(NULL);
    shmctl(shmid, IPC_RMID, NULL);
}
```

J'ai décidé de créer une fonction *verifGagnant( )* pour plus de lisibilité (le *main( )* étant déjà très [trop] chargé).
Cette fonction va tester toutes les lignes/colonnes/diagonales et regarder si les cases contiennent le même pion, et de quel joueur il s'agit. La case 13 de la mémoire partagée sera alors modifiée en conséquence :

```
void verifGagnant(int *grille){

    /* grille[12] peut prendre plusieurs valeurs :
    • 0 : la grille n'est pas encore remplie, le jeu continue
    • 1 : le joueur 1 a gagné (rempli 3 cases consécutives)
    • 2 : le joueur 2 a gagné (rempli 3 cases consécutives)
    • 3 : la grille est remplie mais personne n'a gagné
    */

    /* Vérification première diagonale */
    if(grille[0]==grille[4] && grille[4]==grille[8] && grille[0]!=0){
        if(grille[0]==1){
            grille[12]=1;
        }
        else{
            grille[12]=2;
        }

    }

    /* Vérification geuxième diagonale */
    if(grille[2]==grille[4] && grille[4]==grille[6] && grille[2]!=0){
        if(grille[2]==1){
            grille[12]=1;
        }
        else{
            grille[12]=2;
        }
    }
```

On vérifie aussi si la grille est remplie (une case vide contient la valeur 0) :

```
/* Vérifie si la grille est remplie */

int compteur=0;
for(int i=0; i<9; i++)
{
    if(grille[i]!=0){
        compteur+=1;
    }
}

if(compteur==9 && grille[12] == 0){
    grille[12] = 3;
}
```

Dans les fils :

On englobe les fils dans un while(*grille[12]==0*) : tant que la partie n'est pas finie, on exécute les fils.
On fait une attente active : c'est là que *numFils* entre en jeu, le fils correspondant s'exécutera, l'autre attendra.
Si la partie est finie (*grille[12] !=0*), on fait un exit(0) : on stop ainsi les processus fils (et on évite les zombies).
Selon le numéro entrée au clavier au début, les différents joueurs sont appelés (switch).

```
499         else{
500             while(grille[12]==0){
501                 /* Processus fils */
502                 /* Attente active */
503                 while(grille[10]!=numFils){
504                     //sleep(1);
505                 }
506
507                 //verifGagnant(grille);
508                 if(grille[12]!=0){
509                     exit(0);
510                 }
511
512                 /* JOUEUR 1 */
513                 if(numFils==1){
514                     switch(joueur1){
515                         case 1 :
516                             joueurDebutant(grille,pion1);
517                             break;
518                         case 2 :
519                             joueurMalin(grille,pion1);
520                             break;
521                         case 3 :
522                             joueurTricheur(grille,pion1);
523                             break;
524                         case 4 :
525                             joueurPolytech(grille,pion1);
526                             break;
527                         default :
528                             printf("ERROR");
529                             break;
530                     }
531                 }
```

```
532
533             if(numFils==2){
534                 /* JOUEUR 2 */
535                 switch(joueur2){
536                     case 1 :
537                         joueurDebutant(grille,pion2);
538                         break;
539                     case 2 :
540                         joueurMalin(grille,pion2);
541                         break;
542                     case 3 :
543                         joueurTricheur(grille,pion2);
544                         break;
545                     case 4 :
546                         joueurPolytech(grille,pion2);
547                         break;
548                     default :
549                         printf("ERROR");
550                         break;
551                 }
552             }
553
554
555             grille[11]=numFils;
556             grille[10]=3;
557
558
559         }
560     }
```

Quand le fils a terminé son tour, il entre son numéro (*numFils*) dans la case 11 de la grille, et il inscrit 3 dans la case 10 : c'est au tour du père.

Question 2 : Le débutant

Le joueur débutant va simplement tirer une valeur au hasard entre 0 et 8. *joueurDebutant( )* prend en paramètre la grille et le pion du joueur.

```
void joueurDebutant(int *grille, int pion){

    printf("%s\n","_____________________________________");
    printf("            Joueur DEBUTANT\n");
    printf("%s\n","_____________________________________");

    // Génère un nombre aléatoire entre 0 et 8
    int nb_aleatoire = 0 ;

    do
    {
        nb_aleatoire = rand()%9;
    }while(grille[nb_aleatoire] != 0);

    grille[nb_aleatoire]=pion;
    grille[9]=nb_aleatoire;

    sleep(1);
    afficheGrille(grille);


}
```

Si la case correspondant à cette valeur est déjà occupée (elle vaut 1 ou 2), alors on relance un nombre aléatoire. Sinon, on pose le pion.
On enregistre ce nombre aléatoire dans la case 9 de la grille : ce sera utile pour le joueur Malin.
On affiche enfin la grille.

Question 3 : Le malin

On crée une variable *jeuPrecedent* qui prendra la valeur de la case 9. On crée un tableau des possibilités de placement du pion. On créé aussi une variable *intervalle* (elle servira pour tirer une case aléatoirement).
Selon la valeur de *jeuPrecedent*, on aura plusieurs possibilités de placement (les cases adjacentes à *jeuPrecedent*). On les écrit toutes :

```
void joueurMalin(int *grille, int pion){

    printf("%s\n","_____________________________________");
    printf("            Joueur MALIN\n");
    printf("%s\n","_____________________________________");

    int jeuPrecedent = grille[9];
    int possibilite[9]={-1,-1,-1,-1,-1,-1,-1,-1,-1};
    int nb_aleatoire = 0;
    int intervalle = 0;
    int caseChoisie = 0;


    do{
        switch(jeuPrecedent){
            case 0 :
                possibilite[0]=1;
                possibilite[1]=3;
                possibilite[2]=4;
                intervalle=3;
                break;

            case 1 :
                possibilite[0]=0;
```

Il reste à traiter le cas où toutes les cases de possibilité sont occupées. Dans ce cas, on va donner à possibilité toutes les valeurs de la grille :

```
        default:
            possibilite[0]=0;
            possibilite[1]=1;
            possibilite[2]=2;
            possibilite[3]=3;
            possibilite[4]=4;
            possibilite[5]=5;
            possibilite[6]=6;
            possibilite[7]=7;
            possibilite[8]=8;
            intervalle = 9;
            break;
    }

    /* Si toutes les cases "possibilite" sont prises, alors possibilite = "grille" */
    int compteur=0;
    for(int i=0; i<intervalle; i++){
        if(grille[possibilite[i]]!=0){
            compteur++;
        }
    }

    if(compteur==intervalle){
            possibilite[0]=0;
            possibilite[1]=1;
            possibilite[2]=2;
            possibilite[3]=3;
            possibilite[4]=4;
            possibilite[5]=5;
            possibilite[6]=6;
            possibilite[7]=7;
            possibilite[8]=8;

            intervalle = 9;
    }
```

On créé un compteur qu'on incrémente si la case est occupée. Si le compteur est égal à l'intervalle (taille de *possibilite*), alors il n'y a plus de places disponibles. *possibilite* prend alors toutes les valeurs de la grille.

On tire alors au hasard une valeur dans *possibilite* :

```
        nb_aleatoire=rand()%intervalle;
        //printf("Nombre aléatoire = %d \n",nb_aleatoire);
        //printf("Possibilité étudiée = %d \n",possibilite[nb_aleatoire]);
        caseChoisie = possibilite[nb_aleatoire];
    }while(grille[caseChoisie] != 0);
```

Toutes les opérations précédentes sont effectuées dans un do while pour s'assurer de ne pas placer son pion sur un autre pion.

Malin place alors son pion sur la case choisie, et inscrit la valeur de cette case dans *grille[9]* :

```
grille[caseChoisie]=pion;
grille[9]=caseChoisie;

sleep(1);
afficheGrille(grille);
```

On finit en affichant la grille.

Question 4 : Le tricheur

Le tricheur est implémenté de la même façon que le débutant :

```c
void joueurTricheur(int *grille, int pion){

    printf("%s\n","_____________________________________");
    printf("              Joueur TRICHEUR\n");
    printf("%s\n","_____________________________________");

    // Génère un nombre aléatoire entre 0 et 8
    int nb_aleatoire = 0 ;
    do{
        nb_aleatoire = rand()%100; // On prend un intervalle de 100 pour pas avoir les mêmes positions que Débutant
        nb_aleatoire %=9;
    }while(grille[nb_aleatoire]==pion); //On veut tricher, on évite de rejouer sur sa propre case.


    grille[nb_aleatoire]=pion;
    grille[9]=nb_aleatoire;

    sleep(1);
    afficheGrille(grille);

}
```

J'ai décidé de faire en sorte qu'il ne puisse pas rejouer sur ses propres pions (do while) car s'il triche, c'est dans le but de gagner.
Aussi, je tire un nombre au hasard dans un intervalle de 100 pour qu'il n'ait pas toujours les mêmes valeurs, et qu'il ait une action différente du joueur débutant.

Question 4 : Game Over

On créé des entiers *gainJ1*, *gainJ2* et *gainexAeq*.
On modifie un peu le code : on supprime tous les affichages, et on englobe les processus père et fils (après avoir fork()) d'une boucle for, et on supprime les *sleep(1)*. On va lancer 100 parties.
Au début de chaque tour de boucle, on réinitialise les cases du tableau :

```
for(int k=0; k<100; k++){


    /* On initie la grille */
    for(int i=0; i<10; i++){
        grille[i]=0;
    }
    grille[9] = -1; // dernier coup joué, personne n'a encore joué
    grille[10] = 0; // synchronisation des processus
    /* Synchronisation des processus : */
    // grille[10] = 1 => fils 1
    // grille[10] = 2 => fils 2
    // grille[10] = 3 => père
    grille[11] = 0; // contient le numéro du joueur qui a joué
    grille[12] = 0; //Jeu toujours actif quand elle vaut 0

    grille[10] = 3; //Le père initialise la partie
    grille[11] = 2; // Le joueur 1 commencera
```

A chaque tour de boucle, on incrémente les différentes variables créées dans le switch processus père. A la fin, on les divise par 1000 pour obtenir une probabilité.

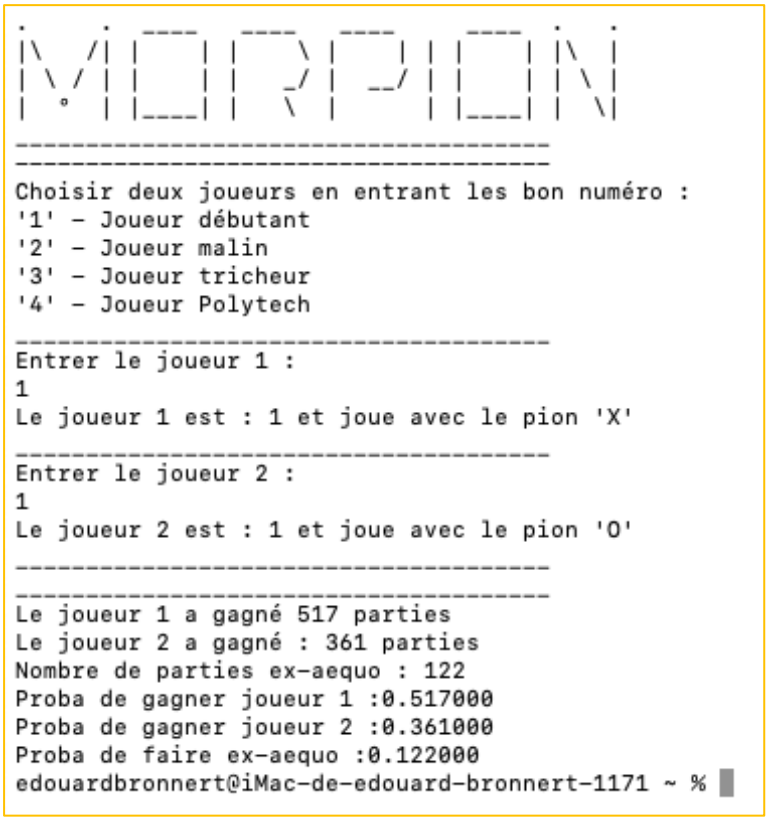
```
switch(grille[12]){
    case 0 :
        break;
    case 1 :
        gainJ1++;
        break;
    case 2 :
        gainJ2++;
        break;
    default :
        gainexAeq++;
        break;
}
```

```
printf("Le joueur 1 a gagné %d parties\n", gainJ1);
printf("Le joueur 2 a gagné : %d parties\n", gainJ2);
printf("Nombre de parties ex-aequo : %d\n", gainexAeq);

float probaGagner1 = (float)gainJ1/1000;
float probaGagner2 = (float)gainJ2/1000;
float exAequo = (float)gainexAeq/1000;

printf("Proba de gagner joueur 1 :%f\n",probaGagner1);
printf("Proba de gagner joueur 2 :%f\n",probaGagner2);
printf("Proba de faire ex-aequo :%f\n",exAequo);
```

Pour faire les calculs en dessous, j'ai fait 1000 tours de boucle et non pas 100.



```
.     .  ____   ____   ____     ____  .   .
|\   /| |    | |    \ |    |   |    | |\  |
| \ / | |    | |   _/ |  __/   |    | | \ |
|  °  | |____| |   \  |        | |____| |  \|
--------------------------------------
--------------------------------------
Choisir deux joueurs en entrant les bon numéro :
'1' - Joueur débutant
'2' - Joueur malin
'3' - Joueur tricheur
'4' - Joueur Polytech
--------------------------------------
Entrer le joueur 1 :
1
Le joueur 1 est : 1 et joue avec le pion 'X'
--------------------------------------
Entrer le joueur 2 :
1
Le joueur 2 est : 1 et joue avec le pion 'O'
--------------------------------------
--------------------------------------
Le joueur 1 a gagné 517 parties
Le joueur 2 a gagné : 361 parties
Nombre de parties ex-aequo : 122
Proba de gagner joueur 1 :0.517000
Proba de gagner joueur 2 :0.361000
Proba de faire ex-aequo :0.122000
edouardbronnert@iMac-de-edouard-bronnert-1171 ~ %
```

On rentre les différentes probabilités dans le tableau suivant :

JOUEUR 1	JOUEUR 2	JOUEUR 1 gagne	ex-aequo
Débutant	Débutant	0.517	0.122
Débutant	Malin	0.404	0.217
Débutant	Tricheur	0.371	0.067
Malin	Tricheur	0.381	0.069
Malin	Malin	0.411	0.26
Tricheur	Tricheur	0.562	0.025

Question Bonus : Le Polytech

Le joueur Polytech prend une simple entrée clavier. Si la case est déjà occupée, il redemande de rentrer une valeur :

```c
void joueurPolytech(int *grille, int pion){

    printf("%s\n","_____________________________________");
    printf("            Joueur POLYTECH\n");
    printf("%s\n","_____________________________________");
    printf("•\n");
    printf("•\n");

    /* On montre les numéros de la grille : */

    printf("%s\n","|0|1|2|" );
    printf("%s\n","|3|4|5|" );
    printf("%s\n","|6|7|8|" );
    printf("•\n");


    int casePopo = 0;

    do{
    printf("Entrer un numéro de case correspondant à la grille au dessus ; cette case doit être libre !\n");
    scanf("%d",&casePopo);

    }while(grille[casePopo]!=0);

    grille[casePopo] = pion;
    grille[9] = casePopo;

    sleep(1);
    afficheGrille(grille);

}
```

J'ai opté pour une implémentation simple, sans vérification si le joueur entre un caractère autre qu'un chiffre. C'est un choix, je me suis dit que ce n'était pas essentiel pour ce TP.

```
---------------------------------------
                 COUP  4
---------------------------------------
---------------------------------------
            Joueur POLYTECH
---------------------------------------
•
•
|0|1|2|
|3|4|5|
|6|7|8|
•
Entrer un numéro de case correspondant à la grille au dessus ; cette case doit être libre !
5
```

Je n'ai pas mis beaucoup de captures de l'exécution, mais le code est joint au document.