TD 2

Processus

Rappels : On rappelle ci-dessous les principales primitives pour la création et l'attente de processus. Dans le cas de `wait()` et de `waitpid()`, on indique comment ces primitives peuvent être appelées de façon simplifiée.

— `fork()` permet à un processus d'en créer un autre. Retourne le PID du fils créé dans le père, et 0 dans le fils.

— `waitpid(pid, NULL, 0)` permet à un processus d'attendre la terminaison de son fils d'identifiant `pid`.

— `wait(NULL)` permet à un processus d'attendre la terminaison de l'un de ses fils (n'importe lequel).

Deux de plus pour la route :

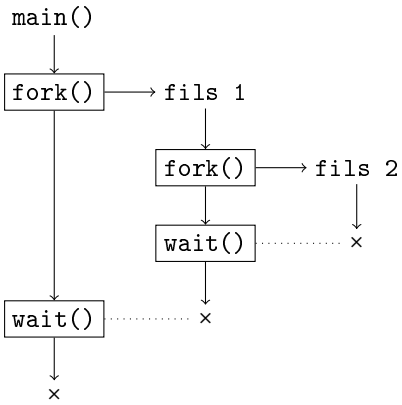
— `getpid()` retourne le PID du processus appelant.

— `getppid()` retourne le PID du père du processus appelant.

Dans les exercices qui suivent, vous supposez que tous les appels à ces primitives sont des succès : vous ne vous préoccupez pas des cas d'erreurs (ce n'est pas l'important ici).

EXERCICE 1 ▶ Un mode de représentation de l'exécution des processus

1) On considère le diagramme suivant qui représente l'exécution de 3 processus : le processus principal `main()` et deux procesus fils, `fils1` et `fils2`.



L'appel à `wait()` a été représenté sur le diagramme à l'instant où le processus fils attendu se termine, et donc où le processus parent sort de cet appel. Il vous est demandé d'écrire la fonction `main()` qui réalise à l'exécution le comportement décrit sur ce diagramme.

2) En vous inspirant du diagramme de la 1ère question, donnez une représentation d'une exécution possible du programme suivant.

```
int main(void) {
  if(fork() == 0) return 0;
  if(fork() == 0) return 0;
  wait(NULL);
  wait(NULL);
  return 0;
}
```

3) Même question avec le programme suivant. Vous noterez sur votre diagramme, pour chaque processus créé, les valeurs prises par la variable `i`, ce qui vous permettra de dérouler plus facilement l'exécution du programme.

```
int main(void) {
  for(int i = 0; i < 3; i++) {
    if(fork() > 0) {
      wait(NULL);
      break;
    }
  }
  return 0;
}
```

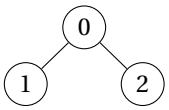
EXERCICE 2 ▶ Arborescences de processus

Le but est de vous entraîner à créer une arborescence de processus donnée. À chaque fois, le processus principal `main()` reçoit pour indice 0, et les autres processus sont indicés à partir de 1. Les processus créés ne font rien hormis :

— attendre individuellement les processus qu'ils ont créés avec `waitpid()`,

— se terminer en retournant 0.

1) Vous devez créer l'arborescence de processus ci-dessous. Vous donnerez deux programmes.



Dans le premier vous organiserez vos conditions sur la valeur de retour de `fork()` sous la forme

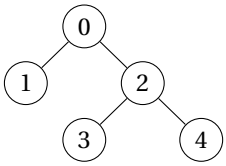
```
if(fork() == 0) { // code exécuté par le fils
  // ...
  return 0;
}
// code exécuté par le père
```

Dans le second vous organiserez ces mêmes conditions sous la forme

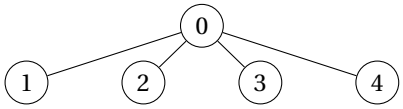
```
if(fork() > 0) { // code exécuté par le père
  // ...
else { // code exécuté par le fils
  // ...
}
```

Vous comparerez ensuite les deux formes du point de vue de la lisibilité !

2) Donnez deux programmes pour créer les arbres suivants. Pour (b), vous utiliserez un tableau d'entiers pour stocker le PID des processus fils, et deux boucles, l'une pour les créer les processus, l'autre pour les attendre individuellement.



(a)



(b)

EXERCICE 3 ▶ Temps d'exécution de programmes créant des fils

On travaille sur un système suffisamment peu chargé pour que, à chaque fois que l'on crée un processus avec `fork()` son exécution commence immédiatement. Dans cette exercice, on utilise des appels à `sleep(n)` qui permettent à un processus de se mettre en sommeil pendant `n` secondes. Donnez, à la second près, les temps d'exécutions des programmes suivants.

```
// programme 1

int main(void) {
  if( fork() > 0 ) {
    wait(NULL);
  }
  else {
    sleep(5);
  }
  return 0;
}
```

```
// programme 2
int main(void) {
  if( fork() > 0 ) {
    sleep(5);
    wait(NULL);
  }
  else {
    sleep(5);
  }
  return 0;
}
```

```
// programme 3
int main(void) {
  if( fork() > 0 ) {
    wait(NULL);
    sleep(5);
  }
  else {
    sleep(5);
  }
  return 0;
}
```

```
// programme 4



int main(void) {
  if( fork() > 0 ) {
    wait(NULL);
  }
  sleep(5);
  return 0;
}
```