Module ITC313 - Informatique

Partie C / C++

TP5 + TP6 hiérarchie de processus, signaux

Benoît Darties - benoit.darties@u-bourgogne.fr
Université de Bourgogne

Année universitaire 2016-2017

Dans ce TP, nous allons revenir sur quelques commandes en shell pour illustrer le fonctionnement de manière simple, puis nous déporterons ces concepts au travers d'appels systèmes dans une fonction C.

1 Gestion des signaux : envoi et reception

Les exercices de cette question se concentrent sur les mécanismes d'envoi et de reception des signaux. Ils visent prioritairement à comprendre comment on envoie un signal, quelles sont les conditions nécessaires pour qu'un processus accepte de recevoir un signal, et comment dérouter l'exécution normale d'un programme lorsque ce dernier reçoit un signal. Nous montrons notamment que le concept d'envoi et réception de signaux s'applique aussi bien pour des scripts shell que pour des programmes écrits dans un langage de programmation, ici le C. Dans la section d'après, nous verrons comment ces signaux sont utilisés pour gérer les processus, notamment les phases d'exécution et d'arrêt prématuré.

Exercice 1 : *Droits et signaux*

Le premier exercice de cette section s'intéresse à la notion de droits associés aux signaux.. Il vise à répondre à la question suivante : "est ce qu'un processus accepte les signaux envoyés par n'importe quel autre processus, ou pas ?". Pour cela, nous utiliserons plusieurs commandes shell, notamment la commande `ps` *qui permet de lister les processus actifs sur une machine, et la commande* `kill` *qui permet d'envoyer un signal à un processus identifié par son* `pid`*.*

Dans cet exercice, vous allez envoyer des signaux à des processus dont certains n'ont pas été lancés par vous (dont vous n'êtes pas les propriétaires) afin de déterminer s'il est possible d'envoyer des signaux à d'autres processus que ceux que l'on a lancé.

1. **Syntaxe d'envoi d'un signal** :
   (a) À l'aide de la commande `kill` et de l'option `-l`, listez l'ensemble des signaux que vous pouvez envoyer avec cette commande.
   (b) En vous aidant du manuel de la commande `kill`, notez la syntaxe permettant d'envoyer un signal identifié par un nom à un processus identifié par son `pid`.
   (c) Quel est normalement l'effet du signal `SIGKILL` ?
2. **Envoi d'un signal à un processus dont on est le propriétaire** :
   (a) En environnement shell, la variable `$$` affiche le `PID` du processus courant, et que l'instruction `while :` permet de boucler de manière infinie. On vous propose le script shell `boucleShell.sh` :

"Programme `boucleShell.sh`"

```
1 #!/bin/bash
2
3 echo "lancement du processus $$";
4
5 while :
6 do
7     sleep 1;
8     echo ".";
9 done
```

Ecrivez ce programme en langage C dans un fichier nommé `boucleShell.c` et compilez-le sous le nom `boucleInfinie`.

(b) Exécutez le programme précédemment rédigé dans un premier terminal et notez son `pid`. Ouvrez ensuite un second terminal et en utilisant la commande `kill`, envoyez le signal `SIGKILL` au processus exécutant `boucleInfinie`. Que se passe-t'il ?

3. **Envoi d'un signal à un processus dont on n'est pas le propriétaire** :

(a) Reprenez la question précédente, mais au lieu de lancer le programme `boucleInfinie` vous-même, demandez à un camarade de se s'identifier sur votre machine et de lancer le processus à votre place. Essayez ensuite d'envoyer le signal `SIGKILL` avec votre identité. Que se passe-t'il ?

(b) La commande `ps` permet de lister les processus de la machine. Pour avoir un affichage comprenant l'ensemble des processus, même ceux dont vous n'êtes pas propriétaire, ainsi que leur `pid`, utilisez la commande `ps aux`. Notez le `pid` de quelques processus appartenant à l'utilisateur `root`. Puis en utilisant la commande `kill`, essayez d'envoyer le signal `SIGKILL` à ce processus appartenant à l'utilisateur `root`. Que constatez-vous ? Le processus s'est-il terminé ? Vérifiez à l'aide de la commande `ps`.

Exercice 2 : *capture de signal et traduction en langage C*

Certains signaux peuvent être capturés, c'est à dire que l'on peut détecter la réception d'un signal envoyé par un autre processus, et redéfinir le comportement à adopter, c'est à dire la suite d'instructions à exécuter, lorsqu'un signal est détecté.

En vous inspirant des commandes et exemples présentés dans le cours :

1. Ecrivez un script shell comportant une boucle infinie (commande `while : do ... done`) à l'intérieur de laquelle le processus affiche un point sur une ligne chaque seconde. Utilisez la commande `sleep 1` pour attendre une seconde. Ajoutez à votre script une instruction pour qu'il affiche un message de votre choix si l'utilisateur appuie sur CTRL+C (SIGINT) pendant son exécution, et testez votre script.
2. Comment l'utilisateur pourrait-il envoyer un autre signal, par exemple SIGUSR1 au processus ?
3. Complétez votre script pour que ce dernier affiche un autre message s'il reçoit le signal SIGUSR1 lors de son exécution.
4. Comment arrêter ce processus ?
5. Traduisez votre script shell en langage C.

Remarques / questions : benoit.darties@u-bourgogne.fr

Exercice 3 : *Capture de signaux et redirections (exercice difficile)*

Certains signaux peuvent être capturés et redirigés, c'est à dire que lorsque le signal est reçu, on redéfinit le comportement par défaut qu'aurait du avoir le processus. Dans un premier temps, nous allons voir quel est le comportement par défaut des processus lors de la réception d'un signal. Dans un second temps, nous allons redéfinir le comportement d'un processus à la réception d'un signal.

Lorsqu'ils sont reçus, beaucoup de signaux, mais pas tous, entrainent par défaut la fin du processus. Dans cet exercice, nous allons d'abord essayer de déterminer quels sont les signaux qui entrainent la fin du processus. Puis nous essayerons de limiter cet effet en redirigeant les signaux pour que le processus ne s'arrête pas.

1. **Identification des signaux qui mettent fin au processus par défaut :** Par défaut, de nombreux signaux terminent l'exécution d'un processus. Nous allons déterminer lesquels en utilisant l'idée suivante, qui sera mise en place au travers d'un script : pour chaque signal numero i, nous allons lancer un processus en arrière-plan qui fait une boucle infinie, et on notera son `pid`. Puis on lui enverra le signal i. Ensuite on vérifiera si le processus existe toujours ou pas. S'il existe toujours, on en concluera que le signal numéro i ne tue pas par défaut le processus. Sinon, on affichera un message indiquant que le signal numéro i tue le processus par défaut et on relance un nouveau processus en arrière-plan pour recevoir un prochain signal. Les questions suivantes ont pour objectif de vous donner les différentes pièces permettant de constituer un tel script en bash.
   (a) En vous basant sur le cours, quel est le signal qui ne peut pas être redirigé et dont la réception tue toujours le processus ?
   (b) Créez ou reprenez en le modifiant à votre guise le programme `boucleShell.sh`, afin de disposer d'un programme qui tourne de manière indéfinie lorsqu'il est lancé (il est recommandé de mettre une instruction de pause de 1 seconde à chaque itération, plutôt qu'une boucle infinie sans instruction, qui monopoliserait tout le temps le processeur.
   (c) Pour lancer un processus en arrière-plan, il suffit de rajouter le caractère `&` à la fin d'une ligne de commande. Testez ceci sur le programme `boucleShell.sh`
   (d) En vous inspirant sur la documentation disponible sur internet via une recherche sur Google, ou via l'adresse `http://wiki.bash-hackers.org/syntax/shellvars`, déterminer à quoi correspond la variable '!' ? vérifiez votre réponse en lançant successivement la commande `boucleShell.sh` puis l'affichage de la variable `!`, en faisant tourner `boucleShell.sh` soit en premier plan, soit en arrière-plan.
   (e) En vous référant au cours, quelle variable permet de récupérer le code retour de la dernière commande lancée dans un shell ?
   (f) A quoi correspond l'option `-p` de la commande `ps` ? Testez cette commande avec l'option `-p` en précisant soit un numéro de processus existant, soit un numéro de processus inexistant. Notez à chaque fois la valeur du code retour renvoyé par la commande `ps`, selon que le numéro de processus passé en paramètre existait ou non. identifiant de processus actif ou non, et affichez son code retour à chaque essai. Que constatez vous ?
   (g) (Pou des raisons de confort visuel seulement) : sachant que le fichier `dev/null` est une sorte de fichier fantôme dans lequel on peut rediriger ce que l'on ne souhaite pas garder, comment redirige-t'on en shell les messages standard et d'erreur d'un processus de sorte à ignorer ces derniers ?
   (h) En utilisant tous les éléments présentés dans les questions précédentes, proposez un script shell `TestKillerSignal.sh` permettant de tester quels sont les signaux qui, par défaut, tuent un processus.
2. **Redirection de signaux :**
   (a) Rappeler quels sont les deux signaux qui ne peuvent pas être redirigés.
   (b) La commande `trap` vue en cours permet de rediriger le processus en cas de réception d'un signal. Utilisez l'option `-l` de cette commande pour lister l'ensemble des signaux ainsi que leur nom usuel.

Remarques / questions : benoit.darties@u-bourgogne.fr

(c) Utilisez ensuite l'option `-p` de cette commande pour lister l'ensemble des signaux actuellement redirigés. En utilisant `trap` de manière analogue à ce qui a été vu en cours, définissez une instruction permettant de rediriger un signal reçu par le shell, de sorte a afficher un message de votre choix lorsque le shell reçoit ce signal. Vérifiez que la liste des signaux redirigés a été mise à jour.

(d) Modifiez votre programme `boucleShell.sh` de sorte à ce que tous les signaux qui provoquaient la fin du processus soient redirigés. A chaque fois que l'un de ces signaux est reçu, affichez un message spécifique mentionnant le nom du signal redirigé.

(e) Testez enfin vos redirections en relançant le script `TestKillerSignal.sh` que vous avez développé dans la partie précédente, désormais en utilisant le programme `boucleShell.sh` modifié comme victime. Combien de signaux tuent encore le programme `boucleShell.sh` ? Ce résultat était-il attendu ?

Exercice 4 : *envoi multiples et capture de signal en C*

Au terme des exercices précédents, vous devez avoir compris les notions de signal, de redirection, et de mise en oeuvre en langage shell. L'objectif de cet exercice est la simple traduction de ce qui a été vu en shell en langage C, et l'étude d'une autre commande nommée ***`killall`***

En vous inspirant des commandes et exemples présentés dans le cours :

1. Completez le script shell `captureShell.sh` qui comporte une boucle infinie au sein de laquelle on attend 1 seconde (instruction `sleep 1`), en ajoutant une instruction pour qu'il affiche un message de votre choix si l'utilisateur appuie sur `ctrl +C` (signal `SIGINT`) pendant son exécution.
2. Traduisez votre script shell en un programme en langage C nommé `captureC.c` qui effectue la même chose et compilez le en un exécutable nommé `captureC`.
3. Lancez 3 instances de votre programme `captureC` dans différents terminaux. Pour envoyer un signal `SIGINT` à chacun de ces processus autrement qu'avec le clavier, nous pourrions utiliser la commande `kill`. L'inconvénient est qu'il faut exécuter cette commande 3 fois, et ce après avoir récupéré l'identifiant de chacun des processus. Une commande permet de faire tout ceci plus simplement, en identifiant les processus non pas via leur `pid` mais via leur nom de fichier : `killall`. En vous aidant de la documentation de cette commande, envoyez le signal `SIGINT` à chacun des processus `captureC`.
4. Mettez en pause tous ces processus avec une seule commande, puis relancez-les tous en même temps. Enfin, tuez-les tous en une seule commande.

2 Gestion des processus

Les exercices de cette section s'intéressent à la façon dont les processus sont créés, et leur gestion en avant-plan ou arrière-plan, et la notion de recouvrement de processus.

Exercice 5 : *Processus en premier-plan / Arriere-plan*

Comme vu dans un exercice précédent, il est possible de lancer un processus en arrière-plan. Le principal soucis est alors qu'il est beaucoup plus difficile de dialoguer avec le processus puisque les caractères entrés au clavier sont envoyés au shell, et non au processus en arrière-plan. Nous allons voir ici comment manipuler les processus pour les passer tantôt en premier-plan, tantôt en arrière-plan, les stopper et les relancer. Ces manipulations sont rendues possibles par l'utilisation de signaux, et les fonctionnalités de deux programmes : ***`bg`*** *(background) et* ***`fg`*** *(foreground)*

Remarques / questions : benoit.darties@u-bourgogne.fr

"Programme boucleShell.sh"

```
1 #!/bin/bash
2
3 echo "lancement du processus $$";
4
5 while :
6 do
7     sleep 1;
8     echo ".";
9 done
```

Pour les besoins de cet exercice, Nous allons réutiliser le script shell `boucleShell.sh`, à nouveau présenté ci-après :

Nous allons montrer qu'il est possible, avec les signaux, de stopper puis reprendre l'exécution de processus, de les basculer en arrière-plan ou au premier-plan, et de gérer leur exécution à partir d'un autre shell.

1. Rajoutez à ce script une instruction pour que ce dernier affiche son identifiant lorsqu'il reçoit le signal `SIGINT` (touches `ctrl+c` sur le clavier). Pour tuer ce processus, il nous faudra envoyer le signal `SIGKILL` à ce processus depuis un autre terminal.
2. Lancez l'exécution de ce script dans un premier terminal ; puis envoyez-lui le signal `SIGINT` afin d'identifier son numéro de processus.
3. Dans un second terminal, exécutez la commande `ps l -p` suivie du numéro de processus précédemment relevé. Cette commande vous donne une liste d'informations sur l'exécution du processus. La colonne `STAT` de l'affichage doit contenir la valeur `R` ou `R+` selon les systèmes. Parcourir le manuel de la commande `ps` afin de déterminer à quel état correspond la lettre `R`. Quels sont les autres états possibles ? A quoi correspond le signe `+` (si ce dernier est présent).
4. De retour dans le premier terminal, envoyez le signal `SIGTSTP` (touches `ctrl+z` sur le clavier) au processus. Que se passe-t'il ? Notez l'apparition du 1 encadré par des crochets. Vérifiez l'état de ce processus au moyen de la commande `ps` dans l'autre terminal. Quel est ce nouvel état ?
5. Envoyez le signal `SIGCONT` au processus via la commande `kill`. Affichez son état au moyen de la commande `ps`. A présent le processus est actif en **arrière plan**.
6. Dans le premier terminal, tapez la commande `fg`(pour *foreground*) suivie de numéro présent entre crochets (ici 1). Notez que le processus est revenu au premier plan est est à nouveau actif.
7. Renvoyez un signal `SIGTSTP` pour arrêter le processus, puis tapez la commande `bg`(pour *foreground*) suivie de numéro présent entre crochets (ici 1). Notez ensuite l'état du processus au moyen de la commande `ps`. Concluez-vous que `bg` permet bien d'envoyer un signal `SIGCONT` à un processus actuellement arrêté ? Ré-exécutez la commande `bg 1` et notez le message d'erreur.
8. Lancez dans le même terminal de nouvelles instances du script. Identifiez leur `PID`, puis envoyez-leur le signal `SIGTSTP`. Notez l'incrémentation du numéro entre crochets. Enchaînez les manipulations de sorte à passer les processus tantôt au premier plan, tantôt en arrêt, tantôt en arrière plan.
9. Lancez une nouvelle instance du script en faisant suivre votre commande du caractère `&`. Quelle conséquence sur l'exécution du processus ? Basculez ce processus en premier plan avec la commande `fg`.
10. Notez le PID de quelques processus, passez tous les processus en arrière-plan, puis déconnectez-vous du terminal (fermeture de la fenêtre ou commande `exit`). Vos processus tournent-ils encore ? Vérifiez dans un autre terminal la présence ou non de vos processus au moyen de la commande `ps`.

11. Lorsqu'un terminal est déconnecté (par exemple en , le shell détecte la fin de session et doit normalement envoyer un signal `SIGHUP` à ses processus enfants, qui doit théoriquement tuer ces derniers, sauf si le processus a été lancé avec la commande `nohup`. Certains shell (bash par exemple) capturent le signal `NOHUP` afin de ne pas l'envoyer aux processus enfants, ce qui peut expliquer que ces derniers continuent à tourner alors que l'utilisateur est déconnecté. Consultez le manuel de `nohup`. Notez les particularités concernant la redirections des sorties standard et erreur. Lancez le script tantôt avec `nohup`, tantôt sans `nohup`, et essayer d'envoyer le signal `SIGHUP`. Notez la différence.

Exercice 6 : *Duplication et recouvrement de processus*

Un des points partiellement abordés dans le cours est le recouvrement de processus. Un processus enfant naît par clonage d'un processus parent. Sans autres mécanismes, tous les processus seraient toujours identiques. Lorsque l'on souhaite lancer un nouveau programme dont le code est contenu dans un fichier, il faut donc d'abord cloner un processus, et dans le processus enfant utiliser une instruction supplémentaire qui dise d'annuler le code actuel du processus et de le remplacer par celui contenu dans le fichier contenant le code du nouveau programme a exécuter. C'est ce qu'on appelle le recouvrement de processus.

Dans cet exercice, nous recouvrons un processus existant, d'abord au travers d'un script shell, puis en appliquant ce concept au sein d'un programme C.

1. **Manipulation de la commande exec sur un shell**
   (a) Que fait la commande `exec` ? D'après le cours, quel appel système est associé à cette commande ?
   (b) La commande `sleep 3` consiste à attendre simplement 3 secondes. Testez cette commande une première fois. Exécutez ensuite la commande `exec` dans un terminal avec comme paramètre la commande `sleep 3`. Notez que la commande `sleep` s'est bien exécutée, mais que le shell a désormais disparu.
   (c) La commande `echo $$` permet de voir le `pid` d'une invite de commande shell. Nous disposons de plusieurs shell sur le système, parmi lesquels `/bin/bash`, `/bin/sh`, et `/bin/tcsh`. Affichez le numéro de processus de votre invite shell, puis lancez un nouveau shell dans le shell en cours. Affichez les `pid` avant et après lancement du shell. Constatez qu'il s'agit bien de processus différents. Répétez la manipulation avec différents shells. Que se passe-t'il si vous quittez le shell nouvellement créé avec la commande `exit` ?
   (d) Refaites la même manipulation, mais cette fois en lançant le nouveau shell avec la commande `exec`. La valeur du `pid` a-t'elle changé ? Un nouveau processus a t'il été créé, ou est-ce l'ancien processus qui a été écrasé ? Que se passe-t'il si l'on exécute la commande `exit`
2. **Application dans un script shell**
   (a) Recopiez ou récupérez le script `testExec.sh` suivant :

"Programme testExec.sh"

```
1 #!/bin/bash
2
3 exec echo "bonjour";
4 exec echo "bonsoir";
```

   (b) Exécutez-ce script. Quel sera/seront le(s) message(s) affiché(s) à l'écran et pourquoi ?
3. **Application dans un programme C** : Dans cet partie, nous allons créer un lanceur de programme.

Remarques / questions : benoit.darties@u-bourgogne.fr

(a) En réutilisant les éléments de programmes vus précédemment, créez un programme `launcher.c` qui demande à l'utilisateur de saisir un nombre (utilisez la fonction `scanf`). Si le nombre est égal à 1 ou 2, ce programme se duplique ensuite au moyen de l'appel système `fork()`.

(b) Dans le processus fils, on veut que si le nombre entré était 1, le processus soit écrasé avec le code du programme `/bin/hostname`. Si la valeur était 2, on souhaite que le processus soit écrasé avec le code du programme `/bin/date`. Utilisez la fonction `execve()`, en vous aidant de son manuel, pour réaliser ce programme. Pour vous aider, sachez qu'il n'est pas nécessaire ici de passer une liste de variables d'environnement, ni d'arguments lors de l'appel à l'un ou l'autre des programmes écrasants. Vous pouvez donc utiliser le paramètre `NULL` lors de l'appel à la fonction `execve()`.

3 Gestion des processus

Les exercices de cette section se consacrent à l'étude des relations de parenté entre processus. On rappelle qu'un processus est toujours créé par réplication d'un processus parent, puis écrasement du code. La réplication d'un processus s'effectue à l'aide de l'instruction `fork()`, qui permet de dupliquer le processus en conservant son état d'avancement. Seul le code retour de la fonction `fork()` est différent entre le processus parent et le processus enfant.

Exercice 7 : *Duplication de processus*

L'objectif de cet exercice est de comprendre la notion de processus parent et de processus enfant lors de la duplication d'un processus par l'appel à la fonction `fork()`. Nous y revoyons la notion de `PID` et `PPID` (ou `PID` du parent) vus en cours. L'objectif est de mettre en évidence le role de l'appel à l'instruction `fork()` ainsi que la particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent. On rappelle que `fork()` retourne 0 dans le processus nouvellement créé (fils) et une valeur strictement supérieure à 0 sinon (qui correspond au `PID` du processus nouvellement créé)

Dans tout l'exercice, on suppose que la mémoire est suffisante et la table des processus n'est pas pleine. L'obtention du `PID` ou du `PPID` peut s'obtenir au moyen des fonctions `getpid()` et `getppid()` (vues en cours). Pour afficher la valeur du `PID` à l'écran, on pourra utiliser les instructions suivantes :

```
1  int pid;
2  pid = getpid();
3  printf("la valeur du pid est %d \n", pid);
```

1. Ecrire un programme C qui se duplique avec la fonction `fork()`, tel que chaque processus affiche son rôle (parent ou enfant) à l'écran ainsi que son `PID` et le `PID` de son parent.
2. Ecrire un programme C qui se duplique avec la fonction `fork()` et dont le fils se duplique également, tel que chaque processus affiche son rôle (grand-père, père ou fils) à l'écran ainsi que son `PID` et le `PID` de son parent.

Exercice 8 : *Creation et destruction de processus*

Cet exercice rappelle les notions de processus zombie et processus orphelin, et définit les modalités d'apparition de ces deux types particuliers de processus.

Remarques / questions : benoit.darties@u-bourgogne.fr

1. Qu'est ce qu'un processus zombie ? en décrivant les grandes lignes du programme, expliquez comment créer un processus zombie.
2. Qu'est ce qu'un processus orphelin ? Une fois orphelin, à qui est-il rattaché ? Décrire les grandes lignes d'un programme permettant de créer un orphelin.

Exercice 9 : *Evaluation du nombre de processus*

L'objectif de cet exercice est de comprendre le résultat de l'appel à l'instruction `fork()` *ainsi que la particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent. On rappelle que* `fork()` *retourne 0 dans le processus nouvellement créé (fils) et une valeur strictement supérieure à 0 sinon (qui correspond au PID du processus nouvellement créé)*

Pour chacun des programmes suivants, précisez combien de processus sont créés lors de l'exécution de ces derniers :

1. premier programme :

"Programme programme1.c"

```
1 int main() {
2     fork();
3     fork();
4     fork();
5 }
```

2. second programme :

"Programme programme2.c"

```
1 int main() {
2     if (fork() > 0) {
3         fork();
4     }
5 }
```

3. troisième programme :

"Programme programme3.c"

```
1 int main() {
2     int cpt=0;
3     while (cpt < 3) {
4         if (fork() > 0)
5             cpt++;
6         else
7             cpt=3;
8     }
9 }
```

Exercice 10 : *Conjonctions, Disjonctions, et Duplication*

L'objectif de cet exercice est de comprendre le résultat de l'appel à l'instruction `fork()` *ainsi que la*

particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent. On rappelle que

1. Dessiner l'arborescence des processus engendrée par le programme `conjonction1.c` suivant :

"Programme conjonction1.c"

```
1 int main() {
2     fork() || (fork() && fork() );
3     exit(EXIT_SUCCESS);
4 }
```

2. Même question avec le programme `conjonction2.c` suivant :

"Programme conjonction2.c"

```
1 int main() {
2     fork() \&\& (fork() || fork());
3     exit(EXIT_SUCCESS);
4 }
```

Exercice 11 : *Terminaison normale de processus*

Lorsqu'un processus est correctement créé, une bonne pratique consiste à ce que le processus parent, qui a créé le nouveau processus, attende systématiquement la fin de l'exécution ce dernier au moyen de la fonction `wait()`*. Ainsi, si l'instruction* `fork()` *a été appelée n fois, il doit en être autant pour l'instruction* `wait()`*, ni plus ni moins, et autant de fois qu'un processus a d'enfant. L'objectif affiché est de placer correctement les instructions* `wait()`

Dans les programmes suivants, rajouter les instructions `wait()` et les éventuels tests conditionnant l'exécution de ces dernières de sorte que chaque processus enfant informe correctement son parent de sa fin d'exécution.

1. premier programme :

"Programme programme1.c"

```
1 int main() {
2     int result,  a=0;
3     result = fork();
4     if (result  > 0)
5         a=5;
6 }
```


Remarques / questions : benoit.darties@u-bourgogne.fr

2. second programme :

"Programme programme2.c"

```
1  int main() {
2      int result1, result2, result3;
3      result1 = fork();
4      result2 = fork();
5      result3 = fork();
6  }
```

3. troisième programme :

"Programme programme3.c"

```
1  int main() {
2      int result1, result2, result3;
3      result1 = fork();
4      if (result1 ==0) {
5          result2 = fork();
6          if (result2 == 0) {
7              result3 = fork();
8          }
9      }
10 }
```