Systèmes d'exploitation

TP 2 - Processus - Correction

2023-03-06

1 Créer des processus avec l'API Unix

1. Écrivez un programme qui appelle `fork()`. Avant d'appeler `fork()`, demandez au processus principal d'accéder à une variable (par exemple, x) et de lui donner une valeur (par exemple, 100). Quelle est la valeur de la variable dans le processus enfant ? Qu'arrive-t-il à la variable lorsque l'enfant et le parent changent la valeur de x ?

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

int main(int argc, char *argv[]) {
    int x = 100;

    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("[child] x = %d\n", x);
        printf("[child] set x = %d\n", 200);
        x = 200;
        printf("[child] x = %d\n", x);
    } else {
        printf("[parent] x = %d\n", x);
        printf("[parent] set x = %d\n", 300);
        x = 300;
        printf("[parent] x = %d\n", x);
    }
    return 0;
}
```

Le processus parent ou enfant peut modifier sa propre copie de la variable x indépendamment et elle changera sans effet notable sur l'autre processus.

2. Écrivez un programme qui ouvre un fichier (avec l'appel système `open()`) et qui appelle ensuite `fork()` pour créer un nouveau processus. L'enfant et le parent peuvent-ils tous deux accéder au descripteur de fichier renvoyé par `open()` ? Que se passe-t-il lorsqu'ils écrivent dans le fichier simultanément, c'est-à-dire en même temps ?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>


int main(int argc, char *argv[]) {
    int fd = open("q2.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRWXU);

    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        int pos = lseek(fd, 0, SEEK_CUR);
        printf("[child] writing at file position %d\n", pos);

        char s[] = "child";
        int written = write(fd, s, strlen(s));

        printf("[child] %d bytes written\n", written);

        close(fd);
    } else {
        int pos = lseek(fd, 0, SEEK_CUR);
        printf("[parent] writing at file position %d\n", pos);

        char s[] = "parent";
        int written = write(fd, s, strlen(s));

        printf("[parent] %d bytes written\n", written);

        close(fd);
    }
    return 0;
}
```

L'appel système `open` renvoie un descripteur de fichier.

> The return value of open() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is

used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file.

https://man7.org/linux/man-pages/man2/open.2.html

Le processus parent et le processus enfant partagent le même descripteur de fichier. Cela signifie que la position d'écriture (*file offset*) est partagée. Les écritures des deux processus dans le fichier sont susceptibles de s'entrelacer.

> The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see open(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of F_SETOWN and F_SETSIG in fcntl(2)).

https://man7.org/linux/man-pages/man2/fork.2.html

3. Écrivez un autre programme en utilisant `fork()`. Le processus enfant doit imprimer "hello" ; le processus parent doit imprimer "goodbye". Vous devez essayer de faire en sorte que le processus enfant imprime toujours en premier ; pouvez-vous le faire sans appeler `wait()` dans le processus parent ?

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello\n");
    } else {
        printf("goodbye\n");
    }
    return 0;
}
```

On pourrait essayer de permettre à l'enfant de s'exécuter en appelant successivement `sleep` ou `yield` dans le parent. Cependant, cette méthode ne garantit pas que l'enfant va afficher en premier. Sans variable partagée entre les processus et sans utiliser `wait`, il n'est pas possible d'avoir cette garantie.

4. Écrivez un programme qui appelle `fork()` et qui appelle ensuite une forme de `exec()` pour exécuter le programme `/bin/ls`. Voyez si vous pouvez essayer toutes les variantes de exec(), y compris (sous Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()` et `execvpe()`. Pourquoi pensez-vous qu'il existe autant de variantes du même appel de base ?

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("[child] calling /bin/ls\n");

        char *args[] = {"/bin/ls", ".", NULL};
        execvp(args[0], args);
        printf("this line will not be printed if execvp() runs correctly\n");
    } else {
        printf("[parent] terminating\n");
    }
    return 0;
}
```

Les différentes variantes de `exec()` correspondent à différents cas d'utilisation. De nouvelles variantes ont été ajoutées pour introduire de nouvelles fonctionnalités tout en conservant une compatibilité avec les programmes déjà existants.

5. Maintenant, écrivez un programme qui utilise `wait()` pour attendre que le processus enfant se termine dans le parent. Que retourne `wait()` ? Que se passe-t-il si vous utilisez `wait()` dans le processus enfant ?

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1);
    } else {
        int wait_code = wait(NULL);
        printf("hello, I am parent of %d (wait_code:%d) (pid:%d)\n", rc,
↪   wait_code, (int) getpid());
```

```
    }
    return 0;
}
```

`wait()` est utilisé pour attendre les changements d'état d'un enfant du processus appelant. Le processus appelant n'ayant pas d'enfant, `wait()` ne bloquera pas son exécution.

6. Écrivez une légère modification du programme précédent, en utilisant cette fois-ci `waitpid()` au lieu de `wait()`. Quand `waitpid()` serait-il utile ?

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1);
    } else {
        int wait_code = waitpid(rc, NULL, 0);
        printf("hello, I am parent of %d (wait_code:%d) (pid:%d)\n", rc,
 ↪  wait_code, (int) getpid());
    }
    return 0;
}
```

`waitpid()` est notamment utile lorsqu'un processus parent a plusieurs enfants et qu'il doit attendre un processus enfant en particulier.

7. Écrivez un programme qui crée un processus enfant, et ensuite dans l'enfant ferme la sortie standard (`STDOUT_FILENO`). Que se passe-t-il si le fils appelle `printf()` pour imprimer une sortie après avoir fermé le descripteur ?

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
```

```
        exit(1);
    } else if (rc == 0) {
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        close(STDOUT_FILENO);
        printf("this will not be printed\n");
    } else {
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

8. Écrivez un programme qui crée deux enfants et connecte la sortie standard de l'un à l'entrée standard de l'autre, en utilisant l'appel système `pipe()`. Vous aurez également besoin de l'appel système `dup2()`.

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

void write_to_pipe_as_stdout(int pipe_fds[]) {
    close(pipe_fds[0]);
    dup2(pipe_fds[1], STDOUT_FILENO);

    printf("Hello from the write child!\n");
}

void read_from_pipe_as_stdin(int pipe_fds[]) {
    close(pipe_fds[1]);
    dup2(pipe_fds[0], STDIN_FILENO);

    char msg[101];
    scanf("%100[^\n]", msg);
    printf("Read child message from the pipe: %s\n", msg);
}

int main(int argc, char const *argv[]) {
    int pipe_fds[2];
    if (pipe(pipe_fds) < 0) {
        fprintf(stderr, "pipe failed\n");
        exit(1);
    }

    int rc1 = fork();
    if (rc1 == -1) {
        fprintf(stderr, "fork failed\n");
```

```
        exit(1);
    }

    if (rc1 == 0) {
        write_to_pipe_as_stdout(pipe_fds);
    } else {
        int rc2 = fork();
        if (rc2 == -1) {
            printf("fork failed\n");
            exit(1);
        }
        if (rc2 == 0) {
            read_from_pipe_as_stdin(pipe_fds);
        }
    }

    return 0;
}
```

Cette question nécessite l'utilisation de l'appel système `dup2` qui permet de faire de la *read end* du *pipe* (`pipe_fds[0]`) l'entrée standard du processus (`STDIN_FILENO`) et de la *write end* du *pipe* (`pipe_fds[1]`) la sortie standard du processus (`STDOUT_FILENO`).

> pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.
>
> https://man7.org/linux/man-pages/man2/pipe.2.html

> The dup() system call allocates a new file descriptor that refers to the same open file description as the descriptor oldfd. […] The dup2() system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in newfd. In other words, the file descriptor newfd is adjusted so that it now refers to the same open file description as oldfd.
>
> https://man7.org/linux/man-pages/man2/dup.2.html