osj

April 9, 2025

Contents

0.1 Introduction

faComment Simplified Explanation

Imagine a process as a task running on your computer. A parent task can create a child task, similar to how a manager can delegate a task to an employee. The child task inherits certain attributes from the parent, just like how a child inherits traits from their parents. When a task is running, it moves between different queues, much like how a person can move between different tasks throughout the day. When a task is done, it becomes a 'zombie', and needs to be cleared away to prevent clutter, just like how you would clean up your workspace after finishing a task.

0.2 Process Creation and Termination

In a computer system, one process, which we can refer to as the parent, has the ability to create another process, known as the child. This is achieved through the allocation and initialization of a new Process Control Block (PCB). For a practical understanding of this, you can run the command 'ps auxwww' in the shell; the PPID displayed is the parent's Process ID (PID).

0.2.1 Inheritance in POSIX

In POSIX, a child process inherits most of the parent's attributes. These attributes include the User ID (UID), open files, current working directory (cwd), and so on. It is important to note that open files should be closed if they are not needed. Why is this so? This is because keeping them open could lead to resource leaks and potential system instability.

0.2.2 Process Execution and State Changes

While a process is executing, the PCB moves between different queues according to the state change graph. These queues include the runnable queue, and the sleep/wait for event queues (i=1,2,3...).

0.2.3 Process Termination

After a process dies, either through calling exit() or being interrupted, it becomes a zombie. The parent process uses the wait* system call to clear the zombie from the system. Why is this necessary? This is to prevent the accumulation of zombie processes which could waste system resources. The wait system call family includes wait, waitpid, waitid, wait3, and wait4. An example of a wait system call is:

```
pid_t wait4(pid_t, int *wstatus, int options, struct rusage *rusage);
```

The parent process can either wait for its child to finish execution or run in parallel with it. The wait*() system call will block unless WNOHANG is given in 'options'. For a deeper

understanding of this, you can read 'man 2 wait'.

faComment Vulgarisation simple

Imaginez que vous voulez créer un clone de vous-même pour vous aider à accomplir une tâche. Vous utilisez une machine spéciale (la fonction fork()) qui crée une copie exacte de vous (le processus enfant). Cette machine donne à votre clone un numéro d'identification unique (pid) et le met au travail. Mais si quelque chose se passe mal avec la machine (fork() échoue), elle vous donne un message d'erreur pour vous dire ce qui ne va pas (perror()).

0.3 Création d'un processus enfant avec fork()

Dans un système informatique, un processus peut créer un autre processus. C'est ce que nous appelons un processus parent et un processus enfant. La fonction fork() est utilisée pour initialiser un nouveau Bloc de Contrôle de Processus (PCB) basé sur la valeur du processus parent. Le PCB est ensuite ajouté à la file d'attente exécutable.

0.3.1 Fonctionnement de fork()

La fonction fork() retourne deux fois. Elle retourne une fois dans le processus parent, avec un pid (Process ID) supérieur à 0, et une autre fois dans le processus enfant, avec un pid égal à 0. À ce stade, il y a deux processus au même point d'exécution. L'espace d'adressage du processus enfant est une copie complète de l'espace du processus parent, à une différence près...

0.3.2 Gestion des erreurs avec errno

'errno' est une variable globale qui contient le numéro d'erreur du dernier appel système. Si la fonction fork() échoue, elle imprime la chaîne associée à errno avec la fonction perror().

```
int main(int argc, char *argv[])
{
  int pid = fork();
  if( pid==0 ) {
   //
   // child
      //
      printf("parent=%d son=%d\n",
             getppid(), getpid());
  }
  else if( pid > 0 ) {
      //
      // parent
      //
      printf("parent=%d son=%d\n",
```

```
            getpid(), pid);
  }
  else { // print string associated
         // with errno
      perror("fork() failed");
  }
  return 0;
}
```