osj

April 9, 2025

Contents

0.1 Introduction

faComment Simple Explanation

Think of the parent process as a factory. It creates a new product (the child process) and sets up the basic attributes for it. The child process then goes on to do its own thing, but it still carries the 'DNA' (attributes) of its parent. When the child process is done (or 'dies'), it doesn't just disappear. It turns into a 'zombie' and the parent has to clean it up. This is important to keep the system running smoothly.

0.2 Process Creation and Termination

In the world of operating systems, one process, which we can call the parent, has the ability to create another process, known as the child. This is done by allocating and initializing a new Process Control Block (PCB).

- Homework: Try running 'ps auxwww' in the shell. The PPID you see is the parent's PID.

0.2.1 Inheritance in POSIX

In POSIX, a child process inherits most of the parent's attributes. This includes the User ID (UID), open files, current working directory (cwd), and so on.

- Note: If any open files are unneeded, they should be closed. Can you think of why this might be important?

0.2.2 PCB Movement

While a process is executing, its PCB moves between different queues. This movement is dictated by the state change graph.

- Queues: runnable, sleep/wait for event i (where i can be any integer)

0.2.3 Process Death and Zombies

After a process dies, either by 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 might this be necessary?
- The wait system call family includes: wait, waitpid, waitid, wait3, wait4. For example:

0.2.4 Parent Process Behavior

The parent process can either sleep/wait for its child to finish or run in parallel. The wait*() function will block unless WNOHANG is given in 'options'.

- Homework: Read 'man 2 wait'

```latex

\begin{tcolorbox}[colback=blue!10,colframe=blue,title={\fontfamily{lmr}\selectfont \\faComment\ Vulgarisation simple},fonttitle=\bfseries,fontupper=\fontfamily{lmr}\selectfont,boxrule=1pt,sharp corners]
Imaginez que vous êtes en train de faire une tâche et que vous voulez la dupliquer pour la faire en parallèle. Dans le monde des systèmes d'exploitation, c'est ce que fait la fonction fork(). Elle crée une copie du processus en cours (le parent) pour créer un nouveau processus (l'enfant). Ces deux processus peuvent alors s'exécuter en parallèle, chacun avec son propre espace d'adressage. C'est comme si vous aviez cloné vous-même pour faire deux tâches en même temps !
\end{tcolorbox}
\section{Création d'un processus enfant avec fork()}

Dans le monde des systèmes d'exploitation, la fonction fork() est utilisée pour créer un nouveau processus, appelé processus enfant, à partir d'un processus existant, appelé processus parent. Lorsque fork() est appelé, un nouveau Bloc de Contrôle de Processus (PCB) est initialisé basé sur la valeur du processus parent et ajouté à la file d'attente des processus prêts à être exécutés.

\begin{itemize}
\item fork() initialise un nouveau PCB basé sur la valeur du processus parent
\item Le PCB est ajouté à la file d'attente des processus prêts à être exécutés
\item Il y a maintenant deux processus au même point d'exécution
\item L'espace d'adressage du nouveau processus enfant est une copie complète de l'espace du processus parent, avec une différence...
\end{itemize}

La fonction fork() retourne deux fois : une fois dans le processus parent avec un pid > 0 et une fois dans le processus enfant avec un pid = 0.

\begin{itemize}
\item fork() retourne deux fois : au parent, avec pid > 0 et à l'enfant, avec pid = 0
\end{itemize}

La variable globale 'errno' contient le numéro d'erreur du dernier appel système. Si fork() échoue, 'errno' contiendra le numéro d'erreur correspondant.

\begin{itemize}
\item 'errno' est une variable globale qui contient le numéro d'erreur du dernier appel système
\end{itemize}
\begin{lstlisting}[language=C]
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;
}
```