Appels systèmes et processus

SYE

3 - Appels Systèmes et Processus

Résumé du document

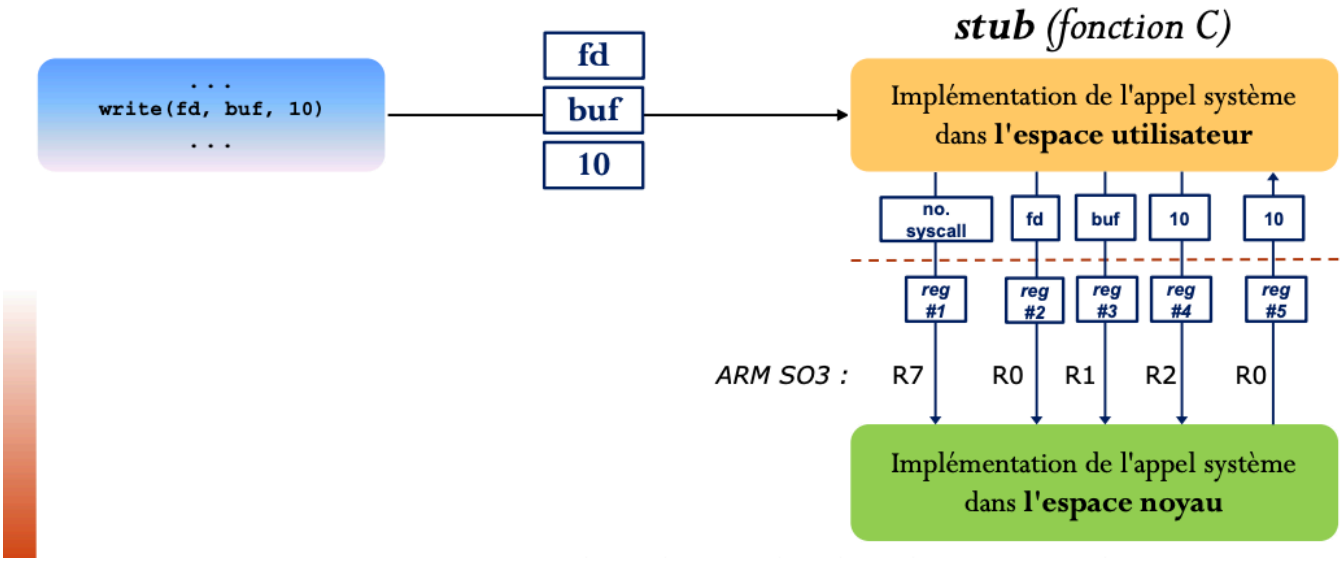
Definition

Table des matières

1. Appels systèmes

L'appel système permet de passer de l'espace utilisateur à l'espace noyau en exécutant du code appartenant au système d'exploitation. Ce processus nécessite un basculement du processeur en mode noyau via une interruption logicielle, telle que l'instruction `int n` sur les processeurs Intel, où n désigne le numéro de l'interruption. Bien que ces interruptions soient principalement réservées au processeur pour gérer les erreurs d'exécution, elles peuvent également être déclenchées par le programme. Les interruptions les plus couramment utilisées pour les appels systèmes incluent `sysenter` ou `int 0x80` sur Intel, `svc` (ou `swi`) sur ARM, et `syscall` sur MIPS.

- Un appel système implique une interruption logiciel réservée.
- Passage d'arguments via les registres et/ou la pile
- Invocation d'un stub dans l'espace utilisateur
- Convention d'appel définie par l'Application Binary Interface (ABI)



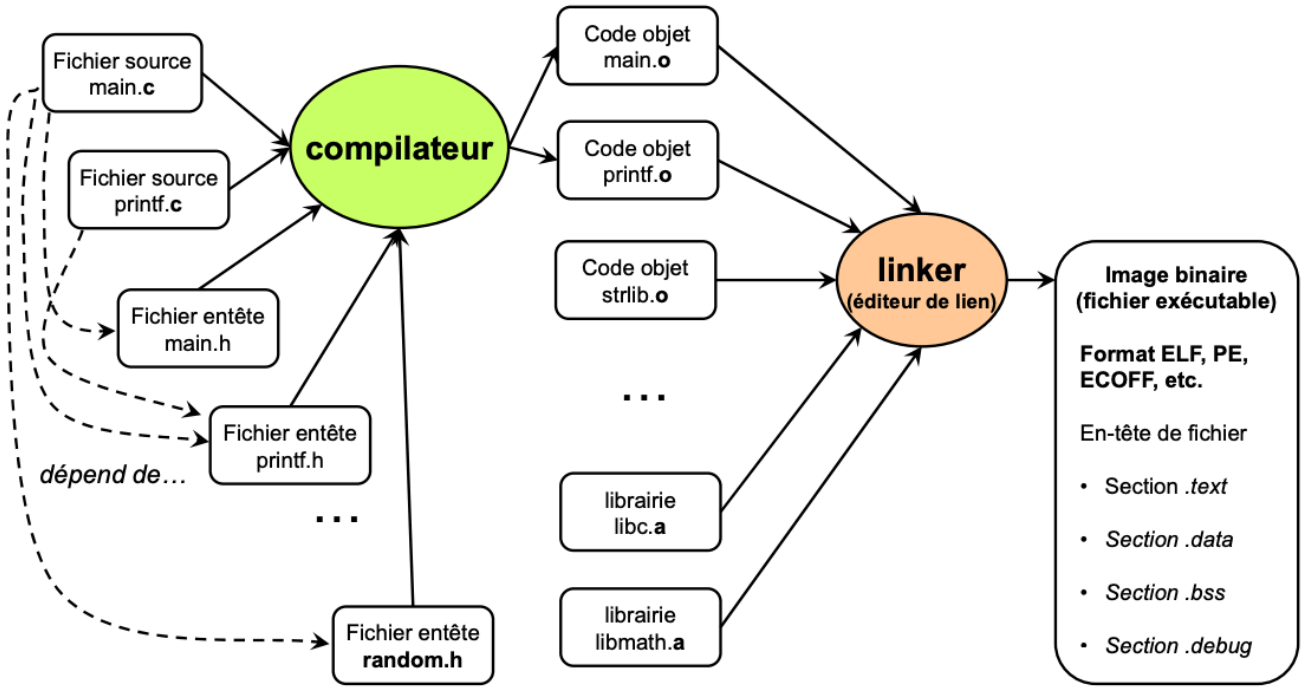
1.1. POSIX

POSIX (Portable Operating System Interface) est une norme qui standardise les appels système sous forme de fonctions C, facilitant la portabilité des applications entre différents systèmes d'exploitation. Bien que Windows ait ses propres appels système via la librairie Win32, qui est documentée dans MSDN, il supporte également POSIX, mais dans une moindre mesure. Sous Linux, les appels système sont intégrés dans la bibliothèque libc de l'espace utilisateur, où se trouvent les implémentations des stubs des appels système.

D'autres normes relatives aux APIs des appels système incluent ANSI, SVr4, X/OPEN et BSD. En général, les appels système sont coûteux en raison des changements d'état du processeur nécessaires pour passer de l'espace utilisateur à l'espace noyau. Pour cette raison, il est souvent préférable d'utiliser des fonctions de plus haut niveau qui minimisent les appels système, par exemple en gérant des buffers dans l'espace utilisateur lors d'opérations comme `read()`. Ces fonctions sont regroupées dans diverses librairies de l'espace utilisateur.

2. Construction d'une image binaire

La chaîne de compilation convertit le code source en une image binaire exécutable. Cette image est composée de plusieurs sections : `.text` pour les instructions, `.data` pour les variables initialisées, `.bss` pour les variables non initialisées, et d'autres. L'éditeur de liens (linker) résout les dépendances entre le code et les bibliothèques, créant ainsi une image binaire complète.

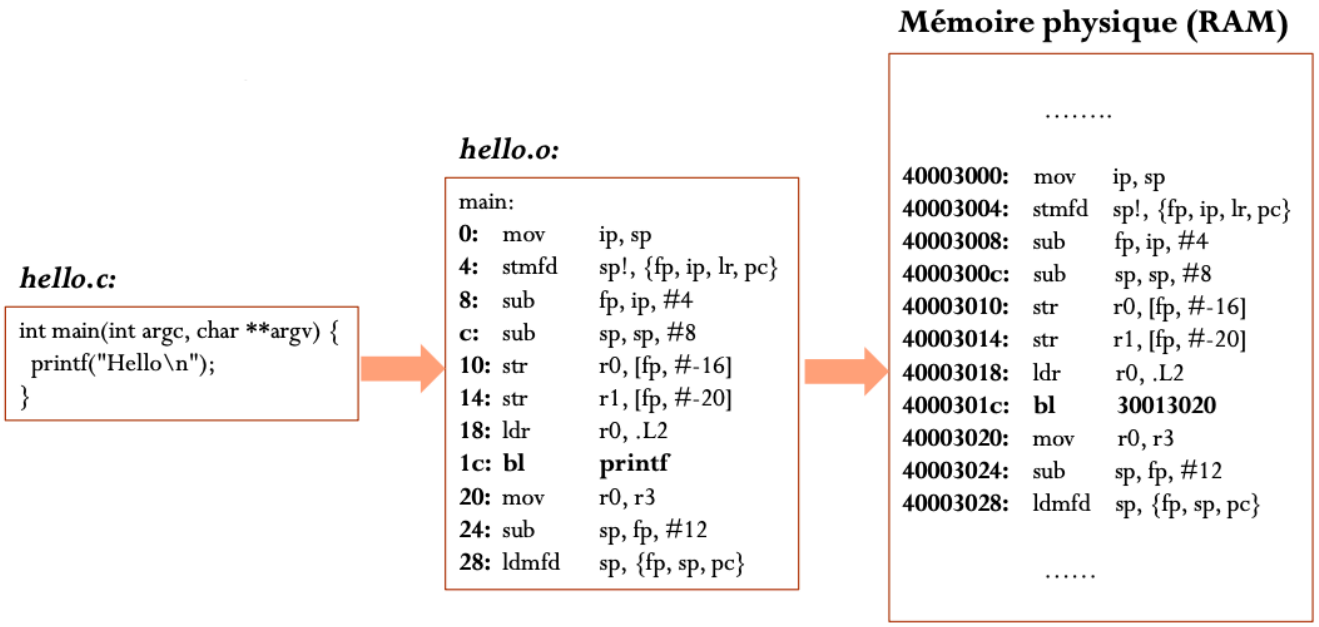


La chaîne de compilation permet de construire une application sous forme d'un fichier binaire exécutable, appelé aussi image binaire. L'image binaire est donc le résultat d'une compilation et d'une opération de linkage ou édition des liens.

Le compilateur traduit le code source (C, C++, Java, assembleur, etc.) en un fichier binaire intermédiaire, appelé fichier objet (ou code objet).

2.1. Types d'adresses

- 3 types d'adresse
  - Adresses relatives
  - Adresses absolues
  - Adresses symboliques



2.1.1. Adresses relatives

Elles sont basées sur un offset (positif ou négatif) et permettent au processeur de déterminer l'adresse effective par rapport à la position courante. Le compilateur ne connaît pas l'emplacement exact des instructions ou des données dans la mémoire physique.

2.1.2. Adresses absolues

Ces adresses indiquent un emplacement final dans la mémoire et peuvent être générées par le compilateur dans le code binaire. Elles ne changent pas lors du chargement du code en mémoire.

2.1.3. Adresses symboliques

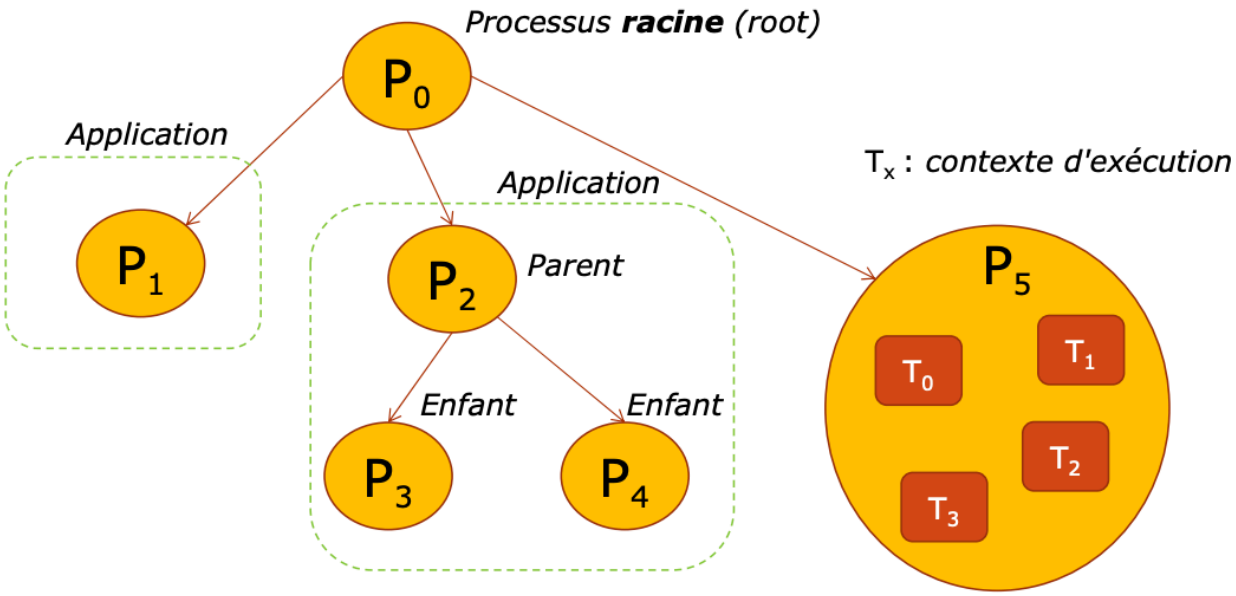
Utilisées par le compilateur lorsque l'emplacement d'une instruction ou d'une donnée n'est pas connu, généralement parce qu'elle réside dans un autre fichier ou en mémoire physique. Ces adresses sont représentées par des noms symboliques et sont résolues ultérieurement par l'éditeur de lien (linker).

3. Processus

Un processus est chargé en mémoire par le noyau et commence son exécution, avec la fonction main() comme point d'entrée pour un programmeur en C, bien qu'il ne s'agisse pas des premières instructions réellement exécutées. Un code de démarrage (C runtime) est ajouté à l'exécutable par l'éditeur de lien.

Pour qu'un processus soit chargé, il doit exister un fichier exécutable sur un support de stockage, représentant l'image binaire du processus. Chaque processus a un contexte d'exécution, déterminé par ses instructions et nécessitant des structures d'état, comme une pile, et un contexte mémoire, qui inclut un espace mémoire contenant toutes les instructions et le code.

Dans un système multitâche, plusieurs processus peuvent être chargés en mémoire et exécutés à tour de rôle sur le processeur, souvent pour des durées équivalentes. Ce partage du temps processeur crée l'illusion que les processus s'exécutent simultanément pour l'utilisateur.



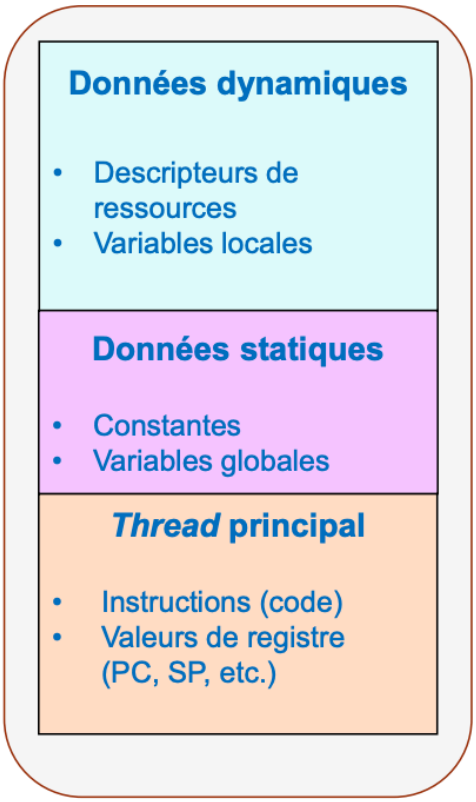
Après la phase de démarrage du noyau, ce dernier lance le premier processus, généralement un processus spécial de type init ou idle, qui peut ensuite démarrer d'autres processus, comme un shell. Dans certains systèmes, le shell peut être le premier processus.

Une application peut comprendre un ou plusieurs processus, et plusieurs contextes d'exécution peuvent également exister au sein d'un même processus. La création d'un nouveau processus se fait toujours à partir d'un processus existant, établissant une relation parent-enfant. Lorsqu'un processus se termine, il doit informer son parent, qui peut alors récupérer l'état et libérer les ressources allouées par le processus enfant.

Si un processus parent se termine prématurément alors qu'il a encore des enfants, cela crée des processus orphelins. Le noyau doit alors "rattacher" ces processus orphelins à un autre processus parent de niveau supérieur, comme le processus racine (init ou idle). Cela est crucial, car lorsque le processus orphelin termine son exécution, son nouveau parent doit libérer les ressources qui lui étaient allouées.

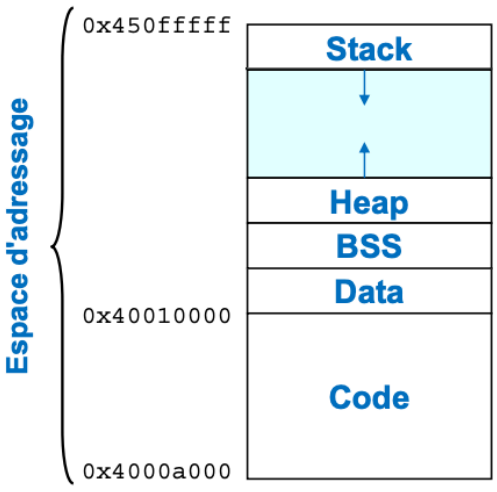
- **Image binaire**
  - Fichier exécutable pouvant être chargé en mémoire.
- **Contexte mémoire**
  - Variables globales & locales
  - Constantes
- **Contexte d'exécution**
  - Code (instructions)
  - Valeurs de registres
- **Ressources logiques**
  - Descripteurs de fichiers, IPCs, etc.

Mémoire (RAM)



Un processus est chargé en mémoire à partir de son image binaire, c'est-à-dire du fichier exécutable structuré, qui contient des sections pour le code, les données initialisées et non initialisées, ainsi que des symboles permettant le débogage. La plupart de ces sections sont chargées dans la RAM au sein du contexte mémoire du processus.

Les fichiers exécutables respectent des formats standards : sous Windows, on trouve le COFF, ECOFF, ou PE ; sous Linux, le format ELF est utilisé, remplaçant "a.out" ; et sous MacOS X, le format utilisé est Mach-O. Ces fichiers exécutables contiennent généralement trois types d'adresses : relatives, absolues, et symboliques.



- *Stack*
  - Variables locales d'une fonction
  - Arguments d'une fonction
  - Adresse de retour
  - Local au contexte d'exécution
- *Heap*
  - Allocation dynamique (*malloc, new, etc.*)
  - Global au(x) contexte(s) d'exécution
- *BSS*
  - *Block Started by Symbol*
  - Variables statiques **non**-initialisées
- *Data*
  - Variables statiques initialisées
  - Constantes