Programmation Système

Les processus

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2020/2021

Programmes et Processus

Définitions

- Un programme est un fichier exécutable qui réside dans le système de fichiers
- Un processus est une instance d'un programme en exécution

Plusieurs instances d'un même programme peuvent s'exécuter en même temps.

A chaque processus est associé :

- un identifiant de processus (**Process ID** ou **pid**)
- un espace d'adressage en mémoire vive pour stocker la pile, les données, ...

Un programme

Un programme contient toutes les informations nécessaires à la création du processus qui découle de son exécution :

Définitions

- Un format binaire ELF (historiquement, a.out, COFF, . . .)
- Des instructions compréhensible par la machine (qui découle de la compilation)
- Le point d'entrée de la première instruction
- Des données
- Des informations pour le debugger
- Des informations sur les librairies partagées

ELF

ELF (Executable and Linkable Format, format exécutable et liable) est un format de fichier binaire utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions).

Il a été développé par l'USL (Unix System Laboratories) pour remplacer les anciens formats a.out et COFF qui avaient atteint leurs limites. Aujourd'hui, ce format est utilisé dans la plupart des systèmes d'exploitation de type Unix (GNU/Linux, Solaris, IRIX, System V, BSD), à l'exception de Mac OS X.

Un programme du point de vue du noyau

Un processus est une entité abstraite connue du noyau qui lui alloue des ressources pour son exécution.

Du point de vue du noyau un processus consiste en :

- En mode utilisateur (user-land)
  - Un portion d'espace mémoire
  - le code du programme
  - les variables auxquels le code a accès
- En mode noyau (kernel-land)
  - Une structure de données de l'état du processus
  - Table des fichiers ouverts
  - Table de la mémoire allouée
  - Répertoire courant
  - Priorité
  - Gestionnaire d'événements (signaux, fin de programme, ...)
  - Les limites du processus

ID de processus

Définition

Chaque processus a un identifiant de processus (**PID**) qui l'identifie de façon unique sur le système. Cet identifiant est un entier positif non nul.

- Le PID est utilisé en interne par le noyau identifié le processus.
- Le PID est également utilisé dans l'espace utilisateur (kill, ps, nice)

getpid

```c
#include <unistd.h>

pid_t getpid(void);

// renvoie l'identifiant du processus appelant.
// Ceci est souvent utilisé par des routines qui
// créent des fichiers temporaires uniques.
```

- *pid_t* est un type abstrait
- On peut transtyper un *pid_t* en entier
- Sous Linux, la valeur maximale est définir par */proc/sys/kernel/pid_max* (modifiable)

getpid (Python)

```
import os

pid = os.getpid()

type(pid)
<type 'int'>

# renvoie l'identifiant du processus appelant.
```

Exécution du main

Un programme C commence par convention par l'exécution de la fonction *main* :

```
int main(int argc, char **argv);
```

- **argc**: nombre d'arguments du programme
- **argv**: tableau de pointeurs sur les arguments du programme

L'invocateur du programme (ex: le shell) exécute l'appel système **execvp** et utilise les arguments passé au programme comme paramètres.

Exemples de programmes avec argv

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

int main (int argc, char **argv) {
    for(int i=0; i<argc; i++) {
        printf("argv[%d]=%s\n", i, argv[i]);
    }
    exit(EXIT_SUCCESS);
}
```

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

int main (int argc, char **argv) {
    for(int i = 0; argv[i] != NULL; i++) {
        printf("argv[%d]=%s\n", i, argv[i]);
    }
    exit(EXIT_SUCCESS);
}
```

Exemples de programmes avec argv (Python)

```python
#!/usr/bin/python3

import sys

def main():
    i = 0
    for arg in sys.argv:
        print("argv[%d]=%s" % (i, arg))
        i += 1

main()
```

```python
#!/usr/bin/python3

import sys

def main():
    for i,arg in enumerate(sys.argv):
        print("argv[%d]=%s" % (i, arg))

main()
```

Terminaison du main

Il y a plusieurs façons de terminer un programme :

- Terminaison normale
  - retour du main
  - exit() (ou _exit() - syscall)
  - Nettoyage de toutes les ressources de la librairie standard
    - appel de *close* sur tous les FD ouverts
    - appel des gestionnaires de sorties (exit handlers)
- Terminaison anormale
  - réception d'un signal (ex: kill)

Par convention sous UNIX, un programme qui termine avec un code de retour égal à 0 a été exécuté avec succès. Dans le cas contraire, le programme a échoué.

```
#include <stdlib.h>
exit(EXIT_SUCCESS); // ou exit(EXIT_FAILURE)
```

atexit

```c
#include <unistd.h>

int atexit(void (*function)(void));

// La  fonction  atexit() enregistre la fonction donnée
// pour que celle-ci soit automatiquement appelée lorsque le
// programme se termine normalement avec exit(3) ou lors de la
// fin de la fonction main. Les fonctions ainsi enregistrées sont
// invoquées dans l'ordre inverse de leur enregistrement ; aucun
// argument n'est transmis
//
// atexit() renvoie 0 en cas de succès et une valeur non nulle
// en cas d'échec
```

POSIX.1-2001 exige que l'implémentation permette d'enregistrer au moins ATEXIT_MAX (32) de ces fonctions. La limite actuelle de l'implémentation peut être obtenue avec sysconf(3).

```
> getconf ATEXIT_MAX
2147483647
```

Exemple de programme avec atexit

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

void trigger_exit(void) {
    printf("Invoking exit handler!\n");
}

int main (int argc, char **argv) {
    printf("Invoking main()...\n");
    if (atexit(trigger_exit) == 0) {
        printf("Exit handler successfully registered\n");
    }
    else {
        printf("Failed to register exit handler\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

```
>./atexit
Invoking main()...
Exit handler successfully registered
Invoking exit handler!
```

Exemple de programme avec atexit (Python)

```
#!/usr/bin/python3

import sys
import os
import atexit

def trigger_exit():
    print("Invoking exit handler!")

def main():
    print("Invoking main()...")
    atexit.register(trigger_exit)
    print("Before exiting()...")
    sys.exit(os.EX_OK);

main()
```

```
> python3 atexit.py
Invoking main()...
Before exiting()...
Invoking exit handler!
```

Exemple de programme avec atexit (Python) v2

```
#!/usr/bin/python3

import sys
import os
import atexit

@atexit.register
def trigger_exit():
    print("Invoking exit handler!")

def main():
    print("Invoking main()...")
    print("Before exiting()...")
    sys.exit(os.EX_OK);

main()
```

```
> python3 atexit.py
Invoking main()...
Before exiting()...
Invoking exit handler!
```

Anatomie d'un processus en mémoire

La mémoire allouée a chaque processus est composée de parties appelées **segments**.

Ces segments sont les suivants :

- Le segment texte (**text segment**) contient les instructions en language machine du programme (ie: le binaire du programme au format ELF). Ce segment est en lecture seule.
- Le segment des données initialisés (**initialized data segment**). Il contient les variables globales et statiques qui sont explicitement initialisés. Les valeurs de ces variables sont lues depuis le fichier exécutable quand il est chargé en mémoire.

Anatomie d'un processus en mémoire (suite)

- Le segment des données non initialisés (**uninitialized data segment** ou **bss segment**). Il contient les variables globales et statiques qui n'ont pas explicitement été initialisés. L'espace mémoire requit par ces variables est alloué par le noyau à l'exécution du programme.
- La pile (**stack**) est un segment dynamique qui s'étend et se réduit et qui contient des 'blocs de pile' **stack frames**. Une *stack frame* par fonction appelée. Une **frame** contient les variables locales de la fonctions et la valeur de retour.
- Le tas (**heap**) est une zone où la mémoire (pour les variables) peut être dynamiquement allouée à l'exécution du programme.

Anatomie d'un processus en mémoire (schéma)

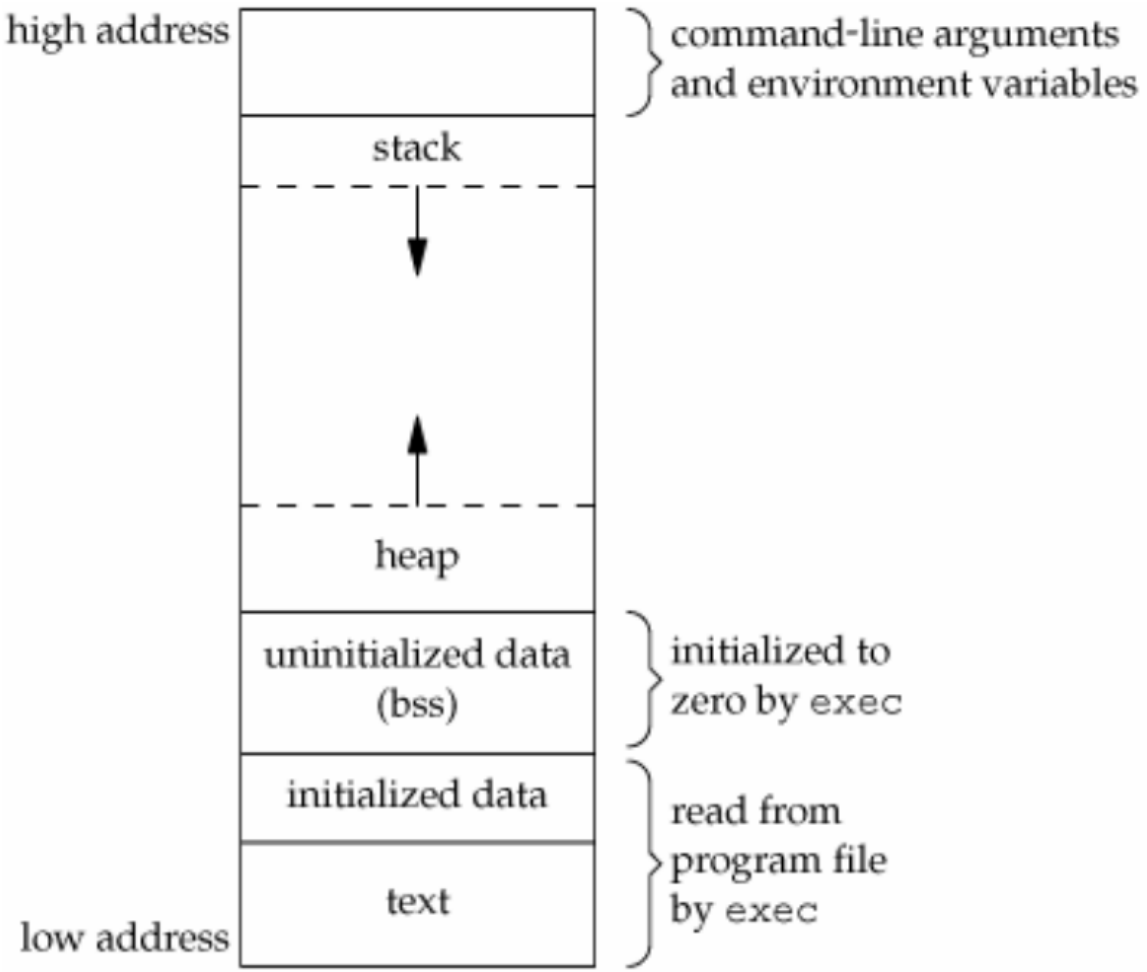

Figure: **APUE** 7.6

Anatomie d'un processus en mémoire (fin)

- L'espace mémoire d'un processus est virtuel. Un processus n'adresse pas directement la mémoire physique.
- Les processus sont isolés des autres et du noyau.
- Le noyau maintient un tableau de pages mémoire qui adresse des zones de la mémoire physique.
- Un processus peut adresser plus de mémoire que le système n'en a de disponible.

Hiérarchie des processus

Sous UNIX, les processus définis sous forme arborescente :

- Chaque processus (autre que init, PID=1) a un unique processus parent
- Chaque processus peut avoir 0 ou plus processus fils
- Exemples : **ps faux**, **pstree**, etc.

Création de processus

```c
#include <unistd.h>

pid_t fork(void);

// En cas de succès, le PID du fils est renvoyé au parent,
// et 0 est renvoyé au fils. En cas d'échec -1 est renvoyé
// au parent, aucun processus fils n'est créé, et errno
// contient le code d'erreur.
```

- Le processus fils commence son exécution juste après le fork
- Le processus parent continue son exécution après le fork
- On distingue le processus père du processus fils par la valeur de retour de *fork*
- Le processus fils est une copie de son parent
- Le segment text est partagé entre le père et le fils
- La copie pure n'est pas nécessaire, seul les zones modifiées sont vraiment copiées et occupe de la mémoire supplémentaire

Exemple de programme avec fork

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

int main (int argc, char **argv) {
    pid_t pid;

    printf("Starting process with PID=%d\n", getpid());
    pid = fork();
    if (pid < 0) {
        perror("Unable to fork");
    }
    else if (pid == 0) {
        printf("Starting child with PID=%d (my parent PID=%d)\n", getpi
    }
    else {
        printf("Still in process with PID=%d\n", getpid());
    }
    printf("Finishing process with PID=%d\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Exécution du programme avec fork

```
> gcc -Wall fork.c -o fork
> ./fork
Starting process with PID=14122
Still in process with PID=14122
Finishing process with PID=14122
Starting child with PID=14123 (my parent PID=14122)
Finishing process with PID=14123
```

Exemple de programme avec fork (Python)

```
#!/usr/bin/python3

import os
import sys

def main():
    print("Starting process with PID=%d" % (os.getpid()))
    try:
        pid = os.fork()
    except OSError as e:
        print("Unable to fork")

    if pid == 0:
        print("Starting child with PID=%d (my parent PID=%d)" % (os.get
    else:
        print("Still in process with PID=%d" % (os.getpid()))
    print("Finishing process with PID=%d" % (os.getpid()))

    sys.exit(os.EX_OK);

main()
```

Exécution du programme avec fork (Python)

```
> python3 fork.py
Starting process with PID=26824
Still in process with PID=26824
Finishing process with PID=26824
Starting child with PID=26825 (my parent PID=26824)
Finishing process with PID=26825
```

Terminaison d'un processus

- Quelque soit la façon dont il termine, le noyau nettoie les ressources allouées (fermeture des descripteurs de fichiers, libération de la mémoire. etc.)
- Quelques soit la façon dont il termine le noyau stocke le status de terminaison du processus jusqu'à ce que le processus parent en prenne connaissance.
- Si le processus parent se termine avant le processus fils, le processus fils devient le fils du processus de PID $= 1$

Attendre la fin d'un processus

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status); // waitpid(-1, &status, 0);

// en cas de réussite, l'identifiant du processus fils
// terminé (ou changé) est renvoyé ; en cas d'erreur,
// la valeur de retour est -1
```

- L'appel bloque jusqu'à ce qu'un processus fils termine (sauf pour certains *options*)
- Retourne une erreur s'il n'y pas de processus fils
- Si un processus fils s'est terminé, la variable *status* est remplie avec le code de retour

Exemple de programme avec waitpid

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

int main (int argc, char **argv) {
    pid_t pid;
    int status;

    printf("Starting parent process with PID=%d\n", getpid());
    for (int i=1; i<=3; i++) {
        pid = fork();
        if (pid < 0) {
            perror("Unable to fork");
        }
        else if (pid == 0) {
            printf("Starting child process with PID=%d (my parent PID=%d)\n", getpid(), getppid());
            sleep(i*3);
            printf("Finishing child process with PID=%d after %d sec\n", getpid(), i*3);
            exit(EXIT_SUCCESS);
        }
    }
    while((pid = waitpid(-1, &status, WNOHANG)) >= 0) {
        if (pid > 0) {
            printf("Process with PID=%d has terminated with status=%d\n", pid, status);
        }
    }
    printf("Finishing parent process with PID=%d\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Exécution du programme avec waitpid

```
> gcc -Wall waitpid.c -o waitpid
> ./waitpid
Starting parent process with PID=18990
Starting child process with PID=18991 (my parent PID=18990)
Starting child process with PID=18992 (my parent PID=18990)
Starting child process with PID=18993 (my parent PID=18990)
Finishing child process with PID=18991 after 3 sec
Process with PID=18991 has terminated with status=0
Finishing child process with PID=18992 after 6 sec
Process with PID=18992 has terminated with status=0
Finishing child process with PID=18993 after 9 sec
Process with PID=18993 has terminated with status=0
Finishing parent process with PID=18990
```

Exemple de programme avec waitpid (Python)

```
#!/usr/bin/python3

import os, sys, time

def main():
    print("Starting parent process with PID=%d" % (os.getpid()))
    for i in range(1,4):
        try:
            pid = os.fork()
        except OSError as e:
            print("Unable to fork")

        if pid == 0:
            print("Starting child process with PID=%d (my parent PID=%d)" % (os.getpid(), os.getppid()))
            time.sleep(i*3)
            print("Finishing child process with PID=%d after %d sec" % (os.getpid(), i*3))
            sys.exit(os.EX_OK)

    cont = True
    while cont:
        try:
            pid, status = os.waitpid(-1, os.WNOHANG)
        except OSError as e:
            cont = False
        if pid > 0:
            print("Process with PID=%d has terminated with status=%d" % (pid, status))

    print("Finishing parent process with PID=%d" % (os.getpid()))
    sys.exit(os.EX_OK)

main()
```

Exécution du programme avec waitpid (Python)

```
> python3 waitpid.py
Starting parent process with PID=29429
Starting child process with PID=29430 (my parent PID=29429)
Starting child process with PID=29431 (my parent PID=29429)
Starting child process with PID=29432 (my parent PID=29429)
Finishing child process with PID=29430 after 3 sec
Process with PID=29430 has terminated with status=0
Finishing child process with PID=29431 after 6 sec
Process with PID=29431 has terminated with status=0
Finishing child process with PID=29432 after 9 sec
Process with PID=29432 has terminated with status=0
Process with PID=29432 has terminated with status=0
Finishing parent process with PID=29429
```

Zombie

Zombie

Un processus zombie est un processus qui a terminé son exécution mais dont le processus parent n'a pas encore pris connaissance du status de terminaison.

Un processus zombie consomme encore quelques ressources : le noyau stocke le statut de terminaison, maintient une entrée dans la table des processus, etc.

Ressources partagées

Après un fork la table des fichiers ouverts est partagée entre les processus parent et enfant.

Si pour un fichier donné (identifié par un descripteur), si l'offset est modifié (read, write, lseek) par un des processus, cela s'applique à l'autre.

Cela permet (entre autre) de faire collaborer (traitements en parallèle) plusieurs processus sur les mêmes données.

- utilisateur / group (réel, effectif), groupe de processus, id de session
- répertoire courant, variables d'environnement
- mémoire partagée, limites de ressources
- terminal de contrôle (tty), umask

Cas d'usage du fork

- Un programme qui se duplique pour éxécuter en parallèle des parties de lui même
  - un serveur web qui crée plusieurs processus pour recevoir et traiter plusieurs requêtes en même temps
  - un programme scientifique qui récupère par le réseau des données et dans le même temps commence à traiter ce qu'il a déjà reçu
- Un programme qui souhaite exécuter un autre programme pour en récupérer les résultats
  - un shell
  - Un éditeur de code qui lance la compilation du programme en cours d'édition et qui affiche le résultat

Exécution d'un autre programme

Sous unix, la création d'un nouveau processus et l'exécution d'un programme sont deux opérations distinctes.

L'exécution d'un nouveau processus ne crée pas un nouveau processus, par contre cela remplace le programme appelant par le nouveau en allant le chercher sur le système de fichiers.

A l'exécution, tous les segments (text, data, bss, heap et stack sont réinitialisés comme si le programme avait été exécuté normalement.

L'exécution du programme se fait grâce à l'appel système **execve**.

```
> man 2 execve
```

La famille exec

```c
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ..., (char *) NULL);
int execlp(const char *file, const char *arg, ..., (char *) NULL);
int execle(const char *path, const char *arg, ...,
        (char *) NULL, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);

// Une sortie des fonctions exec() n'intervient que si une erreur s'est
// produite. La valeur de retour est -1, et errno contient le code
// d'erreur.
```

```
> man 3 exec
```

La famille exec (Python)

```
import os

os.execl(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
```

Exercices

Il est important de gérer les ereurs en utilisant errno et la sortie erreur, on privilégiera la librairie standard : `man 2 stdio`.

- **Question 1.** Combien y'a t'il de processus créés avec le programme suivant ?

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

int main(int argc, char **argv) {
    fork();
    fork();
    fork();
    sleep(10);
    exit(EXIT_SUCCESS);
}
```

Dessinez l'arbre des processus et vérifiez vos résultats avec la commande `pstree` ou `ps fux`

Exercices (suite)

- **Question 2.** Écrire un programme en C qui exécute la commande qui est lui est passée en argument à l'aide d'un processus fils. Le processus parent attendra la fin de son processus fils avant de terminer lui même.

```
# Exemple d'appel
> ./prog ls -l /dev/null
```

- **Question 3.** Écrire un programme en C qui s'exécute lui même récursivement au maximum `n` fois avec `n` passé en paramètre de la ligne de commande du programme.
  On utilisera la fontion `atoi` (`man 3 atoi`) pour la conversion de chaînes de caractères en entier et la fonction `sprintf` (`man 3 sprintf` pour réaliser l'opération inverse.

Bibliographie

[**APUE**] Advanced Programming in the UNIX Environment.
W. Richard Stevens and Stephen A. Rago.
Addison-Wesley Professional, 2005.

[**TLPI**] The Linux Programming Interface.
Michael Kerisk.
No Starch Press, 2010.