Programmation Système

Cours 5: Appels Systèmes

Khaddouja ZELLAMA
Khaddouja.zellama@dauphine.psl.eu

Licence L2 Mathématiques et Informatique
Département MIDO

Problème

Un seul programme s'exécute à la fois sur le processeur:

▶ Quand une application s'exécute, **le système ne s'exécute pas**

Comment exécuter une application qui n'est pas fiable?
En particulier:

- Comment s'assurer que l'application va rendre la main?
- Comment s'assurer que l'application ne va pas exécuter d'opération interdite?

Appels systèmes

- Appel au SE (fonctions primitives).
- Le programme s'interrompt pour demander au SE d'accomplir pour lui une certaine tâche.
- Fonctions fournies par le SE.
- Le noyau utilise ces primitives qui sont pour lui la seule porte entre une programme et le « monde extérieur ».

Les appels systèmes de UNIX:

- Système de fichiers: create, open, close, write, read, …
- Contrôle des processus: exec, fork, wait, signal, kill, …
- Communication inter-processuss: pipe, …

mode noyau/mode utilisateur

Le processeur a deux modes d'exécution:

Mode noyau ou mode privilégié :
Le processeur peut exécuter n'importe quelle instruction

Mode utilisateur :
Le processeur ne peut exécuter qu'un sous ensemble limité d'instructions non critiques.

- Un programme qui s'exécute en mode noyau peut passer en mode utilisateur
- Un programme qui s'exécute en mode utilisateur ne peut pas passer en mode noyau

Interruptions

Une interruption est un évènement matériel qui peut être déclenché par

- Un périphérique horloge, évènement sur le disque ou la carte réseau . . .

▶ **interruption matérielle**

- Un programme en cours d'exécution calcul impossible, appel système

▶ **interruption logicielle**

Interruptions

Une interruption est un évènement matériel qui peut être déclenché par

- Un périphérique horloge, évènement sur le disque ou la carte réseau . . .

▶ **interruption matérielle**

- Un programme en cours d'exécution calcul impossible, appel système

▶ **interruption logicielle**

Lorsqu'une interruption est levée, le processeur

- interrompt l'exécution en cours
- passe en mode noyau
- donne la main au gestionnaire d'interruption

Le système installe le gestionnaire d'interruption au démarrage

Exécution protégée

Mécanisme des appels systèmes

- Une application s'exécute en mode utilisateur seulement
- Lorsqu'elle a besoin d'effectuer une opération critique
- Elle écrit le numéro de l'opération dans un registre
- Déclenche une interruption logicielle (appel système)
- Le système prend la main (gestionnaire d'interruption)
- Exécute (ou non) l'opération en mode privilégié
- Repasse en mode utilisateur
- Rend la main à l'application

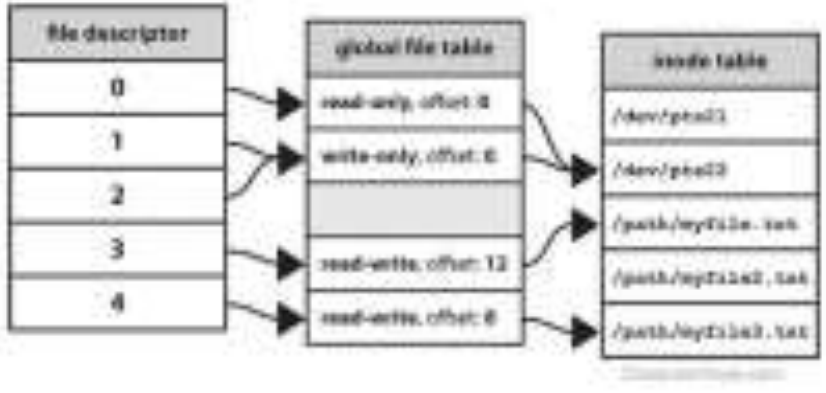
Exemple d'appel système: getpid(), read(), write(), fork(), exec()

open() et close()

```
int   open(const char *pathname, int flags);
int   open(const char *pathname, int flags, mode_t mode);
int   close(int fd);
```

- open(pathname, flags, [mode]):
  - Vérifie que le processus à accès au fichier
  - Créer un nouveau descripteur de fichier dans la table des fichier du processus
  - Initialise le curseur au début du fichier
  - Retourne le descripteur de fichier
- close(fd) ferme le fichier et libère l'entrée dans la table des fichiers

```
1 file1=open("/path/myfile.txt", O_RDWR);
2 file2=open("/path/myfile3.txt", O_RDWR);
3 ...
4 close(file1)
5 close(file12)
```



Flags de open

Au moins l'une de ces trois valeurs

- O_RDONLY, O_WRONLY ou O_RDWR
  - Doit être compatible avec les permissions du fichier

Autres options possibles

- O_CREAT : crée le fichier s'il n'existe pas (spécifier le mode)
- O_APPEND : curseur d'écriture positionné à la fin
- O_TRUC : tronque le fichier.
- . . . (voir man 2 open)

Combiner plusieurs Flags

- L'ensemble des Flags est représenté avec une valeur entière
- Les flags sont combinées avec un OU bit-a-bit : '|'

```
1 open("/tmp/test", O_WRONLY | O_CREAT, 0644);
```

Echec de l'ouverture

open retourne

- le numéro du descripteur de fichier en cas de succès
- -1 sinon.

Pas bien:

```
1 fd = open("/ce/chemin/n/existe/pas", O_RDONLY);
2 while(read(fd, buf, 1)){ // fd = -1 !!
3     ...
4 }
```

Mieux:

```
1 if(fd = open("/ce/chemin/n/existe/pas", O_RDONLY) < 0){
2   perror("Ouverture:");
3   exit(EXIT_FAILURE);
4 }
5 else{
6   while(read(fd, buf, 1)){ ... }
7 }
```

Echec de l'ouverture

open retourne

- le numéro du descripteur de fichier en cas de succès
- -1 sinon.

Pas bien:

```
1 fd = open("/ce/chemin/n/existe/pas", O_RDONLY);
2 while(read(fd, buf, 1)){ // fd = -1 !!
3     ...
4 }
```

Mieux:

```
1 if(fd = open("/ce/chemin/n/existe/pas", O_RDONLY) < 0){
2   perror("Ouverture:");
3   exit(EXIT_FAILURE);
4 }
5 else{
6   while(read(fd, buf, 1)){ ... }
7 }
```

Attention: toujours vérifier que les appels système ont fonctionné!!

▸ En cas d'échec d'un appel système, **errno** est définie, et **perror** permet d'afficher l'erreur associée

read()

```
ssize_t read(int fd, void *buf,          size_t count);
```

read()

- lit les prochains n octets (au plus count) dans fd
- stocke les données dans la variable buf
- retourne le nombre d'octets n lus (au plus count)

Curseur:

- A l'ouverture le curseur est positionné au début du fichier
- Chaque lecture fait avancer le curseur de n (ou moins)
- Possibilité de manipuler la position du curseur avec lseek

Lire au moins n octets

Mauvaise solution:

```
1 #define SIZE = 16
2 char buf[SIZE];
3 open("/tmp/fichier", O_RDONLY);
4 rcount = read(fd, buf, SIZE - 1);
5 buf[rcount] = '\0';
6 printf("Chaine lue: '%s'\n", buf);
```

Bien?

Lire au moins n octets

Mauvaise solution:

```
1 #define SIZE = 16
2 char buf[SIZE];
3 open("/tmp/fichier", O_RDONLY);
4 rcount = read(fd, buf, SIZE - 1);
5 buf[rcount] = '\0';
6 printf("Chaine lue: '%s'\n", buf);
```

Bien?

- read lit au plus SIZE octets!
- si read échoue rcount == -1!

Lire au moins n octets

Solution 1: un octet par un octet

```
1 int i;
2 char buf[SIZE];
3 for(i = 0; i < SIZE - 1; i++){
4    if(read(fd, buf+i; 1) < 0) exit(EXIT_FAILURE);
5 }
6 buf[i] = '\0';
7 printf("Chaine lue: '%s'\n", buf);
```

Lire au moins n octets

Solution 1: un octet par un octet

```
1 int i;
2 char buf[SIZE];
3 for(i = 0; i < SIZE - 1; i++){
4    if(read(fd, buf+i; 1) < 0) exit(EXIT_FAILURE);
5 }
6 buf[i] = '\0';
7 printf("Chaine lue: '%s'\n", buf);
```

Solution 2: par blocs (moins d'appels système)

```
1   char buf[SIZE];
2   int pos = 0;
3   int rcount = 0;
4   do {
5     rcount = read(fd, buf + pos, SIZE - pos - 1);
6     if(rcount < 0) exit(EXIT_FAILURE);
7     else pos += rcount;
8   }while(rcount > 0)
9   buf[pos] = '\0';
```

read/write vs. fread/fwrite

open/read/write/close:

- Appels système
- Manipule des descripteurs de fichiers (type entier)
- Chaque appel nécessite de basculer en mode noyau
- Relativement lent

fopen/fread/fwrite/fclose

- Primitive de la librairie standard C
- Manipule des FILE *
- Lectures/écritures accumulées dans une mémoire tampon
- Plus rapide

read/write vs. fread/fwrite

open/read/write/close:

- Appels système
- Manipule des descripteurs de fichiers (type entier)
- Chaque appel nécessite de basculer en mode noyau
- Relativement lent

fopen/fread/fwrite/fclose

- Primitive de la librairie standard C
- Manipule des FILE *
- Lectures/écritures accumulées dans une mémoire tampon
- Plus rapide

- fdopen permet d'obtenir un FILE * à partir d'un descripteur de fichier ouvert
- fileno permet d'obtenir le descripteur correspondant au FILE *.

Dans le cadre de ce cours, on utilisera plutot read/write.

Duplication de descripteurs de fichiers avec dup

int dup(int oldfd);
int dup2(int oldfd, int newfd);

dup(): duplique un descripteur de fichier

```
1 int fd = open("/tmp/test", ...);
2 dprintf(fd, "J'écris dans le fichier /tmp/test");
3
4 int fd2 = dup(fd);
5 dprintf(fd2, "J'écris aussi dans le fichier /tmp/test");
```

Redirection de sortie standard avec dup/dup2

Avec dup

```
1 int fd = open("/tmp/test", ...);
2 close(1); // on ferme la sortie standard (le descripteur 1 est disponible)
3 dup(fd); // le descripteur 1 pointe vers le fichier "/tmp/test"
4 printf("J'écris dans le fichier /tmp/test avec printf!\n");
```

Avec dup2 (plus propre)

```
1 int fd = open("/tmp/test", ...);
2 dup2(fd, 1); // plus propre
3 printf("J'écris dans le fichier /tmp/test avec printf!\n");
```

fork()

pid_t fork(void);

fork()

- Crée un nouveau processus en clonant le processus parent
- Retourne 0 au processus fils, le pid du fils au processus parent
- Les deux processus reprennent l'exécution après à la suite du programme

fork()

pid_t fork(void);

fork()

- Crée un nouveau processus en clonant le processus parent
- Retourne 0 au processus fils, le pid du fils au processus parent
- Les deux processus reprennent l'exécution après à la suite du programme

```
1 pid_t pid = fork();
2 if(pid == 0){
3     printf("Je suis la copie\n");
4 }
5 else{
6     printf("je suis l'original\n");
7 }
```

```
1 $ gcc forktest.c
2 $./a.out
3 Je suis la copie
4 Je suis l'original
```

Exemple

testfork.c

```
1  int main(){
2   printf("Pid du processus parent %d\n",
          getpid());
3
4    pid_t pid = fork();
5    if (pid == 0)
6      printf("je suis le processus %d (fils
           )\n", getpid()); }
7    else {
8      printf("je suis le processus %d (
           parent)\n", getpid());
9      wait(NULL);
10   }
11 }
```

```
1  $ ./testfork
2  Pid du processus parent 12425
3  je suis le processus 12425 (parent)
4  je suis le processus 12426 (fils)
5  $
```

wait/waitpid

wait(int *wstatus) waitpid(pid t pid, int *wstatus, int options

- Attend qu'un processus fils termine (crée avec fork)
- Le code de retour du processus est stocké dans wstatus
- Retourne le pid du fils qui a terminé

```
1  if(fork() == 0){ // processus fils
2      printf("Salut, je suis le processus %d\n", getpid());
3      sleep(10);
4      printf("J'ai fini!\n");
5      exit(EXIT_SUCCESS);
6  }
7  else{ // processus parent
8      int r;
9      pid_t p = wait(&r);
10     printf("Le processus fils %d a terminé avec le code %d\n", p, WEXITSTATUS(r));
11 }
```

```
1  $ ./a.out
2  Salut, je suis le processus 147497
3  J'ai fini!
4  Le processus fils 147497 a terminé avec le code 0
```

execvp()

int execvp(const char *file, char *const argv[]);

- Remplace le code du programme en cours d'exécution par un autre programme
- initialise argv et argc
- initialise les variables d'environnement
- reprend l'exécution du programme au début de la fonction main

```
1  int main(){
2      printf("Je suis sur le point de me transformer en ls\n");
3      char *bin = "ls";
4      char *args[] = { bin, "-la", "exec.c", NULL};
5      execvp(bin, args);
6      printf("blablablabla\n"); // ?
7  }
```

execvp()

int execvp(const char *file, char *const argv[]);

- Remplace le code du programme en cours d'exécution par un autre programme
- initialise argv et argc
- initialise les variables d'environnement
- reprend l'exécution du programme au début de la fonction main

```c
int main(){
    printf("Je suis sur le point de me transformer en ls\n");
    char *bin = "ls";
    char *args[] = { bin, "-la", "exec.c", NULL};
    execvp(bin, args);
    printf("blablablabla\n"); // ?
}
```

```
$ ./a.out
Je suis sur le point de me transformer en ls
-rw-r--r-- 1 bnegreve bnegreve 391 Mar 8 09:21 exec.c
$
```

Autre exemple

printargv.c

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

testexec.c

```c
int main(int argc, char *argv[]){
  printf("testexec démarre\n");
  char *bin = "printargv";
  char *args[] = { bin, "arg1",
                   "arg2", NULL };
  char *env[] = {};
  execve(bin, args, env);
  printf("testexec termine\n");
}
```

```
$ gcc printargv.c -o printargv
$ gcc testexec.c -o testexec
$ ./testexec
testexec démarre
printargv démarre
argv[0]=printargv
argv[1]=arg1
argv[2]=arg2
printargv termine
```

fork + exec

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

int main(){
  pid_t pid = fork();
  if (pid == 0){
    printf("je suis le fils\n");
    char *bin = "/bin/ls";
    char *args[] = { bin, "-la", "
          forkexec.c"};
    char *env[] = {};
    execve(bin, args , env);
  }
  else {
    printf("je suis le parent\n");
    wait(NULL);
    printf("le processus fils àterminé\n"
          );
  }
}
```

```
$ ./a.out
je suis le parent
je suis le fils
-rw-r--r-- 1 bnegreve bnegreve 391
      Mar 8 09:21 forkexec.c
le processus fils a termine
```

Pseudocode bash

```
1  while(true){
2    afficher_prompt("$ "); // avec write
3    lire_commande; // avec read
4    bin = extraire_binaire(cmd);
5    argv = extraire_arguments(cmd);
6    pid = cloner_processus(); // avec fork
7    rediriger_ES(); //dup, dup2
8    executer_programme(pid, bin, argv); //avec execve
9    if(premier_plan)
10     attendre_terminaison_processus(pid); //avec wait
11 }
```