Utilisation des systèmes UNIX

Benjamin Negrevergne

PSL University – Paris Dauphine – Équipes *MILES*

Problème

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

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

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?

Le road trip d'Alice et Bob

- A et B partent en road trip

- Quand A conduit B dort, quand B conduit A dort
  - ▶ A et B se relaient pour prendre le contrôle de la voiture

Voiture = **Processeur**

Le road trip d'Alice et Bob

- A et B partent en road trip
- Quand A conduit B dort, quand B conduit A dort
  - ▶ A et B se relaient pour prendre le contrôle de la voiture

  Voiture = **Processeur**
- A est fiable, connaît la route et ne s'endort pas au volant

  A = **Système**

Le road trip d'Alice et Bob

- A et B partent en road trip
- Quand A conduit B dort, quand B conduit A dort
  - ▶ A et B se relaient pour prendre le contrôle de la voiture

  Voiture = **Processeur**
- A est fiable, connaît la route et ne s'endort pas au volant

  A = **Système**
- B n'est pas fiable, il/elle conduit mal, se trompe, s'endort au volant

  B = **Application**

Le road trip d'Alice et Bob

- A et B partent en road trip
- Quand A conduit B dort, quand B conduit A dort
  - ▶ A et B se relaient pour prendre le contrôle de la voiture

  Voiture = **Processeur**
- A est fiable, connaît la route et ne s'endort pas au volant

  A = **Système**
- B n'est pas fiable, il/elle conduit mal, se trompe, s'endort au volant

  B = **Application**

Le road trip d'Alice et Bob

- A et B partent en road trip
- Quand A conduit B dort, quand B conduit A dort
  ▶ A et B se relaient pour prendre le contrôle de la voiture

  Voiture = **Processeur**
- A est fiable, connaît la route et ne s'endort pas au volant

  A = **Système**
- B n'est pas fiable, il/elle conduit mal, se trompe, s'endort au volant

  B = **Application**

Comment A s'assure d'arriver à destination ?

Solutions ?

- A prend le volant et ne le lache jamais
  - bien mais pas efficace

Solutions ?

- A prend le volant et ne le lache jamais
  - bien mais pas efficace

- A se réveille à intervalle régulier

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

- B réveille A avant chaque opération critique

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

- B réveille A avant chaque opération critique
  ▶ bien mais insuffisant: B n'est pas obligé de réveiller A

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

- B réveille A avant chaque opération critique
  ▶ bien mais insuffisant: B n'est pas obligé de réveiller A

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

- B réveille A avant chaque opération critique
  ▶ bien mais insuffisant: B n'est pas obligé de réveiller A

- A interdit/empêche B de faire des opérations critiques
  ▶ bien mais impossible si A et B sont des citoyens égaux

Solutions ?

- A prend le volant et ne le lache jamais
  ▶ bien mais pas efficace

- A se réveille à intervalle régulier
  ▶ bien mais insuffisant: n'empêche pas B de prendre une mauvaise décision entre deux réveils

- B réveille A avant chaque opération critique
  ▶ bien mais insuffisant: B n'est pas obligé de réveiller A

- A interdit/empêche B de faire des opérations critiques
  ▶ bien mais impossible si A et B sont des citoyens égaux

Solution:

- On introduit des privilèges en faveur de A.
- B est obligé de solliciter A lorsqu'il veut faire une opération critique

▶ Nécessite un support matériel

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.

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**

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

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()

getpid() #39

```
pid_t getpid(void);
```

```c
#include <unistd.h>
int main(){
  for(;;){
     getpid();
}
```

getpid() #39

```
pid_t getpid(void);
```

```c
#include <unistd.h>
int main(){
  for(;;){
     getpid();
}
```

Fonction `main()`

```
   0x000055555555513c <+0>:   push   %rbp
   0x000055555555513d <+1>:   mov    %rsp,%rbp
   0x0000555555555140 <+4>:   callq  0x555555555030 <getpid@plt>
   0x0000555555555145 <+9>:   jmp    0x555555555140 <main+4>
```

getpid() #39

```
pid_t getpid(void);
```

```c
#include <unistd.h>
int main(){
  for(;;){
     getpid();
}
```

Fonction main()

```
   0x000055555555513c <+0>:   push   %rbp
   0x000055555555513d <+1>:   mov    %rsp,%rbp
   0x0000555555555140 <+4>:   callq  0x555555555030 <getpid@plt>
   0x0000555555555145 <+9>:   jmp    0x555555555140 <main+4>
```

Fonction getpid()

```
   0x00007ffff7eb7770 <+0>:   mov    $0x27,%eax
   0x00007ffff7eb7775 <+5>:   syscall
   0x00007ffff7eb7777 <+7>:   retq
```

getpid() #39

```
pid_t getpid(void);
```

```c
#include <unistd.h>
int main(){
  for(;;){
     getpid();
}
```

Fonction `main()`

```
   0x000055555555513c <+0>:   push   %rbp
   0x000055555555513d <+1>:   mov    %rsp,%rbp
   0x0000555555555140 <+4>:   callq  0x555555555030 <getpid@plt>
   0x0000555555555145 <+9>:   jmp    0x555555555140 <main+4>
```

Fonction `getpid()`

```
   0x00007ffff7eb7770 <+0>:   mov    $0x27,%eax
   0x00007ffff7eb7775 <+5>:   syscall
   0x00007ffff7eb7777 <+7>:   retq
```

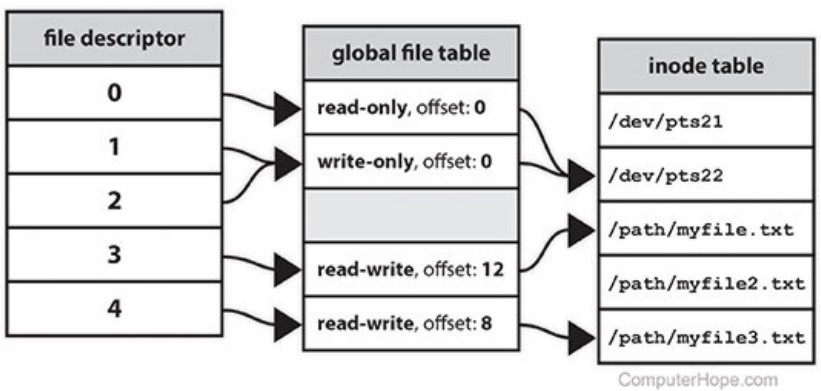
▶ `getpid()` ne fait rien d'autre que d'écrire 0x27 (=39) dans le registre %eax et d'exécuter l'instruction syscall

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", "rw");
2 file2=open("/path/myfile3.txt", "rw");
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 etre compatible avec les permissions du fichier

Flags de `open`

Au moins l'une de ces trois valeurs

- `O_RDONLY, O_WRONLY ou O_RDWR`
  - ▶ Doit etre 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)

Flags de open

Au moins l'une de ces trois valeurs

- `O_RDONLY`, `O_WRONLY` ou `O_RDWR`
  - ▶ Doit etre 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`)

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:

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

Bien?

Lire au moins `n` octets

Mauvaise solution:

```
1 char buf[SIZE];
2 open("/tmp/fichier", O_RDONLY);
3 rcount = read(fd, buf, SIZE);
4 buf[rcount] = '\0';
5 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';
```

⚠Mieux mais ne termine pas si le nombre de caractères écrits est insuffisant et que le fichier n'est pas fermé.

Bug fréquent

```
1 int fd = open("/tmp/fichier", O_RDONLY);
2 int rcount = 0;
3 char buf[SIZE];
4 for(int i = 0; i < 4; i++){
5    rcount = read(fd, buf, SIZE-1);
6    buf[rcount] = '\0';
7    printf("Chaine lue: %s\n", buf);
8 }
```

- Résultat (possible):

```
1 Chaine lue:   pGtU
2 Chaine lue:   pGtU
3 Chaine lue:   pGtU
4 Chaine lue:   pGtU
5 Chaine lue:   pGtU
6 Chaine lue:   pGtU
7 ...
```

Pourquoi?

Bug fréquent

```
1 int fd = open("/tmp/fichier", O_RDONLY);
2 int rcount = 0;
3 char buf[SIZE];
4 for(int i = 0; i < 4; i++){
5    rcount = read(fd, buf, SIZE-1);
6    buf[rcount] = '\0';
7    printf("Chaine lue: %s\n", buf);
8 }
```

- Résultat (possible):

```
1 Chaine lue:   pGtU
2 Chaine lue:   pGtU
3 Chaine lue:   pGtU
4 Chaine lue:   pGtU
5 Chaine lue:   pGtU
6 Chaine lue:   pGtU
7 ...
```

Pourquoi?

Bien tester le retour de `read`!!

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 fprintf(fd, "J'écris dans le fichier /tmp/test");
3
4 int fd2 = dup(fd);
5 fprintf(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 }
```

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  je suis le processus 12425 (
       parent)
3  je suis le processus 12426 (fils)
4  $
```

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 je suis le processus 12425 (
      parent)
3 je suis le processus 12426 (fils)
4 $
```

A votre avis, que fait `while(1)fork();`?

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

```c
if(fork() == 0){ // processus fils
    printf("Salut, je suis le processus %d\n", getpid());
    sleep(10);
    printf("J'ai fini!\n");
    exit(EXIT_SUCCESS);
}
else{ // processus parent
    int r;
    pid_t p = wait(&r);
    printf("Le processus fils %d a terminé avec le code %d\n", p, r);
}
```

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

execve()

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- 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 = "/bin/ls";
    char *args[] = { bin, "-la", "exec.c"};
    char *env[] = {};
    execve(bin, args , env);
    printf("blablablabla\n"); // ?
}
```

execve()

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- 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 = "/bin/ls";
    char *args[] = { bin, "-la", "exec.c"};
    char *env[] = {};
    execve(bin, args , env);
    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");
}
```

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

```
while(true){
  afficher_prompt("$ "); // avec write
  lire_commande; // avec read
  bin = extraire_binaire(cmd);
  argv = extraire_arguments(cmd);
  pid = cloner_processus(); // avec fork
  rediriger_ES(); //dup, dup2
  executer_programme(pid, bin, argv); //avec execve
  if(premier_plan)
    attendre_terminaison_processus(pid); //avec wait
}
```