MC-Info3-Système : Programmation Système

Semestre 3, année 2011-2012

Département d'informatique
IUT Bordeaux 1

septembre 2011

Objectif du cours : MC Info3-Système

A l'issue des cours d'informatique jusqu'à présent vous savez :

- Programmer en langage C.
- Utiliser un système d'exploitation UNIX.

En **8** Semaines (Cours + TD de 2h) :

Comment faire communiquer 2 programmes d'une même machine ?

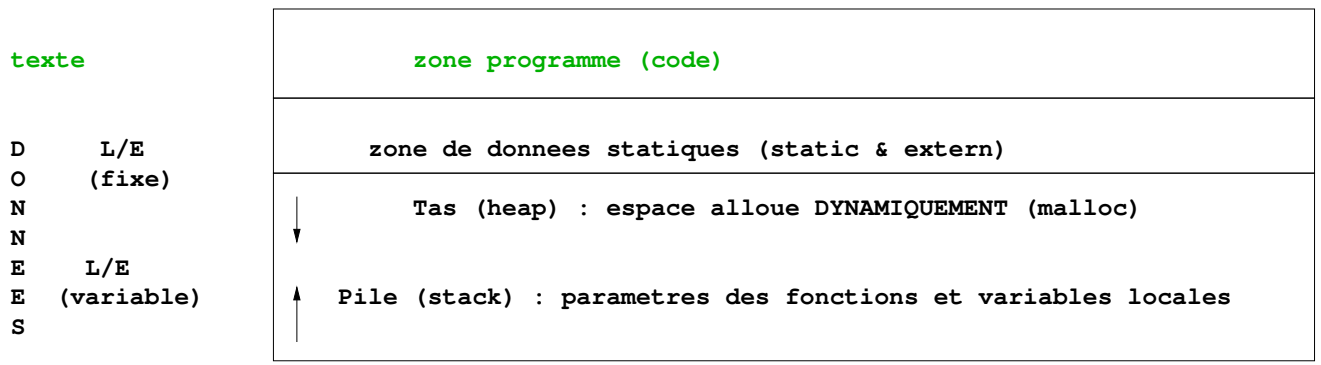
MC Info3-Système : Programmation Système

MC Info3-Système : Programmation Système

Remarque importante : la plupart des programmes exemples à exécuter (indispensable) et imprimer sont dans `/net/Bibliotheque/Systeme_a2/` :
`/net/Bibliotheque/Systeme_a2/INTERRUPTION/int1.c` , par exemple. (liste : `int1.c`, `int2.c`, `clock.c`, `dupli.c`, `attendre.c`, `course.c`, `process.c/fils.c`, `filtre.c`, `tube.c`)

- Un processus est un programme en cours d'exécution.
- Une zone mémoire est allouée à chaque processus.

Espace memoire d'un processus Unix



\+ le contexte :

. la valeur des registres
. la table des descripteurs de fichiers
. le repertoire courant
. le proprietaire,
. etc.

Notions sur les processus

- Lors de sa création, tout processus reçoit un numéro unique (entier positif de 0 à 32767) qui est son identificateur (**pid**).
- Tout processus est créé par un autre processus, excepté le **processus initial**, de nom **swapper** et de **pid 0**, créé artificiellement au chargement du système :

```
swapper (0)
    |
init (1)
```

Notions sur les processus (suite)

- Le swapper crée alors un processus appelé **init**, de **pid 1**, qui initialise le temps-partagé.
- Par convention, on considère que l'ensemble des processus existants à un instant donné forme un arbre dont la racine est le processus initial *init*.
- l'arbre des processus est obtenue par la commande : **pstree**.
- Tout processus a accès (par l'intermédiaire de fonctions système) à :
  - son pid (**getpid()**)
  - le pid de son père (**getppid()**)

Table des processus

- Elle est gérée par le noyau.
- Chaque entrée de la table contient des informations à propos d'un processus en cours d'exécution (structure sys/proc.h).
- Une entrée est allouée à la création du processus, désallouée à son achèvement.
- Listable par la commande ps : **ps -el**, **ps axu**.

La fonction system() : exemple en langage C

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main() {
printf("Je suis le processus N° : %d \n", getpid());

printf("Le pid de mon père est : %d \n", getppid());

system("ps -l");
exit(0);
}
```

La fonction system() : permet de lancer l'exécution d'un shell

```
#include <stdlib.h>

int system(const char* command);

Code de retour : 0 si OK, -1 sinon
```

La fonction system() : nouvel exemple en langage C

```
/* dater.c */
void main() {
    printf("La date est :"); fflush(stdout);
    system("/bin/date");
    /* suite du traitement */
}
```

La fonction system() : interprétation de l'exemple

- Lors de l'exécution de **dater**, un processus **shell** est créé à l'invocation de la fonction **system()** ;
- ce shell interprète la chaîne passée en argument :

```
dater
  |
sh (bash)
  |
/bin/date
```

Synchronisation des processus sous UNIX : les signaux

- Le traitement réalisé par un processus peut être interrompu par divers mécanismes d'interruptions.
- La réception d'un signal provoque une interruption logicielle : l'exécution d'un programme est interrompue pour traiter le signal reçu, puis reprend à l'endroit de son interruption.
- Les signaux sont en nombre fini (32 avec Linux).
- L'information véhiculée par un signal se borne à l'identité (le numéro) du signal.
- cf. `man -k signal` ou `man 7 signal`
- La liste des signaux disponibles sur le système peut être obtenue par la commande UNIX : **kill -l**

Synchronisation des processus sous UNIX : kill -l

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Synchronisation des processus sous UNIX : le principe

Exemples : signaux visant à "terminer" un processus :

SIGHUP (1)	Lors de la déconnexion (fin du shell), ce signal est envoyé à tous les processus du même terminal.
SIGINT (2)	Interruption : généré au clavier par la touche Ctrl-C.
SIGQUIT (3)	Généré au clavier par Ctrl-\. Par rapport au précédent, son objectif est l'obtention d'un fichier **core**.
SIGTERM (15)	Terminaison : peut être généré par la commande **kill** (kill pid).
SIGKILL (9)	Peut également être généré par la commande **kill** (kill -9 pid). Par rapport au précédent, ce signal ne peut être intercepté par le processus.

Synchronisation des processus sous UNIX : le principe

Exemples : signaux visant à "stopper/reprendre" un processus :

SIGSTOP (19)	Stopper processus.
SIGTSTP (20)	Stopper le processus. Généré au clavier par Ctrl-Z. (reprise par les commandes **fg** ou **bg**)
SIGCONT (18)	Reprise du processus.

Synchronisation des processus sous UNIX : le principe

Pour émettre un signal :

- l'utilisateur peut "agir" sur le processus actif attaché au terminal : émission des signaux Ctrl-C (SIGINT), Ctrl-Z (SIGTSTP), Ctrl-\(SIGQUIT) au clavier.
- via la commande **kill**
- via des appels système dans les programmes (expliqués ci-après, par exemple **kill()**)

Synchronisation des processus sous UNIX : le principe

- Lorsqu'un processus est chargé en mémoire, le système initialise sa **table de traitement des signaux** : à chaque signal correspond un élément de la Table de Traitement des Signaux **TTS**.
- Par la suite, lorsque le processus recevra un signal, le traitement qu'il réalisait sera interrompu, et il exécutera la fonction associée au signal reçu.

La fonction kill() envoie un signal à un autre processus

La fonction système **kill()** permet à un processus d'envoyer un signal à un autre processus (voire plusieurs) :

```
#include<unistd.h>
int kill (pid_t pid, int signum);
```

La fonction signal() permet de changer la fonction de traitement d'un signal

Les processus ont tous un traitement prédéfini aux signaux. Néanmoins, celle-ci peut être redéfinie par le programmeur pour la plupart des signaux.

- La fonction système **signal() (ou sigaction())** permet à un processus de changer la fonction de traitement d'un signal :

```
#include<stdio.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

La fonction système signal()

- Ainsi, dans la table de traitement des signaux **TTS**, la fonction associée au signal *signum* est remplacée par la fonction *handler()*.
- Deux fonctions ont un rôle particulier :
  - **SIG_IGN** : permet d'ignorer un signal,
  - **SIG_DFL** : permet de repositionner la fonction de traitement d'un signal à la fonction par défaut.

La fonction système signal()

Les appels les plus simples à la fonction **signal()** sont de la forme suivante :

- **signal(signum, fct_user) ;**
- **signal(signum, SIG_IGN) ;**
- **signal(signum, SIG_DFL) ;**

La fonction système signal()

- La fonction initiale de traitement de certains signaux (fonction par défaut) ne peut être modifiée ou ignorée : c'est notamment le cas des signaux **SIGSTOP et SIGKILL**
- Variantes suivant les signaux et les systèmes UNIX : lorsqu'un processus reçoit un signal, le système peut repositionner la fonction de traitement du signal à la fonction par défaut ...

La fonction système signal() : exemple 1 interruption

```
/* /net/Bibliotheque/Systeme_a2/INTERRUPTION/int1.c */
#include <unistd.h>
#include <signal.h>
void interruption (int), arret (int);
char cmpt = '1';
main ()
{
      signal(SIGINT, interruption); /* récuperation de Ctrl-C */
      signal(SIGQUIT, arret);      /* récuperation de Ctrl-\ */
      signal(SIGTSTP, SIG_IGN);     /* on ignore Ctrl-Z */
      for (;;) {
         write(1,&cmpt,1);
         sleep(1);
      }
}
void arret (int k) {
     write (1,"\n",1);
     write (1,"Au revoir\n",10);
     signal(SIGQUIT, SIG_DFL);
     exit(0);
}
void interruption (int k) {
     signal(SIGINT, interruption);
     cmpt++;
}
```

Le signal d'alarme SIGALRM / la primitive alarm()

```
/* /net/Bibliotheque/Systeme_a2/INTERRUPTION/int2.c */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#define DELAI 1
void initialise(), calcule(), sauve(), onalrm(int);
unsigned long i;
main () {
      initialise();
      signal(SIGALRM, onalrm);
      alarm(DELAI);
      calcule();
      fprintf(stdout,"calcul terminé\n");
      exit(0);
}
void onalrm (int k) {
      sauve();
      signal(SIGALRM,onalrm);
      alarm(DELAI);
}
void initialise () { i=0; }
void sauve() { fprintf(stderr,"sauvegarde de i : %lu\n",i); }
void calcule() { while (i += 2); }
```

Exercice sur les signaux

- On dispose du programme source écrit en langage C **/net/Bibliotheque/Systeme_a2/INTERRUPTION/clock.c** dont le rôle est d'afficher la date et l'heure en gros caractères sur la console.
- Ce programme est normalement appelé sans arguments. On vous demande de le modifier de façon à étendre ses possibilités, et notamment qu'il puisse être lancé sous les formes suivantes :
- $ **clock** // provoque l'affichage habituel de l'heure
- $ **clock** <délai> // provoque l'affichage habituel de l'heure // avec réveil au bout de *délai* secondes

nécessite la librairie *curses* : utiliser l'option -lcurses

Création d'un processus sous UNIX

La **création d'un nouveau processus Unix passe par deux mécanismes** utilisés en général de façon complémentaire :

1. la **duplication** d'un processus existant provoqué par la fonction système **fork()** : mécanisme de *fourche*,
2. le **recouvrement** d'un processus par un nouveau code : fonction système **exec()**.

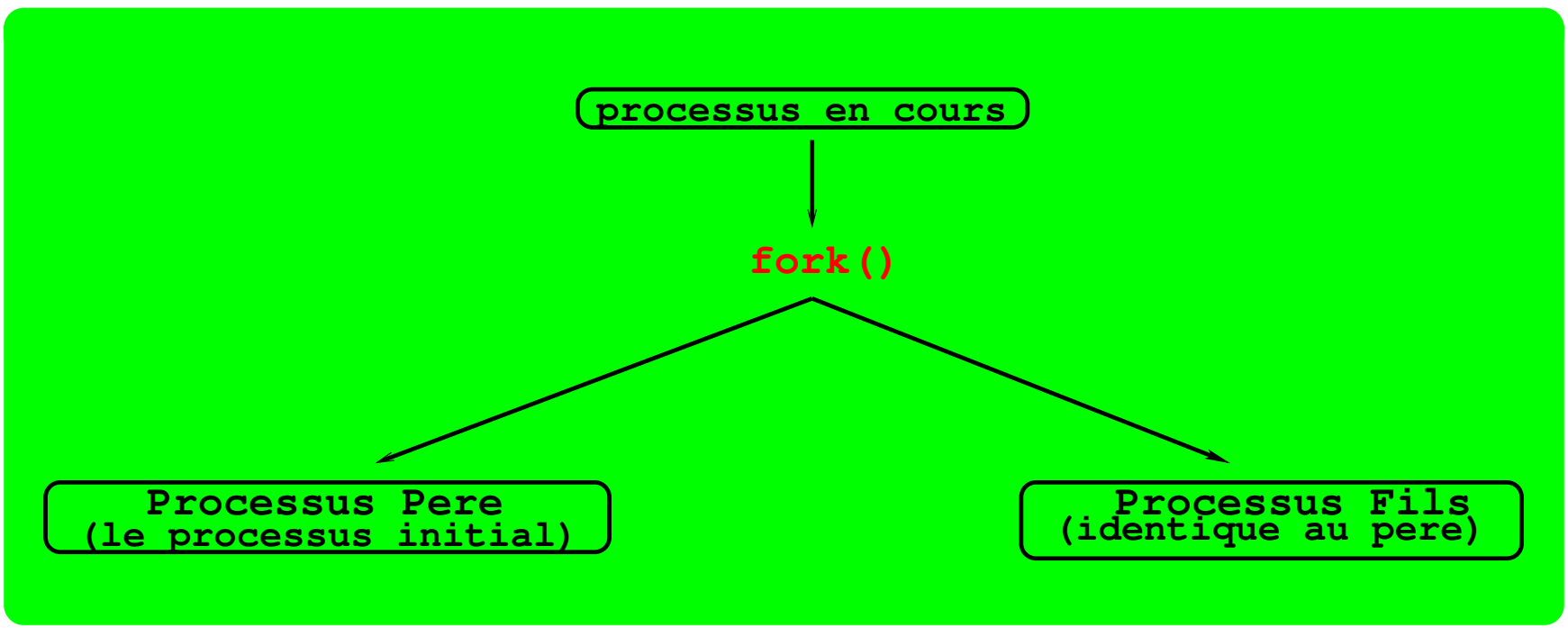
Création d'un processus sous UNIX

Ces mécanismes sont tels que **les processus** ainsi créés **pourront** :

- **se synchroniser** : envoi de signaux (appel système **kill()**), déroutement des fonctions de traitement des signaux (appel système **signal()**), mise en attente (appel système **wait()**), ...
- **communiquer** entre eux (appel système **pipe()**).

Duplication d'un processus

La fonction système **fork()** permet de dupliquer un processus en **créant dynamiquement un nouveau processus analogue au processus initial** :



Duplication d'un processus

Le processus créé (**processus fils**) hérite du **processus père** de certains de ses attributs :

- le même code,
- une copie de la zone de données,
- une copie de l'environnement,
- les différents propriétaires,
- une copie de la table des descripteurs de fichiers,
- une copie de la table de traitement des signaux,
- ...

Duplication d'un processus

Question : comment distinguer le processus père du fils ?
Réponse : leur **pid**
Plus précisément, le seul moyen (dans le code) de distinguer le *processus père* du *processus fils* est la **valeur de retour** de la fonction **fork()** qui est :

- **0 dans le processus fils**,
- **le pid du fils dans le processus père**.

Duplication d'un processus : exemple dupli.c

```
/* /net/Bibliotheque/Systeme_a2/FORK/dupli.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(){
      int n;
      if((n=fork())==0) {/* processus fils ici */
        printf("pid processus fils %d \n", getpid());
        exit(EXIT_SUCCESS);
      }
      else {/* le processus père vient ici */
        sleep(5);
        printf("pid processus père %d \n", getpid());
        printf("mon fils porte le N° %d \n", n);
      }
exit(EXIT_SUCCESS);
}
```

```
% dupli # Exécution
pid processus fils 4647
pid processus père 4646
mon fils porte le N° 4647
%
```

Duplication d'un processus : remarques

- En cas de problème lors de la création du processus *fils* (impossibilité de création en général), la valeur retournée par *fork()* est -1. Cette éventualité n'est pas testée dans *dupli.c*
- Le **processus père** et le **processus fils** sont concurrents : ils **s'exécutent en parallèle**.
- Le **processus père** et le **processus fils** peuvent se **synchroniser** par l'envoi de signaux : en effet, le père connaît le *pid du fils* (valeur de retour de *fork()*) et le fils connaît le *pid du père* (fontion *getppid()*) : voir le source `pere_fils1.c`

Synchronisation de processus Unix : Père-Fils

```
/* /net/Bibliotheque/Systeme_a2/FORK/pere_fils1.c */
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int nb_recu;

void hand (int sig){
  if(sig == SIGUSR1){ signal(SIGUSR1, hand); nb_recu++; printf("."); fflush(stdout); }
  else { printf("Nombre d'exemplaires reçus : %d\n", nb_recu); exit(EXIT_SUCCESS); }
}

void initialise () { nb_recu=0; }

main (){
  signal(SIGUSR1, hand);
  signal(SIGINT, hand);
  initialise();
  printf("Patientez 10 secondes svp ...\n");
  if(fork() == 0) {
    int i;
    for (i=0; i<10 ; i++){kill(getppid(), SIGUSR1); sleep(1); }
    printf("\nFin du fils \nVous pouvez taper Ctrl-C ...\n");
    exit(EXIT_SUCCESS);
  }
  while(1) pause();
}
```

La synchronisation de processus : l'appel système wait()

La primitive **wait()** provoque la **suspension** (mise en attente) du processus **jusqu'à** ce que l'un de ses **processus fils se termine**.

```
#include<unistd.h>
pid_t wait(int *status);
```

Cette primitive permet également à un processus d'**attendre un évènement** (voir **kill()**).

Synchronisation de processus Unix : attendre.c

```
/* /net/Bibliotheque/Systeme_a2/FORK/attendre.c */

#include <stdlib.h>
#include <stdio.h>
main(){
   int m, p;
   if(fork()==0){/* processus fils */
     printf("pid processus fils : %d \n", getpid());
     sleep(3);
     exit(3);
   }
   else{/* processus père */
     m=wait(&p);
     printf("fin du processus fils %d avec valeur de retour %d \n", m, p);
   }
}
```

```
Exécution :
% attendre
pid processus fils : 153
fin du processus fils 153 avec valeur de retour 768
%
```

Synchronisation de processus Unix : explications sur attendre.c

- La fonction **wait() retourne le pid du fils** qui s'est terminé (et qui a donc provoqué le réveil du père) ; dans le cas où il n'y a pas de fils, **wait()** retourne **-1**.
- **Le paramètre passé par adresse (int *status)** permet d'obtenir des **informations** sur la façon dont **s'est terminé le processus fils**.
- Cette information de 16 bits doit être interprétée de la manière suivante :
  - si le processus se **termine normalement** par un **exit(k)**, alors la valeur est $k \times 256$ (d'où 768 dans l'exemple),
  - si le processus se **termine anormalement** (signal), les deux octets permettent d'obtenir le **numéro de ce signal** (cf. man wait)...

Synchronisation de processus Unix (synthèse) : course.c

```
       /* /net/Bibliotheque/Systeme_a2/FORK/course.c */
#include <stdlib.h>
#include <stdio.h>
#define NB 5                    // nombre de concurrents
#define ARRIVEE 100000          // distance à parcourir
int main(){
   int i;             // compteur
   int r;             // retour fork
   pid_t pid;         // pid
   int status;        // valeur de retour
   for(i=1; i<=NB; i++) {
      if ((r = fork()) < 0) { // traitement erreur fork()
      fprintf(stderr,"Erreur fatale : fork()\n");
      exit(EXIT_FAILURE);
      }
      if (r == 0) { // un fils : il court ...
        int j;
        for (j=0; j<ARRIVEE; j++);
        exit(i);
      }
   } // dans le père : poursuite de la boucle for
   for (i=1; i<=NB; i++) {
   pid = wait(&status);
   fprintf (stdout, "le processus %d parti N° %d est arrivé N° %d\n", pid, status/256, i);
   }
   exit(EXIT_SUCCESS);
}
```

Synchronisation de processus Unix : résultats course.c

```
Exemples d'exécution ''course'' :
$ course
le processus 443 parti N° 4 est arrivé N° 1
le processus 442 parti N° 3 est arrivé N° 2
le processus 441 parti N° 2 est arrivé N° 3
le processus 440 parti N° 1 est arrivé N° 4
le processus 444 parti N° 5 est arrivé N° 5
$ course
le processus 447 parti N° 2 est arrivé N° 1
le processus 446 parti N° 1 est arrivé N° 2
le processus 450 parti N° 5 est arrivé N° 3
le processus 449 parti N° 4 est arrivé N° 4
le processus 448 parti N° 3 est arrivé N° 5
$
```

Synchronisation de processus Unix : Processus zombie

Lorsqu'un processus fils est terminé, il devient un **processus zombie** jusqu'à ce que :

- il soit rattrapé par un **wait** dans le processus père, ou
- le processus père meurt

Les processus zombie peuvent empêcher la création de nouveaux processus (32k max) : importance du **wait**.

Synchronisation de processus Unix : zombie.c

```
        /* /net/Bibliotheque/Systeme_a2/FORK/zombie.c */
#include <stdlib.h>
#include <stdio.h>
int main ()
{
   if (fork()==0) {
      // le fils dort 10 secondes puis termine
      printf("Le fils (pid %d) dort... ", getpid()); fflush(stdout);
      sleep(10);
      printf("et termine...\n"); fflush(stdout);
      exit(0);
   } else {
      // le pere boucle...
      while (1) pause();
   }
}
```

Exercice sur la création de processus

Jeu du ShiFuMi

- **Objectif :** écrire un programme qui fait jouer un nombre quelconque de processus au ShiFuMi (≥ 2 joueurs).
- **Déroulement :** le processus père lance *n* processus fils qui tirent aléatoirement PIERRE, PAPIER et CISEAUX. Les fils se synchronisent avec le père qui s'occupe de récupérer les valeurs jouées, compte les points et les affiche.
- **Indications :** s'inspirer des exemples vus précédemment notamment sur la synchronisation via la primitive *wait()*.

Exercice sur la création de processus

Jeu du ShiFuMi : trace d'exécution

```
$ ./shifumi 3
Jeux des Processus

Processus 7235 joue PIERRE
Processus 7236 joue CISEAUX
Processus 7237 joue PAPIER

Points des Processus

Processus : 7235 Point : 1
Processus : 7236 Point : 1
Processus : 7237 Point : 1
```

Recouvrement de processus : primitive **exec[l,v]()**

- La primitive système **exec[l,v]()**de recouvrement (ou **substitution**) permet de **lancer l'exécution d'un nouveau code**. Ce nouveau code recouvre l'ancien.
- Ainsi, il n'y a pas de création de nouveau processus. Il ne peut **pas** y avoir **de retour d"'exec réussi"**, et dans le cas où le recouvrement n'a pu se faire, la fonction *exec()* retourne -1.
- Le "nouveau" processus possède les mêmes caractéristiques (même contexte) que l'ancien :
  - même *pid*,
  - même *père*,
  - même priorité,
  - même propriétaire,
  - même répertoire de travail,
  - mêmes descripteurs de fichiers ouverts.

Le recouvrement par un nouveau code primitive **exec[l,v]()**

```
#include <unistd.h>
int execv(const char *filename, char *const argv[]);

int execl(const char *filename, const char *arg0, ...);
```

Duplication et recouvrement (1/3) : exemple process.c

```
            /* /net/Bibliotheque/Systeme_a2/FORK/process.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(){
   int pid, a, i;
   fprintf(stdout,"début du processus de numéro %d \n",getpid());
   a = fork();  /* création d'un second processus */
   if (!a) {  /* cette partie de programme ne s'exécute que pour le processus fils créé */
      execl( "fils","fils",0);
      fprintf(stderr,"pb execl ");
      exit(3);
   }
   fprintf(stdout,"Je suis le père de %d\n", a);
   for (i=0;i < 10; i++){
      fprintf(stdout,"le père de numéro %d continue\n",getpid());
      sleep(1);
   }
   /* le reste */
   sleep(2);
   fprintf(stdout,"fin du père de numéro : %d \n",getpid());
   exit (0);
}
```

Duplication et recouvrement (2/3) : exemple fils.c et résultats

```
          /* /net/Bibliotheque/Systeme_a2/FORK/fils.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
   int i;

   fprintf(stdout,"Début du fils de numéro %d \n",getpid());
   for (i=1;i<6;i++)
   {
      sleep(1);
      fprintf(stdout,"Le fils de numéro %d s'exécute\n",getpid());
   }
   fprintf(stdout,"Fin du processus de numéro : %d \n",getpid());
   exit(0);
}
```

Duplication et recouvrement (3/3) : résultats process/fils

```
Résultats ‘‘process/fils’’ :
% process
Début du processus de numéro 7326
Je suis père de 7327
Le père de numéro 7326 continue
Début du fils de numéro 7327
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Fin du fils de numéro : 7327
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Fin du père de numéro : 7326
%
```

Recouvrement et redirection : exemple substi.c et résultats

```
/* /net/Bibliotheque/Systeme_a2/EXEC/substi.c */
main(){
   close(STDOUT_FILENO);
   open("toto",O_RDWR|O_CREAT|O_APPEND);
   execl("/bin/ls", ".", "-l", 0);
   printf("test execl, ça fonctionne !");
   fprintf(2,"ERREUR EXEC\n");
}
```

Recouvrement et redirection : résultat

```
% substi
% cat toto
-rwxrwx--x+ 1 tmorsell info_perso 9724 sep 22 02:12 substi
-rw-r--r--+ 1 tmorsell info_perso 0 sep 22 02:12 toto
```

Recouvrement et redirection : exercice

ShiFuMi avec exec[l,v]

Reprendre l'exercice du ShiFuMi :

- placer le code d'un joueur dans un fichier **joueur.c**
- utiliser **exec[l,v]** pour appeler ce code
- le résultat attendu est identique à l'exercice d'origine