TD 1 – Processus (Unix)

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, sept 2022

1. Notion de processus

Question A Rappeler ce qu'est un processus et donner des exemples de processus que vous lancez très régulièrement sur votre ordinateur ou téléphone(3/4 lignes max).

2. Contexte d'un processus

Question B Lister au moins 3 informations qui composent le contexte d'un processus. On rappelle que le contexte est automatiquement conservé par le système lorsqu'un processus est commuté.

3. Création d'un processus (Unix)

On détaille ci après comment se passe la création d'un processus. La fonction système permettant de créer un processus est la fonction ***fork()***. Le processus qui appelle *fork()* est classiquement appelé le processus père, et le processus crée par la fonction *fork()* est appelé le processus fils. Comme toutes les fonctions systèmes, la fonction *fork()* retourne une valeur négative en cas de problème.

Aussi surprenant que cela puisse paraître, la fonction *fork()* ne reçoit aucun argument. On peut donc se demander quel programme elle va exécuter ? En fait, *fork()* permet de créer et démarrer un nouveau processus qui exécute le même programme que le processus courant. Plus précisément, le processus créé correspond à un *clone* du processus courant. Cela signifie que le processus fils hérite du contexte du processus père, à l'exception :

- du pid (le pid du fils est différent du pid du père)
- du résultat du fork (**dans la pile du processus père, le résultat du fork indique le pid du fils, alors que dans la pile du processus fils, le résultat du fork indique 0**)

Le schéma d'usage de base est le suivant

```
int r ;
r = fork() ;
if (r<0) ..          /* error */
else if (r==0) ..  /* child's code */
else ..              /* father's code */
```

Question C Essayez de bien comprendre ce qui se passe durant l'exécution du bloc d'instructions ci dessus. Pour cela, nous vous demandons de détailler ce qui se passe au niveau des espaces d'adressage du processus père et du processus fils, ainsi qu'au niveau des registres.

Question D Considérons l'exemple basique suivant. On vous demande de compléter les /* *todo* */ afin que le processsus fils affiche « I am the child» et que le processus affiche « I am the father, just launched a child having pid .. ».

```
int pid ;
…
pid = fork() ;
if (pid<0) {
        perror("fork error\n");
        exit(-1);
} else if (pid == 0) {
    /* todo */
} else {
    /* todo */
    }
```

Question E Combien de processus sont créés ? (on suppose que l'on est jamais en cas d'erreur)

```
(i1) fork() ;
(i2) fork() ;
(i3) fork() ;
```

Question F Combien de processus sont créés ? (on suppose que l'on est jamais en cas d'erreur)

```
for (int i=0 ; i<3 ; i++)
        fork() ;
```

Question G Nous avons vu que le contexte d'un processus fils est créé comme un clone du contexte de son père. Etendre le programme précédent pour faire en sorte que le processus père crée 4 processus fils et que chaque fils affiche son index.

```
int pid ;
…
for (int i=0; i<4; i++) {
        pid = fork() ;
        if (pid<0) { <error> }
        } else if (pid == 0) {..}
        } else { ..}
```

}

Question H Nous avons vu précédemment que lors d'un fork, le processus fils hérite du contexte du processus père. Ce contexte contient notamment les descripteurs de fichiers ouverts. Cela signifie que père et fils vont partager les mêmes descripteurs. D'après vous, est-ce que cela signifie qu'ils partagent les même pointeurs de lecture/écriture dans ces fichiers ?

Nous vous demandons de chercher cette information en lisant les spécifications associées à la fonction fork() par rapport aux fichiers.

.

4. Exécution d'un programme par un processus (Unix)

Pour faire en sorte qu'un processus exécute un programme différent de celui de son père, le système d'exploitation fournit les appels systèmes de la famille ***exec*** (execl, execv, execvp, ..). Ces appels prennent tous en argument le nom (ou chemin d'accès) du nouveau programme binaire exécutable à exécuter.

Le schéma d'usage de base est le suivant

```
int r ;
r = exec..(newprog, ..) ;
if (r<0) ..     /* error */
```

Un appel de type *exec* réinitialise le processus courant, en « rechargeant » son contexte (code, données, registres, ..) avec le nouveau programme binaire exécutable. Ainsi le PC sera réinitialisé pour pointer sur la méthode *main()* du nouveau programme.

Par exemple, quelque soit le code exécuté par le processus courant, le bloc de code suivant va le dérouter pour le faire exécuter la commande « ls –al ».

```
char  * argv[3];
…
argv[0] = "ls ";
argv[1] = "-al ";
argv[2] = 0;
execvp("ls", argv);
```

Question I Qu'affiche le programme suivant ? (on supposera que l'appel à *execl()* se passe bien).

```
char filename[32] ;
char arg1[32], arg1[32], arg3[32] ;
int ret ;
```

```
…
strcpy(filename, ”AddPerson“) ;
strcpy(arg0,”32”);
strcpy(arg1, “Bob”);
strcpy(arg2, “Marley”);
printf(”Launch pgm AddPerson“) ;
ret = execl(filename,arg0,arg1,arg2,NULL) ; // execv si nb arg inconnu statiq.
if (ret == -1) {
        perror(“execl error with %s\n”, filename);
        exit(-1);
} else {
        printf(“execl done with %s\n”, filename);
}
```

Question J Complétez le pseudo-code suivant pour que le processus courant crée un fils qui va exécuter le programme de nom « mypgm ».

```
    int pid ;
    …
    pid = fork() ;
    if (pid<0) { <error> }
     } else if (pid == 0) {.. }
     } else { ..}
}
```

5. Interruptions

Un processus peut envoyer des interruptions à l'un de ses fils. Il peut notamment envoyer SIGINT, SIGSTOP, SIGTERM et SIGKILL pour effectuer un arrêt (Ctl-C), suspendre (Ctl-Z), terminer ou tuer un processus au travers de la fonction système kill.

Sur la réception d'une interruption, un traitant est exécuté (appelé handler). Ce traitant peut être obtenu et modifié via l'appel système *sigaction()*. Bien entendu, on n'a pas le droit de changer le traitement par défaut de certains signaux, tel que SIGKILL ou SIGSTOP (ces signaux ne peuvent être ni interceptés, ni ignorés ; ils sont traités par le système et pas par le processus).

```
int kill (int pid, int sig);
typedef void (*sighandler t)(int); //  handler
sighandler t signal(int signum, sighandler t handler);
// set a handler
```

A titre d' exemple, quand vous tapez Ctl-C dans un shell, le processus shell envoie le signal SIGTERM (ou SIGINT selon les versions) à tous les processus qu'il a lancé et qui ne se sont pas encore terminés.

L'exemple ci-après permet d'associer la fonction *child_handler ()* comme handler de l'interruption SIGCHLD. Cette interruption qui est envoyée à un processus père à chaque fois que l'un de ses fils termine.

```
static void child_handler(int sig) {
pid_t pid;
int status;
while((pid = waitpid(-1, &status, WNOHANG)) > 0) ; }


/* Establish handler */
/* look at sigaction specification for details */
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = child_handler;
sigaction(SIGCHLD, &sa, NULL);
```

Pour information, certains signaux peuvent être *bloqués* au travers de l'appel système *sigprocmask*(): les signaux bloqués sont mis en attente, en général pour être délivrés ultérieurement. Certains signaux (comme SIGSTOP, SIGKILL) ne peuvent pas être bloqués.

La compréhension de l'aspect signaux n'est pas étudiée en profondeur ici. Vous la verrez en détail lors de la réalisation du TP Shell.

6. Synchronisation père-fils

La plupart des systèmes permettent de synchroniser un processus père et ses fils. Dans Unix, cela se fait au travers des appels systèmes de la famille ***wait*** (wait, waitpid, ..). *wait()* permet à un père d'attendre la terminaison d'un processus fils, quelqu'il soit.

Lorsqu'un processus père appelle une fonction de type wait() et qu'aucun processus fils n'a encore terminé, alors le système place le processus père passe dans un état *bloqué*. Ce processus est donc commuté et ne sera replacé dans la file des prêts que lorsqu'il pourra poursuivre son exécution (donc lorsqu'un processus fils aura terminé).

Lorsqu'un processus père appelle une fonction de type *wait()* et qu'un processus fils a terminé, alors l'appel à *wait()* n'est pas bloquant et retourne immédiatement au père le pid du fils qui a terminé. Si plusieurs fils ont terminé, les appels successifs à *wait()* renvoient les pids de ces fils, dans l'ordre de leur terminaison.

Il y a différentes variantes de la fonction *wait()*. Notamment, *waitpid(p)* permet à un père d'attendre la terminaison du processus fils de pid *p* (sauf si *p* vaut -1). De plus, *waitpid*

accepte un argument (*WNOHANG* dans l'exemple ci dessous) qui rend la fonction non bloquante : si aucun fils n'a terminé, la fonction renvoie simplement une valeur nulle.

```
int status = 0;
/* création des fils */ ;
int p1 = fork() ; …
int p2 = fork() ; …
int p3 = fork() ; …

/* attente d'un fils */
int pid = wait(&status) ;  // status contient des infos sur la terminaison du fils
                           // status != NULL signifie terminaison anormale

/* variante */
int pid = waitpid(-1, &status, WNOHANG) ;

/* autre variante */
waitpid(p1, &status, NULL) ;   // attend la terminaison de p1
```

Pour forcer la libération des ressources d'un processus qui se termine, il y a deux moyens : (i) le père « entérine » la terminaison d'un fils par un appel à wait() ou waitpid(), (ii) le père a explicitement demandé à ignorer les signaux de terminaison de ses fils, via un appel à la fonction *signal(SIGCHLD, SIG_IGN)*). Dans ce cas, tout fils qui se termine verra ses ressources automatiquement libérées au niveau du système.

Autrement dit, si vous créez des processus et ne pensez pas à gérer leur terminaison, votre machine va rapidement saturer en ressources.

Question K Rappelez les 3 situations principales qui provoquent la terminaison d'un processus

Notons que le cas d'un mauvais fonctionnement (ex : div/0) est traité par le système par l'envoi d'une interruption (SIGINT).

Question L Ecrire un programme qui imprime « I am the father », après avoir crée un fils qui imprime « I am the child ».

7. Exemple d'enchaînement de processus et travail pratique

Pour information, voici ci après un exemple typique d'enchainements de processus qui a lieu dans toute machine Unix.

Le processus initial (pid 0) :

- lit le fichier /etc/ttys qui donne le nombre de terminaux et leur description
- crée un processus fils par tty
- crée les démons système (rlogin, NFS, …)
- devient le démon de gestion de la pagination

Chaque processus fils affecté à un terminal (tty) exécute le programme login qui :

- affiche « login »
- attend une entrée clavier
- vérifie le nom et le passwd
- exécute le programme shell de l'utilisateur qui vient de se logger (le shell à exécuter est indiqué dans le fichier /etc/passwd)

Le programme shell :

- affiche un prompt
- attend une commande tapée au clavier
- analyse cette commande pour distinguer le programme demandé et les arguments associés (chaque commande correspond en effet à un programme binaire exécutable, par exemple, /bin/ls)
- crée un processus pour exécuter la commande demandée (sauf si la commande est de type *built-in*, voir ci après)
- attend la fin du processus crée sauf si la commande est en background (le dernier argument est le caractère '&')

Les commandes de type ***built-in*** correspondent aux commandes qui ont pour effet de modifier des variables environnement du shell (par exemple, la variable qui indique le répertoire courant ($PWD), la variable qui donne le PATH, etc).

Question M D'après vous, les commandes suivantes sont-elles de type built-in ou classique ? Vous devez en fait identifier 6 commandes built-in et 6 commandes classiques.

- ls, set, move, cp, pwd, source, mkdir, history, echo, alias, bg

Question N Nous vous demandons de produire le pseudo-code en C de mise en œuvre d'un shell. Ce travail est à réaliser en vue du projet que vous allez réaliser dans lequel vous allez implémenter un programme shell assez complet, qui devra passer un grand nombre de jeux de tests.

Rappel : compilation via gcc de 3 modules sources C vers un exécutable nommé mypgm:

gcc m1.c m2.c m3.c –o mypgm

Code de la fonction read_command :

```
#define FOREGROUND 0
#define BACKGROUND 1
/*-------------------------------------------------------------*/
/* void read_command                                           */
/* reads a shell command from stdin                            */
/* command: contain as result a pointer on the command string */
/* args: array that contains as result pointers on command    */
/*       arguments                                             */
/* type : contains command type as result (BACKGROUND or      */
/*        FOREGROUND                                           */
/*-------------------------------------------------------------*/
void read_command(char **command, char*args[], int *type) {
int nargs;

  do {
      printf(">");
      if (fgets(line, sizeof(line), stdin) != NULL) {
      line[strlen(line) - 1] = (char) 0;
         *command = strtok(line, " ");
      }
   } while (!*command);

  nargs = 0;
  args[nargs++] = *command;

  while (nargs < MAX_PARAMS) {
    args[nargs] = strtok(NULL, " ");
    if (args[nargs] == NULL)
      break;
    nargs++;
  }
  if (nargs == MAX_PARAMS) printf("shell: too much arguments\n");
  if ((nargs > 0) && !strcmp(args[nargs-1], "&")) {
      *type = BACKGROUND; args[nargs-1] = NULL;
  } else {
      *type = FOREGROUND;
  }
 }
```