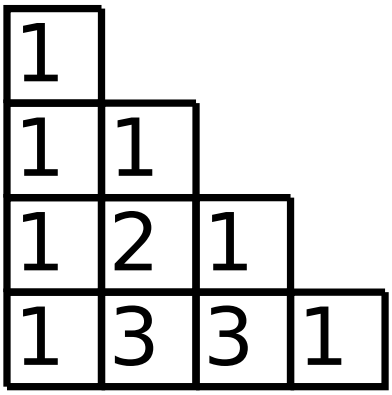
Université Grenoble Alpes

Triangle de Pascal

Introduction aux systèmes d'exploitation



Bastien Delseny

M2 - CCI

11 Janvier 2015 — 11 Juin 2015

Table des matières

1 Compilation et génération du fichier exécutable

1.1 Compilation de triangle

L'exécution du programme triangle permet d'obtenir les instructions en PostScript pour la construction d'un triangle de Pascal de taille de police et de nombre de lignes voulu. Pour exécuter ce programme il suffit d'exécuter la commande >./triangle 12 4 ce qui donnera les instructions en PostScirpt pour créer un triangle de Pascal de taille de police 12 et de 4 lignes (code 1).

Listing 1 – ./triangle 12 4

```
newpath 24 245 moveto 38 245 lineto 38 260 lineto stroke newpath
    ↪ 24 245 moveto 24 260 lineto 38 260 lineto stroke /Courier
    ↪ findfont 12 scalefont setfont newpath 25 248 moveto (1)
    ↪ show
newpath 24 230 moveto 38 230 lineto 38 245 lineto stroke newpath
    ↪ 24 230 moveto 24 245 lineto 38 245 lineto stroke /Courier
    ↪ findfont 12 scalefont setfont newpath 25 233 moveto (1)
    ↪ show
etc ...
```

1.2 Redirection de la sortie standard

La commande >./triangle 12 4 > triangle_pascal_12_4.ps permet de rediriger la sortie standard vers le fichier triangle_pascal_12_4.ps qui contient le code PostScript 1. À l'aide de la commande > cat triangle_pascal_12_4.ps on affiche dans le terminal le code PostScript (code 1) enregistré dans triangle_pascal_12_4.ps. À l'aide de la commande >gv triangle_pascal_12_4.ps l'interprète de langage PostScript nous permet d'observer la figure 1.

1.3 Redirection de l'entrée standard

L'exécution de la commande > tr "bc" "BC" < Makefile permet de mettre en majuscule les lettres c et b du fichier Makefile.

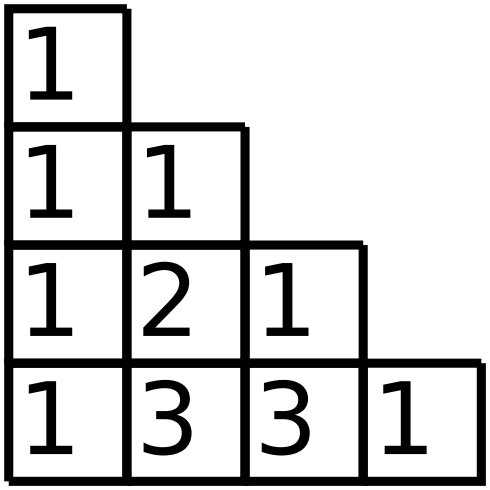


FIGURE 1 – Triangle de Pascal de taille de police 12 de 4 lignes.

1.4 Tube et chaînage de commandes

Lorsque l'on exécute la commande > date > d on redirige la sortie de l'exécution de la commande > date vers le fichier d. L'exécution de la commande > tr ":" "_" < d permet de récupérer ce que contient le fichier d afin de remplacer les : par des _. À l'aide de l'utilisation d'un tube de redirection il est possible d'effectuer la même manoeuvre sans passer par un fichier intermédiaire et afficher le résultat directement dans le terminal. C'est ce que permet de faire la commande date \| tr ":" "_".

Ainsi en utilisant les commandes 2 on obtient la même figure 1 qu'en effectuant les démarche de la section 1.2 qui sera affichée par un interpréteur de PostScript.

Listing 2 – Tubes de redirection

```
> ./triangle 12 4 | gs -sDEVICE=x11 -q
> ./triangle_12_4 | gv -
```

2 Lancement de l'interprète PostScript par exec

Les exécutables des commandes gv et gs se trouvent dans les répertoire /usr/bin/gv et /usr/bin/gs, chemin qu'il a été possible de trouver grâce à la commande which. Ainsi exécuter la commande

>/usr/bin/gv triangle.ps revient au même que d'exécuter la commande >gv triangle.ps.

Afin d'exécuter directement l'interpreteur de PostScript lors de l'exécution du programme il faut modifier le code source de affiche_triangle. Il faudra commencer par initialiser deux variables. Un texte contenant le chemin de la commande de l'interpréteur PostScript à utiliser. Ainsi qu'un tableau de texte contenant la commande ainsi que ces arguments. Le code nécessaire pour l'interpréteur gv et celui pour gs est ici : 3.

Listing 3 – Initialisation des variables pour exec de gv et gs

```
/*      Arguments pour gv       */
char nomGV [] = "/usr/bin/gv";
char *argumentsGV [] = {"/usr/bin/gv", "triangle.ps", NULL};

/*  Arguments pour gs   */
char nomGS [] = "/usr/bin/gs";
char *argumentsGS [] = {"/usr/bin/gs", "-q", "-sDEVICE=x11", "
    ↪ triangle.ps", NULL};
```

Ensuite il faudra ajouter la commande execve(...) correspondante à l'interprète choisi à la fin du main() (code 4)

Listing 4 – Appel de la fonction execve() pour gv et gs

```
/*  Execution de gv     */
execve(nomGV, argumentsGV, envp);

/*  Execution de gs     */
execve(nomGS, argumentsGS, envp);
```

Puis il faut recompiler le code avec >make. Il suffira d'exécuter le programme à l'aide de la commande >./triangle 12 4 > triangle.ps pour obtenir le même résultat que pour la section 1.4. On peut en plus rediriger la sortie standard d'erreur dans un fichier à l'aide de l'option 2> fichier.log comme suit >./triangle 12 4 > triangle.ps 2> triangle.log.

3 Lancement de gs par fork et exec

3.1 Vérification de la taille de police et du nombre de lignes

Afin de vérifier si la taille de la police et si le nombre de ligne sont correctes on peut ajouter le code suivant 5 dans le code source de affiche_triangle.c

Listing 5 – Vérification de la taille de police comprise entre 8 et 24, et du nombre de lignes comprit entre 1 et 40.

```
if(taille_police < 8 || taille_police > 24){
        fprintf(stderr, "Taille de police incorrecte, valeur
          ↪ entre 8 et 24 attendue\n");
        exit(1);
}
if(nb_lignes < 1 || nb_lignes > MAX_LIGNES){ // MAX_LIGNES = 40
        fprintf(stderr, "Nombre de lignes incorrecte, valeur
          ↪ entre 1 et %d attendue\n", MAX_LIGNES);
        exit(1);
}
```

3.2 Une histoire de père et de fils

La fonction C fork() permet de créer un fils. Il faut commencer par initialiser deux variables dans affiche_triangle.c. Une variable qui permettra de récupérer le pid du fils grâce à la commande pid_t p;. Il faut aussi une variable qui permettra de récupérer le code de retour, code d'erreur ou de réussite de la fin d'exécution du fils, grâce à la commande int retour;. Afin que le processus fils exécute gv ou gs il faut procéder comme indiquer dans le code 6.

Listing 6 – Création d'un fils qui execute la fonction execve permettant la lecture du fichier PostScript par un interpréteur de PostScript.

```
/* Generation des triangles par le pere */
sortie = "stdout";
taille_triangle = nb_lignes;
```

```
postscript_triangle (taille_police);
sleep (3);

/* Creation d'un processus fils */
p = fork();

/* Verification de la creation d'un fils a l'aide de la commande
   ↪ fork */
if(p < 0) {
        fprintf(stderr, "Luke␣(fils)␣n'a␣pas␣pu␣etre␣creer␣par␣
           ↪ fork\nVador␣(pere)␣est␣triste\n");
        exit(1);
}

/* Fils creer  plus qu'a attendre qu'il affiche les triangles*/
if(p != 0){
        /* Attendre la terminaison du fils */
        wait(&retour);
        fprintf(stderr, "Generation␣et␣affichage␣du␣triangle␣de␣
           ↪ Pascal␣termine\n");
} else {
        /* Execution de l'interpreteur de PostScript */
        execve(nomGV, argumentsGV, envp);
        //execve(nomGS, argumentsGS, envp);
}
```

4 Génération du Postscript dans un fichier.

Après avoir généré le Postscript du triangle dans triangle.ps il est possible de connaître la taille de ce fichier Postsrcipt. Pour cela il suffit d'effectuer la commande ls -l triangle.ps dans le terminal. On observe que la taille du fichier est de $1,9ko$, appelons $T1$ cette taille.

4.1 Un problème de taille

Après avoir supprimé le fichier de $1,9ko$ triangle.ps et modifié le code de la procédure main pour affecter la sortie à un fichier triangle.ps il faut compiler le code et exécuter le programme. À la fin de l'exécution il y a un nouveau fichier triangle.ps. Ce fichier fait une taille $T2$ de $185o$. La taille de ce nouveau fichier $T2$ est donc largement inférieur à la taille du fichier précédent $T1$. Lorsque l'on ouvre triangle.ps avec un interpréteur de Postscript on se rend compte qu'un seul triangle a été créé.

4.2 Il nous manque des cases

La ligne 66 de la fonction boite_chiffre() du code boite_chiffre.c permet de définir les options d'écriture dans le fichier de sortie. On remarque que le code est le suivant : fsortie = fopen(sortie, "w"). L'option w permet de créer un nouveau fichier et d'écrire dans ce fichier. Or dans le code affiche_triangle.c on observe que la fonction postscript_triangle() fait appelle à une boucle sur la fonction boite_chiffre(). Donc à chaque appelle de boite_chiffre() un nouveau fichier va être créé et une seule boite sera dessinée. Pour palier à ce problème il nous faut changer l'option w de la fonction fopen() par l'option a qui permet de continuer l'écriture dans le fichier existant si celui-ci existe, sinon le fichier sera créé.

4.3 Un problème de superposition

Maintenant un nouveau problème fait face. Lorsque nous lançons plusieurs fois l'exécution du programme les triangles se superposent. Ce qui rend à la fois la présentation des triangles illisibles et la taille des fichiers exponentielle. Il faut donc qu'au début de l'exécution du programme un nouveau fichier soit créé afin de ne pas réécrire à la suite d'un fichier existant. Pour ce faire il faut modifier le code de la fonction postscript_triangle() du code affiche_triangle.c. Il faut en début de cette fonction initialiser un fichier si la sortie n'est pas la sortie standard. On rajoute alors le code 7 en début de la fonction postscript_triangle()

Listing 7 – Initialisation du fichier de sortie.

```
if( strcmp(sortie, ''stdout'') != 0 ) {
  fopen(sortie, ''w'');
}
```

On remarque que suite à cette modification la taille du fichier sera la même en mettant la sortie vers un fichier dans le code source du programme ou en redirigeant la sortie standard vers un fichier. Ainsi à la fin de ces modification $T1 = T2$.

5 Création d'un tube entre triangle et gs

Le but ici est que le processus père créer un tube et créer deux fils qui intéragissent à l'aide de ce tube. Ainsi on commence par créer un tube de redirection à l'aide de la commande pipe(). Une fois le tube créer on créer un processus fils qui va générer le Postscript de triangle vers l'entrée du tube. Ainsi la sortie standard est redirigée vers l'entrée du tube de redirection. On termine le processus fils et on créer un nouveau processus fils.
On redirige l'entrée standard, qui sera donc celle du nouveau fils, vers la sortie du tube de redirection. Le processus fils lance alors l'interprète de Postscript.
Une fois que l'interpréte de Postscript est terminé on termine le processus fils puis on ferme le tube de redirection.
L'ensemble du code est détaillé ici : 8.

Listing 8 – Utilisation d'un tube dans la fonction main.

```
int main(int argc, char *argv[], char *envp[])
{
  /* Initialisation des variables */
  unsigned int taille_police, nb_lignes;
  char nom_executable[200];
  pid_t p_gauche, p_droit;
  int retour_gauche, retour_droit;
  int tube[2];

  lire_args(argc,argv,3,message_usage,
        "%s",nom_executable,"",
        "%d",&taille_police,"taille_de_police_incorrecte",
        "%d",&nb_lignes,"nombre de lignes incorrect");
```

```
/* Verification des valeurs */
/* de taille_police [8,24] et nb_lignes [1,MAX_LIGNES]  */
if(taille_police < 8 || taille_police > 24){
        fprintf(stderr, "Taille de police incorrecte, valeur
          ↪ entre 8 et 24 attendue\n");
        exit(1);
}
if(nb_lignes < 1 || nb_lignes > MAX_LIGNES){
        fprintf(stderr, "Nombre de lignes incorrecte, valeur
          ↪ entre 1 et %d attendue\n", MAX_LIGNES);
        exit(1);
}

      /* Creation d'un tube */
      pipe(tube);

      /* Fils gauche */
      p_gauche = fork();

      if(p_gauche < 0) { // erreur
              fprintf(stderr, "Le fils n'a pas pu etre cree\n")
                ↪ ;
      }

              /* Le fils gauche genere le Postscript,
                      envoie la sortie standard vers l'entree
                        ↪ du tube */
      if(p_gauche == 0){
              close(tube[0]); // fermeture de la sortie du tube
              dup2(tube[1],1);
```

```
            sortie = "stdout";
            taille_triangle = nb_lignes;
            postscript_triangle (taille_police);
            sleep (3);
            close(tube[1]);
            exit(0);

    }

            p_droit = fork();
    if(p_droit < 0) { // erreur
            fprintf(stderr, "Le fils n'a pas pu etre cree\n")
               ↪ ;
    }

            /* Le fils droit recupere la sortie du tube,
                    ouvre l'interprete de Postscript */
    if(p_droit == 0){
            close(tube[1]); // On ferme la sortie du tube
            dup2(tube[0],0);
            execve(nomGV, argumentsGV, envp);
            close(tube[0]);
            exit(0);
    }

    /* fermeture des tubes du pere */
    close(tube[0]);
    close(tube[1]);

    /* Attente de la fin des fils */
    waitpid(p_gauche, &retour_gauche, 0);
```

```
        waitpid(p_droit, &retour_droit, 0);

        fprintf(stderr, "Generation et affichage du triangle de
            ↪ Pascal termine\n");


    return 0;
}
```