Solution en Python

runfic

```
import os, sys, os.path

# Vérifie qu'on a passé un seul paramètre à l'appel du programme
if len(sys.argv) != 2:
    print(f"Usage: python {sys.argv[0]} fichier_source_c++", file=sys.stderr)
    sys.exit(1)

# Teste si le fichier source existe
if not os.path.isfile(sys.argv[1]):
    print(f"{sys.argv[1]} n'existe pas", file=sys.stderr)
    sys.exit(2)

if os.fork() == 0:          # Le processus fils exécute gcc
    fic = os.open("/tmp/erreurs", os.O_CREAT|os.O_WRONLY|os.O_TRUNC)
    os.dup2(fic, 2)         # Redirection des erreurs dans /tmp/erreurs
    os.execlp("c++", "c++", sys.argv[1])

(pid, status) = os.wait()   # Le père attend la fin de c++

if status != 0:              # erreurs de compilation...
    if os.fork() == 0:      # le nouveau fils exécute more
        os.execlp("more", "more", "/tmp/erreurs")
    os.wait()               # Le père attend la fin du more
    os.remove("/tmp/erreurs")
    input("Appuyez sur Entrée pour continuer")
    os.execlp("code", "code", sys.argv[1])  # Puis lance Visual Code
else:                       # la compilation s'était bien passée
    os.remove("/tmp/erreurs")
    os.execlp("./a.out", "a.out")
```

La même chose en Go

```
package main

import (...)

func checkSyntaxe(nomfic string) {
  if !strings.HasSuffix(nomfic, ".cpp") {
    fmt.Fprintf(os.Stderr, "Erreur : %s n'est pas un fichier C++\n", nomfic)
    os.Exit(2)
  }
}

func main() {
  nbParams := len(os.Args)
  if nbParams != 2 {
    fmt.Fprintf(os.Stderr, "Usage: %s fic\n", filepath.Base(os.Args[0]))
    os.Exit(0)
  }
  fic := os.Args[1]
  checkSyntaxe(fic)

  compile := exec.Command("c++", fic)
  ficErr, _ := os.Create("/tmp/erreurs")
  compile.Stderr = ficErr // Redirection des erreurs vers /tmp/erreurs
  compile.Start()         // Lancement en arrière plan
```

La même chose en Go (suite et fin)

```
    if err := compile.Wait(); err != nil { // il y a eu erreur de compilation
      more := exec.Command("more", "/tmp/erreurs")
      more.Stdout = os.Stdout
      more.Start()                         // Exécution parallèle

      more.Wait()
      os.Remove("/tmp/erreurs")
      exec.Command("code", fic).Run()      // Exécution séquentielle
    } else {
      // La compil s'est bien passée, on lance a.out
      os.Remove("/tmp/erreurs")
      run := exec.Command("./a.out")
      run.Stdout = os.Stdout
      run.Run()
    }
}
```

La même chose en Rust

```
use (...)

fn check_syntaxe(nomfic: &str) {
    if !nomfic.ends_with(".cpp") {
        eprintln!("Erreur: {nomfic} n'est pas un fichier C++");
        std::process::exit(2);
    }
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    let program_name = args[0].split('/').last().unwrap();

    if args.len() != 2 {
        eprintln!("Usage: {program_name} fic");
        std::process::exit(0);
    }

    let fic = &args[1];
    check_syntaxe(&fic);

    let fic_err = File::create("/tmp/erreurs").expect("Pb de création du fichier erreurs");
    let mut compile = std::process::Command::new("c++")
        .arg(fic)
        .stderr(fic_err)
        .spawn()                                    // lancement en //
        .expect("pb de créeation du processus c++");
```

La même chose en Rust

```
    let code = compile.wait().unwrap();
    if !code.success() {
        // La compilation a échoué
        std::process::Command::new("more")
            .arg("/tmp/erreurs")
            .spawn()
            .expect("pb de création du processus more")
            .wait()
            .unwrap();

        remove_file("/tmp/erreurs").unwrap();
        std::process::Command::new("code")
            .arg(fic)
            .output()
            .expect("pb de création du processus Visual Code");
    } else {
        // La compilation a réussi
        remove_file("/tmp/erreurs").unwrap();
        let prog = std::process::Command::new("./a.out")
            .output()
            .expect("Erreur d'exécution du programme");
        io::stdout().write_all(&prog.stdout).unwrap();
    }
}
```

Communication inter-processus

On veut écrire une commande qui permette à un processus fils de saisir une information au clavier et au processus père d'afficher cette même information.

> *Peut-on transmettre des informations de type quelconque entre les deux processus ?*
> *On rappelle que chaque processus dispose d'un segment de données* **privé** *et que deux processus distincts n'ont strictement aucun espace mémoire en commun...*

- Le père ouvre un fichier, le fils écrit dedans ? pb de synchro, manipulation de fichiers (lenteur...), etc.
- Utilisation des status renvoyés par les processus ? mais ce sont uniquement des entiers...

Tubes de communication

Un tube de communication (*pipe*) est défini comme un fichier :

- En mémoire centrale
  - Structure analogue à une simple variable en mémoire centrale, les opérations pour manipuler un tube sont : *read*, *write*, *dup*, *close*.
  - Des entrées dans la table des fichiers ouverts sont associées aux tubes.
- De taille limitée
  - Il ne sera pas possible d'écrire directement de très longues séquences d'octets dans un tube.
- Avec des lectures destructrices
  - À l'inverse d'une lecture classique dans un fichier, une lecture dans un tube entraîne la suppression des informations lues.
- Avec synchronisation des accès
  - Les lectures et les écritures effectuées par les processus utilisateurs devront respecter certaines règles quant à l'ordre d'exécution et aux conditions dans lesquelles elles devront être effectuées.

Synchronisation de l'accès aux tubes

- Les processus utilisateurs
  - Certains processus sont amenés à *écrire* dans le tube : ce sont les *producteurs*.
  - Certains processus sont amenés à *lire* dans le tube : ce sont les *consommateurs*.
- La synchronisation repose sur le principe suivant :
  - Lecture bloquante si le nombre d'informations demandé n'est pas disponible dans le tube.
  - Écriture bloquante si le tube ne peut contenir les informations à écrire.

Tubes systèmes en Python : création

- Le module *os* fournit la fonction *pipe()* qui renvoie un tuple de deux descripteurs *(lecture, ecriture)*.

Remarque

- Tous les processus fils créés *après* la création du tube connaissent ce tube (à cause de la duplication des descripteurs) : on a donc une communication limitée à une branche de processus.
- Pour que deux processus puissent communiquer, il suffit donc qu'ils aient un ancêtre commun qui ait créé un tube.

Tubes systèmes en Python : fermeture

- On ferme les deux extrémités du tube par un appel à *os.close()* (et non par un appel à la méthode *close()* des fichiers...)
- Le consommateur récupère alors ce qui reste dans le tube à concurrence de ce qu'il avait demandé (le nombre d'octets voulu ou moins...).
- Du point de vue d'un producteur qui *écrit* dans le tube, la communication prend fin lorsque le tube n'est plus ouvert qu'en écriture.

Conséquences

- Les fermetures de tubes sont essentielles ! Sinon, les processus bouclent car ils ne sauront jamais que la communication est terminée.
- Tout processus n'utilisant pas/plus une extrémité du tube *doit* fermer cette extrémité.

Principe général

- Le père crée le tube *avant* de créer le fils, pour que les deux le connaissent.
- Chaque processus (le père et le fils) ferment ensuite l'extrémité qui ne les concerne pas.
- Les deux processus communiquent en écrivant et en lisant dans le tube
- Quand cette communication est finie, les processus ferment l'extrémité du tube

Remarque

On peut utiliser plusieurs tubes entre deux processus pour communiquer dans les deux sens...

Tubes systèmes en Python

grepsortwc.py

```
import os, sys

if len(sys.argv) != 3:
    print(f"Usage: {sys.argv[0]} regex fic", fic=os.stderr)
    sys.exit(1)

rd, wr = os.pipe()        # Création du pipe sort | wc
if os.fork() == 0:        # Le fils exécute sort
    os.close(rd)          # Il ne lira donc jamais dans le pipe
    os.dup2(wr, 1)        # Redirection de stdout vers wr
    os.close(wr)

    rd, wr = os.pipe()    # Création du pipe grep | sort
    if os.fork() == 0:    # Le fils du fils exécute grep
        os.close(rd)      # Il ne lira donc jamais dans le pipe
        os.dup2(wr, 1)    # Redirection de stdout vers wr
        os.close(wr)
        os.execlp("grep", "grep", sys.argv[1], sys.argv[2])
    else :                 # Le père exécute sort
        os.close(wr)       # Il n'écrira donc jamais dans le pipe
        os.dup2(rd, 0)     # Redirection de stdin vers rd
        os.close(rd)
        os.execlp("sort", "sort", "-uf")
```

Tubes systèmes en Python

grepsortwc.py

```
else:                       # Le père fait le wc
    os.close(wr)            # Il n'écrira donc jamais dans le pipe
    os.dup2(rd, 0)          # Redirection de stdin vers rd
    os.close(rd)
    os.execlp("wc", "wc", "-l")
```

- Notez l'utilisation de *os.close(rd)* et *os.close(wr)* et non *rd.close()* ou *wr.close()*...
- La *méthode close()* s'applique à des fichiers (ou assimilés) or *rd* et *wr* sont des descripteurs systèmes (donc des entiers) : il faut donc utiliser la *fonction os.close(descripteur)*.
- Le *sort* ne sert à rien ici (puisqu'on compte les lignes...). Il n'était présent que pour ajouter la création d'un troisième processus pour les besoins de l'exemple...

Les tubes nommés (FIFO)

- Les tubes système n'ont pas de nom, ils ne peuvent donc être partagés qu'entre des *processus ayant un ancêtre commun*. Autrement dit, deux processus distincts ne peuvent pas utiliser un tube commun pour communiquer.
- Une *FIFO* est un tube auquel on a associé un *nom*, il est donc accessible par tous les processus, qui pourront l'utiliser pour communiquer : on les appelle aussi *tubes nommés*.
- En Python, on crée une FIFO en appelant la fonction *os.mkfifo()* auquel on précise le nom du tube, puis on ouvre cette FIFO comme un fichier classique, avec *open()* en mode lecture ou écriture.
- On lit ou on écrit ensuite dans cette FIFO exactement comme pour un tube système (la lecture est destructrice et peut être bloquante s'il n'y a rien dans le tube).

Exemple minimal d'utilisation d'une FIFO

Communication entre deux processus distincts

```
# ------------------------------------------------------------
# Contenu du fichier emetteur.py
import os

nom = "/tmp/ma_fifo"
os.mkfifo(nom)   # c'est l'émetteur qui commence à parler, donc c'est lui
                        # qui crée la fifo

with open(nom, "w") as fifo:   # L'emetteur enverra des données dans la fifo
  fifo.write("Coucou...)
# fermeture automatique de fifo

# ------------------------------------------------------------
# Contenu du fichier recepteur.py
import os

nom = "/tmp/ma_fifo"
with open(nom, "r") as fifo:  # Le récepteur lira des données dans la fifo
  for ligne in fifo:
     print("Reçu : " + ligne)
# fermeture automatique de la fifo

os.remove(nom)       # Le récepteur et l'émetteur n'ont plus besoin de la fifo...
```

Conclusion

- Un processus Unix classique (créé par l'appel *fork()*) contient :
  - un code en cours d'exécution
  - un ensemble de ressources en mémoire, comme sa table des descripteurs de fichiers et un espace adressable pour ses données.
- Chaque processus a ses propres données, sa propre mémoire et ne les partage pas avec les autres (sauf via des mécanismes spéciaux).
- Communication entre processus difficile.
- La création d'un processus est donc relativement coûteuse puisque que le processus fils est créé à l'image de son père : initialement, le fils est une copie du père. C'est la raison pour laquelle on les appelle aussi **Processus lourds** (par opposition aux processus légers que sont les threads).