



Programmation système

Introduction au parallélisme

Sommaire

- Motivations pour le parallélisme
- Les signaux
- Processus (lourds): définition et création
- Les tubes de communications
- Les processus légers

Motivations pour le parallélisme

- Il est fort probable que la vitesse des processeurs ne continuera pas à augmenter à la cadence actuelle (~double tous les 18 mois) après 2005.
- Certaines applications au parallélisme naturel voient leurs performances multipliées par des facteurs entre 2 et 500, ces facteurs sont largement supérieurs à ce qu'on peut espérer de l'augmentation de la vitesse des processeurs pour les années à venir.
- Si la vitesse des processeurs double tous les 18 mois, ce n'est malheureusement pas le cas de la mémoire et de la rapidité d'accès à celle-ci. Le parallélisme est également un moyen de traiter de grosses masses de données et d'augmenter les performances d'accès à celles-ci (meilleure utilisation des mémoires à accès rapide)
- la vitesse des réseaux augmentant, c'est aussi la possibilité d'utiliser les ressources réparties dans un laboratoire, sur un site et actuellement sur plusieurs sites distants.

Motivations pour le parallélisme

- Matérielles
 - Utilisation de calculateurs multiprocesseurs à mémoire partagée
 - Plusieurs processeurs ayant accès à une mémoire principale commune
 - Nécessité sur un ordinateur monoprocesseur
 - Mettre à profit les temps de blocage
- Logicielles
 - Des parties de programmes sont relativement indépendantes et peuvent être exécutées en même temps
- Logiques
 - Multi activités mises en évidence dans la conception

Introduction

- Un signal est émis à destination d'un processus afin de lui notifier l'occurrence d'un événement le concernant.
- Le processus doit alors traiter le signal le plus rapidement possible.
- Le signal peut être émis alors que le processus n'est pas actif. Ce processus devra être actif pour traiter le signal.
- Un signal qui a été émis mais n'a pas encore été traité par le processus est dit "bloqué" (pending).
- Le concept de signal est à rapprocher de:
 - le concept d'interruption du niveau matériel
 - le concept d'exception du niveau langage
- Dans tous les cas, il s'agit de la survenue d'un événement conduisant à un traitement spécifique:
 - le sous-programme d'interruption
 - le traite-exception

Principes fondamentaux

- Un processus peut associer un traitement spécifique à exécuter lorsque le signal doit être traité: le handler.
- Dans le cas où aucun traitement n'est prévu par l'utilisateur, un traitement standard est prévu par le système pour traiter le signal.
- Un processus peut ignorer un signal. Le déclenchement de ce signal n'a alors aucun effet sur le processus.
 - Un tel signal est dit « ignoré ».
- Un processus peut bloquer un signal. Son traitement est alors retardé.
 - Un signal qui s'est produit mais n'a pas encore été traité est dit « bloqué ».
- L'ensemble des signaux susceptibles d'être bloqués est défini par un masque de signaux.
 - Un tel signal est dit « masqué ».
- Lorsque le signal se produit alors que le processus n'est pas actif, son traitement sera effectué dès que le processus deviendra actif, avant même de reprendre l'exécution du code interrompu.

Interception et Traitement

- Afin de prendre en compte un signal, il est nécessaire d'associer un traitement à ce signal: le traitement sera effectué lors de la réception du signal associé ou dès la reprise d'activité si le signal arrive alors que le processus n'est pas actif
- Le traitement associé peut être:
 - le traitement par défaut du signal (qui conduit très souvent à la destruction du processus)
 - l'ignorance du signal
 - un traitement spécifique défini dans le programme

Quelques signaux

- Sous Unix, les signaux sont définis dans le fichier `sys/signal.h`, qui associe à chaque signal (un entier positif) une constante symbolique commençant par `SIG`.
- Exemples de signaux
 - `SIGINT ()` est le signal d'interruption : il est envoyé à tous les processus créés à partir d'un terminal lorsque l'on fait `Ctrl-C` dans celui-ci.
 - `SIGQUIT ()` a le même effet que `SIGINT` mais crée en plus un fichier core que l'on pourra analyser avec un debugger. Il est envoyé lorsque l'on fait `break` ou `Ctrl-X`.
 - `SIGKILL ()` est l'arme fatale pour détruire un processus (on ne peut pas se protéger contre ce signal). Le processus cesse immédiatement son exécution en laissant tout en l'état...
 - `SIGBUS ()` est produit lorsque des pointeurs ont été mal initialisés
 - `SIGALARM ()` est un signal expédié à l'expiration d'un délai (par `sleep()`, notamment)
 - `SIGCHLD ()` est envoyé à un processus père quand l'un de ses fils se termine.
 - `SIGUSR1 ()` et `SIGUSR2 ()` n'ont pas de rôles prédéfinis. Ils sont mis à la disposition du programmeur pour synchroniser les processus ou pour gérer les entrées dans les fichiers journaux.

Emission d'un signal

- Un processus envoie le signal `SIG` à un autre processus identifié par `PID` en utilisant l'appel système `kill(PID, SIG)` :
 - Si `PID` vaut 0, le signal est envoyé à tous les processus appartenant au même groupe que le processus émetteur.
 - Si `PID` vaut -1, le signal est envoyé à tous les processus, sauf les processus systèmes (ceux ayant l'indicateur `P_SYSTEM`, le processus 1 (init) et celui qui a envoyé le signal). Seul le super-utilisateur peut émettre un tel appel.
 - Si `PID` vaut -N, le signal est envoyé à tous les processus appartenant au groupe N.
 - Dans tous les autres cas, le signal est envoyé au processus ayant le `PID` indiqué.
 - Si `SIG` vaut 0, aucun signal n'est envoyé : cela permet de tester l'existence de `PID` (l'appel échouera si le processus visé n'existe pas et `errno` vaudra `ESRCH`, il ne fera rien si le processus existe...)

En Python : émission

- Les processus sont gérés via le module `os` qui fournit une interface aux appels systèmes.
- Ce module contient la fonction `kill` qui prend en paramètre un pid et un signal exprimé par une constante du module `signal` ou par un entier positif ou nul.
- Les signaux sont eux-mêmes gérés via le module `signal`, qui définit notamment les constantes décrivant les signaux (`signal.SIGTERM`, par exemple).

```
import os
try:
    os.kill(12345, 0)
    print("Le processus 12345 existe")
except ProcessLookupError:
    print("Le processus 12345 n'existe pas")
```

En Python : interception

- Ignorer un signal

```
import os
signal.signal(signal.SIGINT, signal.SIG_IGN)
```

- Revenir au traitement par défaut

```
import os
signal.signal(signal.SIGINT, signal.SIG_DFL)
```

- Associer un gestionnaire de traitement

```
import signal, sys
nb_sigint = 0

def handler(num_sig, frame): # Un handler doit avoir deux params
    global nb_sigint
    nb_sigint += 1
    if nb_sigint == 5: sys.exit(0)

# Programme principal
signal.signal(signal.SIGINT, handler) # association du handler
while True: pass                     # boucle sans fin
```

Définition

Un processus =
Un programme séquentiel
en exécution

- Un même programme exécuté 2 fois produit 2 processus distincts
- La juxtaposition de plusieurs processus permet de décrire des activités qui ne sont pas séquentielles

Définition

Une ressource =
Un élément de l'environnement
utilisé par un ou plusieurs processus

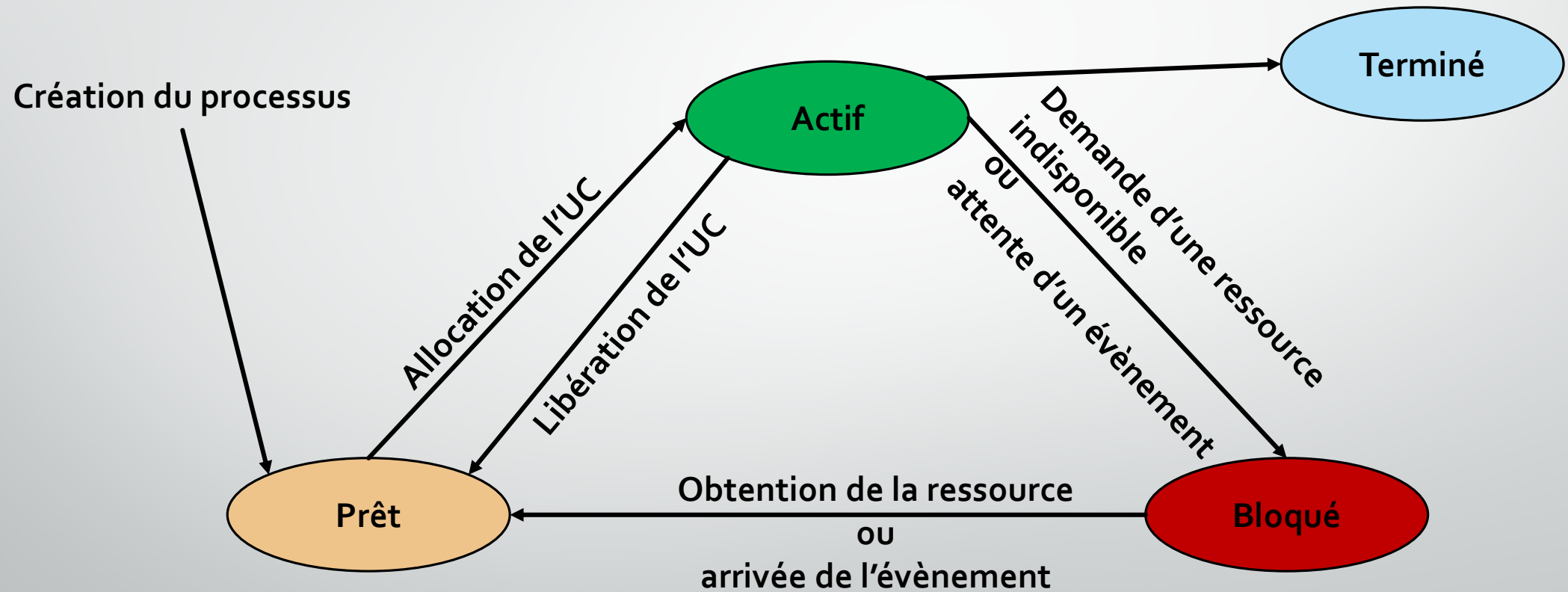
- Exemples de ressources
 - matérielles : imprimante, mémoire
 - Logicielles : fichiers, sémaphore

Taxonomie

- Processus indépendants
 - Aucune relation entre les processus
- Processus coopérants
 - Activités visant un objectif commun
- Processus concurrents
 - Processus relativement indépendants utilisant des ressources communes. Ils doivent se synchroniser et communiquer entre eux afin d'obtenir les ressources dont ils ont besoin.

Le processus parent est le processus responsable de la création du **processus enfant**

Graphe d'états



Graphe d'états : les opérations

- Création
 - L'opération de création comporte
 - la création d'un PCB (Process Control Block),
 - l'allocation des ressources initiales.
 - Le processus est créé dans l'état prêt (ou suspendu).
- Destruction
 - L'opération de destruction entraîne la libération des ressources détenues par le processus (PCB, mémoire, ...)
 - Selon les systèmes, les descendants du processus peuvent:
 - être détruits en même temps que le processus concerné,
 - être conservés et recueillis par un processus d'accueil.

Graphe d'états : les opérations

- Activation
 - Ce qui caractérise un processus prêt par rapport à un processus actif est que ce dernier dispose d'une UC (unité centrale) pour s'exécuter. L'opération d'activation consiste donc à choisir un processus prêt pour chaque UC libre.
 - Les processus sont gérés en file d'attente (FIFO, priorité)
- Prémption
 - L'opération de prémption consiste à retirer l'UC à un processus actif. Le processus passe alors dans l'état prêt.
 - L'UC est alors libre et le système doit effectuer une opération d'activation

Graphe d'états : les opérations

- Blocage et Déblocage
 - L'opération de **blocage** consiste à rendre non éligible un processus actif. La raison peut en être :
 - une ressource demandée par ce processus n'est pas disponible,
 - un événement particulier est attendu par le processus.
 - L'opération de **déblocage** consiste à replacer le processus dans l'état prêt puisque la raison de son blocage n'est plus fondée.

Attention: Un danger de la programmation parallèle est qu'un processus bloqué le reste indéfiniment suite à une erreur de programmation.

Image d'un processus

- A tout processus est associé un ensemble d'informations appelé image. Lorsqu'un programme est "lancé", le système crée un processus et construit son image.
- On y trouve:
 - Un **descripteur** contenant les informations générales au processus (PCB)
 - Un **segment de données privé** au processus:
 - Il contient les constantes, les données statiques et la zone dynamique
 - Un **segment pile privé** au processus:
 - Il permet de gérer les appels de sous-programmes
 - Un **segment de code** partageable entre plusieurs processus:
 - Il contient le code à exécuter par le processus

Pour que le segment de code soit partageable, le code doit être réentrant

Process Control Block (PCB)

- Le PCB constitue le descripteur du processus. On y trouve:
 - Le numéro interne du processus
 - Le numéro du processus père
 - Les variables d'environnement du processus
 - Autres informations
 - Priorité
 - Descriptif des fichiers ouverts
 - ...

Les processus Unix : la commande PS

- La commande **ps** donne des informations sur les processus du système
 - **UID** : numéro propriétaire
 - **PID** : numéro processus
 - **PPID** : numéro du processus père
 - **PRI** : priorité du processus
 - **RSS** : mémoire occupée (en Koctets)
 - **STAT** : état (S: sleeping < 20 s, I: Idle > 20 s, R: running, Z: zombie)



Les
signaux

```
luciole %ps -al
  UID    PID  PPID  CPU  PRI  NI       VSZ    RSS  WCHAN  STAT  TT        TIME COMMAND
   501    239   237    0   31   0    27812   864  -      Ss    p1    0:00.17 -bash
      0    320   239    0   31   0    27292   404  -      R+    p1    0:00.01 ps -al
   501    247   237    0   31   0    27812   892  -      Ss+   p2    0:00.05 -bash
luciole %
```

Les processus Unix : Création

- Un processus UNIX est créé par un autre processus sur un appel de la primitive **fork**
- Seul le processus initial est créé statiquement
- Le nombre de processus évoluant dans un système est variable et peut croître jusqu'à atteindre une valeur limite (dans l'ensemble du système ou pour un utilisateur)
- Le type `pid_t` est un type entier (int ou long)

Les processus Unix : fork()

Important

- Un appel à **fork()** crée un processus fils à l'image de son père et renvoie un entier :
 - -1 si le processus fils n'a pas pu être créé.
 - 0 si on est chez le fils au retour du fork
 - le PID du fils si on est chez le père au retour du fork
- L'ordre de retour du fork est indéfini : on ne sait pas si c'est le code du fils ou du père qui s'exécutera en premier.
- Initialement, le processus créé est un clone de son père, mais a un PID différent.
 - Ses segments de données et de pile contiennent les mêmes valeurs que ceux de son père.
 - Ses descripteurs de fichiers sont identiques à ceux de son père.
 - Son segment de code est identique à celui de son père : il exécute donc aussi les instructions qui suivent l'appel du fork.

Avec Python

Dans Python: utilisation du module **os**

- **os.fork()**
 - Création d'un processus fils
 - retourne 0 si on est chez le fils au retour du fork
 - le PID du fils si on est chez le père au retour du fork
 - `OSError` en cas d'erreur
- **os.getpid()** → retourne le pid du processus courant
- **os.getppid()** → retourne le pid du père

Avec Python: Exemple

```
import os

if os.fork() == 0:
    # On est donc dans le fils...
    print("Je suis le fils {}, mon père est {}".format(os.getpid(), os.getppid()))
else:
    # on est donc dans le père...
    print("Je suis le père {}".format(os.getpid()))

# Attention : le code qui suit est exécuté par le fils et le père
print("Je suis qui ? Réponse : {}".format(os.getpid()))
```

Processus Unix : terminaison

- Normale et prévue
 - Le processus décide lui-même de se terminer
 - Il s'agit de la fin de l'algorithme du processus
 - Le processus informe le système de cette décision
- Anormale ou imprévue
 - Destruction par un autre processus
 - Destruction par le système
 - Suite à une anomalie de fonctionnement du processus
 - Dans le cadre d'une politique de gestion des processus



exit (int raison)



Les signaux

Processus Unix : terminaison

- Actions réalisées par le système
 - Récupérer les ressources détenues par le processus et mettre à jour les statistiques,
 - Fermer automatiquement les fichiers ouverts par le processus,
 - Réveiller et informer le processus père en cas de blocage en attente de la terminaison d'un fils
 - **Orphelins**: le processus père n'attend pas la terminaison de ses fils avant de se terminer. Les processus fils en cours d'exécution deviennent des **orphelins**. Le système les rattache au processus d'accueil (le processus init de pid égal à 1) qui les fait disparaître.
 - **Zombies**: le processus père ne consulte pas la raison de terminaison du processus fils. Le processus fils n'existe plus vraiment, mais il continue d'apparaître dans la table des processus du système.

Processus Unix : Attente de la terminaison

➤ `(pid, status) = os.wait()`

- Suspension de l'exécution du processus parent jusqu'à la terminaison d'un fils ou la réception d'un signal
- Pas de suspension si le fils est déjà terminé et n'a pas été attendu ou s'il n'y a plus de processus fils à attendre
- Les valeurs retournées
 - le Pid du fils terminé
 - l'état (status) du fils terminé (si 0 alors ok, sinon problème)

➤ `(pid, status) = os.waitpid(pidFils, options)`

- attente d'un fils en particulier (`pidFils > 0`), attente de n'importe quel fils (`pidFils = -1`)
- Les valeurs retournées identique à `os.wait()`
- options (`=0` attente normale, `os.WNOHANG` attente non bloquante, etc.)

Python : Exemple zombies

```
import os

if os.fork() == 0:
    print("Je suis le fils :{} ".format(os.getpid()))
    os._exit(0)

print("Je suis le père :{} ".format(os.getpid()))
input("Vérifiez que le fils est zombie avec 'ps -ax' et tapez return pour continuer")
(pid, status) = os.wait()

input("Vérifiez que le fils n'est plus zombie avec 'ps -ax' et tapez return pour continuer")
if status == 0:
    print("le fils s'est terminé normalement")
else:
    print("le fils s'est mal terminé")
    exit(0)
```

Python : Exemple orphelins

```
import os, time

pid_fils = os.fork()
if pid_fils == 0:
    print("Je suis le fils : {}".format(os.getpid()))
    time.sleep(20)
    os._exit(0)

print("Je suis le père : {}".format(os.getpid()))
print("Vous avez 20 secondes pour vérifier que le fils est orphelin avec 'ps -ef|grep {}'.format(pid_fils))

exit(0) # Le père se termine avant le fils...
```

Python : Ni zombies, ni orphelins

```
import os, signal
```

```
if os.fork() == 0:  
    print("Je suis le fils : {}".format(os.getpid()))  
    os._exit(0)
```

```
# le processus père détourne SIGCHLD, envoyé par un processus fils lorsqu'il se termine.  
signal.signal(signal.SIGCHLD, lambda sig, frame: os.wait())
```

```
print("Je suis le père : {}".format(os.getpid()))  
print("Mon fils n'est pas un zombie (vérifier avec la commande 'ps -ax')")  
input("Tapez sur Entrée pour continuer...")
```

```
# Ce n'est pas parce qu'on chasse les zombies qu'il faut faire des orphelins...
```

```
# Le père ne doit pas se terminer avant ses fils !
```

```
try:
```

```
    os.wait()
```

```
except ChildProcessError:
```

```
    pass
```

```
finally:
```

```
    exit(0)
```

```
# Le fils s'était déjà terminé : wait() a échoué
```

```
# Dans tous les cas, on termine le père
```




Les signaux

Commutation d'image

Après un appel à *fork*, le segment de code du processus fils contient la même chose que le segment de code de son père → ils exécutent donc le même programme

Est-il possible de créer un processus capable de faire autre chose?



Principe de la commutation d'image:
Un processus peut dynamiquement modifier l'image qu'il exécute

Commutation d'image: Exec

- Les segments code et données sont modifiés à partir des informations présentes dans le fichier exécutable spécifié en paramètre de l'appel
- La pile est vidée
- Les autres caractéristiques du processus ne sont pas modifiées ((p)pid, fichiers ouverts, redirections etc.)
- L'exécution va commencer dans le nouveau code à la première instruction exécutable
- En principe, si la commutation d'image a correctement fonctionné, l'instruction qui suit l'appel de la primitive exec n'est pas exécutée
- Les paramètres à transmettre à la fonction peuvent être spécifiés
 - sous forme de liste (execl)
 - sous la forme d'un tableau (execv)

Commutation d'image : Exercice

- Donner l'algorithme du programme « *runfic* » qui compile un fichier source en C et exécute le programme résultant s'il n'y a pas eu d'erreur. Si la compilation échoue, *runfic* appelle la commande *more* pour visualiser les messages d'erreur, puis charge le fichier source dans l'éditeur *vi* pour qu'on puisse le corriger.

Commutation d'image: Exemple

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

if not os.path.isfile(sys.argv[1]):    # Teste si le fichier source existe
    print("{} n'existe pas".format(sys.argv[1]), 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("gcc", "gcc", sys.argv[1])

(pid, status) = os.wait() # Le père attend la fin de gcc
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("vi", "vi", sys.argv[1]) # Puis lance vi
else: # la compilation s'était bien passée
    os.execlp("./a.out", "a.out")
```

Communication entre processus

Ecrire une commande qui permet 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 ?



- ✓ Le père ouvre un fichier, le fils écrit dedans
 - pb de synchro, manipulation de fichiers, etc.
- ✓ Utilisation des **status** mais uniquement des entiers...

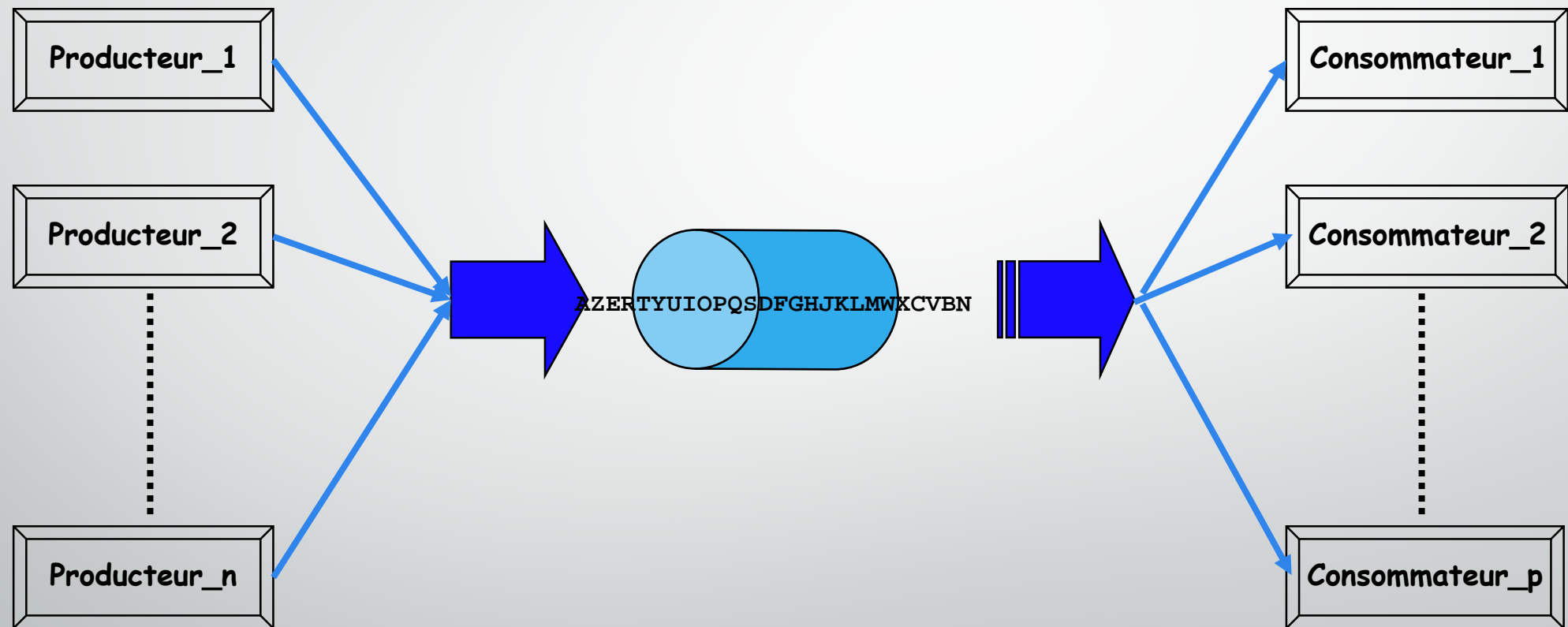
Définition



`ls -l | wc -l`

- 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 SYNTAXIQUEMENT IDENTIQUES aux opérations pour manipuler un fichier classique (à l'exception de la création et de l'ouverture): 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
 - A l'inverse d'une lecture classique dans un fichier, une lecture dans un tube entraîne la suppression des informations lues. (Il s'agit d'une extraction).
 - 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.

Illustration



Synchronisation

- 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 :**
 - Une opération qui n'est pas possible est BLOQUANTE pour le processus appelant.
 - Par exemple :
 - La lecture est bloquante si le tube est vide.
 - L'écriture est bloquante si le tube est plein.

Création

- `os.pipe (int tube[2])` crée un tube de communication et renvoie 0 si Ok, -1 sinon
- Cette fonction alloue deux descripteurs de fichiers et place leurs numéros dans le tableau *tube*.
 - Le premier descripteur, *tube[0]*, permet de *lire* dans le tube.
 - Le deuxième, *tube[1]* permet d'y *écrire*.
- Les accès à un tube sont identiques à ceux des fichiers classiques, sauf qu'il y a *synchronisation* entre l'écriture et la lecture : une lecture peut donc être bloquante si le tube ne contient pas les données demandées.

- 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.

Fermeture



Important

- `os.close()` permet la fermeture des deux extrémités du tube (pas de `close()`)
- Du point de vue d'un processus consommateur qui lit le tube, la communication prend fin lorsque plus aucun processus n'a le tube ouvert en écriture.
- 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.

- 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.

Illustration : Bonne pratique

- La création du tube

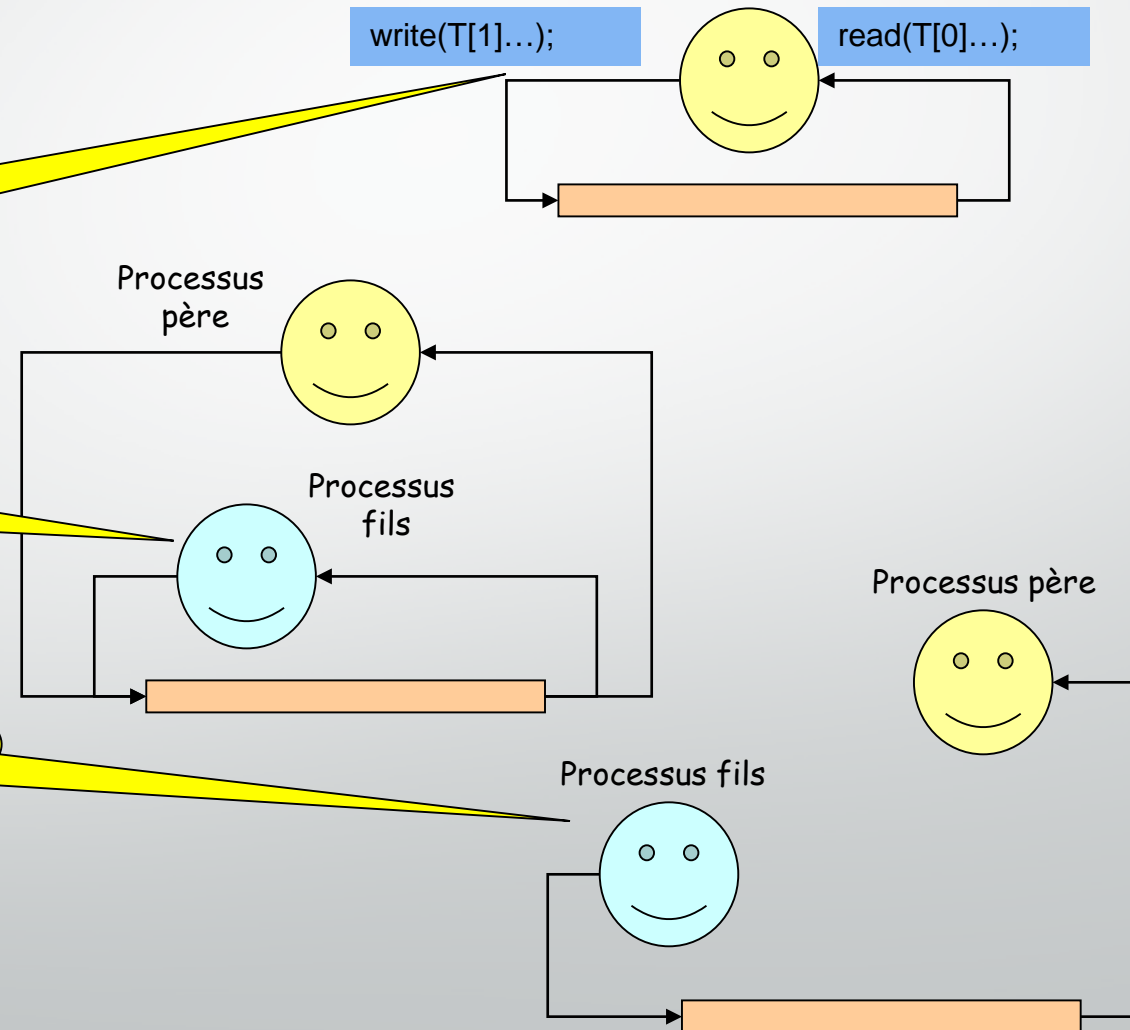
`os.pipe()`

- La création du fils

`os.fork()`

- L'orientation du tube

Fermetures de l'entrée
non utilisée
par chacun des processus



Exercice

- Donner l'algorithme d'une application dans laquelle le processus père lit les caractères au clavier et le processus fils affiche ces mêmes caractères à l'écran de l'utilisateur.
- Donner le schéma de communication ainsi que l'algorithme d'une application qui exécute la commande:

```
grep regexp fic1 | sort -uf | wc -l
```

Exemple: grepsortwc.py

```
grep regexp fic1 | sort -uf | wc -l
```

```
import os, sys
if len(sys.argv) != 3:
    print("Usage: {} regex fic".format(sys.argv[0]))
    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")

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")
```

Les tubes nommés

- Les tubes nommés associent les propriétés des tubes avec certaines propriétés des fichiers
 - Ils sont de taille limitée
 - Les lectures et écritures sont effectuées suivant un ordre FIFO
 - Les lectures sont destructrices
 - Ils sont utilisables par des processus qui ne sont pas obligatoirement de la même famille.
- 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 comme pour un tube système (la lecture peut être bloquante).

Exemple

```
# -----  
# 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'émetteur 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



Processus Lourds