Programmation système

Elsy Kaddoum & Éric Jacoboni

19 janvier 2022

Université de Toulouse-Le Mirail

Sommaire

Introduction

Motivations pour le parallélisme

- 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 calculateur 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.

Les processus lourds

Processus : définition

Définition

Un processus = un programme en cours d'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.

Processus : ressources

Ressource d'un processus

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

- Exemples de ressources :

  Matérielles : imprimante, mémoire.

  Logicielles : fichiers, sémaphores.

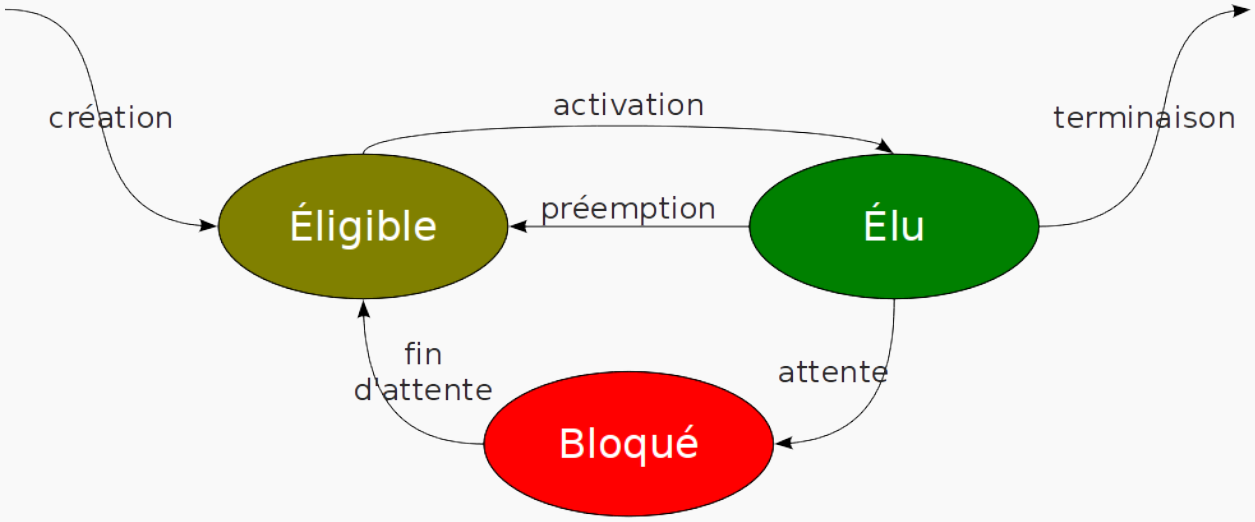
Processus : taxonomie

- Processus *indépendants* : Aucune relation entre les processus.
- Processus *coopératifs* : 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.
- Ne pas confondre *processus parallèles* et *processus concurrents* !

Hiérarchie de processus

Le **processus parent** *est le processus responsable de la création du* **processus fils**.

Graphe d'états d'un processus



- Le passage de l'état « Élu » à l'état « Éligible » correspond à une libération de l'UC.
- Le passage de l'état « Élu » à l'état « Bloqué » correspond à l'attente d'une ressource non disponible ou d'un événement (saisie au clavier, par exemple).

Opérations

Création :

L'opération de création d'un processus comporte :

- La création de son PCB (*Process Control Block*)
- L'allocation des ressources initiales.

Destruction :

Libération des ressources détenues par le processus (PCB, mémoire, etc.). Selon le système, les fils du processus peuvent :

- Être détruits en même temps que leur père.
- Être conservés et recueillis par un processus d'accueil.

Opérations

Activation :

- Ce qui caractérise un processus éligible (prêt) par rapport à un processus élu (actif) est que ce dernier dispose d'une UC (unité centrale) pour s'exécuter. L'opération d'activation consiste donc à choisir un processus éligible (prêt) pour chaque UC libre.
- Les processus sont gérés à l'aide d'une file d'attente (FIFO, priorité)

Préemption :

- L'opération de préemption consiste à retirer l'UC à un processus élu (actif). Le processus repasse alors dans l'état éligible (prêt).
- L'UC est alors libre et le système doit effectuer une opération d'activation pour activer un nouveau processus éligible.

Opérations

Blocage

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.

Déblocage

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

- À 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, contenant les constantes, les données statiques et la zone dynamique.

  Un segment pile privé au processus permettant de gérer les appels de sous-programmes.

  Un segment de code partageable entre plusieurs processus contenant le code à exécuter par le processus.

Pour que le segment de code soit partageable, le code doit être **réentrant** *(utilisable simultanément par plusieurs tâches, afin d'éviter sa duplication en mémoire).*

Le Process Control Block

Le PCB constitue la pièce d'identité du processus. Il contient notamment :

- Le numéro interne du processus (son PID)
- Le numéro du processus père (son PPID)
- Les variables d'environnement du processus
- D'autres informations, comme sa priorité, les descripteurs de fichiers ouverts, etc.

Outils du shell pour gérer les processus et les signaux

Les commandes suivantes permettent de gérer les processus depuis une session shell (pour plus d'informations, consultez leurs pages de manuel respectives) :

- *ps* affiche la liste des processus en cours d'exécution.
- *kill* envoie un signal au processus de PID indiqué.
- *killall* envoie un signal à tous les processus exécutant un programme donné.
- *pgrep* recherche les processus correspondant à un ou plusieurs critères donnés.
- *pkill* envoie un signal à tous les processus correspondant à un ou plusieurs critères donnés.
- *pstree* affiche une représentation arborescente des processus.
- *top* et ses variantes affiche une représentation « live » des processus en cours d'exécution et permet de leur envoyer des signaux.

Les signaux

- Les signaux sont des mécanismes permettant de prendre en compte les *événements* pouvant se produire pendant l'exécution d'un processus.
- Ce sont des *interruptions logicielles asynchrones*.
- Ces événements peuvent être :
  - des événements inattendus (`Ctrl-C`, par exemple) ;
  - des erreurs de programmation (erreurs d'adressage, par exemple) ;
  - des événements prévus et donc programmés.
- Lorsque de tels événements surviennent, le processus reçoit un *signal* et exécute alors un traitement associé à ce signal (un *gestionnaire de signal*) puis, généralement, se termine (mais un processus peut se protéger contre certains signaux, afin de les ignorer s'ils surviennent).

Traitement d'un signal

- Le traitement d'un signal par un processus consiste à annuler le traitement par défaut pour ce signal en fournissant un traitement défini par le programmeur (un *handler*). Ce traitement peut consister :
  - à exécuter du code défini par une fonction ;
  - à se masquer contre un signal (sauf *SIGKILL*) ;
  - à revenir au traitement par défaut.
- 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.
- Sous Unix, les signaux sont définis dans le fichier *sys/signal.h* (ou dans un fichier inclus par celui-ci, *bits/signum.h* sous Linux, par exemple). Ce fichier associe à chaque signal (un entier positif) une constante symbolique commençant par *SIG*.

Liste des 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`.
- *SIGBUS* est produit lorsque des pointeurs ont été mal initialisés (voir TP de C/C++...)
- *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...
- *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.
- *SIGALARM* est un signal expédié à l'expiration d'un délai (par *sleep(3)*, notamment)
- *SIGCHLD* est envoyé à un processus père quand l'un de ses fils se termine.

Émission d'un signal

- Un processus envoie le signal `SIG` à un autre processus identifié par `PID` en utilisant l'appel système *kill(PID, SIG)* (voir *kill(2)*) :
  - 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...)

Émission d'un signal en Python

- En Python, 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).

Envoi d'un signal en Python

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

Traitement des signaux en Python

- Ignorer un signal :

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

- Revenir au traitement par défaut :

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

- Mise en place un gestionnaire de traitement associé au signal :

```
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
```

Les processus Unix : création

- Un processus UNIX est créé par un autre processus via un appel à la primitive *fork*.
- Seul le processus initial (*init*) 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).

L'appel à fork

- Un appel à *fork* crée un processus fils à l'image de son père et renvoie un entier de type *pid_t* (entier sur 32 bits sur les plateformes actuelles) :
  - -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.

La fonction fork de Python

La fonction *fork()* du module *os* de Python fonctionne exactement comme l'appel système que nous venons de décrire :

Exemple de fork

```
import os

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

# Qui exécute le code qui suit ?
print(f"Je suis qui ? Réponse : {os.getpid()}")
```

Terminaison d'un processus

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 (appel à *exit()*)

Anormale ou imprévue :

- Destruction par un autre processus (réception d'un signal)
- Destruction par le système (suite à une anomalie du processus ou dans le cadre d'une politique de gestion des processus).

Terminaison d'un processus

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 s'il était bloqué en attente de la terminaison d'un fils.

Terminaison d'un processus

- Un processus père doit attendre la fin de tous ses fils avant de se terminer, sous peine de créer des *processus orphelins* qui seront automatiquement adoptés par *init* (le processus 1) avant d'être détruits.
- Inversement, si un fils se termine et que son père ne consulte pas la raison pour laquelle il s'est terminé, ce processus fils devient *zombie*. Il n'existe plus vraiment, mais il continue d'apparaître dans la table des processus du système.

Remarques

- Les fonctions système *wait* et *waitpid* bloquent le processus appelant en attente de la terminaison d'un fils (sauf option contraire pour *waitpid*).
- Pour éviter les processus fils orphelins, il suffit que le processus père détourne *SIGCHLD*, qui est envoyé par un processus fils lorsqu'il se termine.

Exemples

Zombies

```
import os

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

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

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

Exemples

Orphelins

```
import os, time

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

print(f"Je suis le père : {os.getpid()}")
print("Vous avez 20 secondes pour vérifier "
      f"que le fils est orphelin avec 'ps -ef|grep {pid_fils}'")
exit(0)  # Le père se termine avant le fils...
```

Exemples

Ni zombies, ni orphelins...

```
import os, signal

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

# Le processus père détourne SIGCHLD pour faire un wait() dès qu'un fils se termine
signal.signal(signal.SIGCHLD, lambda sig, frame: os.wait())

print(f"Je suis le père : {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:   # Le fils s'était déjà terminé : wait() a échoué
    pass
finally:
    exit(0)                 # Dans tous les cas, on termine le père
```

Exercice

- Que produit le code suivant ? (essayez de déduire la réponse avant d'exécuter le code).
- Représentez les créations de processus par un arbre.

```
import os

for i in range(4):
    if os.fork() != 0:
        print(f"PID = {os.getpid()}, i = {i}")
```

Commutation d'image : exec

- 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...
- Pour qu'ils exécutent des codes différents, il faut "commuter l'image du processus" : c'est-à-dire *modifier dynamiquement le code qu'il exécute*.

Commutation d'image : exec

- Les appels de la famille *exec(3)* permettent de charger un segment de code avec le code d'un *autre* programme : il y a donc *écrasement* du code antérieur et il n'est plus possible de revenir à l'ancien code après cet appel. Si les instructions qui suivent l'appel à *exec* sont exécutées, c'est donc que l'appel à échoué.
- Les segments de code et de données du processus sont donc modifiés.
- Son segment de pile est vidé.
- Toutes ses autres caractéristiques sont conservées (PID, descripteurs de fichiers ouverts, redirections, etc).
- L'exécution du nouveau code commence...
- Les paramètres transmis dépendent de la fonction :
  - fournis les uns après les autres (*execlp*)
  - mis dans un tableau passé à la fontion (*execvp*)

Exercice

Donner l'algorithme du programme *runfic* qui compile un fichier source 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.