

RAPPORT DE DOSSIER  
Projet de Système d'Exploitation  
UNIX - Python



« Gestion de processus en batch »

Benjamin Rath  
Laurene Cladt  
Rodolphe Aubry

# Introduction

Le projet de Système d'exploitation pour le 1<sup>er</sup> semestre de M1 consiste au lancement d'un programme pour tout utilisateur à une date ou heure précise de façon cyclique. L'utilisateur voulant lancer un programme devra utiliser une commande **pgcycl** en précisant des paramètres comme la date, l'horaire, la commande ou encore les fichiers de sorties.

L'utilisateur pourra alors spécifier quelles commandes doivent être effectuées périodiquement, connaître celles en cours d'exécution, mais également en supprimer certaines.

Un fichier nommé **fbatch** sera modifié tout au long de l'exécution d'un daemon **gobatch** afin d'avoir connaissance des commandes ou programmes exécutés périodiquement.

Ce projet permet de mettre en application directe les notions vues en cours, du développement Python aux usages **UNIX** tels que les threads ou les files de messages en passant par les sémaphores.

Nous utilisons **Python 2.7** pour ce projet ainsi que différents OS Unix (Elementary OS, Ubuntu et Ubuntu Gnome) afin d'assurer une compatibilité complète avec la bibliothèque **posix\_ipc**.

Ci-dessous se trouve la table des matières donnant le plan général du dossier.

# Table des matières

<b>Dossier Utilisateur</b>	<b>5</b>
Prérequis	5
Introduction à la commande Pgcycl	5
Ajouter une ligne au fichier	5
Lister le contenu du fichier	6
Détruire une ligne du fichier	6
<b>Dossier Concepteur</b>	<b>7</b>
Algorithmes synthétiques	7
Sémaphores	7
Gobatch	7
Pgcycl	8
<b>Dossier de Programmation</b>	<b>10</b>
Bibliothèques	10
OS	10
SYS	10
POSIX_IPC	10
DATETIME	10
TIME	10
RE	11
MULTIPROCESSING	11
SHLEX	11
Application	11
<b>Dossier de Tests</b>	<b>12</b>
pgcycl -l	12
pgcycl -d	12
pgcycl -a	13
<b>Conclusion</b>	<b>15</b>

# Dossier Utilisateur

## 1. Prérequis

Le programme de gestion de processus en batch est développé en python 2.7, sous un système tel qu'Ubuntu cette version est installée par défaut.

Il se peut que la bibliothèque posix\_ipc ne soit pas installée par défaut, pour résoudre ce problème, la commande suivante doit être tapée : **pip install posix\_ipc**

## 2. Introduction à la commande Pgcycl

La commande pgcycl permet à n'importe quel utilisateur d'effectuer des exécutions cycliques de commandes ou de programmes. Grâce à cette commande, il est possible d'automatiser des tâches redondantes et de permettre d'obtenir un résultat cohérent et ainsi de répondre au besoin de l'utilisateur. Un fichier est créé, modifié ou interrogé lors de l'exécution d'une commande pgcycl. Il existe trois paramètres principaux avec cette commande détaillés ci-dessous.

## 3. Ajouter une ligne au fichier

Pour pouvoir exécuter une commande de façon cyclique à une date et/ou un horaire donné, il est nécessaire de remplir un fichier avec pgcycl. La syntaxe est la suivante :

**./pgcycl -a minute heure jourmois mois joursemaine commande sortie erreur**

Où **minute** est la minute à laquelle on souhaite exécuter la commande.

Où **heure** est l'heure à laquelle on souhaite exécuter la commande.

Où **jourmois** est le jour du mois à laquelle on souhaite exécuter la commande.

Où **mois** est le mois à laquelle on souhaite exécuter la commande.

Où **joursemaine** est le jour de la semaine à laquelle on souhaite exécuter la commande.

Où **commande** est la commande que l'on souhaite exécuter.

Où **sor**tie est le fichier de sortie standard.

Où **erreur** est le fichier de sortie des erreurs.

Chaque paramètre horaire doit être valide en fonction de la donnée, voici un tableau des différentes valeurs possibles pour chaque paramètre.

Paramètre	Valeurs possibles
minute	0 à 59
heure	0 à 24
jourmois	1 à 31
mois	1 à 12
joursemaine	0 à 6

Il est possible de mettre “\*” pour un paramètre de date ou horaire que l'on souhaite ignorer.

De plus il n'est pas obligatoire de spécifier tous les paramètres de date ou d'horaire, dans ce cas les paramètres manquants seront remplacés par une étoile.

Les fichiers de sortie standard et d'erreurs permettront de connaître le déroulement des différentes exécutions cycliques.

**Note :** Dans le cas où l'utilisateur voudrait utiliser une commande avec plusieurs arguments, il est nécessaire d'utiliser des doubles quotes. Par exemple pour la commande ls -l, l'utilisateur devra entrer “ls -l” (Apostrophes à l'intérieur et guillemets à l'extérieur).

#### 4. Lister le contenu du fichier

Pour lister le contenu du fichier retenant les commandes et s'exécutant de façon cyclique la syntaxe est la suivante :

**./pgcycl -l**

Cette commande va alors lister le contenu du fichier fbatch et formater celui-ci afin d'afficher proprement les différentes commandes qui seront exécutées cycliquement. Chaque ligne est numérotée ce qui permettra de connaître facilement la ligne que l'on souhaitera détruire.

#### 5. Détruire une ligne du fichier

Afin d'arrêter l'exécution cyclique d'une commande, il est nécessaire de supprimer cette dernière du fichier. La syntaxe est la suivante :

**./pgcycl -d x**

Où **x** est le numéro de la ligne à supprimer.

Cette commande sera alors effacée du fichier et son exécution sera alors arrêtée.

## Dossier Concepteur

### Détail des phases de développement

La première étape a été de créer des processus communicants par le biais de sémaphores. Le processus **gobatch** fait P(S) pour prendre la sémaphore, ce qui le bloque jusqu'à ce que **pgcycl** relâche la sémaphore et lui permet ainsi d'interagir avec le fichier **fbatch**. Pour ce faire, nous avons utilisé la librairie **posix\_ipc**.

Nous avons ensuite implémenté la gestion de l'écriture dans le fichier **fbatch** par **pgcycl** et la lecture par **gobatch**, via la librairie **os**.

Pour permettre aux deux processus de communiquer entre eux, nous avons utilisé les files de messages, avec un système de priorité représentant les différents appels de paramètre **-l**, **-a** et **-d**. Le processus **gobatch** est ainsi en mesure d'agir en fonction du paramètre précisé avec l'appel de **pgcycl**.

Pour gérer efficacement les paramètres passés à **pgcycl**, nous avons utilisé la librairie **re** pour permettre l'utilisation d'expressions régulières. Cela nous a permis de différencier les différents paramètres et de vérifier leur légitimité. Une série de conditions a également été élaborée afin de gérer tous les cas possibles : ainsi, si l'utilisateur ne précise pas un paramètre de temps, il sera remplacé par **\*\***. De même, les paramètres superflus seront tronqués et on s'assure que l'utilisateur a bien renseigné une commande et les fichiers de sortie désirés.

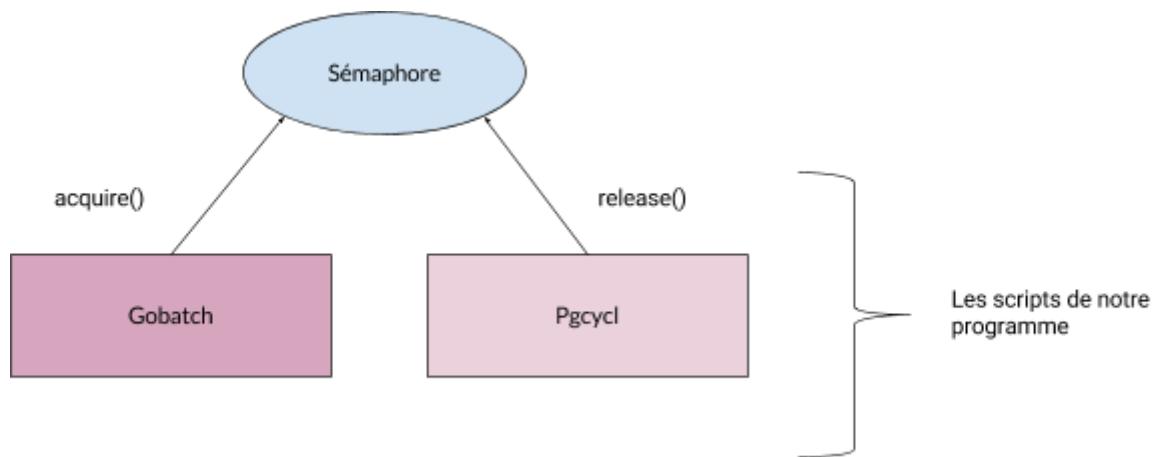
L'étape suivante a été d'implémenter la création et gestion de threads par **gobatch** à chaque exécution de l'appel de **pgcycl** avec le paramètre **-a**. Un thread doit être lancé pour chaque nouvelle ligne écrite dans **fbatch**, et les threads doivent être gardés en mémoire pour pouvoir les terminer par la suite. Nous avons tout d'abord utilisé la librairie **threading**, avant de se tourner vers **multiprocessing** pour permettre l'arrêt des threads lors de la suppression d'une ligne du fichier via le paramètre **-d**.

Nous avons ensuite élaboré une fonction permettant de calculer la date de prochaine exécution d'une commande en fonction des paramètres passés à **pgcycl** et de la date actuelle. Cette fonction nous permet de calculer le nombre de secondes que le thread devra attendre avant de lancer la commande précisée.

La prochaine étape a donc été de gérer l'exécution de la commande, ses éventuels arguments ainsi que les fichiers de sortie standard et erronée. Pour cela, nous avons

également utilisé la librairie **shlex** afin de gérer correctement les espaces et les guillemets.

Pour finaliser le projet, nous nous sommes penchés sur le bon formatage du fichier **fbatch** et de l'affichage des commandes, puis sur la gestion des erreurs par les processus. Ainsi, **gobatch** et **pgcycl** avertissent l'utilisateur si ce dernier entre une instruction erronée, et gèrent les éventuels problèmes au niveau du fichier **fbatch** ou de la date.



## Algorithmes synthétiques

### Sémaphores

**S1** initialisé à 0.

### Gobatch

#### **lectureFichier():**

On ouvre le fichier **fbatch.txt** se trouvant dans le dossier **home** de l'utilisateur en lecture.

Pour chaque ligne du fichier **fbatch.txt**, on fait un “print” de ce qu'elle contient après avoir formaté les données afin qu'elles soient plus lisibles pour l'utilisateur.

#### **creationThread():**

On ouvre le fichier fbatch.txt se trouvant dans le dossier home de l'utilisateur et on récupère la dernière ligne du fichier.

On crée le thread qui va utiliser la fonction à laquelle on passera les arguments.

On démarre le thread en tant que daemon et on l'ajoute à un tableau.

#### **execFonction(commande,stdout,stderr):**

On fait un "print" notifiant de l'exécution de la commande.

On exécute ensuite la commande avec un fichier de sortie et d'erreur précisé.

#### **supprimerThread(message):**

On fait un "print" du tableau de threads.

On arrête le thread correspondant à la ligne supprimée.

On supprime ensuite la référence du thread de la liste.

On affiche de nouveau le tableau de threads pour donner un visuel sur l'action venant de se dérouler.

#### **run():**

Tant que VRAI on fait tourner gobatch (boucle infinie : le processus ne doit pas s'arrêter).

On affiche une ligne signalant le fait que gobatch est bien lancé et qu'il attend une commande.

On attend ensuite une commande venant de pgcycl : tant que rien n'est arrivé le processus est bloqué par **S.acquire()**.

L'action de gobatch dépend ensuite du message reçu si la priorité (remplaçant la notion de type) est :

- 1 : on lance la fonction **lectureFichier()**
- 2 : on lance la fonction **supprimerThread(message)**
- 3: on crée un thread via la fonction **creationThread()**

## Pgcycl

**pgcycl** relâche le sémaphore via **S.release()** après avoir envoyé son message à gobatch.

### **ecritureFichier(msg):**

On ouvre le fichier fbatch.txt en écriture.

On affiche dans le terminal le fait que l'on est en train d'écrire la commande.

On écrit ensuite le paramètre **msg** à la fin du fichier sur une nouvelle ligne.

### **supprimerLigne(line):**

On ouvre le fichier fbatch.txt en lecture et écriture.

On récupère toutes les lignes du fichier.

On réécrit toutes les lignes du fichier sans la ligne **line** qui a été donnée en paramètre.

On tronque alors le reste.

On ferme le fichier.

### **run():**

On crée la file de messages.

On récupère les arguments donnés par l'utilisateur.

On vérifie ensuite le paramètre, si le paramètre donné est :

- “l” , on envoie à gobatch le message “fbatch list” accompagné d'une priorité égale à 1, on affiche ensuite le fait que le message ait bien été envoyé.
- “d” , on lance la fonction **supprimerLigne(x)** puis on envoie à gobatch le message “x” accompagné d'une priorité égale à 2, on affiche ensuite le fait que le message ait bien été envoyé. **x** correspond au nombre donné en argument après -d.
- “a” , On crée un tableau “**cron**” puis on récupère les arguments correspondant au temps. On récupère ensuite le nom de la commande ainsi que les fichiers de sortie standard et d'erreur. On lance ensuite la commande **ecritureFichier(msg)** où **msg** contient cron ainsi que la commande et les fichiers de sortie et d'erreur. Pour terminer on envoie à gobatch le message “fbatch write” accompagné d'une priorité égale à 3, on affiche ensuite le fait que le message ait bien été envoyé.

# Dossier de Programmation

## Bibliothèques

Pour le fonctionnement du script nous utilisons différentes bibliothèques python :

- os
- sys
- posix\_ipc
- datetime
- time
- re
- multiprocessing
- shlex

## OS

La bibliothèque **os** est utilisée pour lancer les commandes qui ont été ajoutées dans le fichier fbatch.txt mais aussi pour partir du répertoire home de l'utilisateur lors de la lecture du fichier fbatch.txt.

## SYS

La bibliothèque **sys** est utilisée pour récupérer la liste des arguments passés.

## POSIX\_IPC

La bibliothèque **posix\_ipc** est utilisée pour la création et gestion des files de messages mais aussi la création de sémaphores

## DATETIME

La bibliothèque **datetime** nous permet de récupérer la date du jour via **datetime** et de travailler dessus via **timedelta**

## TIME

La bibliothèque **time** est utilisée pour endormir les threads pour un temps défini avant la prochaine exécution de la commande via **time.sleep(temps)**

## RE

La bibliothèque **re** est une bibliothèque regex, elle nous permet de faire un tri plus facile dans les arguments et de vérifier si oui ou non ils sont bons selon notre regex via la fonction **re.match**.

## MULTIPROCESSING

La bibliothèque **multiprocessing** est utilisée pour la gestion des “threads” elle nous permet de créer des threads en tant que daemons, de fermer les threads créés etc.

## SHLEX

La bibliothèque **shlex** nous permet de séparer correctement nos variables en prenant en compte les guillemets.

## Application

L’application est composée de deux scripts python :

- gobatch.py (9.14 KB, 228 lignes)
- pgcycl.py (5.64 KB, 141 lignes)

**gobatch.py** sera lancé une fois et travaillera en arrière plan, l’utilisateur n’aura plus jamais besoin d’interagir avec une fois lancé. Il est le centre de contrôle de l’application, c’est lui qui est en charge de créer les threads demandés par **pgcycl.py**, d’arrêter les threads et d’afficher le contenu de fbatch.txt.

**pgcycl.py** est le script qui sera lancé par l’utilisateur pour afficher le contenu de fbatch.txt, ajouter ou supprimer des commandes à lancer périodiquement.

## Dossier de Tests

Nous avons rigoureusement testé nos processus afin de pouvoir gérer les erreurs éventuellement commises par l'utilisateur et s'assurer de leur bon fonctionnement en fonction des paramètres précisés.

**pgcycl -l**

Cas précis testé	Donnée fournie	Résultat attendu	Résultat obtenu
Fonctionnement normal	-l	Liste des lignes contenues dans le fichier fbatch.txt	✓
Arguments en trop	-l suivi de n'importe quoi	Liste des lignes contenues dans le fichier fbatch.txt	✓
Fichier fbatch vide/n'existe pas	-l	L'utilisateur est averti que le fichier fbatch.txt est vide ou n'existe pas	✓

### Exemple d'utilisation

Affichage de la partie **pgcycl** :

```
rodolphe@rodvm:~/pybatch$ ./pgcycl.py -l
pgcycl : creation de la file de message
pgcycl : message fbatch list envoyé
pgcycl: Relachement du semaphore
```

*On lance la commande via pgcycl*

Affichage de la partie **gobatch** :

```
gobatch : Le message reçu est : fbatch list, de priorité 1
Ligne 1 - Execution du programme ls -l chaque * 4 * à 12 H 11
Ligne 2 - Execution du programme pip install --upgrade pip chaque Mardi 3 Avril à 8 H 40
gobatch: En attente d'une commande
```

Gobatch affiche les lignes de fbatch.txt une par une et les formate pour que l'utilisateur puisse les lire facilement

### pgcycl -d

Cas précis testé	Donnée fournie	Résultat attendu	Résultat obtenu
Fonctionnement normal	-d x (où x est la ligne à supprimer ainsi que le thread correspondant))	Supprime la ligne du fichier fbatch.txt et ferme le thread correspondant	✓
Fichier fbatch vide/n'existe pas mais thread existe	-d x (où x est la ligne à supprimer ainsi que le thread correspondant)	Ferme le thread correspondant et avertit l'utilisateur que le fichier fbatch.txt est vide ou n'existe pas	✓
Il y a un fichier fbatch.txt mais le thread n'existe pas	-d x (où x est la ligne à supprimer ainsi que le thread correspondant)	Supprime la ligne du fichier fbatch.txt et averti l'utilisateur que le thread n'existe pas	✓
Il n'y a ni fichier fbatch.txt ni thread	-d x (où x est la ligne à supprimer ainsi que le thread correspondant)	L'utilisateur est averti que le fichier fbatch.txt est vide ou n'existe pas, de même pour le thread.	✓
Ligne qui n'existe pas	-d x (où x est la ligne à supprimer ainsi que le thread correspondant, mais dépasse le nombre de lignes)	L'utilisateur est averti qu'il y a un problème	✓
Pas d'argument saisi ou un argument qui n'est pas un nombre	-d	L'utilisateur est averti qu'il n'a pas saisi d'argument ou que l'argument est mauvais.	✓

## Exemple d'utilisation

Affichage de la partie pgcycl :

```
rodolphe@rodvm:~/pybatch$ ./pgcycl.py -d 1
pgcycl : creation de la file de message
1
pgcycl : message 1 envoyé
pgcycl: Relachement du semaphore
```

On souhaite supprimer la première ligne de fbatch.txt

Affichage de la partie gobatch :

```
gobatch : Le message reçu est : 1, de priorité 2
[<Process(Process-1, started daemon)>]
[]
```

Gobatch affiche les threads du tableau de thread avant et après la suppression

## pgcycl -a

Cas précis testé	Donnée fournie	Résultat attendu	Résultat obtenu
Fonctionnement normal	-a **** commande stdout.txt stderr.txt (où * correspond à une valeur correcte selon ce qui est attendu)	La commande est exécutée à chaque fois que la date et heure correspondent aux données fournies	✓
Pas d'argument fourni	-a	L'utilisateur est averti que le nombre d'arguments est invalidé	✓

Un nombre comme seul argument	-a x (où x est un nombre)	L'utilisateur est averti qu'un ou des paramètres sont invalides, il reçoit la consigne de renseigner au moins une commande à exécuter ainsi qu'une sortie standard et une sortie d'erreurs	
Uniquement la commande, la sortie standard et la sortie d'erreurs	-a commande stdout.txt stderr.txt	Lance la commande toutes les minutes comme aucun paramètre de temps n'a été fourni	
Pas de commande renseignée, uniquement sortie standard et erreurs par exemple	-a stdout.txt stderr.txt	Ne lance rien	Lance la commande -a toutes les minutes comme aucun paramètre de temps n'a été fourni
Une date erronée	-a ***** stdout.txt stderr.txt (où * correspond à une valeur incorrecte au moins une fois selon ce qui est attendu)	L'utilisateur est averti que sa date est erronée et rien n'est lancé, la ligne correspondante dans le fichier fbatch.txt est supprimée	

## Exemple d'utilisation

Affichage de la partie pgcycl :

```
rodolphe@rodvm:~/pybatch$ ./pgcycl.py -a 11 '*' 5 "'ls -l'" stdout.txt stderr.txt
pgcycl : creation de la file de message
gobatch: Ecriture de la commande
pgcycl : message fbatch write envoyé
pgcycl: Relachement du semaphore
```

On veut que la commande "ls -l" soit exécutée tous les 5 du mois à 11 minutes

Affichage de la partie gobatch :

```
gobatch: En attente d'une commande
gobatch : Le message reçu est : fbatch write, de priorité 3
gobatch: En attente d'une commande
Démarrage du Thread pour la commande : ls -l
-----
Minute : 11
Heure : *
Jour du mois : 5
Mois : *
Jour semaine : *
-----
timenow : 2018-01-12 21:55:50.401043
timesched : 2018-02-05 00:11:00.401066
nbr de secondes : 1995310.00002
-----
```

Gobatch affiche les informations sur le thread nouvellement créé et le temps en seconde avant la prochaine exécution de la commande

# Conclusion

L'application permet de lancer des commandes avec une périodicité spécifiée par l'utilisateur. L'utilisateur a accès à 3 commandes à travers le fichier **pgcycl.py** : lister le contenu du fichier **fbatch.txt**, supprimer une ligne de **fbatch.txt** et son thread correspondant, et enfin ajouter une commande à lancer selon les dates déterminées par l'utilisateur comme arguments ainsi qu'un fichier de sortie et d'erreur.

Il est notamment possible de réutiliser des fonctions utilisées dans ces scripts : en effet, on pourrait de nouveau avoir besoin de la gestion des crons, de tests d'arguments en fonction d'expressions régulières, mais aussi la création de threads s'exécutant de façon périodique.

L'architecture du programme nous permet d'ajouter ou retirer des fonctions très facilement afin de l'améliorer.

Afin d'améliorer l'application nous avons songé à ces différentes idées :

- Intégrer une interface graphique afin de faciliter l'accès à une base d'utilisateurs plus orientée novice.
- Lancer gobatch au démarrage du système en tant que daemon, qui à son tour relancerait les threads contenus dans **fbatch.txt**

Ce projet nous a permis de mettre en pratique la notion de thread, de sémaphore et de file de messages et également appliquer les méthodes vues en cours et en TD. Nous avons également pu découvrir la programmation sous contrainte du temps réel.