NSI Terminale

processus : race condition

qkzk
Lycée des Flandres

avril 2020

Race condition

Race condition

Situation crée quand plusieurs processus essayent d'accéder en même temps à une même ressource.

Définition

Condition de compétition :

> une situation caractérisée par un résultat différent selon l'ordre dans lequel agissent les acteurs du système.
> Une situation de compétition peut survenir dès que plusieurs acteurs tentent d'accéder au même moment à une ressource partagée et que l'un d'entre eux est susceptible de modifier son état.
> Les situations de compétition sont des problèmes particulièrement difficiles à identifier et à corriger puisqu'ils ne surviennent que suite à l'ordonnancement particulier et difficilement reproductible d'une séquence d'événements.

Notre exemple

La ressource : Le programme doit accéder à celle-ci (fichier, bdd, imprimante...). Nous **l'écran L'objectif du programme** : écrire trois fois son numéro (pas trouvé plus simple) **Résultat** : si c'est **1** qui écrit en premier on sera dans l'état **1**. Etc.

Principe

On a un processeur (c'est déjà assez compliqué comme ça !).

Il fait UNE CHOSE À LA FOIS : cf assembleur (fetch, read, execute)

- **Comment donner l'illusion du multi tâche ?**
- **Comment contrôler à l'avance un résultat ?**

Exemple

Deux scripts **bash** qui font la même chose : Ils écrivent dans la console le plus vite possible.

Le premier arrivé à trois affichages a gagné

```
c = 0
Tant que c < 3, faire :
    afficher MON NOM
    c = c + 1
Fin tant que
```

Exemple code bash

contenu de **runner_1.sh**

```
#!/bin/sh

c=0 # compteur initialisé à 0
while ((c<3)); do # tant que compteur < 3
  echo "1" # écrit dans la console
  c=$c+1 # on augmente le compteur
done
```

Problème

ils sont lancés en parallèle sur un même fil d'exécution avec :

```
./runner_1.sh & ./runner_2.sh
```

- Le `&` signifie : exécute et continue.
- Ce qui est **avant** le `&` sera lancé et mis en fond de tâche.
- Ce qui est après le `&` sera exécuté dans la foulée.
- **C'est le processeur qui décide quoi faire**
  - Très difficile de savoir dans quel ordre il va exécuter ça !
  - **C'est imprévisible (presque aléatoire)**

Résultats parallèle : aléatoire

Exécution 1	Exécution 2	Exécution 3
1	1	1
2	1	2
1	2	1
2	1	1
2	2	2
1	2	2

Exécution en série

- Il suffit de changer un symbole (`& ---> ;`) pour que l'exécution se fasse **en série**.

```
./runner_1.sh ; ./runner_2.sh
```

Résultat série : dans l'ordre de l'appel

```
Exécution
1
1
1
2
2
2
```

Exécution avec des priorités

La commande `nice` permet de donner plus ou moins de priorité à un processus.

```
nice -10 ./runner_1.sh & ./runner_2.sh
```

Attention : la valeur ici est **10**. Plus elle est élevée moins c'est important.

Ils sont toujours lancés en parallèle mais `runner_1` est **moins important**

Résultat priorité : dans l'ordre des priorités

`runner_1` est exécuté **après** `runner_2`

```
Execution
2
2
2
1
1
1
```

Pour changer la priorité et rendre PLUS important

```
nice --5 ./runner_1.sh & ./runner_2.sh
```

La priorité de 1 est `-5` : il est **plus** important

Résultat priorité : c'est 1 qui gagne

`runner_1` est exécuté **avant** `runner_2`

```
Execution
1
1
1
2
2
2
```

Résultat prévisible

Comment s'assurer que deux processus lancés en même temps aient un résultat prévisible ?

`nice` la bonne idée ?

Pas vraiment...

- Par défaut il faut être super utilisateur pour donner la priorité maximale,
- S'ils sont lancés **en parallèle avec la même priorité** ça redevient imprévisible,

Interblocage

Une des solutions est de faire communiquer les processus entre eux. Ce n'est pas difficile mais ça demande d'avoir déjà compris les bases.

Une autre solution est de les empêcher de travailler s'ils n'ont pas accès à une ressource.

Par exemple en bloquant un dossier le temps de l'exécution d'un des processus.

Exemple

On lance en parallèle avec :

```
./lockfile.sh 1 & ./lockfile.sh 2
```

Les numéros 1 et 2 sont des paramètres passés à chaque processus.

lockfile

```
if ! mkdir /tmp/dossier.lock 2>/dev/null; then
    echo "Le processus tourne déjà !" >&2
    exit 1
fi
```

```
c=0
while (( c<3 )); do
  echo $1
  sleep 1
  c = $c + 1
done
```

locfile traduction

```
essaye de créer le dossier /tmp/dossier.lock
si le dossier /tmp/dossier.lock existe déjà alors
  affiche une erreur et quitte le programme.

sinon
  faire trois fois :
    affiche ton numéro, dors une seconde
  fin boucle
fin si
```

Résultat

```
1
Le processus tourne déjà !
1
1
```

- Le processus 1 s'est lancé (il a crée le dossier), il a fonctionné.
- Le processus 2 s'est lancé (impossible de créer une dossier existant), il a quitté.

Améliorer la méthode

Au prix d'un peu plus de complexité dans le code, on peut s'assurer que les deux processus :

1. soient lancés en parallèle,
2. aient la même priorité,
3. s'exécutent dans un ordre prévisible.

Conclusion

- Les processus sont lancés **par l'utilisateur** mais c'est **le processeur** qui décide de l'ordre dans lequel ils seront exécutés.
- Deux processus lancés en parallèle s'exécutent dans des ordres difficiles à prédire,
- Deux processus lancés en série s'exécutent l'un après l'autre,
- Avec des priorités on peut contrôler... mais il faut savoir à l'avance quelle priorité donner.
- En utilisant une ressource on peut éviter le problème.