

Laboratoire 0 : Infrastructure (Git, Docker, CI/CD)

PAR

Ibrahim BADRI, BADIO2089900

RAPPORT DE LABORATOIRE PRÉSENTÉ À MONSIEUR FABIO PETRILLO DANS LE
CADRE DU COURS *ARCHITECTURE LOGICIELLE* (LOG430-02)

MONTREAL, LE 13 SEPTEMBRE 2025

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

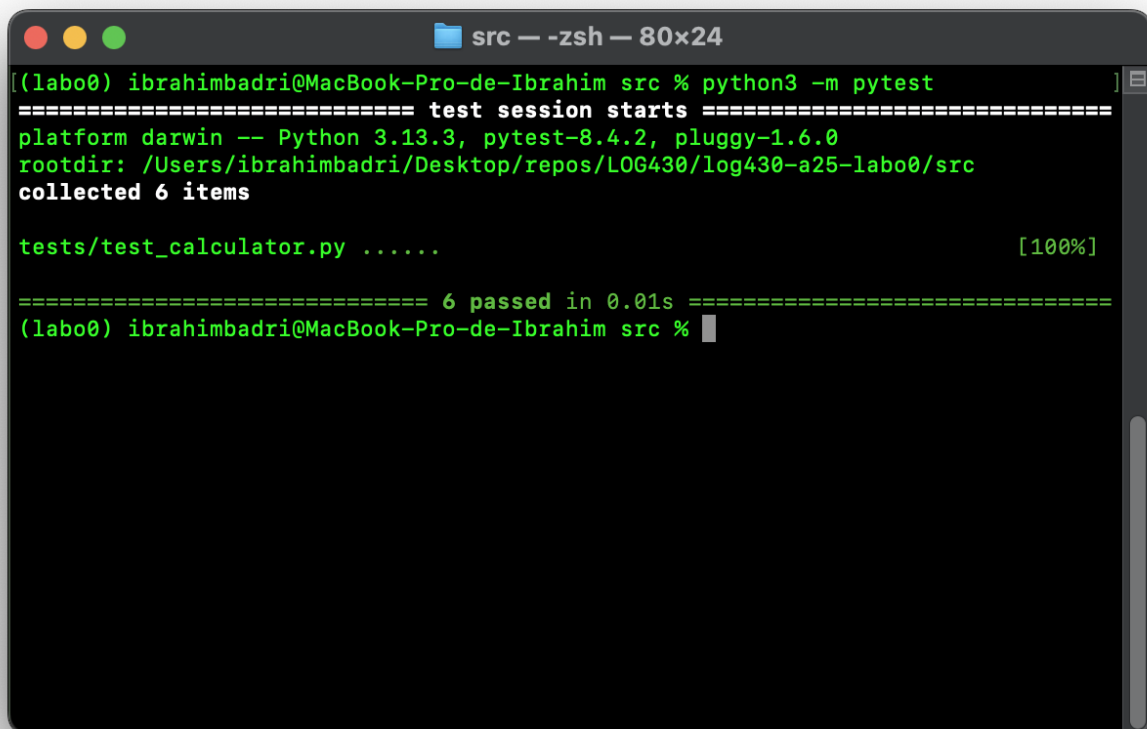
Tables des matières

- [Question 1](#)
- [Question 2](#)
- [Question 3](#)
- [Question 4](#)

Question 1

Si l'un des tests échoue à cause d'un *bug*, comment `pytest` signale-t-il l'erreur et aide-t-il à la localiser ? Rédigez un test qui provoque volontairement une erreur, puis montrez la sortie du terminal obtenue.

Lorsque l'un des tests échoue à cause d'un *bug*, `pytest` l'indique clairement dans le terminal en affichant le nombre total de tests exécutés, réussis et échoués, puis détaille dans la section **FAILURES** le nom de la fonction de test fautive, le fichier et la ligne exacte où l'erreur s'est produite ainsi que le message explicatif qui s'y applique. Dans le cadre de l'exécution de mon test volontairement faux, on obtient le message **DID NOT RAISE <class 'ZeroDivisionError'>** qui découle d'une division d'un chiffre par zéro `(0)`. L'échec provient du fait que le code testé renvoie simplement une chaîne de caractères au lieu de lever l'exception attendue, ce qui provoque le décalage entre le contrat du test et le comportement réel de la fonction.

A terminal window titled 'src — -zsh — 80x24' showing the output of a pytest command. The output indicates a successful test session with 6 items collected and 6 tests passed in 0.01s. The terminal text is as follows:

```
[(labo0) ibrahimbadri@MacBook-Pro-de-Ibrahim src % python3 -m pytest  
===== test session starts =====  
platform darwin -- Python 3.13.3, pytest-8.4.2, pluggy-1.6.0  
rootdir: /Users/ibrahimbadri/Desktop/repos/LOG430/log430-a25-labo0/src  
collected 6 items  
  
tests/test_calculator.py ..... [100%]  
  
===== 6 passed in 0.01s =====  
(labo0) ibrahimbadri@MacBook-Pro-de-Ibrahim src %
```

Figure 2. Résultat de l'exécution des tests exempts d'erreurs

Question 2

Que fait GitLab (GitHub) pendant les étapes de « *Setup repository* » et « *Checkout repository* » ?
Veuillez inclure la sortie du terminal Gitlab CI (GitHub Actions) dans votre réponse.

À l'issue des étapes définies dans le fichier de configuration d'intégration continue (CI), nous obtenons dans un premier temps cette sortie qui correspond à la phase d'initialisation *Set up job* d'un GitHub Actions démarrant d'abord un *runner* éphémère, à l'échéance une machine virtuelle **Ubuntu 24.04 LTS** en prenant le soin de préciser la version du logiciel et l'image utilisée avec tous les outils préinstallés. Il génère ensuite le **GITHUB_TOKEN** et affiche les permissions associées pour permettre aux étapes du *workflow* d'interagir en toute sécurité avec le dépôt GitHub. Le système crée ensuite le répertoire de travail temporaire puis télécharge les actions mentionnées dans le fichier de *workflow* en vérifiant leur intégrité grâce à leurs SHA spécifiques. Une fois ces préparatifs terminés, le *job* nommé **build** peut démarrer et exécuter les étapes de compilation, de tests ou de déploiement définies.

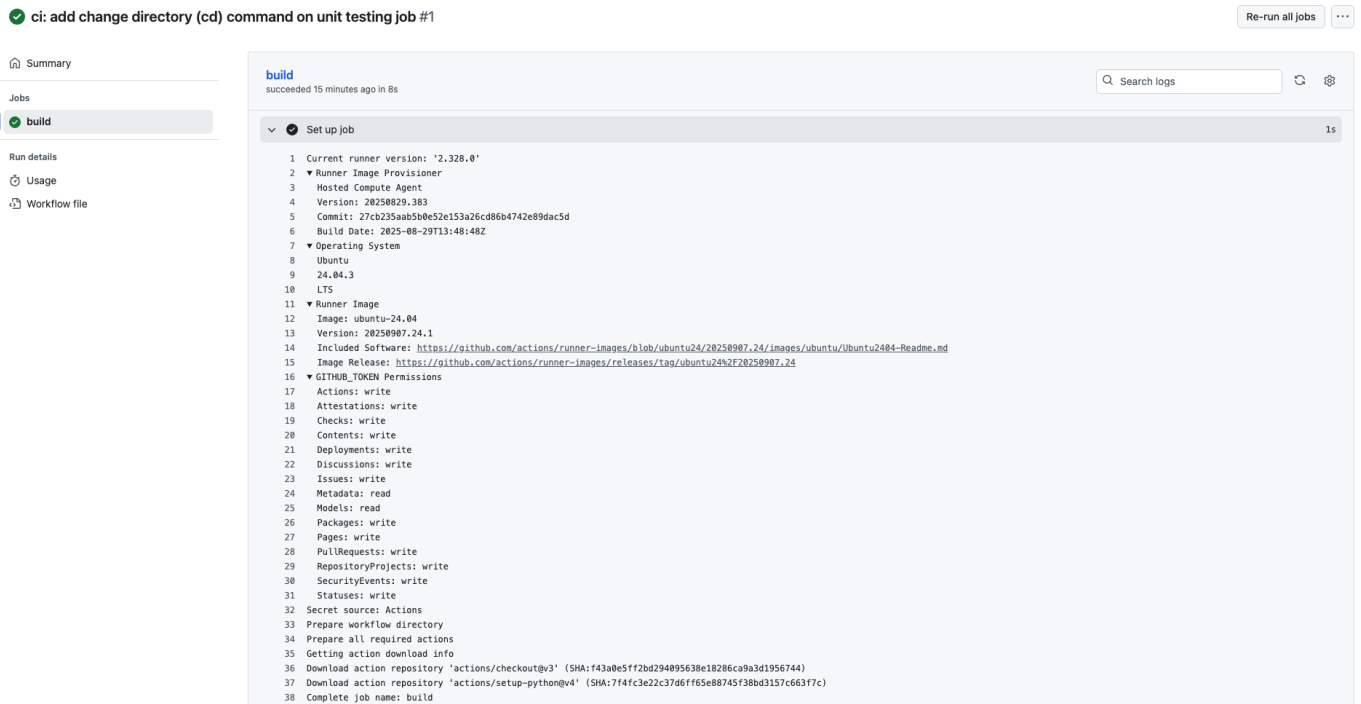


Figure 3. Résultat de sortie de la première étape du build

Dans un second temps, nous obtenons une sortie qui correspond à la deuxième étape intitulée *Checkout* dépôt (*repository*) et qui procède à l'exécution de l'action `actions/checkout@v3`, qui prépare le runner GitHub Actions puis `clone` le dépôt actuel pour le rendre disponible au reste du *workflow*. Par la suite, il procède à l'ajout du répertoire de travail comme *safe directory*, à la suppression de tout le contenu résiduel, à l'initialisation d'un dépôt Git local, à la configuration de l'authentification avec le `GITHUB_TOKEN`, à la récupération du code via un `git fetch` limité au dernier commit et positionne la branche locale sur la révision exacte `3bd502227607c22b629400fea9e68b0a2fafaa33` de manière que le code source complet de ce *commit* soit prêt pour les étapes suivantes.

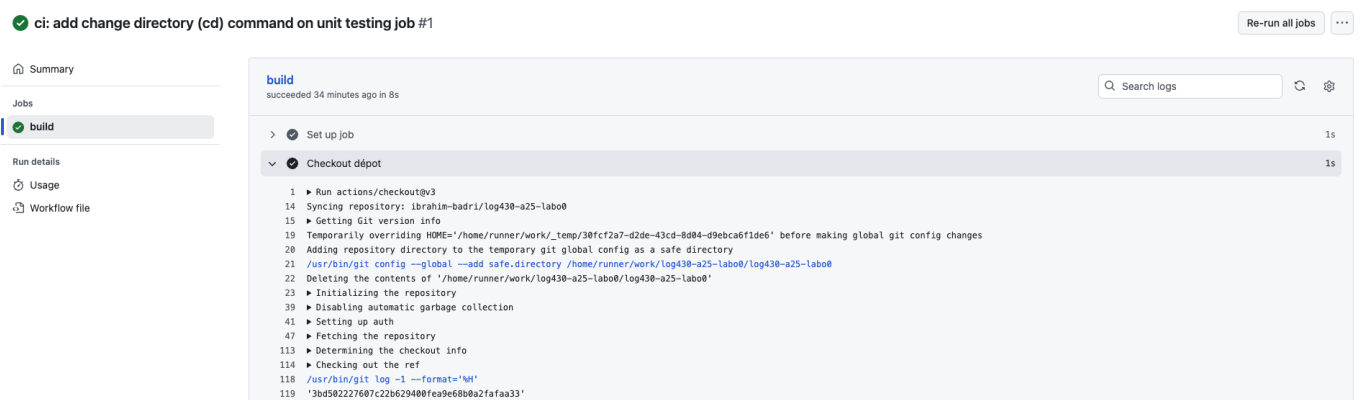


Figure 4. Résultat de sortie de la deuxième étape du build

Question 3

Quel approche et quelles commandes avez-vous exécutées pour automatiser le déploiement continu de l'application dans la machine virtuelle ? Veuillez inclure les sorties du terminal et les scripts Bash dans votre réponse.

Pour mettre en place le déploiement continu (CD) de mon application dans la machine virtuelle, j'ai d'abord élaboré un *pipeline* d'intégration continue (CI) dans GitHub Actions déclenché à chaque **push** ou **pull request** afin d'exécuter automatiquement toutes les étapes de validation du code. Ce *workflow*, défini dans le fichier `.github/workflows/push_pr-build-tests.yml`, télécharge le dépôt avec `actions/checkout@v3`, installe l'environnement Python 3.11 grâce à `actions/setup-python@v4`, met à jour l'outil **pip**, installe l'ensemble des dépendances à partir du fichier `requirements.txt` et installe explicitement **pytest** pour l'exécution des tests. La dernière étape change de répertoire vers `src` puis lance la commande `python3 -m pytest` pour exécuter tous les tests unitaires. Il est d'ailleurs important de noter que l'échec d'un seul test interrompt le *pipeline* et empêche toute suite du processus. Ci-dessous se trouve le contenu intégral du fichier de configuration *Yet Another Markup Language* (YAML) de l'intégration continue.

```
name: CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Repository checkout
        uses: actions/checkout@v3

      - name: Install Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install pytest

      - name: Launch unit tests
        run: |
          cd src
          python3 -m pytest
```

Une fois cette intégration continue validée, j'ai configuré un deuxième workflow de déploiement continu (CD), défini dans `.github/workflows/workflow_run-deploy-to-vm.yml`, déclenché uniquement après la réussite complète du job CI via l'événement `workflow_run`. Ma première approche consistait à utiliser un *runner* GitHub hébergé dans l'espace infonuagique et une connexion SSH non-interactive vers la machine virtuelle provisionné à l'aide de l'utilitaire **sshpass**. Ce *job* installait d'abord **sshpass** puis ouvrait

une session SSH par le biais de la commande `ssh -o StrictHostKeyChecking=no` en lisant l'utilisateur, le mot de passe, le port et l'hôte depuis les GitHub Secrets `USERNAME`, `SSH_PASSWORD`, `SSH_PORT` et `HOST` avant d'exécuter à distance une série de commandes tel que le clonage du dépôt si nécessaire, la mise à jour de la branche main avec `git fetch --all --prune`, `git checkout main` et `git pull --ff-only`, la construction et le lancement des conteneurs via `docker-compose build --no-cache` et `docker-compose up -d --remove-orphans`, le nettoyage des images avec `docker image prune -f` et l'affichage de l'état final des services grâce à `docker-compose ps`. Ci-dessous se trouve l'intégralité du fichier de configuration.

```
name: CD

on:
  workflow_run:
    workflows: ["CI"]
    types:
      - completed

jobs:
  deploy:
    if: ${ github.event.workflow_run.conclusion == 'success' }
    runs-on: ubuntu-latest

    steps:
      - name: Install SSH non-interactive password authentication utility
        run: sudo apt-get update && sudo apt-get install -y sshpass

      - name: Deploy application to remote virtual machine (VM)
        shell: bash
        env:
          SSH_PASSWORD: ${ secrets.SSH_PASSWORD }
          USERNAME: ${ secrets.USERNAME }
          HOST: ${ secrets.HOST }
          SSH_PORT: ${ secrets.SSH_PORT }
          GITHUB_REPOSITORY: ${ github.repository }
        run: |
          sshpass -p "$SSH_PASSWORD" ssh -o StrictHostKeyChecking=no -p
"$SSH_PORT" \
          "$USERNAME@$HOST" "
            set -euo pipefail
            REPOSITORY_NAME=$(basename \"$GITHUB_REPOSITORY\")
            cd ~
            if [ ! -d \"$REPOSITORY_NAME\" ]; then
              git clone https://github.com/$GITHUB_REPOSITORY.git
            fi
            cd \"$REPOSITORY_NAME\"
            git fetch --all --prune
            git checkout main
            git pull --ff-only
            docker-compose build --no-cache
```

```
docker-compose up -d --remove-orphans
docker image prune -f
docker-compose ps
"
```

Cependant, cette méthode a échoué en raison de l'impossibilité d'établir une connexion SSH entrante vers la machine virtuelle celle-ci étant hébergée sur le réseau privé de l'école et donc injoignable depuis Internet. Par conséquent, on constate l'échec déploiement continu via cette approche.



Figure 5. Échec du déploiement continu via connexion SSH

Pour contourner cette limitation, j'ai déployé un *runner* auto-hébergé dit *self-hosted* directement sur la machine virtuelle en suivant les instructions exhaustives énumérées sur GitHub. Ce second *workflow* conserve le même déclencheur `workflow_run` mais s'exécute sur l'étiquette `runs-on: [self-hosted, Linux, X64]`, ce qui lui permet d'exécuter les commandes de déploiement localement sans passer par SSH. Concrètement, le *job* commence par activer l'option `set -euo pipefail` pour que tout échec stoppe immédiatement le processus, puis détermine le nom du dépôt avec `REPOSITORY_NAME=$(basename "$GITHUB_REPOSITORY")`. Il se place dans le répertoire personnel, clone le dépôt s'il n'existe pas déjà, exécute `git fetch --all --prune`, `git checkout main` et `git pull --ff-only` afin d'obtenir la dernière révision de la branche principale, puis reconstruit l'image avec `docker compose build --no-cache` et relance les conteneurs en mode détaché avec `docker compose up -d --remove-orphans`. Enfin, il libère l'espace disque inutile via `docker image prune -f` et affiche l'état actuel des services grâce à `docker compose ps`. Ci-dessous se trouve l'intégralité du fichier de configuration.

```
name: CD
```



```

on:
  workflow_run:
    workflows: ["CI"]
    types:
      - completed

jobs:
  deploy:
    if: ${ github.event.workflow_run.conclusion == 'success' }}
    runs-on: [self-hosted, Linux, X64]

    steps:
      - name: Deploy application to remote virtual machine (VM)
        shell: bash
        env:
          GITHUB_REPOSITORY: ${ github.repository }}
        run: |
          set -euo pipefail
          REPOSITORY_NAME=$(basename "$GITHUB_REPOSITORY")
          cd ~
          if [ ! -d "$REPOSITORY_NAME" ]; then
            git clone https://github.com/$GITHUB_REPOSITORY.git
            "$REPOSITORY_NAME"
          fi
          cd "$REPOSITORY_NAME"
          git fetch --all --prune
          git checkout main
          git pull --ff-only
          docker compose build --no-cache
          docker compose up -d --remove-orphans
          docker image prune -f
          docker compose ps

```

Grâce à cette configuration, chaque **push** validé par les tests unitaires entraîne automatiquement la mise à jour du code et le redémarrage propre des conteneurs sur la machine virtuelle. Il s'avère que cette approche a permis le succès du déploiement continu.

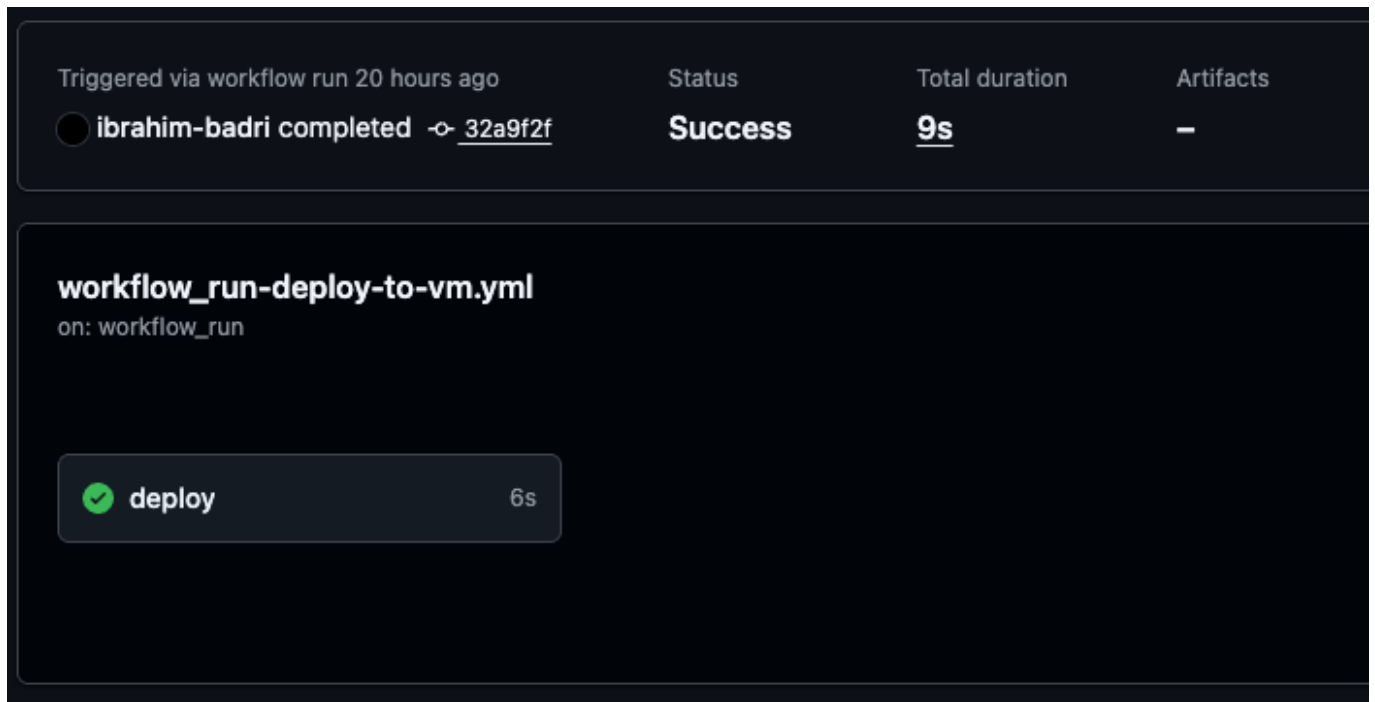


Figure 6. Succès du déploiement continu via auto-hébergement

Pour favoriser la simplicité, la rigueur et la modularité du déploiement, il aurait été judicieux d'extraire certaines parties des commandes directement présentes dans les *jobs* du *workflow* et de les regrouper dans un ou plusieurs scripts Bash dédiés, appelés ensuite par ces *jobs*.

Question 4

Quel type d'informations pouvez-vous obtenir via la commande **top** ? Veuillez inclure la sortie du terminal dans votre réponse.

La commande **top** est un outil de surveillance en temps réel qui permet d'évaluer rapidement l'activité d'un système Linux. Son exécution affiche un tableau qui se met à jour régulièrement et fournit, d'un côté, des informations globales sur la machine telles que la charge moyenne du processeur sur une, cinq et quinze minutes, la durée de fonctionnement depuis le dernier démarrage, l'utilisation du *central processing unit* (CPU) et de la mémoire. D'autre part, elle affiche également la liste des processus en cours. Pour chaque processus, **top** indique notamment l'identifiant (PID), l'utilisateur, la priorité, le pourcentage de CPU et de mémoire utilisés, le temps total d'exécution et la commande qui l'a lancé.

```
src — log430@log430-etudiante-69: ~ — sshpass -p ssh -o StrictHostKeyC...

top - 02:53:36 up 10 days, 5:07, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 132 total, 1 running, 131 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.2 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3622.6 total, 268.1 free, 805.4 used, 2884.0 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used. 2817.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 39482 log430    20   0   12368    5760   3584 R   0.3   0.2   0:00.03 top
     1 root      20   0  22752  13944   9464 S   0.0   0.4   0:40.66 systemd
     2 root      20   0       0       0       0 S   0.0   0.0   0:00.28 kthreadd
     3 root      20   0       0       0       0 S   0.0   0.0   0:00.00 pool_wor+
     4 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
     5 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
     6 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
     7 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
     9 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
    12 root       0 -20       0       0       0 I   0.0   0.0   0:00.00 kworker+
    13 root      20   0       0       0       0 I   0.0   0.0   0:00.00 rcu_tas+
    14 root      20   0       0       0       0 I   0.0   0.0   0:00.00 rcu_tas+
    15 root      20   0       0       0       0 I   0.0   0.0   0:00.00 rcu_tas+
    16 root      20   0       0       0       0 S   0.0   0.0   0:03.40 ksoftir+
    17 root      20   0       0       0       0 I   0.0   0.0   1:32.78 rcu_pre+
    18 root      rt   0       0       0       0 S   0.0   0.0   0:07.94 migrati+
```

Figure 7. Résultat de sortie de la commande top