MINISTÈRE
DE L'ÉDUCATION
NATIONALE
*Liberté*
*Égalité*
*Fraternité*

**EAE INF 3**

**SESSION 2025**

# AGREGATION
# CONCOURS EXTERNE

## Section : INFORMATIQUE

**ÉPREUVE SPÉCIFIQUE SELON L'OPTION CHOISIE :**
- **ÉTUDE DE CAS INFORMATIQUE**
- **FONDEMENTS DE L'INFORMATIQUE**

Durée : 6 heures

*L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.*

*Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.*

*Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.*

**NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.**
**Le fait de rendre une copie blanche est éliminatoire.**

**Tournez la page S.V.P.**

**INFORMATION AUX CANDIDATS**

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

► **Etude de cas informatique** :

| Concours | Section/option | Epreuve | Matière |
|---|---|---|---|
| EAE | 6200A | 103 | 9424 |

► **Fondement de l'informatique** :

| Concours | Section/option | Epreuve | Matière |
|---|---|---|---|
| EAE | 6200A | 103 | 9425 |

*IMPRIMERIE NATIONALE* – 25 0115 – D'après documents fournis

# Système de gestion de projets informatiques

## Préliminaires

L'énoncé s'inspire d'un système de gestion de projets informatiques tel que *SourceForge* ou *GitHub*. Chaque partie s'intéresse à une problématique à résoudre, présente des objectifs concrets et amène une réflexion sur les moyens de les atteindre.

**Attendus.** Il est attendu des candidates et des candidats des réponses construites. Les copies sont évaluées sur la précision, le soin et la clarté de la rédaction. On veillera toujours à rendre saillante la logique générale du code. Les pré-conditions des fonctions demandées n'ont pas besoin d'être vérifiées dans le code mais peuvent être énoncées sous forme de commentaires.

**Dépendances.** Ce sujet contient six parties. Les différentes parties et un grand nombre de leurs questions sont largement indépendants. Il est possible d'aborder les différentes parties dans un ordre quelconque en groupant les questions d'une même partie et en indiquant clairement quelle question est répondue.

**Langages.** Les parties I et V sont à traiter en Python. La partie II est à traiter en SQL. La partie III est à traiter en HTML/CCS et JavaScript. La partie IV est à traiter en C.

## Partie I. Mise en place de chaînes de traitement d'intégration et de déploiement continus (*chaînes CI/CD*)

Une chaîne d'intégration et de déploiement continus (*chaîne CI/CD*) permet de gérer le processus de production de nouvelles versions d'un produit logiciel. La chaîne assure la bonne mise en relation entre les activités des équipes de développement et celles des équipes opérationnelles. Un système d'intégration et de déploiement continus permet la définition et l'exécution *automatique* de chaînes CI/CD.

Une chaîne CI/CD est composée de plusieurs *travaux* où chaque travail définit une suite d'*actions*. Les travaux d'une chaîne sont tous distincts. Ils peuvent être indépendants et s'exécuter en parallèle, ou avoir des dépendances qui imposent qu'un travail attende la terminaison d'un autre. Au sein d'un travail, les actions s'exécutent en séquence l'une après l'autre, chaque action ne débutant qu'après la fin de l'action qui la précède. Un exemple de chaîne CI/CD pour une application informatique est une chaîne qui a plusieurs travaux de construction indépendants (création de différentes versions de l'application), ainsi qu'un travail de packaging qui dépend de ces derniers. Chaque travail de construction est typiquement composé d'une suite d'actions qui mélange compilation et tests.

L'exécution d'une chaîne CI/CD, qui se déclenche suite à un événement ou de manière périodique, consiste en l'exécution des travaux qui la composent.

**Question 1.** Quel est l'intérêt de la mise en place des chaînes CI/CD ? La réponse s'appuiera sur un minimum de deux arguments.

Nous représentons les entités constitutives d'une chaîne CI/CD par trois classes Python : la classe `CICDPipeline`, la classe `Work` et la classe `Action`, qui obéissent aux relations décrites ci-dessus. Chaque instance de l'une des trois classes dispose d'un attribut entier `id` distinct (un identifiant) dont la valeur est fournie par l'usager à la création. Chacune des trois classes est munie d'une méthode `run`. La méthode `run` définit les traitements que la chaîne, le travail ou l'action doit effectuer.

**Question 2.** Proposer un diagramme de classes contenant les trois classes `CICDPipeline`, `Work` et `Action`, ainsi que d'éventuelles classes supplémentaires permettant de factoriser certaines fonctionnalités le cas échéant.

**Question 3.** Quels sont les concepts de la programmation orientée-objet qui permettent à la méthode `run` de s'appliquer à des instances relevant de trois classes différentes ?

**Question 4.** Écrire une implémentation de la classe `Work` en Python.

### Travaux indépendants.

Dans la question 5, nous considérons le cas simplifié des chaînes CI/CD composées de travaux *indépendants*. Une telle chaîne lance l'exécution de ses travaux en parallèle à l'aide de fils d'exécution (ou *threads*) : elle crée un fil d'exécution par travail. L'exécution de la chaîne se termine quand tous les travaux, et donc les fils d'exécution correspondants, se sont terminés.

**Question 5.** Écrire en Python une première version de la classe `CICDPipeline`. Des rappels sur la manipulation de fils d'exécution en Python sont donnés en Annexe A.

### Mise en place des dépendances.

Dans les questions 6 à 13, nous considérons le cas général d'une chaîne où les travaux peuvent avoir des dépendances. L'exécution d'un travail qui a des dépendances ne peut démarrer avant la terminaison de l'exécution de tous les travaux dont il dépend.

**Question 6.** Modifier le code Python de la classe `Work` écrite à la question 4 en adjoignant un troisième argument au constructeur : une liste `dependencies` des travaux dont dépend le travail construit et qui est passée comme attribut à l'objet créé.

Le bon fonctionnement de l'exécution d'une chaîne repose sur la synchronisation des fils d'exécution utilisés pour exécuter les travaux. Dans ce but, une chaîne CI/CD associe un sémaphore distinct à chacun de ses travaux. Quand un travail $t_1$ doit attendre la terminaison d'un autre travail $t_2$, il se bloque sur le sémaphore associé à $t_2$. Il est débloqué par $t_2$ à la fin de l'exécution de $t_2$.

**Question 7.** Modifier le code Python de la classe `CICDPipeline` écrite à la question 5 afin d'y inclure une gestion des sémaphores : définir un attribut, procéder à son initialisation dans le constructeur et fournir un accesseur d'entête `get_semaphore(work_id)`. Des rappels sur la synchronisation de fils d'exécution en Python sont donnés en Annexe B.

Pour que tous les travaux qui attendent un certain travail puissent être débloqués, il est nécessaire de connaître leur nombre. Comme pour les sémaphores, l'information sur le nombre de travaux que chaque travail doit débloquer est à gérer au niveau de la chaîne CI/CD correspondante.

**Question 8.** Dans la continuité de la question 7, modifier la classe `CICDPipeline` pour pouvoir gérer l'association entre un travail de la chaîne et le nombre de travaux qui en dépendent. Rajouter également un accesseur d'entête `get_waiting_for(work_id)`.

**Question 9.** Pour se synchroniser avec les autres travaux, un travail a besoin d'avoir accès aux informations (sémaphores et compteurs) de sa chaîne CI/CD. Modifier la classe `Work` pour y intégrer une référence vers sa chaîne CI/CD.

**Question 10.** Décrire, en langue française, une solution pour la synchronisation de travaux avec dépendances. La mettre en œuvre en réécrivant la méthode `run` de la classe `Work`.

### Interblocages.

**Question 11.** Les dépendances entre travaux étant introduites exclusivement par l'intermédiaire du constructeur de la classe `Work`, selon la spécification de la question 6, démontrer que l'exécution de la méthode `run` de la classe `CICDPipeline` ne connaît pas d'interblocages.

Lors de la définition de grandes chaînes CI/CD avec de nombreux travaux, il peut être commode de rajouter des dépendances au fur et à mesure de la définition de différents travaux.

**Question 12.** Enrichir le code Python de la classe `Work` écrite à la question 6 en adjoignant un transformateur d'entête `set_dependency(prec_work)` de sorte que la commande `work.set_dependency(prec_work)` ajoute le travail `prec_work` comme dépendance devant précéder le travail `work`.

L'utilisation de la méthode `set_dependency` risque d'introduire des interblocages.



**Tournez la page S.V.P.**

**Question 13.** Citer un problème d'algorithmique classique permettant de modéliser la détection d'interblocages. Nommer un algorithme classique le résolvant. Écrire en langage Python un script permettant de détecter si une chaîne fait l'objet d'un interblocage.

**Question 14.** Proposer une modification du code écrit à la question 12 pour informer l'usager de l'introduction d'un interblocage dès que possible.

# Partie II. Modélisation et stockage persistant

**Les projets informatiques.** Les chaînes CI/CD sont définies dans le cadre de *projets informatiques*. Un projet informatique est un projet de développement et de gestion d'un produit logiciel. Un projet dispose d'un identifiant unique au sein du système, ainsi que d'un nom permettant une manipulation plus facile par les équipes de développement et d'intégration. Un projet peut disposer de plusieurs chaînes CI/CD.

**Les utilisateurs.** Les *utilisateurs* qui travaillent sur un projet informatique sont dotés d'un *rôle*. Le rôle délimite les opérations qu'un utilisateur peut effectuer et inclut, entre autres, les rôles de *propriétaire*, *développeur* et *invité*. Un utilisateur peut être impliqué dans plusieurs projets avec des rôles distincts.

**Les opérations de lecture/écriture.** Les utilisateurs peuvent effectuer deux types d'opérations sur les projets : des consultations de projet (*opération de lecture*) et des modifications (*opération d'écriture*). Chaque opération d'un utilisateur sur un projet, qu'elle soit de lecture ou d'écriture, est consignée de manière persistante.

**Question 15.** Élaborer une modélisation relationnelle des données du système en faisant apparaître les tables `User`, `Project`, `CICD_Pipeline`, `Role` et une table de jointure `User_Project`.

**Question 16.** Écrire une requête SQL qui permet de créer la table contenant les données des projets informatiques.

**Question 17.** Écrire une requête SQL qui permet d'identifier les cinq projets ayant le plus d'utilisateurs.

**Question 18.** Écrire une requête SQL qui permet de trier les utilisateurs d'un projet `MyProject` selon le nombre de leurs contributions (modifications).

# Partie III. Application WEB

Dans cette partie nous nous intéressons à l'application web qui permet de travailler avec les projets informatiques.

L'application web doit permettre aux utilisateurs d'accéder aux informations sur leurs projets et leur activité. Chaque utilisateur doit disposer de sa propre adresse pour consulter les informations le concernant. Toutefois, les informations concernant un projet informatique en particulier doivent être accessibles à la même adresse pour tous les utilisateurs.

**Question 19.** Proposer une structuration d'URL pour l'application web qui permette de consulter les informations d'un projet informatique P pour un utilisateur U.

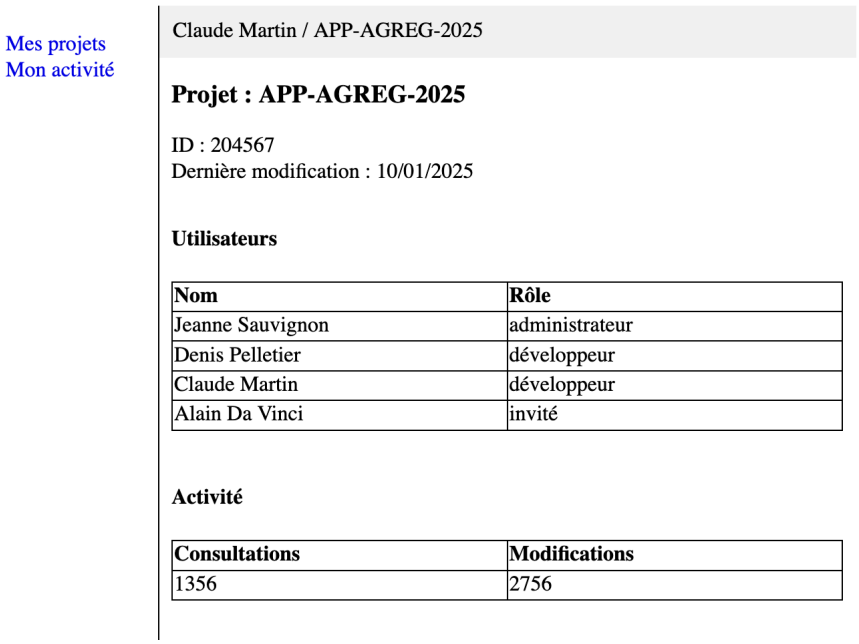


FIGURE 1 – Page web visualisée par l'utilisatrice *Claude Martin* participant au projet informatique `APP-AGREG-2025`. La page donne l'identifiant du projet et la dernière date de modification, avant de lister les utilisateurs impliqués et le nombre de consultations et de modifications. À gauche, un menu permet d'accéder aux informations sur les projets de *Claude Martin* et sur l'historique de ses contributions.

**Question 20.** Donner le code HTML/CSS statique qui correspond à l'esquisse d'IHM décrite dans la Figure 1.

**Question 21.** Enrichir le code HTML/CSS précédent pour proposer une solution supportant la mise-à-jour dynamique, i.e. sans rechargement de la page, du nom du projet, du nombre de consultations et du nombre de modifications.

**Question 22.** Écrire en JavaScript un script qui permet à l'application cliente de recevoir les informations de mise-à-jour du nombre de consultations et du nombre de modifications.



**Tournez la page S.V.P.**

**Question 23.** Quels sont les couches réseau et les protocoles impliqués entre la partie cliente et la partie serveur de l'application web lorsqu'un utilisateur veut consulter la page à l'adresse `https://agregsystem.fr` ?

# Partie IV. Gestion de versions

Dans cette partie nous allons nous pencher sur un des aspects des plus importants d'un système de gestion de projets informatiques : la gestion des versions. La gestion des versions concerne l'ensemble des fichiers qui définissent un produit logiciel : fichiers source, scripts, fichiers de configuration, etc. Ces fichiers sont typiquement organisés de manière arborescente, avec plusieurs répertoires et sous-répertoires.

Pour gérer les différentes versions, le système utilise deux types d'objets : les *objets binaires (blobs)* et les *arbres*. Les *blobs* représentent les fichiers, alors que les *arbres* représentent les répertoires. Pour chaque objet, le système calcule une clé unique d'identification obtenue par hachage du contenu de l'objet.

Nous supposons pour la suite que le système fournit les définitions suivantes, écrites en langage C :

```
// types d'objets de gestion de versions
typedef enum {BLOB_TYPE, TREE_TYPE} object_t;

// structure décrivant un objet de gestion de versions
struct object_info {
  object_t type;
  unsigned long size;
  hash_t oid;
};

// fonction de hachage de contenu
hash_t hash(void* content, unsigned long size);

// structure de blob
struct blob {
    struct object_info header;
    void* content;
};
```

La structure `object_info` est utilisée aussi bien pour les *blobs*, que pour les *arbres*. Elle contient trois champs : le type de l'objet, donné à l'aide du type énuméré `object_t`, la taille en octets du contenu de l'objet, ainsi qu'un identifiant, `oid`, qui correspond au haché de l'objet. L'identifiant est obtenu à l'aide de la fonction `hash`, que nous supposons fournie.

La structure pour représenter un *blob* contient, en plus des informations de type, de taille et d'identifiant, le contenu du fichier correspondant, `content`. Le champ `content` est par conséquent une suite d'octets de taille variable.

**Question 24.** Écrire en C une fonction

```
struct blob* init_blob(void* content, unsigned long size);
```

qui initialise une structure `struct blob` à partir du contenu et de la taille d'un fichier passés en argument. La fonction renvoie la structure initialisée qui doit être allouée par la fonction. En cas d'échec, la fonction renvoie `NULL`.
Des fonctions C qui peuvent vous être utiles pour cette question sont données en Annexe C.

**Arbres.** Pour les *arbres*, le système de gestion de versions s'inspire de la logique des répertoires UNIX. Dans UNIX, le contenu d'un répertoire est une liste comprenant des informations sur des fichiers et des sous-répertoires. Dans le système de gestion de versions, le contenu d'un *arbre* est une liste comprenant des informations sur des blobs et des sous-arbres.

Pour la suite, nous supposons que le contenu d'un arbre est représenté à l'aide d'un tableau de taille prédéfinie. Chaque élément de ce tableau correspond à un objet distinct de l'arbre et contient deux parties : la structure `struct object_info`, qui décrit l'objet, et le nom du fichier ou du répertoire représenté par cet objet.

**Question 25.** En suivant la logique de la structure `struct blob` et les indications ci-dessus, déclarer en C une structure de données `struct tree` pour représenter un arbre. Donner le code C d'une fonction `struct tree* empty_tree()` qui crée un arbre vide. Bien spécifier comment un objet non défini est représenté dans le contenu de l'arbre.

**Question 26.** Donner le code C des deux fonctions suivantes qui permettent de changer le contenu d'un arbre

```
bool tree_add_object(struct tree* tree, char* name, struct object_info* header);
bool tree_remove_object(struct tree* tree, hash_t oid);
```

**Question 27.** Écrire en C une fonction

```
bool diff(void* o1, void* o2);
```

qui vérifie si les deux objets `o1` et `o2` passés en paramètre sont identiques.

**Chemins.** Dans UNIX, l'emplacement d'un fichier dans l'arborescence de fichiers est défini par son *chemin* depuis la racine. UNIX définit des conventions pour représenter un tel chemin sous forme d'une chaîne de caractères. Ainsi, la racine de l'arborescence de fichiers est notée **`/`** et le chemin `/code/test.c` correspond au fichier `test.c` qui se trouve dans le répertoire `code` qui est lui-même un sous-répertoire de la racine.

Le système de gestion de versions sauvegarde tous les objets de blob ou d'arbre dans une table associative entre l'objet et son identifiant. Dans la suite, nous supposerons les fonctions de manipulation de cette table fournies et déclarées comme suit :

```
bool put_hashtable(hash_t oid, void* o);
void* get_hashtable(hash_t oid);
```

La fonction `put_hashtable` lie un identifiant à son objet dont il est le haché (*blob* défini par `struct blob` ou *arbre* défini par `struct tree` ) correspondant. La fonction `get_hashtable` permet d'avoir accès à l'objet identifié par un haché donné.



**Tournez la page S.V.P.**

Nous supposons également fournies les fonctions suivantes :

```
struct tree* get_root();
char** parse_path(char* path);
```

La fonction `get_root` récupère l'objet arbre correspondant au répertoire racine (/). La fonction `parse_path` renvoie la liste des chaînes de caractères qui composent le chemin `path`, terminée par NULL. Ainsi `parse_path("/code/test.c")` renvoie `{"code","test.c",NULL}`.

**Question 28.** Écrire en C une fonction `object_t get_type(char* path)` qui permet de connaître le type de l'objet (*blob* ou *arbre*) accessible à l'emplacement défini par le chemin `path` passé en paramètre. Nous supposons que le paramètre `path` est une chaîne de caractères non vide qui respecte les conventions de nommage des chemins UNIX.

**Question 29.** Déclarer en C une structure de données `struct commit` qui permet de contenir les informations relatives à la définition d'une nouvelle version d'un arbre : l'identifiant de l'arbre en question, la liste des commits précédant le commit en question, le nom de l'utilisateur qui a créé le commit et un commentaire.

## Partie V. Performances à l'exécution

Dans cette partie, nous nous intéressons à la durée d'exécution d'une chaîne CI/CD. Nous supposons que toute action d'un travail CI/CD s'exécute en une unité de temps.

**Question 30.** Enrichir le code Python de la classe `Work`, introduite à la question 4, afin que la commande `work.duration()` renvoie le temps d'exécution du travail `work`.

Le système de gestion de projets informatique exécute les chaînes CI/CD sur des entités dédiées appelées *exécutants*. Les exécutants fournissent les ressources de calcul et de stockage nécessaires aux traitements CI/CD, et permettent une éventuelle exécution parallèle. Pour la suite, nous supposons que les dépendances d'une chaîne CI/CD sont stabilisées et ne peuvent plus changer.

**Question 31.** Enrichir le code Python de la classe `CICDPipeline`, introduite à la question 5, afin que la commande `pipeline.sequential_duration()` renvoie le temps d'exécution de la chaîne `pipeline` dans le cas où un seul fil d'exécution peut être exécuté à la fois.

**Question 32.** Enrichir le code Python de la classe `CICDPipeline`, introduite à la question 5, afin que la commande `pipeline.parallel_duration()` renvoie le temps d'exécution de la chaîne `pipeline` dans le cas où les fils d'exécution peuvent être de nombre illimité. On décrira et on justifiera avec soin l'algorithme employé.

## Partie VI. Autour du CI/CD

**Question 33.** En quoi consiste la phase d'intégration d'un projet informatique ? Quels sont ses liens avec les tests et la gestion de versions ?

**Question 34.** Le déploiement correspond au processus de transfert du produit logiciel modifié vers l'environnement de production. Citer trois risques majeurs pour un déploiement.

**Question 35.** Que signifie le fait que l'intégration et le déploiement soient *continus* ? Quels en sont les bénéfices ?

* *
*



**Tournez la page S.V.P.**

# ANNEXE A : `threading` — Parallélisation à base de Threads

Extrait de la documentation Python 3

**Code source** : `Lib/threading.py`

## Objets `Threads`

La classe `Thread` représente une activité exécutée dans un fil d'exécution distinct. Il existe deux manières de spécifier l'activité : en passant un objet appelable au constructeur ou en redéfinissant la méthode `run()` dans une sous-classe. Aucune autre méthode (à l'exception du constructeur) ne doit remplacée dans une sous-classe. En d'autres termes, remplacez uniquement les méthodes `__init__()` et `run()` de cette classe.

Une fois qu'un objet fil d'exécution est créé, son activité doit être lancée en appelant la méthode `start()` du fil. Ceci invoque la méthode `run()` dans un fil d'exécution séparé.

D'autres fils d'exécution peuvent appeler la méthode `join()` d'un fil. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée soit terminé.

Un fil d'exécution a un nom. Le nom peut être passé au constructeur, et lu ou modifié via l'attribut name.

`class threading.`**`Thread`**`(`*`group=None, target=None, name=None, args=(), kwargs=, *, daemon=None`*`)`

Les arguments sont :

`group` devrait être `None` ;

`target` est l'objet appelable qui doit être invoqué par la méthode `run()`. La valeur par défaut est `None`, ce qui signifie que rien n'est appelé.

`name` est le nom du fil d'exécution. Par défaut, un nom unique est construit de la forme `"Thread-N"` où N est un entier. Si `target` est défini, le nom est de la forme `"Thread-N (target.__name__)"`

`args` est une liste ou un tuple d'arguments pour l'invocation de `target`. La valeur par défaut est ().

`kwargs` est un dictionnaire d'arguments nommés pour l'invocation de l'objet appelable. La valeur par défaut est .

Si la sous-classe réimplémente le constructeur, elle doit s'assurer d'appeler le constructeur de la classe de base (`Thread.__init__()`) avant de faire autre chose au fil d'exécution.

**`start()`**

Lance l'activité du fil d'exécution.

Elle ne doit être appelée qu'une fois par objet de fil. Elle fait en sorte que la méthode `run()` de l'objet soit invoquée dans un fil d'exécution.

**`run()`**

Méthode représentant l'activité du fil d'exécution.

Exemple :

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

`join(timeout=None)`

Attend que le fil d'exécution se termine. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée se termine – soit normalement, soit par une exception non gérée – ou jusqu'à ce que le délai optionnel timeout soit atteint.



**Tournez la page S.V.P.**

# ANNEXE B : `threading` — Synchronisation de Threads en Python

Extrait de la documentation Python 3

**Code source** : `Lib/threading.py`

### Verrous

Un verrou primitif n'appartient pas à un fil d'exécution lorsqu'il est verrouillé. En Python, c'est actuellement la méthode de synchronisation la plus bas-niveau qui soit disponible, implémentée directement par le module d'extension `_thread`.

Un verrou primitif est soit « verrouillé » soit « déverrouillé ». Il est créé dans un état déverrouillé. Il a deux méthodes, `acquire()` et `release()`. Lorsque l'état est déverrouillé, `acquire()` verrouille et se termine immédiatement. Lorsque l'état est verrouillé, `acquire()` bloque jusqu'à ce qu'un appel à `release()` provenant d'un autre fil d'exécution le déverrouille. À ce moment `acquire()` le verrouille à nouveau et rend la main. La méthode `release()` ne doit être appelée que si le verrou est verrouillé, elle le déverrouille alors et se termine immédiatement. Déverrouiller un verrou qui n'est pas verrouillé provoque une `RuntimeError`.

Lorsque plusieurs threads sont bloqués dans `acquire()` en attendant que l'état passe à déverrouillé, un seul thread continue lorsqu'un appel `release()` réinitialise l'état à déverrouillé ; lequel des threads en attente se débloque n'est pas défini et peut varier selon les implémentations.

*class* threading.**Lock**

La classe implémentant des verrous primitifs. Une fois qu'un fil d'exécution a acquis un verrou, tout appel consécutif pour acquérir le verrou est bloquant, jusqu'à libération du verrou. Un verrou peut être libéré par n'importe quel fil d'exécution.

**acquire**(blocking=True, timeout=-1)

Acquiert un verrou, bloquant ou non bloquant.

**release**()

Libérer un verrou. Peut être appelé par n'importe quel fil d'exécution, non seulement par celui qui a acquis le verrou.

### Sémaphores

*class* threading.**Semaphore**(value=1)

Cette classe implémente les sémaphores.

Un sémaphore gère un compteur interne qui est décrémenté à chaque appel `acquire()` et incrémenté à chaque appel `release()`. Le compteur ne peut jamais descendre en dessous de zéro ; lorsque `acquire()` trouve qu'il est nul, il bloque, attendant qu'un autre thread appelle `release()`.

*class* threading.**Semaphore**(value=1)

**acquire**(blocking=True, timeout=None)

Acquiert un sémaphore.

**release**(n=1)

Libérer un sémaphore, incrémente le compteur interne de un.



**Tournez la page S.V.P.**

# ANNEXE C : Fonctions utiles en C

**NAME**

**memcpy** – copy memory area

**SYNOPSIS**

```
#include <string.h>

void *
memcpy(void *restrict dst, const void *restrict src, size_t n);
```

**DESCRIPTION**

The **memcpy()** function copies n bytes from memory area src to memory area dst. If dst and src overlap, behavior is undefined. Applications in which dst and src might overlap should use **memmove(3)** instead.

**RETURN VALUES**

The **memcpy()** function returns the original value of dst.

**NAME**

**memmove** – copy byte string

**SYNOPSIS**

```
#include <string.h>

void *
memmove(void *dst, const void *src, size_t len);
```