Προηγμένες αρχιτεκτονικές υπολογιστών και προγραμματισμός παραλλήλων συστημάτων

LAB-02. Python Threading Library

Python Threads

- Ξεχωριστές ροές εκτέλεσης σε μία διεργασία.
- Το πρόγραμμα μπορεί να εκτελεί δύο ή περισσότερες εργασίες ταυτόχρονα.
- Όμως, στην Python 3 δύο threads ΔΕΝ εκτελούνται ΠΑΝΤΑ πραγματικά ταυτόχρονα.
  - Παρά το ότι δίνουν την αίσθηση για το αντίθετο.
- Οι πολυνηματικές εφαρμογές σε Python ΔΕΝ εγγυώνται ταχύτερη εκτέλεση σε όλες τις περιπτώσεις.

Python Threads και εφαρμογές

- Εφαρμογές που αναμένουν επί μακρόν την εκτέλεση εξωτερικών γεγονότων είναι καλές υποψήφιες για πολυνηματική υλοποίηση.
- Αντίθετα, CPU-intensive προβλήματα ίσως να μην επιταχυνθούν καθόλου.
- Αιτία 1: Python Global Interpreter Lock (GIL).
  - https://realpython.com/python-gil/
- Αιτία 2: Λειτουργικό σύστημα.
- Αιτία 3: CPU.
- Οι πολυνηματικές υλοποιήσεις στη C/C++/Java συχνά παρέχουν καλύτερα αποτελέσματα.

Η βιβλιοθήκη threading της Python

- Παρέχει μία βασική πλατφόρμα υλοποίησης πολυνηματικών εφαρμογών.
- Σε αυτή τη μονάδα το αντικείμενο `Thread` ενθυλακώνει τα threads.
- Παρέχει μία κομψή και καθαρή διεπαφή χρήσης τους.

Εκκίνηση νέου thread

- Η βιβλιοθήκη Python `threading` παρέχει δύο τρόπους εκκίνησης ενός νέου thread:

1. Με κλήση της `start(function, arguments, …)`
   - `function`: περιγράφει και υλοποιεί την εργασία που θα εκτελέσει το thread.
   - `arguments`: παράμετροι για την `function`.
   - επιπλέον ορίσματα.
2. Με ειδική κλάση που:
   - κληρονομεί από την `threading.Thread`.
   - παρακάμπτει (override) τη μέθοδο `run` της `Thread`.

threading.Thread.start()

- Κατά τη δημιουργία του thread περνάμε μία συνάρτηση που περιέχει τον κώδικα που θα εκτελεστεί από το thread.
- Περνάμε επίσης και παραμέτρους στη συνάρτηση.

```
x = threading.Thread(target=thread_function, args=(1,))
x.start()
```

- Στο παράδειγμα, το νέο thread που δημιουργείται θα εκτελέσει την `thread_function` με όρισμα 1.

Συνάρτηση-όρισμα της Thread.start()

- Η `thread_function()` απλά τυπώνει δύο μηνύματα με χρονική διαφορά 2 δευτερολέπτων.

```
import logging
import os
import threading
import time

def thread_function(arg):
    logging.info("Thread %s starting", arg)
    time.sleep(2)
    logging.info("Thread %s finishing", arg)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    logging.info("Main    : all done")
```

```
02:50:44: Main    : before creating thread
02:50:44: Main    : before running thread
02:50:44: Thread 1 starting
02:50:44: Main    : wait for the thread to finish
02:50:44: Main    : all done
02:50:46: Thread 1 finishing
```

Εκκίνηση με κλάση (1)

- Έστω η κλάση `ChefAnna`.
- Για να εκτελεστεί σε ξεχωριστό thread θα πρέπει:
  - να κληρονομεί από την `threading.Thread`.
  - να παρακάμπτει (`override`) τη μέθοδο `run` της `Thread`.

```
class ChefAnna(threading.Thread):
    def __init__(self):
        super(ChefAnna, self).__init__()

    def run(self):
        print('Anna started & waiting for sausage to thaw...')
        time.sleep(5)
        print('Anna is done cutting sausage.')
```

Εκκίνηση με κλάση (2)

- Οι μέθοδοι `init` και `run` είναι οι μοναδικές που επιτρέπεται να παρακαμφθούν.
- Δημιουργία αντικειμένου κλάσης `ChefAnna`.
- Κλήση της μεθόδου `start` του αντικειμένου.

```
anna = ChefAnna()
anna.start()
```

Daemons

- Daemon: Μία διεργασία (process) που εκτελείται διαρκώς στο παρασκήνιο.
- Μοιάζει με server process.
- Αλλά δεν ελέγχεται από το χρήστη.
  - Δεν παρέχει control terminal.
- Τις περισσότερες φορές εκτελεί άλλες (δι)εργασίες (χρονισμένα ή μη), πραγματοποιεί ελέγχους, κλπ.
  - Ορολογία Windows: service.
  - Παραδείγματα: cron daemon, syslogd, sshd, ...

Daemon Threads

- Daemon thread: νήμα που δεν τερματίζεται παρά μόνο όταν τερματιστεί η διεργασία που το δημιούργησε.
  - Ο τερματισμός της διεργασίας τερματίζει αυτόματα και όλα τα daemon threads.
- Αν ένα thread δεν είναι daemon, τότε η διεργασία πρέπει να περιμένει τον τερματισμό του.
- Ορίζεται με την παράμετρο `daemon=True` μέσα στην `Thread.start()`.

```
x = threading.Thread(target=thread_function, args=(1,), daemon=True)
```

Παράδειγμα Daemon Threads

- Έστω ότι στο προηγούμενο παράδειγμα, τα threads δημιουργούνται ως daemons.
- Output:

```
02:53:15: Main    : before creating thread
02:53:15: Main    : before running thread
02:53:15: Thread 1 starting
02:53:15: Main    : wait for the thread to finish
02:53:15: Main    : all done
```

- Λείπει το μήνυμα για τον τερματισμό του thread 1.
- Η `thread_function()` δεν ολοκληρώθηκε.

Αναμονή για ολοκλήρωση - `join` (1)

- Η μέθοδος `join()` επιβάλλει σε ένα thread την αναμονή για την ολοκλήρωση ενός παιδιού – thread.
- Μέχρι να ολοκληρωθεί το παιδί – thread, ο γονέας – thread μπαίνει σε αναμονή.
- Δηλαδή «παγώνει» η εκτέλεση του.
- Blocking

Αναμονή για ολοκλήρωση - join (2)

```
class ChefAnna(threading.Thread):
    def __init__(self):
        super(ChefAnna, self).__init__()

    def run(self):
        print('Anna started & waiting for sausage to thaw...')
        time.sleep(5)
        print('Anna is done cutting sausage.')

# main thread (e.g. ChefGeorge)
if __name__ == '__main__':
    anna = ChefAnna()
    anna.start()
    time.sleep(0.5)
    anna.join()
```

Εδώ το κυρίως thread αναμένει την ολοκλήρωση της συνάρτησης run του παιδιού – thread anna.

Εργασία με πολλαπλά threads

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format,
           level=logging.INFO, datefmt="%H:%M:%S")

    threads = list()
    for index in range(3):
        logging.info("Main: create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()

    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

```
01:45:46: Main    : create and start thread 0.
01:45:46: Thread 0: starting
01:45:46: Main    : create and start thread 1.
01:45:46: Thread 1: starting
01:45:46: Main    : create and start thread 2.
01:45:46: Thread 2: starting
01:45:46: Main    : before joining thread 0.
01:45:49: Thread 0: finishing
01:45:49: Thread 1: finishing
01:45:49: Thread 2: finishing
01:45:49: Main    : thread 0 done
01:45:49: Main    : before joining thread 1.
01:45:49: Main    : thread 1 done
01:45:49: Main    : before joining thread 2.
01:45:49: Main    : thread 2 done
```

Εργασία με πολλαπλά threads (2)

- Στο προηγούμενο παράδειγμα, δημιουργούνται πολλαπλά threads - 3.
- Υπάρχει η ανάγκη αναμονής για τον τερματισμό και των τριών.
- Για το λόγο αυτό, τοποθετούνται στη λίστα `threads`.
- Η αναμονή για τον τερματισμό (`join`) των threads της λίστας γίνεται με loop.
- Ο τερματισμός των threads είναι τυχαίος!
- Δεν γνωρίζουμε εκ των προτέρων ποιο thread θα τελειώσει πρώτο.

Race Conditions

- Μπορούν να συμβούν όταν δύο ή περισσότερα thread αποκτούν πρόσβαση σε κοινόχρηστα δεδομένα ή πόρους. Παρουσιάζεται το πρόβλημα του ποιο thread θα αποκτήσει πρώτο πρόσβαση.
- Μία από τις δυσκολότερες καταστάσεις στον πολυνηματικό προγραμματισμό.
- Η δυσκολία έγκειται στο ότι συμβαίνουν συνήθως τυχαία. Το πρόβλημα δεν είναι συνήθως εμφανές.
- Δύσκολο debugging.
- Οδηγούν σε μπερδεμένα αποτελέσματα.
- Η βασική βιβλιοθήκη παρέχει μηχανισμούς προστασίας από αυτές.

Παράδειγμα

- Ποια είναι η τελική τιμή της `garlic_count`?
- Γιατί είναι διαφορετική κάθε φορά που εκτελείται το πρόγραμμα;

```
import threading
garlic_count = 0

def shopper():
    global garlic_count
    for i in range(1000000):
        garlic_count += 1

if __name__ == '__main__':
    george = threading.Thread(target=shopper)
    anna = threading.Thread(target=shopper)
    george.start()
    anna.start()
    george.join()
    anna.join()
    print('We should buy', garlic_count, 'garlic.')
```

Παράδειγμα (2)

- Τα δύο threads, `George` και `Anna`, διαβάζουν την τιμή της `garlic_count` και την ανανεώνουν κατά 1, 1000000 φορές.
- Όμως, το αποτέλεσμα δεν είναι 2000000 στο τέλος της εκτέλεσης.
- Μάλιστα, είναι διαφορετικό κάθε φορά που εκτελείται το πρόγραμμα.
- Γιατί?

Πείραμα

- Πείραμα:
- Τώρα ο `George` τερματίζει πριν ξεκινήσει η `Anna`.
- Δηλαδή, τα δύο threads δεν εκτελούνται **ποτέ** ταυτόχρονα.
- Τώρα η `garlic_count` παίρνει τη σωστή τιμή: 2000000.

```
if __name__ == '__main__':
    george = threading.Thread(target=shopper)
    anna = threading.Thread(target=shopper)
    george.start()
    george.join()
    anna.start()
    anna.join()
    print('We should buy', garlic_count, 'garlic.')
```

Ταυτόχρονη ανάγνωση/εγγραφή

- Η εξήγηση είναι ότι όταν τα δύο threads εκτελούνται ταυτόχρονα, σε κάποιες τυχαίες περιπτώσεις διαβάζουν ταυτόχρονα την ίδια τιμή για την `garlic_count`.
- Έτσι, όταν την ανανεώνουν κατά 1, επιστρέφουν στην ουσία το ίδιο αποτέλεσμα.
- Π.χ. αν τα δύο threads εκκινήσουν ταυτόχρονα, τότε και τα δύο διαβάζουν `garlic_count=0`.
- Και επιστρέφουν και τα δύο `garlic_count=1`.
- Λάθος: Θα έπρεπε `garlic_count=2`.

Κλείδωμα κοινόχρηστων δεδομένων

- Χρειάζεται μεγάλη προσοχή στη διαχείριση (μεταβολή, ανάγνωση) **κοινόχρηστων** δεδομένων.
- Εάν η εφαρμογή μεταβάλλει κοινόχρηστα δεδομένα, τότε το κάθε thread θα πρέπει να τα κλειδώνει πριν τα τροποποιήσει.
- Με το κλείδωμα, το thread αποκτά αποκλειστική πρόσβαση στα κοινόχρηστα δεδομένα.
- Κανένα άλλο thread δεν έχει δικαίωμα ανάγνωσης και εγγραφής σε κλειδωμένα δεδομένα.
- Ασφάλεια από race conditions.
- Είναι υποχρεωμένα να περιμένουν το ξεκλείδωμα.

Αμοιβαίος Αποκλεισμός και κλειδώματα

- Mutual Exclusion (συχνά MUTEX).
- Mutual Exclusion Locks (συχνά: MUTEX_LOCKS).

```
lock = threading.Lock()
```

- Υποστηρίζονται δύο βασικές μέθοδοι:
  - `acquire()`
  - `release()`

Μέθοδος `Lock.acquire()`

- Επιχειρεί να κλειδώσει τον κώδικα που βρίσκεται μετά από την `acquire()`.
  - Αν πετύχει, το thread αποκτά (αποκλειστικό) δικαίωμα εκτέλεσης του κώδικα μετά την `acquire()`.
  - Αν αποτύχει, τότε κάποιο άλλο thread έχει κλειδώσει τον κώδικα και αυτό το thread είναι υποχρεωμένο να περιμένει το ξεκλείδωμα.
  - Blocking scenario.
  - Κίνδυνος: Με κακή σχεδίαση, το thread μπορεί να περιμένει για πάντα για να κάνει `acquire()`!
  - Μεγάλος κίνδυνος: Με κακή σχεδίαση, ΌΛΑ τα thread μπορεί να περιμένουν για πάντα! – Deadlock.

Μέθοδος `Lock.release()`

- Επιχειρεί να ξεκλειδώσει τον κώδικα που βρίσκεται πριν από την `release`.
  - Αν πετύχει, ο κώδικας ξεκλειδώνει και κάποιο άλλο thread μπορεί να κάνει acquire.
  - Αν αποτύχει, τότε ο κώδικας δεν ήταν κλειδωμένος.
  - Κίνδυνος: Με κακή σχεδίαση, το thread μπορεί να περιμένει για πάντα για να κάνει `release()`!
  - Και αφού όλα τα υπόλοιπα threads περιμένουν τη λήξη, τότε περιμένουν όλοι για πάντα! – deadlock.

Αμοιβαίος Αποκλεισμός και κλειδώματα

```
import threading
garlic_count = 0
lock = threading.Lock()

def shopper():
    global garlic_count
    for i in range(1000000):
        lock.acquire()
        garlic_count += 1
        lock.release()

if __name__ == '__main__':
    george = threading.Thread(target=shopper)
    anna = threading.Thread(target=shopper)
    george.start()
    george.join()
    anna.start()
    anna.join()
    print('We should buy', garlic_count, 'garlic.')
```