2022年12月12日(月) 4限
@研究棟A302

プログラミングA2
第11回

担当：伏見卓恭
連絡先：fushimity@edu.teu.ac.jp
居室：研A1201

プログラミングA2の流れ

第 1回：＜復習編＞関数，ファイル入出力，コンテナデータ型
第 2回：＜復習編＞クラスとオブジェクト
第 3回：＜文法編＞関数の高度な利用法１
第 4回：＜文法編＞関数の高度な利用法２
第 5回：＜文法編＞オブジェクト指向プログラミング
第 6回：＜応用編＞データ構造とアルゴリズム１
第 7回：＜応用編＞データ構造とアルゴリズム２
第 8回：＜実践編＞HTTPクライアント
第 9回：＜実践編＞スクレイピング
第10回：＜実践編＞データベース
第11回：＜実践編＞並行処理
第12回：＜総合編＞総合演習(複合問題)
第13回：＜総合編＞まとめ
第14回：＜総合編＞ Python力チェック ← 確認テストのこと

本日のお品書き

- **並行処理**
  - マルチスレッド
  - マルチプロセス（並列処理）
  - スレッドセーフとロック
  - セマフォ
  - パイプによるプロセス間通信

- ~~**非同期処理**~~

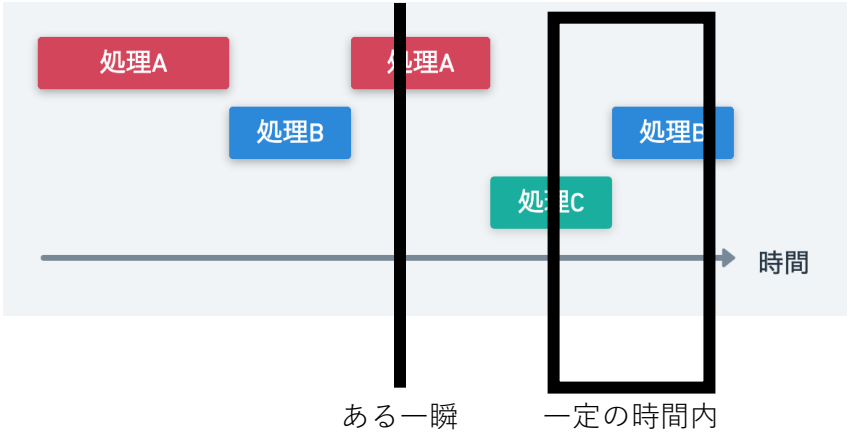
並行処理と並列処理

プログラムは通常，コードを上から順に1つずつ実行するが，途中に重たい処理があった場合，それが終わるまで次の処理に進むことができない．
→ 重たい処理と同時に，別の処理も進められたら効率的である．

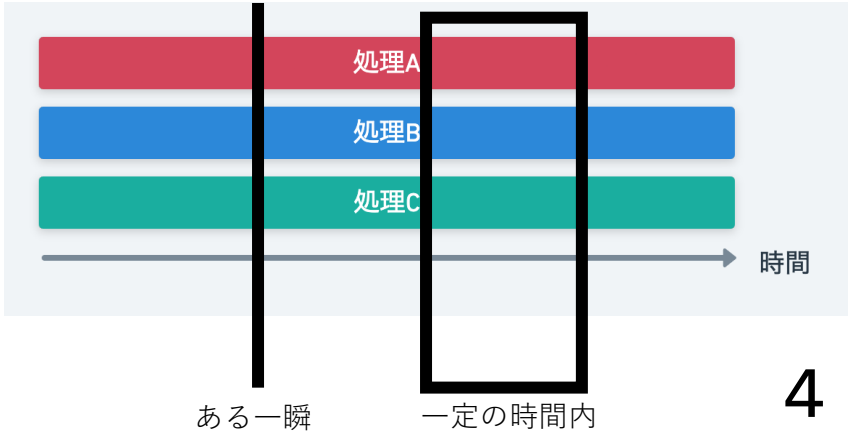
並行処理(Concurrent)とは，**一定の時間内**に複数の処理を同時に進行することである．

類似の概念である並列処理(Parallel)とは，**ある一瞬**を切り取ったときでも複数の処理を同時に進行していることである．

- 並行処理（並列処理ではない）



- 並列処理（並行処理でもある）



ボトルネックとなる処理

- **CPUバウンドな処理**
  - 数値計算のようにCPUに負荷をかけるような処理
  - 処理速度がCPUの計算速度に依存する処理
  - 並列処理で高速化可能
  - 並列処理でない並行処理では高速化できない

- **I/Oバウンドな処理**
  - ファイルの読み書き，NW通信，DB接続などの処理
  - CPUに関係ないI/O部分に負荷がかかる処理
  - I/O処理が終わるまでCPUは待ち状態
  - 並行処理：待ち時間に他の処理を実行することで高速化
    （もちろん，並列処理は並行処理なので高速化可能）

Pythonにおける並行処理

- プロセス内の実行単位
- CPUのコアに命令を与える単位

●**マルチスレッド**

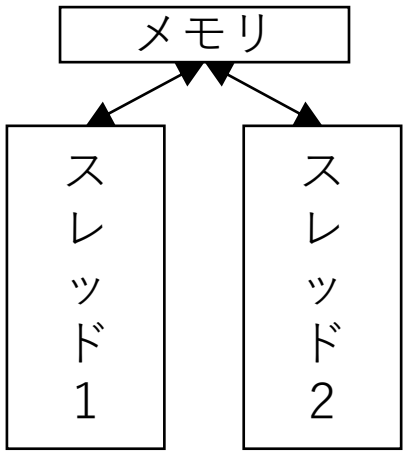
- threadingモジュールのThreadクラス
- concurrent.futuresモジュールのThreadPoolExecutorクラス

OSが実行しているプログラムのインスタンス

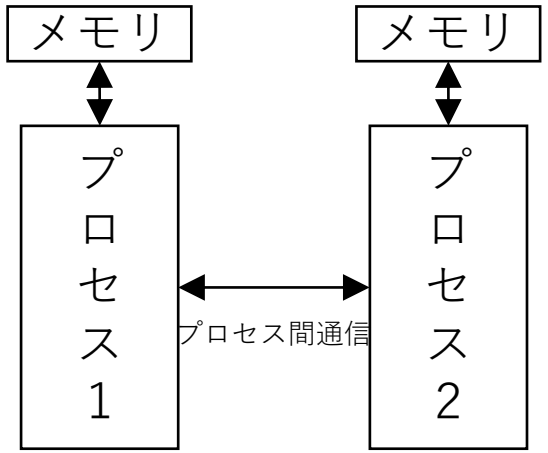
●**マルチプロセス**

- multiprocessingモジュールのProcessクラス
- concurrent.futuresモジュールのProcessPoolExecutorクラス

※スレッド間でメモリを共有



※プロセス毎にメモリを管理



Pythonにおけるスレッドセーフ

スレッドセーフとは，複数のスレッドを並列的に実行しても
問題が発生しないことを意味する.

たとえば，複数のスレッドが共有しているデータに対して
一度に1スレッドのみがアクセスできるようにしておくことで,
処理中に他のスレッドにデータを上書きされることを防ぐ.

CPython(Pythonの実装の1つ)はスレッドセーフでない.
→ PythonはGIL(Global Interpreter lock)という排他ロックにより,
Pythonインタプリタのプロセスは1スレッドしか実行できない.
＝1つのプロセスに複数スレッドが存在してもロックを持つ単一スレッド
でしかコードが実行できず，その他のスレッドは待ち状態になる.

→ Pythonにおけるマルチスレッドは並列処理ではなく並行処理となる

threadingモジュール

参照：https://docs.python.org/ja/3/library/threading.html

Threadクラスの使い方

```
from threading import Thread

th = Thread(target=callableオブジェクト, args=引数タプル)
th.start()
```

fizz_buzz_mt.py

```
from threading import Thread

if __name__ == "__main__":
    bgn0 = time.time()
    num_lst = [i*10000000 for i in range(5, 0, -1)]
    for num in num_lst:
        th = Thread(
                    target=fizz_buzz,
                    args=(num, bgn0)
                   )
        th.start()
    print("start")
```

← 5つのThreadオブジェクトを生成
← 対象：fizz_buzz関数
← fizz_buzz関数の引数：num, bgn0

← スレッドの実行開始

CPUバウンドな処理：fizz_buzz.py

fizz_buzz.py

```
def fizz_buzz(num, bgn0):
    bgn = time.time()
    print(f"fizz_buzz start：{num}")

    省略

    end = time.time()
    print(f"fizz_buzz end：{num}¥t
            所要時間：{end-bgn:.2f}秒¥t
            累積時間：{end-bgn0:.2f}秒")

if __name__ == "__main__":
    bgn0 = time.time()
    num_lst = [i*10000000 for i in range(5, 0, -1)]
    for num in num_lst:
        res = fizz_buzz(num, bgn0)
    print("start")
```

← 呼出開始時

← 実行開始時

← fizz_buzzの繰り返し回数

← fizz_buzzを5回実行

逐次処理の場合

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python fizz_buzz.py
fizz_buzz start：50000000
fizz_buzz end：50000000 所要時間：15.37秒        累積時間：15.37秒
fizz_buzz start：40000000
fizz_buzz end：40000000 所要時間：12.46秒        累積時間：27.83秒
fizz_buzz start：30000000
fizz_buzz end：30000000 所要時間：8.77秒         累積時間：37.24秒
fizz_buzz start：20000000
fizz_buzz end：20000000 所要時間：5.89秒         累積時間：43.64秒
fizz_buzz start：10000000
fizz_buzz end：10000000 所要時間：2.84秒         累積時間：46.81秒
start
```

← for文内の処理がすべて終了してから「print("start")」

5千万の処理がendしてから
4千万の処理がstartし，
4千万の処理がendしてから
3千万の処理がstartしている．

この例では，全体で47秒程度かかっている

マルチスレッドの場合

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python fizz_buzz_mt.py
fizz_buzz start：50000000
fizz_buzz start：40000000
fizz_buzz start：30000000
fizz_buzz start：20000000
fizz_buzz start：10000000
start
fizz_buzz end：10000000 所要時間：11.89秒 累積時間：12.21秒
fizz_buzz end：20000000 所要時間：22.38秒 累積時間：22.62秒
fizz_buzz end：30000000 所要時間：23.82秒 累積時間：24.05秒
fizz_buzz end：40000000 所要時間：30.49秒 累積時間：30.54秒
fizz_buzz end：50000000 所要時間：32.70秒 累積時間：32.70秒
```

← Mainスレッドの「print("start")」

5つのスレッドがほぼ同時にstartし，
1千万，2千万と少ない順に処理がendしている．

この例では，全体で32秒程度かかっている．
※CPUバウンドな処理は，not並列な並行処理では高速化できない．

multiprocessingモジュール

参照：https://docs.python.org/ja/3/library/multiprocessing.html

Threadクラスの使い方

```
from multiprocessing import Process

pr = Process(target=callableオブジェクト, args=引数タプル)
pr.start()
```

fizz_buzz_mp.py

```
from multiprocessing import Process

if __name__ == "__main__":
    bgn0 = time.time()
    num_lst = [i*10000000 for i in range(5, 0, -1)]
    for num in num_lst:
        pr = Process(
                     target=fizz_buzz,
                     args=(num, bgn0)
                    )
        pr.start()
    print("start")
```

← 5つのProcessオブジェクトを生成
← 対象：fizz_buzz関数
← fizz_buzz関数の引数：num, bgn0

← プロセスの実行開始

マルチプロセスの場合

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python fizz_buzz_mp.py
start                                   ← Mainプロセスの「print("start")」
fizz_buzz start：40000000
fizz_buzz start：50000000
fizz_buzz start：30000000
fizz_buzz start：20000000
fizz_buzz start：10000000
fizz_buzz end：10000000 所要時間：3.17秒          累積時間：3.28秒
fizz_buzz end：20000000 所要時間：5.95秒          累積時間：6.06秒
fizz_buzz end：30000000 所要時間：8.25秒          累積時間：8.36秒
fizz_buzz end：40000000 所要時間：10.23秒         累積時間：10.34秒
fizz_buzz end：50000000 所要時間：12.33秒         累積時間：12.44秒
```

Mainプロセスと5つのプロセスがほぼ同時にstartし，同時に進行し，
1千万，2千万と少ない順に処理がendしている．

同時進行しているため，全体としては
最も時間のかかる5千万に対する処理の12秒で終了している．
※CPUバウンドな処理は，並列処理により高速化できる．

threadingモジュールの関数

- **current_thread()：**
  - fizz_buzzなどスレッド処理されている関数の中からスレッドを取得する
  - Threadオブジェクトを返す
  - .nameを付して，スレッド名を取得する際に使われる

- **enumerate()：**
  - 現在処理中のスレッド一覧を返す
  - ※組み込み関数のenumerateと区別がつかなくなるので，名前空間としてモジュール名を付けた方がよい

threadingモジュールのインポート

```
import threading

print(threading.current_thread().name)
for th in threading.enumerate():
    print(th)
```

練習問題：fizz_buzz_mt.py

- fizz_buzz関数内からThreadオブジェクトおよびスレッド名を出力せよ
- 現在のスレッド一覧を取得し，for-in文でThreadオブジェクトおよびスレッド名を出力せよ
- モジュール名を名前空間として利用できるようにimport文を改めよ

実行例：fizz_buzz_mt.py

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python fizz_buzz_mt.py
fizz_buzz start：50000000
fizz_buzz start：40000000
fizz_buzz start：30000000
fizz_buzz start：20000000
fizz_buzz start：10000000
start
```

← Mainスレッドの「print("start")」

```
<_MainThread(MainThread, started 21308)> MainThread
<Thread(Thread-1, started 16624)> Thread-1
<Thread(Thread-2, started 2576)> Thread-2
<Thread(Thread-3, started 22836)> Thread-3
<Thread(Thread-4, started 22972)> Thread-4
<Thread(Thread-5, started 9884)> Thread-5
```

← スレッド一覧

```
<Thread(Thread-5, started 9884)> Thread-5
fizz_buzz end：10000000 所要時間：11.67秒        累積時間：12.13秒
<Thread(Thread-4, started 22972)> Thread-4
fizz_buzz end：20000000 所要時間：21.65秒        累積時間：21.96秒
<Thread(Thread-3, started 22836)> Thread-3
fizz_buzz end：30000000 所要時間：38.44秒        累積時間：38.60秒
<Thread(Thread-2, started 2576)> Thread-2
fizz_buzz end：40000000 所要時間：49.55秒        累積時間：49.59秒
<Thread(Thread-1, started 16624)> Thread-1
fizz_buzz end：50000000 所要時間：51.45秒        累積時間：51.45秒
```

← 各スレッドの終わり間際に実行される

解答例：fizz_buzz_mt.py

fizz_buzz_mt.py

```
def fizz_buzz(num, bgn0):
    bgn = time.time()
    print(f"fizz_buzz start：{num}")
    省略
    end = time.time()
    print(Threadオブジェクト, スレッド名)
    print(f"fizz_buzz end：{num}¥t所要時間：{end-bgn:.2f}秒¥t累積時間：{end-
bgn0:.2f}秒")
    return result_lst

if __name__ == "__main__":
    bgn0 = time.time()
    num_lst = [i*10000000 for i in range(5, 0, -1)]
    for num in num_lst:
        th = threading.Thread(target=fizz_buzz, args=(num, bgn0))
        th.start()
    print("start")
    for th in threading.スレッド一覧:
        print(th, th.name)
```

Timerクラス

- 開始までの時間を設定できるスレッド

Timerクラスの使い方

```
from threading import Timer
th = Timer(インターバル, callableオブジェクト, args=引数タプル)
th.start()
```

fizz_buzz_tmr.py

```
for t, num in enumerate(num_lst, 1):
    th = threading.Timer(t, fizz_buzz, args=(num, bgn0))
    th.start()
print("start")
```

← 1秒後，2秒後，…，5秒後に開始

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python fizz_buzz_tmr.py
start
fizz_buzz start：50000000        インターバル：1.01秒
fizz_buzz start：40000000        インターバル：2.03秒
fizz_buzz start：30000000        インターバル：3.03秒
fizz_buzz start：20000000        インターバル：4.02秒
fizz_buzz start：10000000        インターバル：5.10秒
fizz_buzz end：10000000 所要時間：18.92秒        累積時間：24.08秒
fizz_buzz end：20000000 所要時間：26.09秒        累積時間：30.22秒
fizz_buzz end：30000000 所要時間：46.48秒        累積時間：49.56秒
fizz_buzz end：40000000 所要時間：56.26秒        累積時間：58.30秒
fizz_buzz end：50000000 所要時間：58.08秒        累積時間：59.10秒
```

← 先にMainスレッドの「print("start")」

← 開始時刻のズレ

スレッドセーフの実現

スレッドをロックし，多元同時更新を防ぐ

Lockクラス

※multiprocessing.Processに対しても同じように使用できる

- スレッドをロックし，他のスレッドによる干渉を防ぐ
  - Lockオブジェクトをスレッド処理する関数に引数として渡す
  - acquireメソッド：ロック状態にする
  - releaseメソッド：アンロック状態に戻す

Lockクラスの使い方

```
from threading import Lock
lck = Lock()
th = Thread(target=関数, args=(lck, ))

def 関数(lck):
    lck.acquire()
    ロック状態で
    他スレッドによる干渉を受けない
    lck.release()
```

または,

```
    with lck:
        ロック状態で
        他スレッドによる干渉を受けない
```

← 引数として渡す

← ロックされてない状態

← ロック状態

← with構文で書くこともできる

サンプルコード：counter.py

counter.py

```
def counter(cnt_dct, type_):
    now = cnt_dct[type_]

    rnd = random.randint(1, 5)
    print(f"{rnd}秒sleep")
    time.sleep(rnd)

    cnt_dct[type_] = now+1
    print(f"{type_}を追加", cnt_dct)


if __name__ == "__main__":
    types = read_types("lec11/data/poke_names.txt")
    cnt_dct = collections.defaultdict(int)
    for type_lst in types[:9]:
        for type_ in type_lst:
            counter(cnt_dct, type_)

    print("集計結果：", cnt_dct)
```

← 現在の辞書の値を抽出

← 乱数(1~5)秒のsleep

← 現在の値を更新

← カウント用の空辞書

← 辞書とタイプ文字列

逐次処理の場合

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter.py
2秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 1})
2秒sleep
どくを追加 defaultdict(<class 'int'>, {'くさ': 1, 'どく': 1})
5秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 2, 'どく': 1})
5秒sleep
どくを追加 defaultdict(<class 'int'>, {'くさ': 2, 'どく': 2})
1秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 2})
2秒sleep
どくを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3})
3秒sleep
ほのおを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 1})
1秒sleep
ほのおを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 2})
1秒sleep
ほのおを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3})
2秒sleep
ひこうを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1})
3秒sleep
みずを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1, 'みず': 1})
3秒sleep
みずを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1, 'みず': 2})
1秒sleep
みずを追加 defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1, 'みず': 3})
集計結果： defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1, 'みず': 3})
```

スレッドセーフでないコード

counter_mt.py

```
if __name__ == "__main__":
    types = read_types("lec11/data/poke_names.txt")
    cnt_dct = collections.defaultdict(int)

    for type_lst in types[:9]:
        for type_ in type_lst:
            th = threading.Thread(
                                  target=counter,
                                  args=(cnt_dct, type_)
                                 )
            th.start()

    for th in threading.enumerate():
        if th.name == "MainThread":
            continue
        th.join()

    print("集計結果：", cnt_dct)
```

← 1回の関数呼び出しを1つのスレッドとする

← スレッドを実行する

← メインスレッドはjoinできない

← **join()したスレッドが終了するのを待つ**

← 全スレッドが終了したら

※join()しないと，スレッドの終了を待たずに集計結果が出力される

実行例：counter_mt.py

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter_mt.py
3秒sleep
4秒sleep
 :
ほのおを追加 defaultdict(<class 'int'>, {'くさ': 0, 'どく': 0, 'ほのお': 1, 'ひこう': 0, 'みず': 0})
どくを追加ひこうを追加どくを追加みずを追加   defaultdict(<class 'int'>, {'くさ': 0, 'どく': 1, 'ほのお':
1, 'ひこう': 1, 'みず': 1})
defaultdict(<class 'int'>, {'くさ': 0, 'どく': 1, 'ほのお': 1, 'ひこう': 1, 'みず': 1})
defaultdict(<class 'int'>, {'くさ': 0, 'どく': 1, 'ほのお': 1, 'ひこう': 1, 'みず': 1})
defaultdict(<class 'int'>, {'く
さ': 0, 'どく': 1, 'ほのお': 1, 'ひこう': 1, 'みず': 1})
くさを追加くさを追加 defaultdict(<class 'int'>, {'くさ': 1, 'どく': 1, 'ほのお': 1, 'ひこう': 1, 'みず':
1})
 :

集計結果： defaultdict(<class 'int'>, {'くさ': 1, 'どく': 1, 'ほのお': 1, 'ひこう': 1, 'みず': 1})
```

※各スレッドが，ほぼ同時に辞書の値(0)を抽出し，
スレッドごとに異なる時間のsleepを経て辞書を更新するため，
最終的な集計結果が正しくない．

練習問題：counter_lck.py

Lockオブジェクトを導入し，スレッド実行を制御することで同時更新を防ぐコードを実装せよ.

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter_lck.py
1秒sleep
くさを追加 2秒sleepdefaultdict(<class 'int'>, {'くさ': 1, 'どく': 0})

どくを追加 4秒sleepdefaultdict(<class 'int'>, {'くさ': 1, 'どく': 1})

くさを追加 defaultdict(<class 'int'>, {'くさ': 2, 'どく': 1})
5秒sleep
どくを追加2秒sleep
defaultdict(<class 'int'>, {'くさ': 2, 'どく': 2})
くさを追加 5秒sleepdefaultdict(<class 'int'>, {'くさ': 3, 'どく': 2})

：
集計結果： defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう': 1, 'みず': 3})
```

解答例：counter_lck.py

counter_lck.py

```
def counter(cnt_dct, type_, lck):
    print(f"{type_}を追加", cnt_dct)

if __name__ == "__main__":
    lck =
    for type_lst in types[:9]:
        for type_ in type_lst:
            th = threading.Thread(
                                  target=counter,
                                  args=(cnt_dct, type_, lck)
                                 )
            th.start()
    省略
```

ロック

アンロック

Lockオブジェクトの生成

← アンロックとprintのタイミングを逆にするとどうなる？？

Semaphoreクラス

※multiprocessing.Processに対しても同じように使用できる

セマフォとは，コンピュータで並列処理を行う際，同時に実行されているプログラム間で資源の排他制御や同期を行う仕組みの一つである．
当該資源のうち現在利用可能な数を表す値のこと．

- 同時実行できるスレッド数を制御する
  - Semaphoreオブジェクトをスレッド処理する関数に渡す

Semaphoreクラスの使い方

```
from threading import Semaphore
smp = Semaphore(スレッド数)
th = Thread(target=関数, args=(smp, ))

def 関数(smp):
    smp.acquire()
    セマフォが有効なときだけ処理を実行
    有効でないときは待ち
    smp.release()
または，
    with smp:
        セマフォが有効なときだけ処理を実行
        有効でないときは待ち
```

← 引数として渡す

← ロックされてない状態

← ロック状態

← with構文で書くこともできる

例題：counter_smp.py

counter_smp.py

```
def counter(cnt_dct, type_, smp):
    with smp:
        now = cnt_dct[type_]
        time.sleep()
        cnt_dct[type_] = now+1
    print(f"{type_}を追加", cnt_dct)

if __name__ == "__main__":
    smp = threading.Semaphore(2)
    for type_lst in types[:9]:
        for type_ in type_lst:
            th = threading.Thread(target=counter, args=(cnt_dct, type_, smp))
            th.start()
```

← print()もwith構文に含めるとどうなる？？

← スレッド数を1にするとどうなる？？

実行例

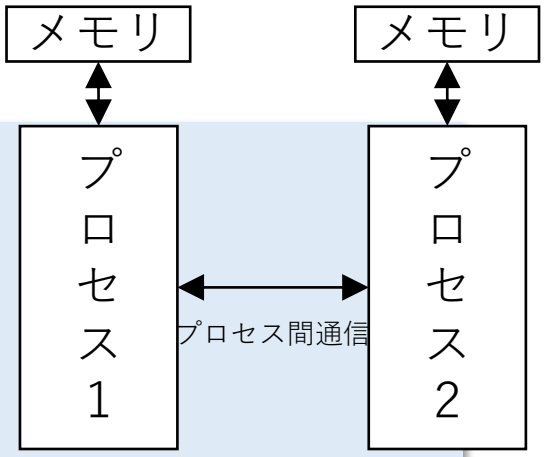
```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter_smp.py
5秒sleep
4秒sleep
どくを追加 4秒sleep
defaultdict(<class 'int'>, {'くさ': 0, 'どく': 1})
くさを追加 3秒sleep
defaultdict(<class 'int'>, {'くさ': 1, 'どく': 1})
くさを追加どくを追加3秒sleep2秒sleep

 :
集計結果： defaultdict(<class 'int'>, {'くさ': 2, 'どく': 3, 'ほのお': 2, 'ひこう': 1, 'みず': 2})
```

← フシギダネの'くさ'と'どく'に対する2つのスレッドが同時に実行される

← フシギソウ

マルチプロセスの場合

※プロセス毎にメモリを管理



counter_mp.py

```
if __name__ == "__main__":
    types = read_types("lec11/data/poke_names.txt")
    cnt_dct = collections.defaultdict(int)
    processes = list()
    for type_lst in types[:9]:
        for type_ in type_lst:
            pr = multiprocessing.Process(
                                          target=counter,
                                          args=(cnt_dct, type_)
                                         )

            pr.start()
            processes.append(pr)
    for pr in processes: pr.join()
    print("集計結果：", cnt_dct)
```

← multiprocessingモジュールにはenumerate関数がない

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter_mp.py
2秒sleep
2秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 1})
ひこうを追加 defaultdict(<class 'int'>, {'ひこう': 1})
くさを追加 defaultdict(<class 'int'>, {'くさ': 1})
どくを追加 defaultdict(<class 'int'>, {'どく': 1})
：
集計結果： defaultdict(<class 'int'>, {})
```

← 集計されてない

※サブプロセスとメインプロセスでメモリ(辞書)を共有する必要がある

パイプによるプロセス間通信

- **フォーク**：メインプロセスのメモリがコピーされ新しい(サブ)プロセスが生成される
- **パイプ**：あるプロセスの出力を他のプロセスの入力とする

Pipeクラスの使い方

```
from multiprocessing import Pipe
pp1, pp2 = Pipe()
pr = Process(target=関数, args=(pp1, ))
pr.start()
pp2.recv()

def 関数(pp):
    処理
    pp.send(共有したい変数)
    pp.close()
```

← 2つで1組

← フォーク／片方を引数として渡す

← パイプを通して変数を受け取る

← 共有したい変数をパイプを通して送る

例題：counter_pp.py

パイプによるプロセス間通信により，タイプカウント用の辞書cnt_dctを共有し，正しい集計結果を出力せよ．

counter_pp.py

```
if __name__ == "__main__":
    types = read_types("lec11/data/poke_names.txt")
    cnt_dct = collections.defaultdict(int)
    processes = list()
    for type_lst in types[:9]:
        for type_ in type_lst:
            pp1, pp2 = multiprocessing.Pipe()
            pr = multiprocessing.Process(
                target=counter,
                args=(cnt_dct, type_, pp1)
            )
            pr.start()
            cnt_dct = pp2.recv()
            processes.append(pr)
    for pr in processes: pr.join()
    print("集計結果：", cnt_dct)
```

← ①③メインプロセスの現在の状態をコピーしてサブプロセスを生成

← ②メインプロセスはサブプロセスから辞書を受け取りcnt_dctは最新の状態

例題：counter_pp.py

counter_pp.py

```
def counter(cnt_dct, type_, pp):
    now = cnt_dct[type_]
    rnd = random.randint(1, 5)
    print(f"{rnd}秒sleep")
    time.sleep(rnd)
    cnt_dct[type_] = now+1
    pp.send(cnt_dct)
    pp.close()
    print(f"{type_}を追加", cnt_dct)
```

← sleep()をsend後に移動するとどうなる？？

← 更新後のcnt_dctをパイプを通して送信

実行例

```
PS C:¥Users¥admin¥Desktop¥ProA2> python counter_pp.py
2秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 1})
3秒sleep
どくを追加 defaultdict(<class 'int'>, {'くさ': 1, 'どく': 1})
4秒sleep
くさを追加 defaultdict(<class 'int'>, {'くさ': 2, 'どく': 1})
：
集計結果： defaultdict(<class 'int'>, {'くさ': 3, 'どく': 3, 'ほのお': 3, 'ひこう':
1, 'みず': 3})
```