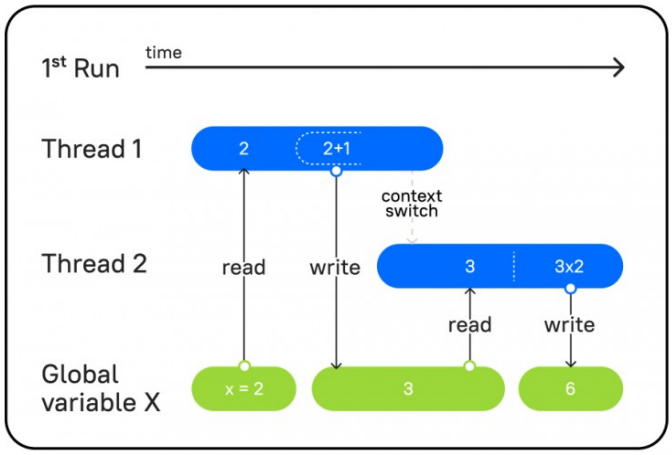
```
import multiprocessing
import time

def task():
    print('Засне на 0.5 секунд')
    time.sleep(0.5)
    print('Прокинеться')

if __name__ == "__main__":
    start_time = time.perf_counter()
    p1 = multiprocessing.Process(target=task)
    p2 = multiprocessing.Process(target=task)
    p1.start()
    p2.start()
    finish_time = time.perf_counter()
    print(f"Програма завершиться на {finish_time-start_time} секунді")
```

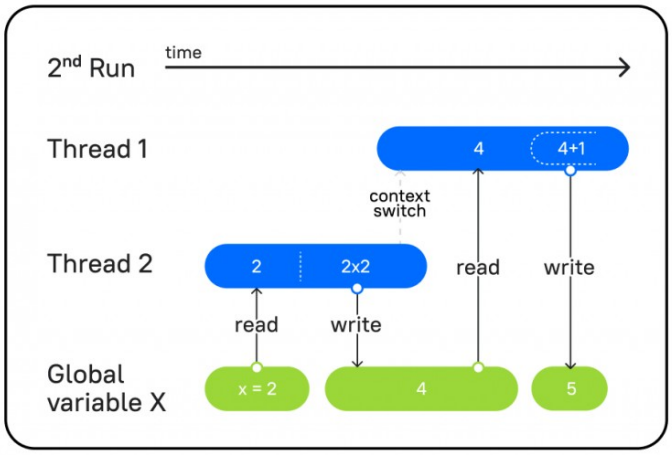
```
import multiprocessing
import time

def task():
    print('Засне на 0.5 секунд')
    time.sleep(0.5)
    print('Прокинеться')

if __name__ == "__main__":
    start_time = time.perf_counter()
    p1 = multiprocessing.Process(target=task)
    p2 = multiprocessing.Process(target=task)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    finish_time = time.perf_counter()
    print(f"Програма завершиться на {finish_time-start_time} секунді")
```

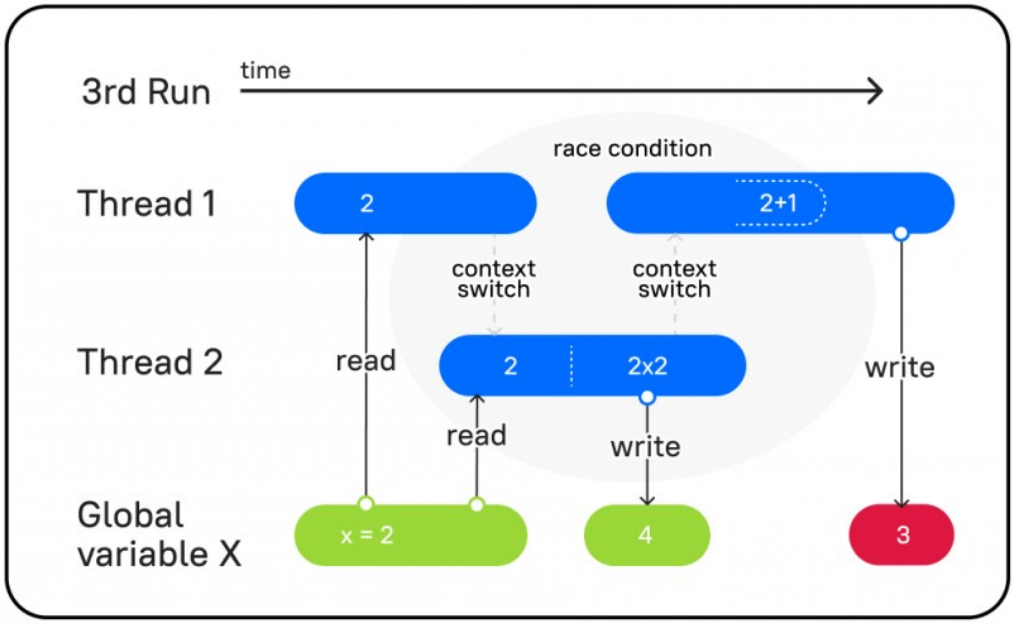
```
import threading
import time

def task():
        print('Засне на 0.5 секунд')
        time.sleep(0.5)
```

```
        print('Прокинеться')

if __name__ == "__main__":
        start_time = time.perf_counter()
        p1 = threading.Thread(target=task, args=(), daemon=True)
        p2 = threading.Thread(target=task, args=(), daemon=True)
        p1.start()
        p2.start()
        p1.join()
        p2.join()
        finish_time = time.perf_counter()
        print(f"Програма завершиться на {finish_time-start_time} секунді")
```

```
i=0 # a global variable
for x in range(100):
    print(i)
    i+=1
```







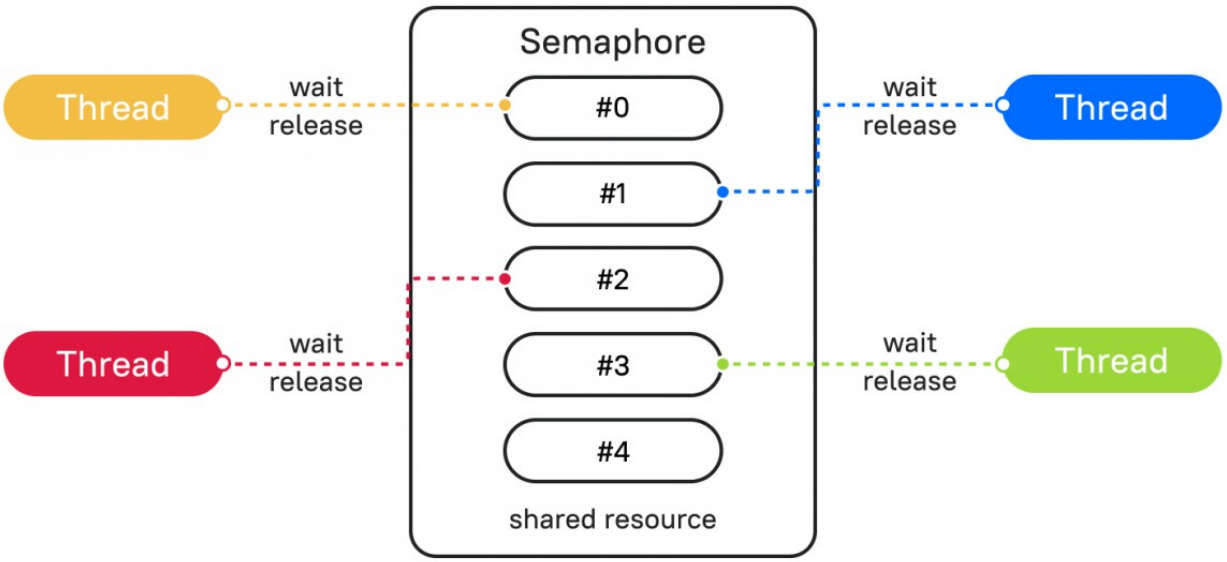

```
from threading import Thread
from time import sleep

counter = 0

def increase(by):
    global counter
    local_counter = counter
    local_counter += by
    sleep(0.1)
    counter = local_counter
    print(f'{counter=}')
```

```
t1 = Thread(target=increase, args=(10,))
t2 = Thread(target=increase, args=(20,))
t1.start()
t2.start()
t1.join()
t2.join()
```

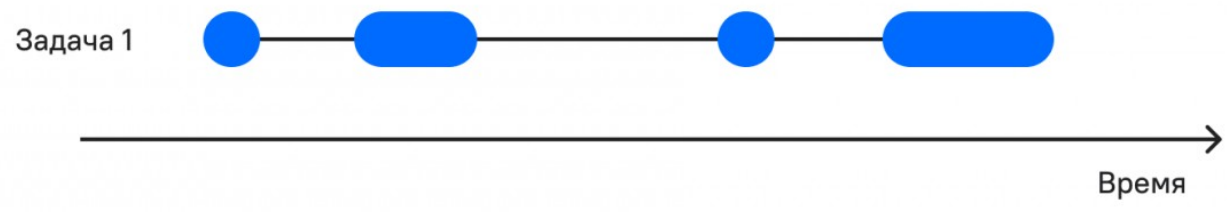
```
from threading import Thread, Lock
from time import sleep

counter = 0

def increase(by, lock: Lock):
    global counter
    lock.acquire()
    local_counter = counter
    local_counter += by
    sleep(0.1)
    counter = local_counter
    print(f'{counter=}')
    lock.release()

lock = Lock()
t1 = Thread(target=increase, args=(10, lock,))
t2 = Thread(target=increase, args=(20, lock,))
t1.start()
t2.start()
t1.join()
t2.join()
```

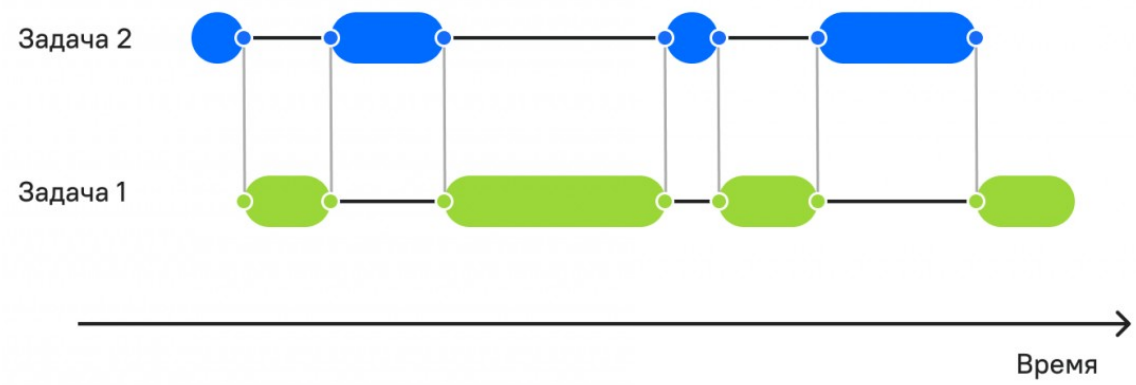
```
from threading import Thread, Lock
from time import sleep

a = 5
b = 10

a_lock = Lock()
b_lock = Lock()

def function_a():
```

```
    global a
    global b
    a_lock.acquire()
    print('Функція a, a_lock = заблокований')
    sleep(1)
    b_lock.acquire()
    print('Функція a, b_lock = заблокований')
    sleep(1)
    a_lock.release()
    print('Функція a, a_lock = розблокований')
    b_lock.release()
    print('Функція a, b_lock = розблокований')

def function_b():
    global a
    global b
    b_lock.acquire()
    print('Функція b, b_lock = заблокований')
    a_lock.acquire()
    print('Функція b, a_lock = заблокований')
    sleep(1)
    b_lock.release()
    print('Функція b, b_lock = розблокований')
    a_lock.release()
    print('Функція b, a_lock = розблокований')

t1 = Thread(target=function_a)
t2 = Thread(target=function_b)
t1.start()
t2.start()
t1.join()
t2.join()
print('Завершено')
```



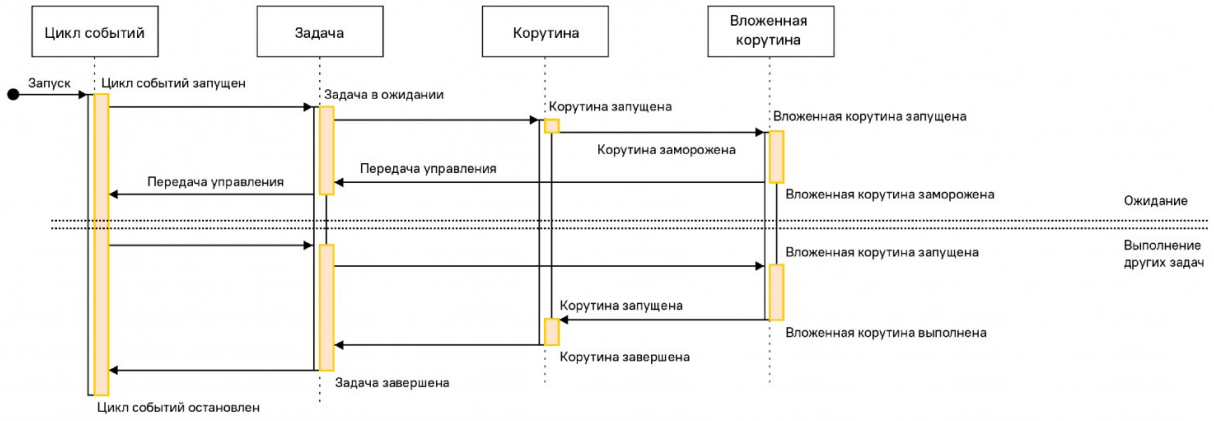

```
import datetime
from threading import Semaphore, Thread
from time import sleep

s = Semaphore(3)

def semaphore_func(payload: int):
    s.acquire()
    now = datetime.datetime.now().strftime('%H:%M:%S')
    print(f'{now=}, {payload=}')
    sleep(2)
    s.release()

threads = [Thread(target=semaphore_func, args=(i,)) for i in range(7)]

for t in threads:
    t.start()

for t in threads:
    t.join()
```





callback hell

asyncio aiofiles aiohttp   IO bound




```
import asyncio


async def hello():
    print('Запуск функції hello')
    await asyncio.sleep(5)  # Віддаємо керування назад до Event loop поки чекаємо
    print('Перемикання контексту у функцію hello')

hello()
```

RuntimeWarning: coroutine 'hello' was never awaited
  hello()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback

---

import asyncio

async def hello():
    print('Запуск функції hello')

```
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print('Перемикання контексту у функцію hello')

asyncio.run(hello())
```

```
import asyncio

async def hello():
    print(' Запуск функції hello')
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print(' Перемикання контексту у функцію hello')

async def starter():
    await asyncio.gather(hello(), hello())

asyncio.run(starter())
```

```
import asyncio
import time

start = time.time() ## точка відліку часу

async def hello():
    print('Запуск функції hello')
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print('Перемикання контексту у функцію hello')

async def starter():
    await asyncio.gather(*[hello() for i in range(10000)])

asyncio.run(starter())

end = time.time() - start
print(end)
```

```
import asyncio

async def hello():
    print('Запуск функції hello')
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print('Перемикання контексту у функцію hello')

async def bye():
    print('Запуск функц bye')
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print('Перемикання контексту у функцію bye')

ioloop = asyncio.get_event_loop()
tasks = [ioloop.create_task(hello()), ioloop.create_task(bye())]
tasks_for_wait = asyncio.wait(tasks)
ioloop.run_until_complete(tasks_for_wait)
ioloop.close()
```

```
import asyncio

async def hello():
    print('Запуск функції hello')
    await asyncio.sleep(5)  # Віддаємо керування назад у Event loop поки чекаємо
    print('Перемикання контексту у функцію hello')
    return 'Виконана функція hello'
async def bye():
    print('Запуск функції bye')
    await asyncio.sleep(2)  # Віддаємо керування назад у Event loop поки чекаємо
    print(' Перемикання контексту у функцію bye')
    return 'Виконана функція bye'
async def starter(ioloop):
    tasks = [ioloop.create_task(hello()), ioloop.create_task(bye())]
    done, pending = await asyncio.wait(tasks,
return_when=asyncio.FIRST_COMPLETED)
    result = done.pop().result()

    for pending_future in pending:
        pending_future.cancel()

    print(result)
ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(starter(ioloop))
ioloop.close()
```

pending
call_soon, call_later, call_at
aiohttp
aiofiles

Concurrent.futures
Class Executor
ThreadPoolExecutor ProcessPoolExecutor
Об'єкт Future має метод done ()

pro.py

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistant-subdomain.python.org/']


def load_url(url, timeout):
        with urllib.request.urlopen(url, timeout = timeout) as conn:
                return conn.read()
```

```
with concurrent.futures.ThreadPoolExecutor(max_workers = 5) as executor:

        future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
        for future in concurrent.futures.as_completed(future_to_url):
                url = future_to_url[future]
        try:
                data = future.result()
        except Exception as exc:
                print('%r generated an exception: %s' % (url, exc))
        else:
                print('%r page is %d bytes' % (url, len(data)))
```

Executor.map()

```
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed
values = [2,3,4,5]
def square(n):
   return n * n
def main():
   with ThreadPoolExecutor(max_workers = 3) as executor:
     results = executor.map(square, values)
   for result in results:
      print(result)
if __name__ == '__main__':
   main()
```

ProcessPoolExecutor замість ThreadPoolExecutor

`pro1.py`

```
from concurrent.futures import ProcessPoolExecutor
from time import sleep
def task(message):
   sleep(2)
   return message

def main():
   executor = ProcessPoolExecutor(5)
   future = executor.submit(task, ("Completed"))
   print(future.done())
   sleep(2)
   print(future.done())
   print(future.result())
```

```
if __name__ == '__main__':
    main()
```

```
with ProcessPoolExecutor(max_workers = 5) as executor
```

pro2.py

```
import concurrent.futures
from concurrent.futures import ProcessPoolExecutor
import urllib.request

URLS = ['http://www.foxnews.com/',
  'http://www.cnn.com/',
  'http://europe.wsj.com/',
  'http://www.bbc.co.uk/',
  'http://some-made-up-domain.com/']

def load_url(url, timeout):
        with urllib.request.urlopen(url, timeout = timeout) as conn:
                return conn.read()

def main():
        with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
                future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
                for future in concurrent.futures.as_completed(future_to_url):
                        url = future_to_url[future]
                try:
                        data = future.result()
                except Exception as exc:
                        print('%r generated an exception: %s' % (url, exc))
                else:
                        print('%r page is %d bytes' % (url, len(data)))

if __name__ == '__main__':
        main()
```

```
from concurrent.futures import ProcessPoolExecutor
from concurrent.futures import as_completed
values = [2,3,4,5]
def square(n):
    return n * n
def main():
    with ThreadPoolExecutor(max_workers = 3) as executor:
```

```
        results = executor.map(square, values)
    for result in results:
        print(result)
if __name__ == '__main__':
    main()
```

```
import time
import concurrent.futures

value = [8000000, 7000000]

def counting(n):
    start = time.time()
    while n > 0:
        n -= 1
    return time.time() - start

def main():
    start = time.time()
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, time_taken in zip(value, executor.map(counting, value)):
            print('Start: {} Time taken: {}'.format(number, time_taken))
    print('Total time taken: {}'.format(time.time() - start))

if __name__ == '__main__':
    main()
```

```
import time
import concurrent.futures

value = [8000000, 7000000]

def counting(n):
    start = time.time()
    while n > 0:
        n -= 1
    return time.time() - start

def main():
    start = time.time()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        for number, time_taken in zip(value, executor.map(counting, value)):
            print('Start: {} Time taken: {}'.format(number, time_taken))
        print('Total time taken: {}'.format(time.time() - start))
```

```
if __name__ == '__main__':
    main()
```

```
import os
if __name__ == "__main__":
    print(f"Головний процес {os.getpid()}")
```

```
import os
import multiprocessing

def child_process():
    print(f"Дочірній процес {os.getpid()}")

if __name__ == "__main__":
    print(f"Головний процес {os.getpid()}")

    # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
     # функцію `child_process` для виконання.
    process = multiprocessing.Process(target=child_process)

   # Запускаємо процес
    process.start()

    # Тут ми приєднуємося до процесу. Це змусить наш скрипт зависнути і
     # потрібно зачекати, поки дочірній процес завершиться.
    process.join()
```

```
import os
import multiprocessing

def child_process():
    print(f"Дочірній процес {os.getpid()}")

if __name__ == "__main__":
    print(f"Головний процес {os.getpid()}")

    # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
     # функцію `child_process` для виконання.
    process = multiprocessing.Process(target=child_process)

    # Запускаємо процес
    process.start()
```

```
    # Тут ми приєднуємося до процесу. Це змусить наш скрипт зависнути і
    # потрібно зачекати, поки дочірній процес завершиться.
    #process.join()

   print("Виконання дочірнього процесу:")
```

```
import os
import multiprocessing

def child_process(id):
   print(f"Дочірній процес {os.getpid()} з id#{id}")

if __name__ == "__main__":
   print(f"Головний процес {os.getpid()}")
   list_of_processes = []

  # Перебирає числа від 0 до 10 і створює процеси для кожного з них
   for i in range(0, 10):
      # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
       # функцію `child_process` для виконання. Зверніть увагу на різницю
       # зараз ми використовуємо параметр `args`, це означає, що ми можемо
передати
       # параметри функції, яка виконується як дочірній процес.
      process = multiprocessing.Process(target=child_process, args=(i,))
      list_of_processes.append(process)

   for process in list_of_processes:
      # Запускаємо процес
      process.start()

      # Тут ми приєднуємося до процесу. Це змусить наш скрипт зависнути і
    # потрібно зачекати, поки дочірній процес завершиться.
      process.join()
```

```
import os
import multiprocessing

def child_process(queue, number1, number2):
   print(f"Дочірній процес {os.getpid()}, який виконує обчислення.")
   sum = number1 + number2

   # Putting data into the queue
   queue.put(sum)
```

```
if __name__ == "__main__":
    print(f"Головний процес {os.getpid()}")

    # Визначення нової черги()
    queue = multiprocessing.Queue()

    # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
     # функцію `child_process` для виконання. Зверніть увагу на різницю
     # зараз ми використовуємо параметр `args`, це означає, що ми можемо
передати
     # параметри функції, яка виконується як дочірній процес.
    process = multiprocessing.Process(target=child_process, args=(queue,1, 2))

    # We then start the process
    process.start()

   # Тут ми приєднуємося до процесу. Це змусить наш скрипт зависнути і
     # зачекайте, поки дочірній процес завершиться.
    process.join()

   # Доступ до результату з черги.
    print(f"Отримуємо результат з дочірнього процесу, який дорвінює:
{queue.get()}")
```

```
import os
import multiprocessing

def child_process():
    print(f"Дочірній процес {os.getpid()}.")
    raise Exception("Помилка! :(")

if __name__ == "__main__":
    print(f"Головний процес {os.getpid()}")

   # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
     # функцію `child_process` для виконання. Зверніть увагу на різницю
     # зараз ми використовуємо параметр `args`, це означає, що ми можемо
передати
     # параметри функції, яка виконується як дочірній процес.
    process = multiprocessing.Process(target=child_process)

    try:
        # Запускаємо
        process.start()
```

```
        # І нарешті ми приєднуємося до процесу. Це змусить наш скрипт зависнути
і
         # зачекайте, поки дочірній процес завершитьсяе.
        process.join()

        print("Виконання дочірнього процесу")
    except Exception:
        print("Помилка?")
```

```
import os
import multiprocessing

def child_process():
    try:
        print(f"Дочірній процес {os.getpid()}.")
        raise Exception("Помилка :(")
    except Exception:
        print("Обійшлося без помилок :) ")

if __name__ == "__main__":
    print(f"Батьківський процес {os.getpid()}")

   # Тут ми створюємо новий екземпляр класу Process і призначаємо нашу
     # функцію `child_process` для виконання. Зверніть увагу на різницю
     # зараз ми використовуємо параметр `args`, це означає, що ми можемо
передати
     # параметри функції, яка виконується як дочірній процес.
    process = multiprocessing.Process(target=child_process)

    # Запускаємо процес
    process.start()

   # Тут ми приєднуємося до процесу. Це змусить наш скрипт зависнути і
     # зачекайте, поки дочірній процес завершиться.
    process.join()

    print("Виконання дочірнього процесу")
```

Thread Threading

__thread