PPC

Rappel

Cette phrase est pour définir que l'environnement est en python3 dans le cas ou les fichier est exécutable

```
#!/usr/bin/env python3
```

sinon on peut lancer ce fichier avec

```
python3 nomdufichier
```

Process

un process se lance soit comme un objet

```
class f(Process):
    def __init__(self, number):
        super().__init__()  # Ne pas oublier
        self.number = number
    def run(self):
        chain = [0,1]
        a, b = 0, 1
        i = 0
        while i < self.number :
            a, b = b, a+b
            chain.append(b)
            i +=1
        print(chain)
        time.sleep(8)
if __name__=="__main__":
    index = 5
    p = f(index)
```

soit depuis une fonction

```
def f(self):
        chain = [0,1]
        a, b = 0, 1
        i = 0
        while i < self.number :
            a, b = b, a+b
            chain.append(b)
            i +=1
        print(chain)
if __name__=="__main__":
    p = Process(target=fArray, args=(index,))
    p.start()
    p.join()
```

Signal

```
def handler(sig, frame):
    print(« Die son »)
    # Emettre le signal SIGKILL (tue le process passé en pid)
    os.kill(childPID, signal.SIGKILL)
def child():
    signal.signal(signal.SIGUSR1, handler)# Executer handler quand SIGUSR1 est emis
    while True:
        print("Hey !")
        time.sleep(1)
childPID=0
if __name__=="__main__":
    p = Process(target=child, args=())
    p.start()
    childPID = p.pid
    time.sleep(5)
    os.kill(childPID, signal.SIGUSR1) # Emettre SIGUSR1
    p.join()
```

Pipes

Un `pipe()` est une paire de process de communication

```
def child(conn, sentence):
    conn.send(sentence[::-1])# [::-1] -> inverser la phrase
    conn.close()
if __name__=="__main__":
    conn_par, conn_child = Pipe() # Creer un pipe
    print("Entrez une phrase")
    sentence = input()
    p = Process(target=child, args=(conn_child,sentence))
    p.start()
    reversed_sent = conn_par.recv()    # Receive cote parent
    print(reversed_sent)
    p.join()
    conn_par.close()
```

Mémoire partagée

Array

```
def fArray(number, mem):
    mem[0] = 0
    a, b = 0, 1
    i = 0
    while i < number :
        a, b = b, a+b
        mem[i+1] = a
        i +=1
    if i+1 < MEMORY_SIZE:
        mem[i+1] = -1  #stop flag
if __name__=="__main__":
          index = 5
    shared_memory = Array(‘l’, MEMORY_SIZE)
    p = Process(target=fArray, args=(index, shared_memory))
          p.start()
    p.join()
    for x in shared_memory[:]:
        if x == -1:
            break
        print(x, end=‘ ’)
```

Manager

```
def fManager(n, lst):
    lst.append(0)
    a,b = 0,1
    i=0
    while i < n :
        a, b = b, a+b
        lst.append(a)
        i += 1
if __name__=="__main__":
    with Manager() as manager:
        lst = manager.list()
        p = Process(target=fManager, args=(index, lst))
                  p.start()
                  p.join()
        print(lst)
```

Passage de message

Client

```
key = 128
try:
    mq = sysv_ipc.MessageQueue(key)
except ExistentialError:
    sys.exit(1)
while True:
    request = getrequest()
    if request == 1:
        m = b"" # Type Byte
        mq.send(m, type=1)
        m, t = mq.receive(type = 3)
        print("Server response: ", m.decode())
    if request == 2:
        m= b""
        mq.send(m, type = 2)
    if request == 3:
        break
mq.remove() # ne pas oublier
```

Server

```
key = 128
try :
    mq = sysv_ipc.MessageQueue(key, sysv_ipc.IPC_CREAT)
except ExistentialError:
    sys.exit(1)
while True:
    mp, t = mq.receive() # Receive message from client
    if int(t) == 2 :
        break
    if int(t) == 1 :
        mq.send(time.asctime(), type = 3)
mq.remove()
```

Thread

Les `thread()`se lancent de la même manière que les `Process()`, en créant un objet qui hérite de `Thread()`est qui redéfinit la methode `run()` ou lance un objet `Thread()`avec une target

```
def countpi(n):
    i  = n
    global inpoint
    while i :
        x = -1 + 2*random.random()
        y = -1 + 2*random.random()
        if x**2 + y**2 < 1 :
            inpoint += 1
        i -= 1
if __name__ == "__main__":
          n = 2000
    threadpi = threading.Thread(target=countpi, args=(n,))
    threadpi.start()
    threadpi.join()
    print(« Estimation of pi : », round((4*(float(inpoint)/float(n))), 5))
```

Thread / Message Queue / Evenement

```
def worker(task_queue, data, data_ready, res_queue):
    data_ready.wait() # Attendre un evenement
    function = task_queue.get() # Récupération d’une fonction
    res = function(data)  # Récupération du resultat de la fonction
    res_queue.put([function.__name__, res]) # Envoyer le nom de la fonction et son resultat

if __name__ == "__main__":
    data = []
    tasks = [min, max, statistics.median, statistics.mean, statistics.stdev]    # Fonction a effectuer
    task_queue = Queue()
    for task in tasks :
        task_queue.put(task) # Queue contenant les fonctions voulues
    data_ready = threading.Event()
    res_queue = Queue() # Queue des resultats
    # Creation d'un thread pour chaque fonction a effetuer
    threads = [threading.Thread(target=worker, args=(task_queue, data, data_ready, res_queue)) for i in range(len(tasks))]
    for thread in threads:
        thread.start() # Démarrer chaque thread
    # Faire une liste de float
    in_str = input(« Entrez une séquence de nombre »).split()
    for s in in_str:
        try :
            data.append(float(s))
        except:
            print(« Bad number », s)
    print(data)
    data_ready.set() # Lancer un evenement
    for thread in threads:
        thread.join()   # Attendre la fin des threads
    while not res_queue.empty():
        print(res_queue.get())  # Afficher les resultats
```

Process pool

```
def is_prime(n):
    print(multiprocessing.current_process().name)
    if n <=3 :
        return n >=1 # Vrai 0, 1 /  Faux 2,3
    elif (n%2 == 0) or (n%3 == 0) :
        return False
    i = 5
    while (i*i<n):
        if (n%i == 0) or (n%(i+2)==0):
            return False
        i += 6
    return True

if __name__ == "__main__":
    numbers = [random.randint(1000, 1000000) for i in range(10)]
    print("Test with numbers :", numbers)
    with multiprocessing.Pool(processes = 4) as pool:
        for x in pool.map(is_prime, numbers):
            print(x) # Synchronous map
        for x in pool.map_async(is_prime, numbers).get():
            print(x) # Asynchronous map
        for x in pool.imap(is_prime, numbers):
            print(x) # Lazy map
        results = [pool.apply_async(is_prime, (n,)) for n in numbers]
        for r in results:
            print(r) # Asynchronous call in one process
```

Thread pool dans un context server/client

Client

```
# initialisation d'une mq avant
t = getRequest() # Recupere la demande
if t == 1:
    pid = os.getpid()
    m = str(pid).encode()
    mq.send(m, type = 1)
    m, t = mq.receive(type = (pid + 3)) # pid+3 = identifiant
    dt = m.decode()
    print("Server response:", dt)
if t == 2:
    m = b""
    mq.send(m, type = 2)
```

Server

```
def worker(mq, m) :
    msg = str(time.asctime()).encode()
    mq.send(msg, type=int(m.decode()) + 3) # == pid + 3
threads=[]
with concurrent.futures.ThreadPoolExecutor(max_workers = 4) as executor :
    while True :
        m, t = mq.receive() # m est l'identifiant
        if t == 1:
            executor.submit(worker, mq, m) # Lance un thread worker
        if t == 2:
            mq.remove()
            break
```

Lock et Semaphore

```
inpoint = 0 # Variable partagé
def countpi(n, lock):
    i  = n
    global inpoint
    while i :
        x = -1 + 2*random.random()
        y = -1 + 2*random.random()
        if x**2 + y**2 < 1 :
            with lock: # protection variable
                inpoint += 1
            i -= 1
if __name__ == "__main__":
    n = 2000
    lock = threading.Lock()
    threadList = []
    NB_THREAD = 10
    for i in range(NB_THREAD):
        threadList.append(threading.Thread(target=countpi, args=(n,lock)))
    for thread in threadList :
        thread.start()
    for thread in threadList :
        thread.join()
    print("Estimation de pi : ", round((4*(float(inpoint)/float(n*NB_THREAD))), 5))
```

Sémaphore Producteur / Consommateur

```
BUFFER_SIZE = 5
def producer(n, buffer, full, empty, lock): # Ecrivain
```

```
    i = p = 0
    a,b = 0,1
    while i <n+1:
        a,b = b, a+b
        if i == n: # Definir la fin
            a = -1
        empty.acquire() # Prendre un semaphore
        with lock:
            print(a)
            buffer[p]= a
            p = (p+1)%BUFFER_SIZE
            i += 1
        full.release() # Debloquer le lecteur
def consumer(buffer, full, empty, lock): # Lecteur
    q = 0
    while True:
        full.acquire()# Attendre qu'un producer envoie un semaphore
        with lock:
            res = buffer[q]
            print("consumer :",res) # lecture
            q = (q+1)%BUFFER_SIZE
        empty.release() # Envoyer un semaphore
        if res == -1: # Fin
            break
if __name__ == "__main__":
    n = 10
    buffer = array.array('l', range(BUFFER_SIZE))
    lock = threading.Lock()
    full = threading.Semaphore(0)
    empty = threading.Semaphore(BUFFER_SIZE)
    prod = threading.Thread(target = producer, args = (n, buffer, full, empty, lock))
    cons = threading.Thread(target = consumer, args = (buffer, full, empty, lock))
    cons.start()
    prod.start()
    cons.join()
    prod.join()
```

Le diner des philosophes

```
N = 5
class State:
    THINKING = 1
    HUNGRY = 2
    EATING = 3
philosopherloop = True
def philosopher(i, Lock, state, sem):
    while philosopherloop:
        time.sleep(5)
        with Lock :
            state[i] = State.HUNGRY
            if (state[(i + N - 1) % N] != State.EATING) and (state[(i + 1) % N] != State.EATING):
                # Si mes voisins de gauche et de droite ne mange pas, je mange
                state[i] = State.EATING
                sem[i].release()
        sem[i].acquire() # Attendre que mon voisin me débloque
        time.sleep(2) # Le philosopher mange
        with Lock :
            state[i] = State.THINKING
            # Si le voisin de gauche a faim et que son autre voisn ne mange pas
            if state[(i+N-1)%N] == State.HUNGRY and state[(i+N-2)%N] != State.EATING:
                state[(i+N-1)%N] = State.EATING # Le faire manger
                sem[(i+N-1)%N].release() # Le debloquer
            # Si le voisin de droite a faim et que son autre voisn ne mange pas
            if state[(i+1)%N] == State.HUNGRY and state[(i+2)%N] != State.EATING:
                state[(i+1)%N] = State.EATING   # Le faire manger
                sem[(i+1)%N].release() # Le debloquer

if __name__ == "__main__":
    state = []
    sem = []
    Lock = threading.Lock()
    for i in range(N) : # Mettre tout les philos en penseur
        state.append(State.THINKING)
        sem.append(threading.Semaphore(0))
    threads = [threading.Thread(target=philosopher, args=(i, Lock, state, sem)) for i in range(N)]
    for thread in threads :
        thread.start()
    time.sleep(10)
    philosopherloop = False
    for thread in threads :
        thread.join()
    print("FIN")
```