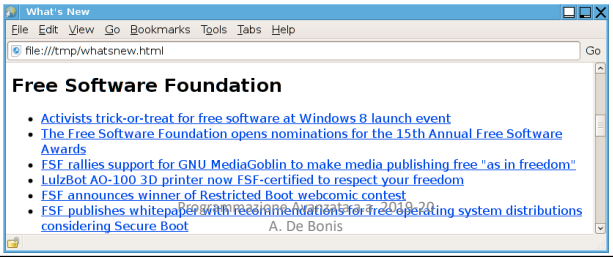
Programmazione Avanzata

Concorrenza I/O Bound

Un esempio

- Scaricare file o pagine web da Internet e` un esigenza molto frequente. A causa dei tempi di latenza della rete, e` di solito possibile fare molti download in modo concorrente e quindi terminare molto piu` velocemente il download.
- Il libro di Summerfield propone un codice che scarica RSS feed (piccoli documenti XML) che riportano storie relative a notizie riguardanti il mondo della tecnologia.
- I feed provengono da diversi siti web e il programma li usa per produrre una singola pagina HTML con i link a tutte le storie.



Un esempio

- La tabella mostra i tempi di varie versioni del programma

Program	Concurrency	Seconds	Speedup
`whatsnew.py`	*None*	172	*Baseline*
`whatsnew-c.py`	16 coroutines	180	0.96×
`whatsnew-q-m.py`	16 processes using a queue	45	3.82×
`whatsnew-m.py`	16 processes using a process pool	50	3.44×
`whatsnew-q.py`	16 threads using a queue	50	3.44×
`whatsnew-t.py`	16 threads using a thread pool	48	3.58×

- Poiche' la latenza della rete varia molto, la velocita` dei programmi puo` variare molto da un minimo di 2 fino ad un massimo di 10 o piu` volte, in base ai siti, la quantita` di dati scaricati e la banda della connessione.
- In considerazione di cio`, le differenze tra la versione basata su multiprocessing e quella basata su multithreading sono insignificanti.
- La cosa importante da ricordare e` che l'approccio concorrente permette di raggiungere velocita` molto piu` elevate nonostante queste varino di esecuzione in esecuzione

Informazioni sul pacchetto threading

- Il modulo threading costruisce interfacce ad alto livello per il threading al top del modulo di basso livello _thread.
- La classe Thread rappresenta un'attivita` che viene eseguita in un thread separato.
- Una volta creato un oggetto thread, si da` inizio alla sua attivita` invocando il metodo start() che invoca il metodo run() del thread in un thread separato.
- Una volta iniziata l'attivita` del thread, il thread viene considerato vivo fino al momento in cui non termina il suo metodo run() (anche se a causa di un'eccezione non gestita)

Informazioni sul pacchetto threading

- Altri thread possono invocare il metodo join() che blocca il thread che lo invoca fino a quando non termina il thread il cui metodo join() e` stato invocato.
- Il thread ha un attributo name il cui valore puo` essere passato al costruttore.
- I thread possono essere contrassegnati come daemon attraverso un flag. Se vi sono solo thread daemon in esecuzione, si esce dall'intero programma. Il valore iniziale del flag e` ereditato dal thread che crea il thread o puo` essere passato al costruttore.
- L'interfaccia di threading.Thread fornita e` simile a quella di multiprocessing.Process.

Un'implementazione con code e threading

- Questo esempio usa thread multipli e due code thread-safe, una per i job (URL) e l'altra per i risultati (coppie contenenti True e un frammento HTML da includere nella pagina HTML da costruire, oppure False e un messaggio di errore)
- La funzione main() comincia ricevendo dalla linea comando il massimo numero di elementi (limit) da leggere da una data URL e un livello di concorrenza (concurrency).
  - La funzione handle_commandline() pone il valore della concorrenza pari a 4 volte il numero di core (si sceglie un multiplo del numero di core, dal momento che il programma e` I/O bound.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

Un'implementazione con code e threading

- Il modulo queue implementa code che possono essere utilizzate da piu` entita`.
- Sono particolarmente utili nel multithreading in quanto consentono a thread multipli di scambiarsi informazioni in modo sicuro.
- Il modulo implementa tre tipi di code: FIFO, LIFO e Coda a priorita`.
- Internamente queste code usano lock per bloccare temporaneamente thread in competizione tra loro.

Un'implementazione con code e threading

- La funzione poi riporta all'utente che sta cominciando a lavorare e mette in filename l'intero percorso del file di dati contenente le URL.
- Poi la funzione crea due code thread-safe e i thread worker.
- Una volta iniziati i thread worker (che sono bloccati perche' non c'e` alcun lavoro da svolgere ancora) vengono aggiunti i job alla coda dei job.
- Si attende quindi nella funzione process() che i job vengano completati e poi vengono forniti in output i risultati.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

Un'implementazione con code e threading

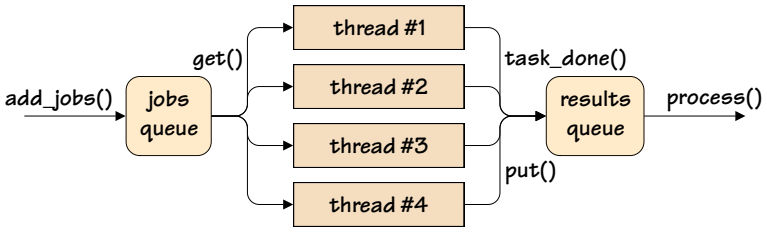
- Questa funzione crea un numero di thread worker pari al valore specificato da concurrency e da` a ciascuno di questi thread una funzione worker da eseguire e gli argomenti con cui la funzione deve essere invocata.
- Ciascun thread viene trasformato in thread daemon in modo che venga terminato al termine del programma.
- Alla fine viene invocato start sul thread che si blocchera` in attesa di un job. In questa attesa sono solo i thread worker ad essere bloccati non il thread principale.

```
def create_threads(limit, jobs, results, concurrency):
    for _ in range(concurrency):
        thread = threading.Thread(target=worker, args=(limit, jobs,
                results))
        thread.daemon = True
        thread.start()
```


47

Un'implementazione con code e threading

- Questa e` la struttura del programma concorrente.



48

Un'implementazione con code e threading

- La funzione Feed.iter() restituisce ciascun feed come una coppia (*title*, *url*) che viene aggiunta alla coda jobs. Alla fine viene restituito il numero di job.
- In questo caso la funzione avrebbe potuto restituire lo stesso valore invocando *jobs*.qsize() piuttosto che computare direttamente il numero di job. Se pero` add_jobs() fosse stato eseguito nel suo proprio thread allora il valore restituito da qsize() non sarebbe stato attendibile dal momento che i job sarebbero stati prelevati nello stesso momento in cui venivano aggiunti.

```
def add_jobs(filename, jobs):
    for todo, feed in enumerate(Feed.iter(filename), start=1):
        jobs.put(feed)
    return todo
```


Un'implementazione con code e threading

- La funzione worker esegue un loop infinito. Il loop infinito termina sicuramente al termine del programma dal momento che il thread e` un daemon.
- La funzione si blocca in attesa di prendere un job dalla coda dei job e non appena prende un job usa la funzione Feed.read() (del modulo Feed.py) per leggere il file identificato dalla URL.
- Se la read fallisce, il flag ok e` False e viene stampato il risultato che e` un messaggio di errore. Altrimenti, sempre che il programma ottenga un risultato (una lista di stringhe HTML), viene stampato il primo elemento (privato dei tag HTML) e aggiunto il risultato alla coda dei risultati.
- Il blocco try ... finally garantisce che jobs.task_done() venga invocato ogni volta che che viene invocato queue.Queue.get() call.

La funzione Feed.read() legge una data URL (feed) e tenta di farne il parsing. Se il parsing ha successo, la funzione restituisce True insieme ad una lista di frammenti HTML. Altrimenti, restituisce False insieme a None o a un messaggio di errore.

```
def worker(limit, jobs, results):
    while True:
        try:
            feed = jobs.get()
            ok, result = Feed.read(feed, limit)
            if not ok:
                Qtrac.report(result, True)
            elif result is not None:
                Qtrac.report("read {}".format(result[0][4:-6]))
                results.put(result)
        finally:
            jobs.task_done()
```

Un'implementazione con code e threading

- Questa funzione viene invocata una volta che i thread sono stati creati e i job aggiunti alla coda. Essa invoca queue.Queue.join() che si blocca fino a quando la coda non si svuota, cioe` fino a che non vengono eseguiti tutti i job o l'utente non cancella l'esecuzione.
- Se l'utente non cancella l'esecuzione, viene invocata la funzione output() per scrivere nel file HTML le liste di link e poi viene stampato un report con la funzione Qtrac.report().
- Alla fine la funzione open() del modulo webbrowser viene invocata sul file HTML per aprirlo nel browser di default.

La funzione output() crea un file whatsnew.html e lo popola con i titoli dei feed e con i loro link. Queste informazioni sono presenti nei result all'interno della coda results. Ogni result contiene una lista di frammenti HTML (un titolo seguito da uno o piu` link).
Al termine output() restituisce il numero di result (numero di jobs terminati con successo) e il nome del file HTML creato.

```
def process(todo, jobs, results, concurrency):
    canceled = False
    try:
        jobs.join() # Wait for all the work to be done
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
    if canceled:
        done = results.qsize()
    else:
        done, filename = output(results)
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done, todo,
            concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```


Un'implementazione che usa Futures e threading

- La funzione main crea un insieme di future inizialmente vuoto e poi crea un esecutore di un pool di thread che lavora allo stesso modo di un esecutore di un pool di processi.
- Per ogni feed, viene creato un nuovo future invocando il metodo concurrent.futures.ThreadPoolExecutor.submit() che eseguira` la funzione Feed.read() sulla URL del feed e restituira` al piu` un numero di link pari a limit.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
        done, filename, canceled = process(futures)
        if canceled:
            executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
            len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```

Un'implementazione che usa Futures e threading

- Ciascun future creato viene aggiunto al pool futures con add().
- Una volta che i future sono stati creati, viene invocata la funzione process() che aspettera` fino a quando non vengono terminati tutti i future o fino a quando l'utente non cancella l'esecuzione.
- Alla fine viene stampato un sunto e se l'utente non ha cancellato l'esecuzione, la pagina HTML generata viene aperta nel browser dell'utente.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
        done, filename, canceled = process(futures)
        if canceled:
            executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
            len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```


Un'implementazione che usa Futures e threading

- Questa funzione scrive l'inizio del file HTML e poi invoca la funzione wait_for() per aspettare che il lavoro venga fatto.
- Se l'utente non cancella l'esecuzione, la funzione itera sui risultati (le coppie gia` descritte) e per quelli che contengono una lista (che consiste di titoli, ciascuno seguito da uno o piu` link) gli elementi della lista vengono scritti nel file HTML.
- Se l'utente cancella l'esecuzione, la funzione calcola semplicemente quanti feed sono stati letti con successo.
- In ogni caso, la funzione restituisce il numero di feed letti, il nome del file e True o False a seconda che l'utente abbia cancellato o meno l'esecuzione.

```
def process(futures):
    canceled = False
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        canceled, results = wait_for(futures)
        if not canceled:
            for result in (result for ok, result in results if ok and
                    result is not None):
                done += 1
                for item in result:
                    file.write(item)
        else:
            done = sum(1 for ok, result in results if ok and result is not
                       None)
        file.write("</body></html>\n")
    return done, filename, canceled
```

Un'implementazione che usa Futures e threading

- Questa funzione itera sui future, bloccandosi fino a quando uno di essi non termina o e` cancellato.
- Una volta ricevuto un future la funzione riporta un errore o un successo e in entrambi i casi appende il Booleano e il risultato (una lista di stringhe o una stringa errore) ad una lista di risultati.

```
def wait_for(futures):
    canceled = False
    results = []
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
                ok, result = future.result()
                if not ok:
                    Qtrac.report(result, True)
                elif result is not None:
                    Qtrac.report("read {}".format(result[0][4:-6]))
                results.append((ok, result))
            else:
                raise err # Unanticipated
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
        for future in futures:
            future.cancel()
    return canceled, results
```


55

Multitasking e coroutine

Programmazione Avanzata

56

Coroutine e concorrenza

- Per svolgere un insieme di operazioni indipendenti, un approccio può essere quello di effettuare un'operazione alla volta con lo svantaggio che se un'operazione è lenta, il programma deve attendere la fine di questa operazione prima di cominciare la prossima.
- Per risolvere questo problema si possono usare le coroutine:
  - ciascuna operazione è una coroutine
  - un'operazione lenta non influenzerà le altre operazioni almeno fino al momento in cui queste non avranno bisogno di nuovi dati da elaborare. Ciò è dovuto al fatto che le operazioni vengono eseguite indipendentemente.

Coroutine e concorrenza

- Supponiamo di avere 3 coroutine che elaborano gli stessi dati e impiegano tempi differenti.
- La coroutine 1 è veloce , la coroutine è lenta, la coroutine3 impiega tempi variabili.
- Una volta che le tre coroutine hanno ricevuto i dati iniziali da eleborare, se una delle tre si trova a dover attendere perché ha finito per prima, le altre continuamo a lavorare minimizzando gli idle time.
- Una volta che le coroutine non servono più , viene invocato close() su ciascuna coroutine in modo che non utilizzino più tempo del processore.

Coroutine e concorrenza

Step	Action	`coroutine1()`	`coroutine2()`	`coroutine3()`
1	Create coroutines	Waiting	Waiting	Waiting
2	`coroutine1.send("a")`	Process `"a"`	Waiting	Waiting
3	`coroutine2.send("a")`	Process `"a"`	Process `"a"`	Waiting
4	`coroutine3.send("a")`	Waiting	Process `"a"`	Process `"a"`
5	`coroutine1.send("b")`	Process `"b"`	Process `"a"`	Process `"a"`
6	`coroutine2.send("b")`	Process `"b"`	Process `"a"` (`"b"` pending)	Process `"a"`
7	`coroutine3.send("b")`	Waiting	Process `"a"` (`"b"` pending)	Process `"b"`
8		Waiting	Process `"b"`	Process `"b"`
9		Waiting	Process `"b"`	Waiting
10		Waiting	Process `"b"`	Waiting
11		Waiting	Waiting	Waiting
12	`coroutine`*`N`*`.close()`	Finished	Finished	Finished

Coroutine e concorrenza:un esempio

- Vogliamo applicare diverse espressioni regolari al testo in un insieme di file HTML.
- Lo scopo è dare in output le URL in ciascun file e gli heading di livello 1 e livello 2.

Coroutine e concorrenza:un esempio

- URL_RE = re.compile(r"""href=(?P<quote>['"])(?P<url>[^\1]+?)""" r"""(?P=quote)""",
  re.IGNORECASE)
- flags = re.MULTILINE|re.IGNORECASE|re.DOTALL
- H1_RE = re.compile(r"<h1>(?P<h1>.+?)</h1>", flags)
- H2_RE = re.compile(r"<h2>(?P<h2>.+?)</h2>", flags)

- Le espressioni regolari (regex) in alto servono a fare il match di una URL e del testo contenuto trai i tag <h1> e <h2>.

Coroutine e concorrenza:un esempio

- Ciascun regex_matcher() è una coroutine che prende una funzione receiver (essa stessa una coroutine) e un regex.
- Ogni volta che il matcher ottiene un match lo invia al receiver.

```
receiver = reporter()
matchers = (regex_matcher(receiver, URL_RE),
            regex_matcher(receiver, H1_RE),
            regex_matcher(receiver, H2_RE))
```

Coroutine e concorrenza:un esempio

- Il matcher entra in un loop infinito e subito si mette in attesa che yield resituisca un testo a cui applicare il regex.
- Una volta ricevuto il testo, il matcher itera su ogni match ottenuto, inviando ciascun match al receiver.
- Una volta terminato il matching la coroutine torna a yield e si sospende nuovamente in attesa di altro testo.

```
@coroutine
  def regex_matcher(receiver, regex):
    while True:
      text = (yield)
      for match in regex.finditer(text):
        receiver.send(match)
```


63

Coroutine e concorrenza:un esempio

- Il programma legge i nomi dei file sulla linea di comando e per ciascuno di essi stampa il nome del file e poi salva l'intero testo del file nella variabile html usando la codifica UTF-8.
- Il programma itera su tutti i matcher e invia il testo ad ognuno di essi.
- Ogni matcher procede indipendentemente inviando ogni match ottenuto alla coroutine reporter
- Alla fine viene invocato close() su ciascun matcher e sul reporter per impedire che i matcher rimangano sospesi in attesa di testo e che il reporter rimanga in attesa di match.

```
try:
    for file in sys.argv[1:]:
      print(file)
      html = open(file, encoding="utf8").read()
      for matcher in matchers:
        matcher.send(html)
  finally:
    for matcher in matchers:
      matcher.close()
    receiver.close()
```


64

Coroutine e concorrenza:un esempio

- La coroutine reporter() è usata per dare in output i risultati.
- Viene creata dallo statement receiver = reporter() ed è passata ad ogni matcher come argomento receiver.
- Il reporter() attende che gli venga spedito un match, quindi stampa i dettagli del match e poi continua ad attendere in un loop infinito fino a quando viene invocato close() su di esso.

```
@coroutine
def reporter():
    ignore = frozenset({"style.css", "favicon.png", "index.html"})
    while True:
        match = (yield)
        if match is not None:
            groups = match.groupdict()
            if "url" in groups and groups["url"] not in ignore:
                print("   URL:", groups["url"])
            elif "h1" in groups:
                print("   H1: ", groups["h1"])
            elif "h2" in groups:
                print("   H2: ", groups["h2"])
```

re.Match.groupdict()
restituisce un dizionario contenente tutti sottogruppi con nome del match e associa come chiave a ciascun sottogruppo il nome del sottogruppo.