Chapter 10 - Processes and Threading

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

1. Using the Multiprocessing Module

2. Using the Threading Module

Processes and Threading

- Computer processors with more than one processing unit are everywhere
  - These are designated as *multicore* processors
  - It is likely that you have one, e.g., in your smartphone
- We want to get the most out of all the available cores
  - By spreading the processing load through them

- There are two main approaches to spreading the workload
  1. Using multiple processes
  2. Using multiple threads

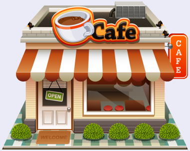
Processes and Threading

- It is beyond the scope of CS 172 to teach all the details about processes and threads
  - These will certainly come later, e.g., in an *Operating Systems* course
- For now, we will care to provide a gentle introduction
- And, more importantly,
  - We will focus on teaching you how to handle processes and threads in Python
  - And illustrate possible uses for those constructions!

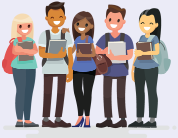
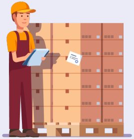
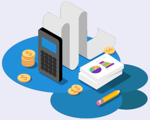
A gentle introduction to Processes and Threading

Imagine you want to increase the business of a coffee shop

- Which is currently being served by (only) one person:

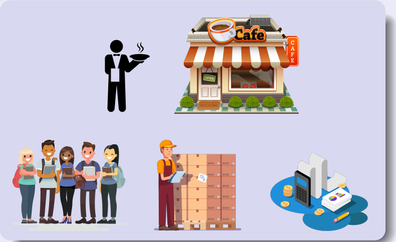
- Do note that the person needs to handle multiple tasks:
  - Even if he/she can only do one task at each time!

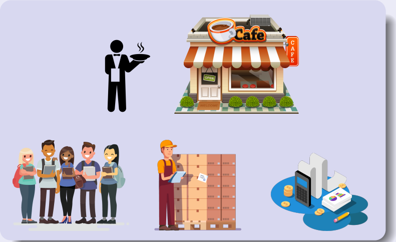




- How can this be achieved?

A gentle introduction to Processes and Threading

One possibility is to open another shop



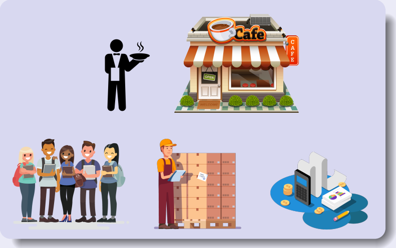


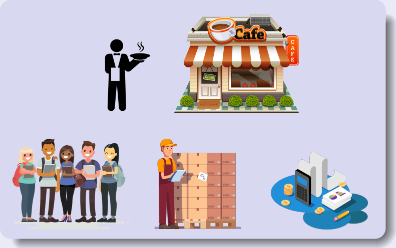
- Each shop runs in its own environment
  - Each shop, e.g., has its own inventory
- So shops are independent!

This corresponds to a **multiple processing** strategy

A gentle introduction to Processes and Threading

One possibility is to open another shop



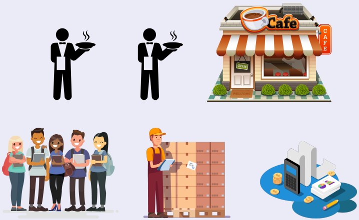


- Each shop runs in its own environment
  - Each shop, e.g., has its own inventory
- So shops are independent!

This corresponds to a **multiple processing** strategy

A gentle introduction to Processes and Threading

An alternative possibility is to hire more waiters

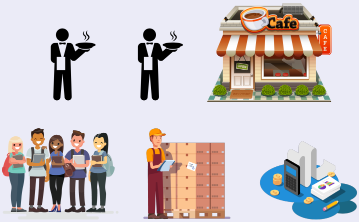


- Waiters can be performing different tasks
- But they are working over the same resources!
  - If only one cake slice exists, they can not serve it to two customers!

This corresponds to a **multiple threading** strategy

A gentle introduction to Processes and Threading

An alternative possibility is to hire more waiters



- Waiters can be performing different tasks
- But they are working over the same resources!
  - If only one cake slice exists, they can not serve it to two customers!

This corresponds to a **multiple threading** strategy

A gentle introduction to Processes and Threading

For computers, it is important to remember that:

- Each program needs its resources to be able of executing:
  - such as, e.g., the values of its variables
  - or the instructions that build it

Using multiple processes:

- Each process is treated as an independent program
  - Each has its own resources
  - Run independently
  - Concurrency is handled by the OS, which simplifies our work
  - Sharing data among processes, and aggregating their results, is hard!

Using multiple threads:

- Only one process is running, but it has multiple collaborating threads
  - Resources are shared, which must be handled with care!
  - Concurrency must be handled by the programmer
  - Simpler to combine/treat the results from the multiple threads!

A gentle introduction to Processes and Threading

For computers, it is important to remember that:

- Each program needs its resources to be able of executing:
  - such as, e.g., the values of its variables
  - or the instructions that build it

Using multiple processes:

- Each process is treated as an independent program
  - Each has its own resources
  - Run independently
  - Concurrency is handled by the OS, which simplifies our work
  - Sharing data among processes, and aggregating their results, is hard!

Using multiple threads:

- Only one process is running, but it has multiple collaborating threads
  - Resources are shared, which must be handled with care!
  - Concurrency must be handled by the programmer
  - Simpler to combine/treat the results from the multiple threads!

A gentle introduction to Processes and Threading

For computers, it is important to remember that:

- Each program needs its resources to be able of executing:
  - such as, e.g., the values of its variables
  - or the instructions that build it

Using multiple processes:

- Each process is treated as an independent program
  - Each has its own resources
  - Run independently
  - Concurrency is handled by the OS, which simplifies our work
  - Sharing data among processes, and aggregating their results, is hard!

Using multiple threads:

- Only one process is running, but it has multiple collaborating threads
  - Resources are shared, which must be handled with care!
  - Concurrency must be handled by the programmer
  - Simpler to combine/treat the results from the multiple threads!

Outline

Using the Multiprocessing Module

Our use case for showcasing the use of multiple processes

- You have a program that implements a desired functionality
- And you want to automate its use!
- The user runs a *parent* program
  - which in turn runs as many instances of a *child* program as necessary
- In Python, this can be done using the `subprocess` module, which
  - Provides facilities for running other programs and
  - Passing any command-line options we want and, if desired,
  - Communicating with them using pipes

Using the Multiprocessing Module

Our example program

- We want to search for a word specified on the command line
- In the files listed after that word
- The *parent* program is provided in file `grepword-p.py`
- The *child* program is provided in file `grepword-p-child.py`
- The following pseudo-invocation

  ```
  > python grepword-p.py _word_ _file1_ _file2_ ... _filen_
  ```

  would search for `_word_` in the provided `n` files
- The output would be the names of the files which include `_word_`

Using the Multiprocessing Module

Our example parent program (1/3)

- The heart of `grepword-p.py` is encapsulated in its `main()` function
  - which we will look at in three parts:

```
def main():
    child = os.path.join(os.path.dirname(__file__), "grepword-p-child.py")
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist) // opts.count
    ...
```

- in the first line, we begin by getting the name of the child program
- then we get the user's command-line options
  - using `parse_options()`, which relies on module `optparse` to return:
    - the `opts` named tuple, which indicates whether the program should recurse into sudirectories and the count of how many processes to use
    - the word to search for and the list of files given on the command line
- the `get_files()` function returns a list of files to be read

Using the Multiprocessing Module

Our example parent program (1/3)

```
def main():
    ...
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1
    ...
```

- We then calculate how many files are given to each (sub)process to work on
- `start` and `end` are used to specify the slice of the `filelist` that will be given to the next child process
  - The number of files may not be a multiple of the number of processes
    - we increase the number of files for the first process by the remainder
- `number` is used purely for debugging (represents the number of a child)

Using the Multiprocessing Module

Our example parent program (2/3)

```
def main():
    ...
    pipes = []
    while start < len(filelist):
        command = [sys.executable, child]
        if opts.debug:
            command.append(str(number))
        ...
```

- For each `start:end` slice of the `filelist` we create a command list
  - consisting of the Python interpreter, `sys.executable`
  - the child program we want to execute
  - and the command-line options
    - in this case, just the child number if we are debugging

Using the Multiprocessing Module

Our example parent program (2/3)

```
...
pipes = []
while start < len(filelist):
    ...
    pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
    pipe.stdin.write(word.encode("utf8") + b"\n")
    for filename in filelist[start:end]:
        pipe.stdin.write(filename.encode("utf8") + b"\n")
    pipe.stdin.close()
    pipes.append(pipe)
    number += 1
    start, end = end, end + files_per_process
```

- We then create a `subprocess.Popen` object (`pipe`)
    - specifying the command to execute (as a list of strings – `command`)
    - and in this case requesting to write to the process's standard input
- We send the word to search to the `stdin`, followed by a newline
- We also send every file in the relevant slice of the file list
- Finally, we close the standard input of this particular process
- We add the process to the list of processes and prepare next iteration

Using the Multiprocessing Module

Our example child program (1/2)

```
number = "{0}: ".format(sys.argv[1]) if len(sys.argv) == 2 else ""
stdin = sys.stdin.buffer.read()
lines = stdin.decode(encoding="utf8", errors="ignore").splitlines()
word = lines[0].rstrip()
```

- We start by setting `number` as a string with a given debugging number it is exists
  - or to an empty string if we are not debugging
- Since the program is running as a child process and the `subprocess` module only reads/writes binary data and uses the local encoding
  - we must read `sys.stdin`'s underlying buffer of binary data
- Once we read the binary data, we decode it into a Unicode string and split it into lines
- The first line contains the search word

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    ...
```

- All lines after the first are filenames with paths (the first is the search word)

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                ...
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

- It is possible that some files are large, and this can be a problem
  - Specially if there are 20 child processes running concurrently
  - We handle this by reading each file in blocks of size `BLOCK_SIZE`
- It if fails to read a block, we go to the next file (`break`)
- Another advantage is that we can stop reading the file as soon as we find the word

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                ...
                current = current.decode(encoding="utf8", errors="ignore")
                ...
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

- The files are read in binary mode (`"rb"`): we must convert blocks to strings
  - Before we can search it, since the search word is a string
  - We assumed that all the files use the UTF-8 encoding
- A more sophisticated program would try to determine the encoding and reopen the file using the correct encoding

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                ...
                if (word in current or
                    word in previous[-len(word):] + current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                ...
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

- If a block contains the search word, we print its name
- We keep the previous block to ensure that we don't miss cases when the only occurrence of the search word happens to fall across two blocks

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                ...
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

- It stops when the current block is smaller than `BLOCK_SIZE` (since there nothing else to read in the file after)

Using the Multiprocessing Module

Our example child program (2/2)

```
for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                current = current.decode(encoding="utf8", errors="ignore")
                if (word in current or
                    word in previous[-len(word):] +
                            current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
    except EnvironmentError as err:
        print("{0}{1}".format(number, err))
```

Outline

Using the Threading Module

- Setting up 2 or more separate threads of execution in Python is simple
- The complexity comes from needing to share data among threads
- Imagine we have two threads sharing a list `L`
  - one might be iterating over it using `for x in L`
  - and the other would come and delete some items in the list
  - at best, this will lead to obscure crashes, at worst at incorrect results
- One common solution is to use some kind of locking mechanism
  - one thread might acquire a lock and *then* start iterating over the list
  - any other thread will then be blocked by the lock
  - Actually, the *second* thread trying to acquire the lock will be blocked until the *first* releases it

Using the Threading Module

- One problem with locking is the risk of **deadlock**
  - Suppose *thread #1* acquires *lock A* so that it can access *shared data a*
  - and then, in the scope of *lock A*, it tries to acquire *lock B* so that it can access *shared data b*
  - but it can not acquire *lock B* because, meanwhile, *thread #2* has acquired *lock B* so that it can access *b*
  - and is itself now trying to acquire *lock A* so that it can access *a*
  - So, *thread #1* holds *lock A* and is trying to acquire *lock B*, while *thread #2* holds *lock B* and is trying to acquire *lock A*
  - **Both threads are blocked**

Using the Threading Module

- Deadlocks might be simple to understand, but they are sometimes not simple to handle in practice
- One way to avoid them is to define a policy with the order in which locks should be acquired
  - if we define that *lock A* must always be acquired before *lock B*
  - if we then wanted to acquire *lock B*, this policy requires us to first acquire *lock A*
  - So, the previously described deadlock would not be possible!
- Also, any thread waiting to acquire a lock is not doing useful work!

Using the Threading Module

- Every Python program has at least one thread, the main thread
- To create multiple threads, we must import and use the `threading` module
- There are two ways to create threads:
  1. calling `threading.Thread()` and pass it a callable object
  2. subclassing the `threading.Thread` class
     - This is the most flexible approach
     - Subclasses can reimplement `__init__()`, whose reimplementation *must* call the base class implementation
     - Subclasses *must* reimplement `run()`, since it is in this method that the thread's work is done
     - The `run()` method must *never* be called by our code
     - Threads are started by calling the `start()` method, which will call `run()`
     - No other `threading.Thread` may be reimplemented, although adding other methods is fine

Using the Threading Module

Our example program

- We go back to our word search example
- We start by reviewing the `grepword-t.py` program
  - Which does the same thing as `grepword-p.py`
  - But does so delegating the work to multiple threads
- `grepword-t.py` does not appear to use any locks at all
- This is possible because the only shared data is a list of files
  - and for this we use the `queue.Queue` class
  - This class is special since it handles all the locking itself internally
  - Whenever we access it to add or remove items, we can rely on the queue itself to *serialize* accesses
  - In the context of threading, serializing access to data means ensuring that only one thread at a time has access to the data
  - Another benefit is that we don't have to share out the work ourselves
    - we simply add items to the queue and leave the worker threads to pick up work whenever they are ready

Using the Threading Module

Our threaded main program

- The heart of grepword-t.py is encapsulated in its main() function:

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

- Getting the user's options and the file list are the same as before
- We then create a queue.Queue
- And loop as many times as there are threads to be created

Using the Threading Module

Our threaded main program

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

- For each thread, we prepare a number string
- And we create a `Worker` instance (a `threading.Thread` subclass)
- Next we start the thread, although at this point it has no work to do:
  - The work queue is empty, so the thread will immediately be blocked trying to get some work

Using the Threading Module

Our threaded main program

- With all the threads created and ready to work, we iterate over all the files, adding each one to the work queue
- Once the 1^{st} file is added, one of the threads can start working on it
  - and so on until all the threads have a file to work on
- As soon as a thread finishes working on a file it can get another one, until all the files are processed
- This differs from `grepword-p.py` where we had to allocate slices of the file list to each child process
  - and the child processes were started and given their lists sequentially

Using the Threading Module

Our threaded main program

- The program will not terminate while it has any threads running
  - This is a problem because once the worker threads have done their work, although they have finished they are technically still running
- The solution is to turn threads into daemons
- The program will terminate as soon as the program has no nondaemon threads running
- The main thread is not a deamon
  - So once the main thread finishes, the program will cleanly terminate each deamon thread and then terminate itself
- Of course, this can now create the opposite problem:
  - Once threads are up and running we must ensure the main thread does not finish until all the work is done
- This is achieved by calling `queue.Queue.join()`
  - This method blocks until the queue is empty

Using the Threading Module

The beginning of the Worker class

```
class Worker(threading.Thread):

    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

- The `__init__()` method must call the base class `__init__()`
- The work queue is the same `queue.Queue` shared by all threads

Using the Threading Module

The beginning of the Worker class

```
class Worker(threading.Thread):

    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

- The `run()` method runs an *infinite* loop
    - at each iteration we call `queue.Queue.get()` to get the next file
        - This call will block if the queue is empty, and does not have to be protected by a lock because `queue.Queue` handles that automatically
    - Once we have a file we process it

Using the Threading Module

The beginning of the Worker class

```
class Worker(threading.Thread):

    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

- Finally, we must tell the queue that we have done that particular job
  - calling `queue.Queue.task_done()` is essential to the correct working of `queue.Queue.join()`
- We will not show the `process()` function
  - its code is very similar to the one shown previously (slides 19-22)