Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

MCS 275 Lecture 32
Programming Tools and File Management
Jan Verschelde, 31 March 2017

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

Serving Multiple Clients

avoid to block clients with waiting

Client/Server computing:

1. server provides hardware or software service,
2. clients make requests to server.

Typically, clients and servers run on different computers.

Operations in server side scripting:

1. define addresses, port numbers, sockets,
2. wait to accept client requests,
3. service the requests of clients.

For multiple clients, the server could either

1. wait till every one is connected before serving, or
2. accept connections and serve one at a time.

In either protocol, clients are blocked waiting.

A Print Server

Imagine a printer shared between multiple users connected in a computer network.

Handling a request to print could include:

1. accepting path name of the file,
2. verifying whether the file exists,
3. placing request in the printer queue.

Each of these three stages involves network communication with possible delays during which other request could be handled.

A multithreaded print server has multiple handlers running simultaneously.

Waiting for Client Information

interactive dialogues

Imagine an automated bank teller:

1. ask for account number,
2. client must provide password,
3. prompt for menu selection, etc...

Each of these questions may lead to extra delays during which other clients could be serviced.

Running multiple threads on server will utilize the time server is idle waiting for one client for the benefit of the request of other clients.

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

Using Sockets and Threads

object oriented design

The main program will

1. define address, port numbers, server socket,
2. launch the handler threads of the server.

Typically we will have as many handler threads as the number of connections the server listens to.

Inheriting from the class `Thread`,

1. the constructor of the handler will store a reference to the socket server as data attribute,
2. the `run` contains the code to accept connections and handle requests.

Locks are needed to manage shared resources safely.

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

Waiting for Data from 3 Clients

a simple multithreaded server

Suppose 3 clients send a message to a server.
Application: collect a vote from three people.

Clients behave as follows:

1. may connect at any time with the server,
2. getting message typed in takes time.

Multithreaded server listens to 3 clients:

1. three threads can handle requests,
2. each thread simply receives message,
3. server closes after three threads are done.

a multithreaded server accepts three clients

```
$ python mtserver.py
give the number of clients : 3
server is ready for 3 clients
server starts 3 threads
0 accepted request from  ('127.0.0.1', 49153)
0 waits for data
1 accepted request from  ('127.0.0.1', 49154)
1 waits for data
1 received  this is B
0 received  this is A
2 accepted request from  ('127.0.0.1', 49155)
2 waits for data
2 received  this is C
$
```

clients run simultaneously in separate terminals

```
$ python mtclient.py
client is connected
Give message : this is A
$

$ python mtclient.py
client is connected
Give message : this is B
$

$ python mtclient.py
client is connected
Give message : this is C
$
```

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

client sends a message to the server

```
from socket import socket as Socket
from socket import AF_INET, SOCK_STREAM

HOSTNAME = 'localhost'   # on same host
PORTNUMBER = 11267       # same port number
BUFFER = 80              # size of the buffer

SERVER_ADDRESS = (HOSTNAME, PORTNUMBER)
CLIENT = Socket(AF_INET, SOCK_STREAM)
CLIENT.connect(SERVER_ADDRESS)

print('client is connected')
DATA = input('Give message : ')
CLIENT.send(DATA.encode())

CLIENT.close()
```

`main()` in the server

First, we define the server socket:

```
def main():
    """
    Prompts for number of connections,
    starts the server and handler threads.
    """
    nbr = int(input('give the number of clients : '))
    buf = 80
    server = connect(nbr)
    print('server is ready for %d clients' % nbr)
```

The definition of the server socket to serve a number of clients happens in the function `connect`.

`main()` continued, lauching the threads

Every client is served by a different thread.

```
handlers = []
for i in range(nbr):
    handlers.append(Handler(str(i), server, buf))
print('server starts %d threads' % nbr)
for handler in handlers:
    handler.start()
print('waiting for all threads to finish ...')
for handler in handlers:
    handler.join()
server.close()
```

Important: *before* the `server.close()`
we *must* wait for all handler threads to finish.

the function `connect()`

```
def connect(nbr):
    """
    Connects a server to listen to nbr clients.
    Returns the server socket.
    """
    hostname = ''          # to use any address
    portnumber = 11267 # number for the port
    server_address = (hostname, portnumber)
    server = Socket(AF_INET, SOCK_STREAM)
    server.bind(server_address)
    server.listen(nbr)
    return server
```

the handler class

```
from threading import Thread

class Handler(Thread):
    """
    Defines handler threads.
    """
    def __init__(self, n, sock, buf):
        """
        Name of handler is n, server
        socket is sock, buffer size is buf.
        """
    def run(self):
        """
        Handler accepts connection,
        prints message received from client.
        """
```

the constructor of the Handler thread

The object data attributes for each handler
are server socket and buffer size.

```
def __init__(self, n, sock, buf):
    """
    Name of handler is n, server
    socket is sock, buffer size is buf.
    """
    Thread.__init__(self, name=n)
    self.srv = sock
    self.buf = buf
```

code for the Handler thread

```
def run(self):
    """
    Handler accepts connection,
    prints message received from client.
    """
    handler = self.getName()
    server = self.srv
    buffer = self.buf
    client, client_address = server.accept()
    print(handler + ' accepted request from ', \
        client_address)
    print(handler + ' waits for data')
    data = client.recv(buffer).decode()
    print(handler + ' received ', data)
```

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

n Handlers for m Requests

Consider the following situation:

1. n threads are available to handle requests,
2. a total number of m requests will be handled,
3. m is always larger than n.

Example application:

- Calling center with 8 operators is paid to handle 100 calls per day.
- Some operators handle few calls that take long time.
  Other operators handle many short calls.

Computational science application: dynamic load balancing.
Handle a number of jobs with unknown duration for each job.

Management Handlers Threads

Previous multithreaded server, after starting threads:

1. request of client accepted by one handler,
2. request is handled by a thread,
3. after handling request, the thread dies.

Simple protocol, but server must wait till all threads have received and handled their request.

To handle *m* requests by *n* server threads:

1. server checks status of thread `t`: `t.isAlive()`,
2. if thread dead: increase counter of requests handled,
3. start a new thread if number of requests still to be handled is larger than number of live threads.

running the server

```
$ python mtnserver.py
give the number of clients : 3
give the number of requests : 7
server is ready for 3 clients
server starts 3 threads
0 is alive, cnt = 0
1 is alive, cnt = 0
2 is alive, cnt = 0
0 accepted request from  ('127.0.0.1', 49234)
0 waits for data
0 is alive, cnt = 0

... to be continued ...
```

running the server continued

```
1 is alive, cnt = 0
2 is alive, cnt = 0
0 received  1
1 accepted request from  ('127.0.0.1', 49235)
1 waits for data
0 handled request 1
restarting 0
1 is alive, cnt = 1
2 is alive, cnt = 1
1 received  2
0 is alive, cnt = 1
1 handled request 2
restarting 1
2 is alive, cnt = 2

... etcetera ...
```

Multithreaded Servers

1. Serving Multiple Clients
   - avoid to block clients with waiting
   - using sockets and threads
2. Waiting for Data from 3 Clients
   - running a simple multithreaded server
   - code for client and server
3. n Handler Threads for m Client Requests
   - managing threads dynamically
   - code to manage handler threads
4. Pleasingly Parallel Computations

modified `main()`

```
def main():
    """
    Prompts for the number of connections,
    starts the server and the handler threads.
    """
    ncl = int(input('give the number of clients : '))
    mrq = int(input('give the number of requests : '))
    server = connect(ncl)
    print('server is ready for %d clients' % ncl)
    handlers = []
    for i in range(ncl):
        handlers.append(Handler(str(i), server, 80))
    print('server starts %d threads' % ncl)
    for handler in handlers:
        handler.start()
    manage(handlers, mrq, server)
    server.close()
```

code to manage Handler threads

A thread cannot be restarted,
but we can create a new thread with same name.

Recall that the server maintains a list of threads.

```
def restart(threads, sck, i):
    """
    Deletes the i-th dead thread from threads
    and inserts a new thread at position i.
    The socket server is sck.  Returns new threads.
    """
    del threads[i]
    threads.insert(i, Handler(str(i), sck, 80))
    threads[i].start()
    return threads
```

the function `manage()`

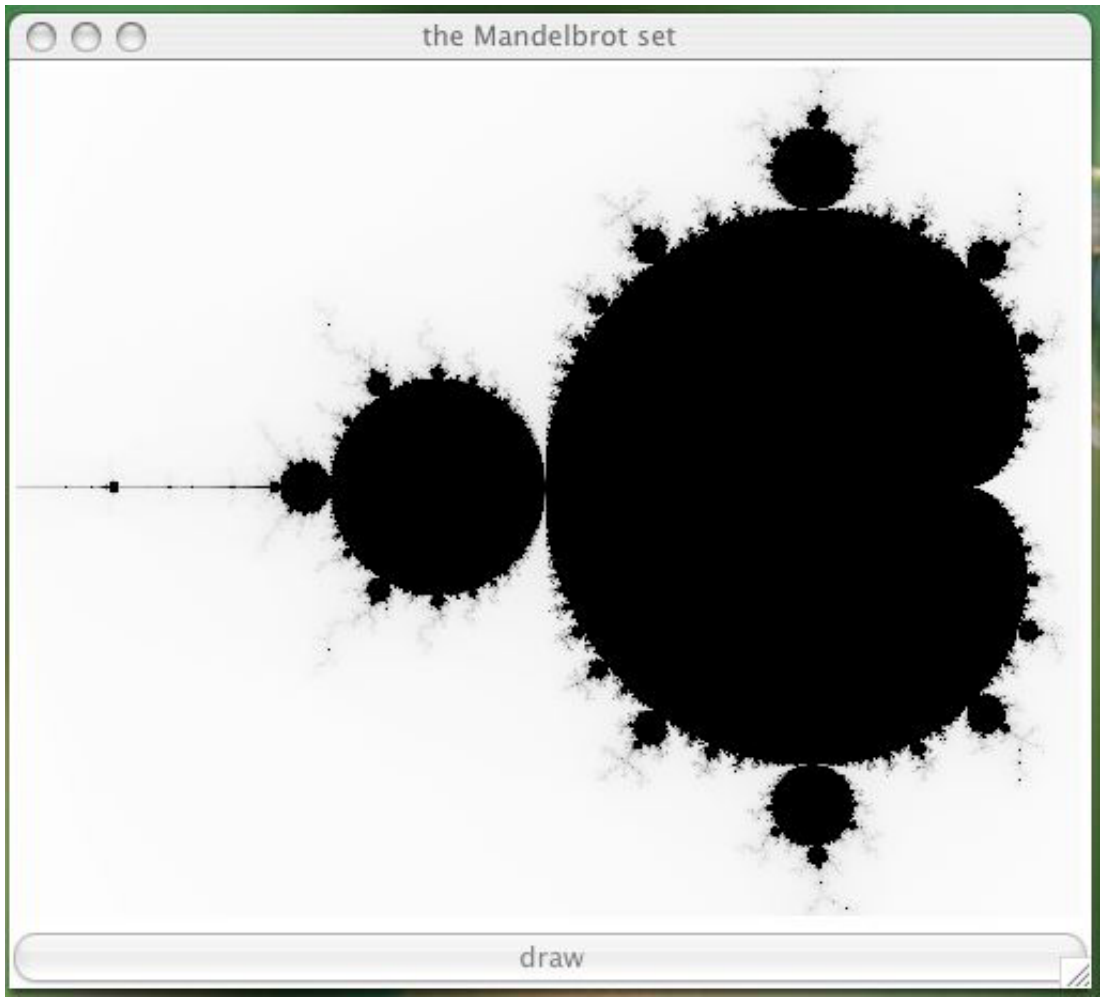
```
def manage(threads, mrq, sck):
    """
    Given a list of threads, restarts threads
    until mrq requests have been handled.
    The socket server is sck.
    """
    cnt = 0
    nbr = len(threads)
    dead = []
    while cnt < mrq:
        sleep(2)
        for i in range(nbr):
            if i in dead:
                pass
            elif threads[i].isAlive():
                print(i, 'is alive, cnt =', cnt)
            else:
                                # continues on the next slide
```

code for `manage()` continued

```
            else:
                cnt = cnt + 1
                print('%d handled request %d' % (i, cnt))
                if cnt >= mrq:
                    break
                elif cnt <= mrq-nbr:
                    print('restarting', i)
                    threads = restart(threads, sck, i)
                else:
                    dead.append(i)
```

Observe that only the main server manipulates the counter, locks are not needed.

The Mandelbrot Set



Definition of the Set

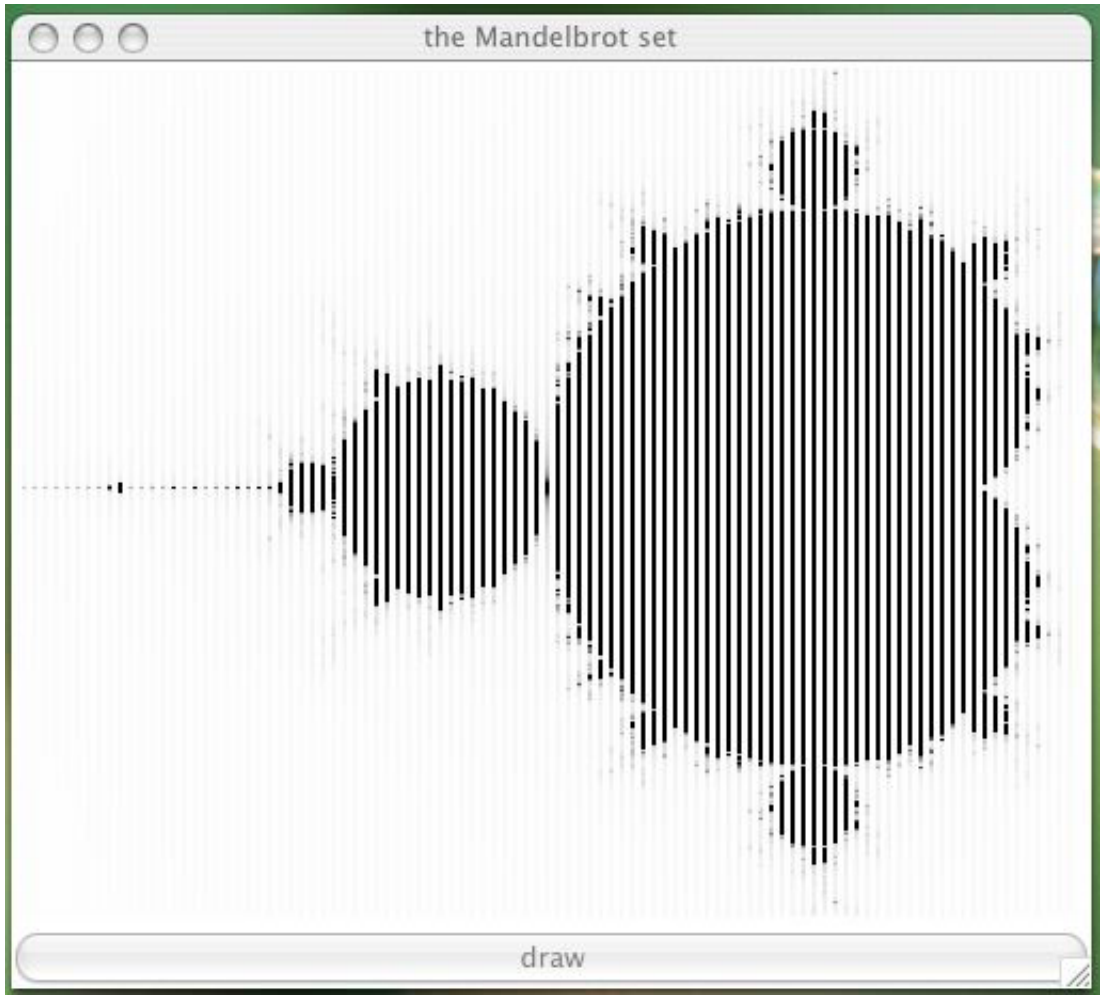
The Mandelbrot set is defined by an iterative algorithm:

- A point in the plane with coordinates (x, y) is represented by $c = x + iy$, $i = \sqrt{-1}$.
- Starting at $z = 0$, count the number of iterations of the map $z \rightarrow z^2 + c$ to reach $|z| > 2$.
- This number k of iterations determines the inverted grayscale $255 - k$ of the pixel with coordinates (x, y) in the plot for $x \in [-2, 0.5]$ and $y \in [-1, +1]$.

To display on a canvas of 400 pixels high and 500 pixels wide we need 16,652,580 iterations.

Pleasingly parallel: different threads on different pixels require no communication.

One Fifth of the Plot



Manager/Worker Paradigm

implemented via client/server parallel computations

The server *manages* a job queue, e.g.: number of columns on canvas to compute pixel grayscales of.

The n handler threads serve n clients. The clients perform the computational *work*, scheduled by the server.

Here we use a simple static workload assignment.

Handler thread t manages all columns k: $k \mod n = t$.
E.g.: $n = 2$, first thread takes even columns,
second thread takes odd columns.

A client receives the column number from the server
and returns a list of grayscales for each row.

For code, see Project 4 solution of Spring 2008.

Summary + Assignments

Assignments:

1. Only threads can terminate themselves. Write a script that starts 3 threads. One thread prompts the user to continue or not, the other threads are busy waiting, checking the value of a shared boolean variable. When the user has given the right answer to terminate, the shared variable changes and all threads stop.
2. Write a multithreaded server with the capability to stop all running threads. All handler threads stop when one client passes the message "stop" to one server thread.
3. Use a multithreaded server to estimate π counting the number of samples $(x, y) \in [0, 1] \times [0, 1]$: $x^2 + y^2 \leq 1$. The job queue maintained by the server is a list of tuples. Each tuple contains the number of samples and the seed for the random number generator.