Network programming with Python

Yves ROUDIER

Yves.Roudier@eurecom.fr

Based on slides from Corrado LEITA

What are we going to talk about?

- 3 lectures
  - Dec 18: Python primer and introduction to sockets
  - Jan 07: Synchronous socket programming and threads
  - Jan 21: Asynchronous socket programming and twisted python

- **Homework**

Where can I find additional material?

- Slides and code downloadable from intranet
- Additional documentation:
  - Python Tutorial (highly recommended) http://docs.python.org/tutorial/
  - Kurose, Ross, "Computer Networking, a Top Down Approach" (Chapter 2)
  - Stevens, "UNIX Network Programming", Volume 1
- You will find in these slides everything you need!

1. Introduction to python
2. Sockets in UNIX
3. Concurrent servers
4. Threads

INTRODUCTION TO PYTHON

Python "propaganda"

- Software Quality
  - Force programmers to write readable code
  - Explicit and minimalistic language design making code easy to understand
- Developer Productivity
  - Optimized for development speed
  - Large standard library
- Program portability
  - Python programs mostly run without modifications on Linux/MacOS/Windows/Android/iOS/… ☺

Python main characteristics

- Automated memory management
- Small core language, very easy to learn
- Large standard library
- Strong and dynamically typed
- Indentation matters
- Easily extendable using C
  - High level of similarity between the standard library and the underlying C APIs
  - Python sockets are almost completely mappable to the POSIX socket API

Running python

- Interpreter can receive the program as argument (python script.py) or can be executed in interactive mode
- Code is compiled into bytecode and executed in a virtual machine
  - Python script: .py extension
  - Python bytecode: .pyc extension
- Compilation and execution are transparent to the user

Automatic memory management

```
corrado@Daphne ~ $ python
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a="just a string"
>>> a+=" a bit more text"
>>> a="a new string"
>>>
```

The string size is dynamically allocated, no need to declare it in advance

New string object allocated, old string object automatically garbage collected

- Python never requires the programmer to explicitly allocate or deallocate memory

Strong and dynamic typing

```
corrado@Daphne ~ $ python
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a="just a string"
>>> b=2
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> b=str(b)
>>> a+b
'just a string2'
>>>
```

type(a)==str
type(b)==int

type(a)==str
type(b)==str

- Strong typing: restrictions on operations among different data types
- Dynamic typing: values have types, but variables have not

Indentation matters

```
corrado@Daphne ~ $ python
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a=12
>>>  print a
  File "<stdin>", line 1
    print a
    ^
IndentationError: unexpected indent
>>> if a==12:
...  print a
...
12
>>>
```

The space before the command is considered as indentation, which is not expected at this point

After the "if" clause an indented block is instead expected. Any line starting with one space will be considered part of the same indentation block

- Best practice:
  - Avoid hard tabs
  - Always use 4 spaces per indentation level
    - See https://www.python.org/dev/peps/pep-0008/

Indented blocks

Multi-line comments are enclosed by """ . The first line of a python function is always considered as a documentation string

```
>>> def my_max_function(a,b):
...         """
...         This is a simple python function
...         """
...         if a>=b:
...                 #a is bigger
...                 return a
...         else:
...                 return b
...
>>> my_max_function(2,3)
3
```

This is a single-line comment

- Of course, indentation can be nested

Objects

- In Python, everything is an object. Every object has:
  - A type (that cannot be changed) which determines the supported operations
  - An identity (that cannot be changed)
  - A value
    - Mutable types: value can be modified after object creation
    - Immutable types: value cannot be modified after creation

Objects

```
corrado@Daphne ~ $ python
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a=12
>>> id(a)
140663588987776
>>> type(a)
<type 'int'>
>>> isinstance(a,int)
True
>>>
>>> l=[1,2,3]
>>> id(l)
4550068992
>>> type(l)
<type 'list'>
>>>
>>> a+=1
>>> id(a)
140663588987752
>>>
>>> l+=[4,5]
>>> id(l)
4550068992
>>>
```

Variable of type "int", identity 140663…6 and value 12

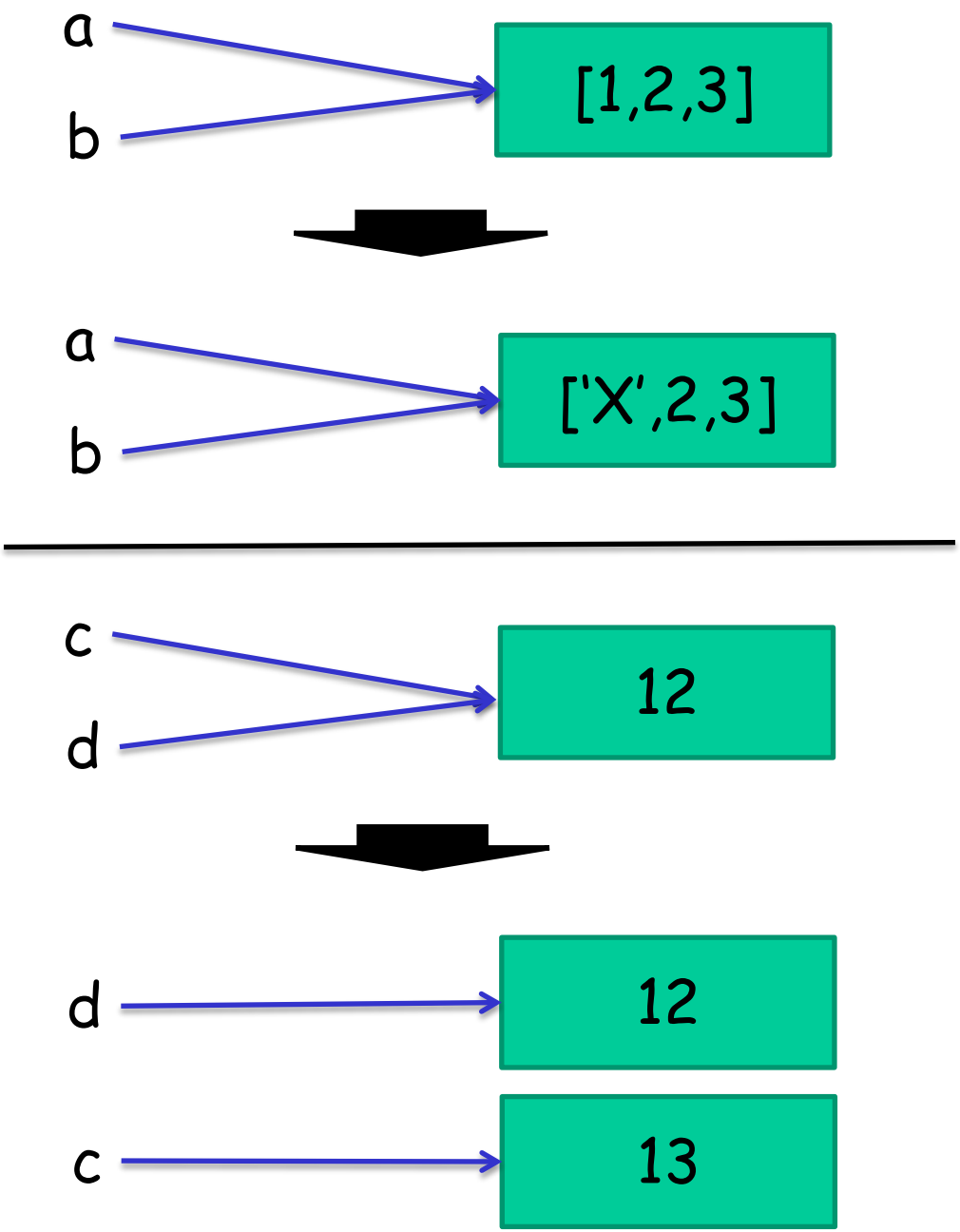
Variable of type "list", identity 455006…2 and value [1,2,3]

Integers are immutable: increasing the value of variable a leads to the generation of a new object with different identity

Lists are mutable: the object value can be modified

Objects

```
>>> a=b=[1,2,3]
>>> a[0]='X'
>>> b
['X', 2, 3]
>>> c=d=12
>>> c+=1
>>> d
12
>>>
```



Objects

```
>>> l=[1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice_
_', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter_
_', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '_
_reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__'
, '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', '
extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> print l.__doc__
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
>>>
```

- Use dir( ) to enumerate the methods and fields of an object
- Use help( ) to access the object documentation (or obj.__doc__)

Modules

- Any python source file can be imported as a module in another source file as a way to organize code in complex programs.
- Example: import os module (part of standard library) and call the popen function
  - import os -> os.popen() **SAFE**
  - import os as o -> o.popen() **SAFE**
  - from os import popen -> popen() **BAD**
  - from os import popen as po -> po() **BAD**
  - from os import * -> popen() **BAD!!!**

Packages

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

- Way to structure python's module namespace by using "dotted module names"
  - import sounds.effects.echo
  - from sounds.effects import echo
  - from sounds.effects.echo import echo_function
- __init__.py must be present in the directory

Builtin data types

- Numbers
  - a=13
  - a/4==3
  - a/4.==3.25
- Strings
  - a="my "+'string'
  - a.find('s')==3
  - a.split(' ') == ["my","string"]
  - "b"*3=="bbb"
  - "val=%d.%d"%(1,2)
- Lists
  - a=[]
  - a.append(2)
  - a+=[3,4,5]
  - a.remove(3)
  - a[0]+=1
- Tuples
  - Same as lists, but immutable
  - a=(1,2,3)
- Sets
  - a=set()
  - a.add(3)
  - a.add(3)
  - len(a)==1
  - set([1,2]).issubset(set([1,2,3,4]))

Builtin data types

- Dictionaries
  - Associative arrays, no constraint on data types
  - Keys must be immutable

  - a={"key1":[1,2,3],"key2":"test"}
  - a.values()==['test', [1, 2, 3]]
  - a.keys()==['key2', 'key1']
  - a.items()==[('key2', 'test'), ('key1', [1, 2, 3])]
  - ('key2' in a)==True
- Null value
  - None

Control structures

```
if <condition>:
   <code>
[elif <condition>:
   <code>]
[else:
   <code>]
```

```
for vars in <iterable>:
   <code>
[else:
   <code>]
```

```
while <condition>:
   <code>
[else:
   <code>]
```

- Continue skips to the next iteration
- Break terminates the loops without executing the else block

Useful constructs

- Traditional loop over numerical values:
  - for x in range(start,end[,step]):
- Looping and keeping track of the index:
  - for index,value in enumerate(['A','B','C']):
- Looping over two lists at the same time:
  - for vala,valb in zip(lista,listb):
- Efficient file reading:
  - for line in file('my_file.txt'):
- List comprehensions:
  - [x**2 for x in range(start,stop)]
- Mapping:
  - map(str,[1,2,3])
  - equivalent to: [str(i) for i in [1,2,3]]

More on functions

Functions are objects: they can be passed as arguments to other functions or can be assigned to other variables

```
>>> def my_function(value1,value2=0):
...     """
...     Function documentation
...     """
...     print "value1=%d"%value1
...     print "value2=%d"%value2
...     return value1+value2
...
>>>
>>> my_function(12)
value1=12
value2=0
12
>>> my_function(12,13)
value1=12
value2=13
25
>>> my_function(value1=1,value2=2)
value1=1
value2=2
3
```

value1 is a mandatory argument

value2 is an optional argument

All arguments are always passed by value

Non-keyword arguments must always precede keyword arguments. For instance, my_function(value2=2,12) would raise an exception

Catching exceptions

- Exceptions are objects in python
- Can be manually raised
  - raise ExceptionClass [,ExceptionParameter]
  - raise ExceptionInstance
- Should be always catched and handled!

```
try:
    #some code

except (RuntimeError, TypeError):
    print 'Runtinme or Type Error'
except IOError, instance:
    print 'IOerror',instance
    raise instance
except:
    print "Any other exception"
else:
    print "Everything was fine"
finally:
    print "In any case..."
```

This code may fail and may raise Exceptions. We need to make sure we handle these Exceptions correctly!

Catch multiple Exceptions types in the same line

Retrieve the Exception instance

Catch all the other Exceptions

Classes

```
class MyList:
    """
    Class-level documentation
    """
    def __init__(self,maxlen):
        """
        Constructor takes as input a value
        """
        self.__maxlen=maxlen
        self.__mylist=[]

    def append(self,value):
        """
        Append an element to the list if there
        is space.
        """
        if len(self.__mylist)<self.__maxlen:
            self.__mylist.append(value)

    def next(self):
        """
        This renders the object a valid iterator
        """
        if len(self.__mylist)==0:
            raise StopIteration
        else:
            return self.__mylist.pop(0)

    def __iter__(self): return self
```

A class can inherit from one or more base classes. Syntax: class Myclass(Baseclass1,Baseclass2). If a method is not found in MyClass, it will be searched for in any of the base classes.

The constructor is a special method invoked upon object instantiation. It can take as input multiple initialization arguments.

Any class method always receives as first argument the object instance itself.

x=MyList(10)

x.append(1)

is equivalent to:

MyList.append(x,1)

Classes

```
class MyList:
    """
    Class-level documentation
    """
    def __init__(self,maxlen):
        """
        Constructor takes as input a value
        """
        self.__maxlen=maxlen
        self.__mylist=[]

    def append(self,value):
        """
        Append an element to the list if there
        is space.
        """
        if len(self.__mylist)<self.__maxlen:
            self.__mylist.append(value)

    def next(self):
        """
        This renders the object a valid iterator
        """
        if len(self.__mylist)==0:
            raise StopIteration
        else:
            return self.__mylist.pop(0)

    def __iter__(self): return self
```

In python classes everything is public. A method or object is rendered "private" by prepending its name with __ . The interpreter will convert __name into __Classname_name . This name mangling technique is solely to avoid "accidents"!

These are instance objects: they are assigned to self and are specific to the instance. Setting MyList.attr=<val> would instead define a class object.

Mylist.attr=2 ➔ class-specific

self.attr=2 ➔ instance-specific

An iterable object is an object that implements a next() method. The __iter__() method must return an iterator for the class, or self is a next() method is defined (as in this case)

Try:

x=MyList(2)

x.append(1)

x.append(2)

for i in x: print i

Code readability

```
c=None
a=2
b=3

def compute():
    global c
    c=a*b
```

BAD!!!

```
class Multiply:

    def compute(self,a,b):
        return a*b

c=Multiply().compute(2,3)
```

SAFE

```
class Multiply:

    def __init__(self,a,b):
        self.__a=a
        self.__b=b

    def compute(self):
        return self.__a*self.__b

c=Multiply(2,3).compute()
```

SAFE

Dangers

```
i=2
class Myclass:
    j=i

i=3

m=Myclass()
n=Myclass()

print m.j
m.j=4
print n.j
Myclass.j=5
print n.j
```

- Q: What are the three values being printed here?

Dangers

```
i=2
class Myclass:
    j=i

i=3

m=Myclass()
n=Myclass()

print m.j
m.j=4
print n.j
Myclass.j=5
print n.j
```

- Q: What are the three values being printed here?
- A: The two values are:
  - 2 (the assignment is done at the execution of the class definition)
  - 2 (assigning a value to m.j makes it become an instance object)
  - 5 (in the last step I have modified the class attribute)

1. Introduction to python
2. Sockets in UNIX
3. Concurrent servers
4. Threads

SOCKETS IN UNIX

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

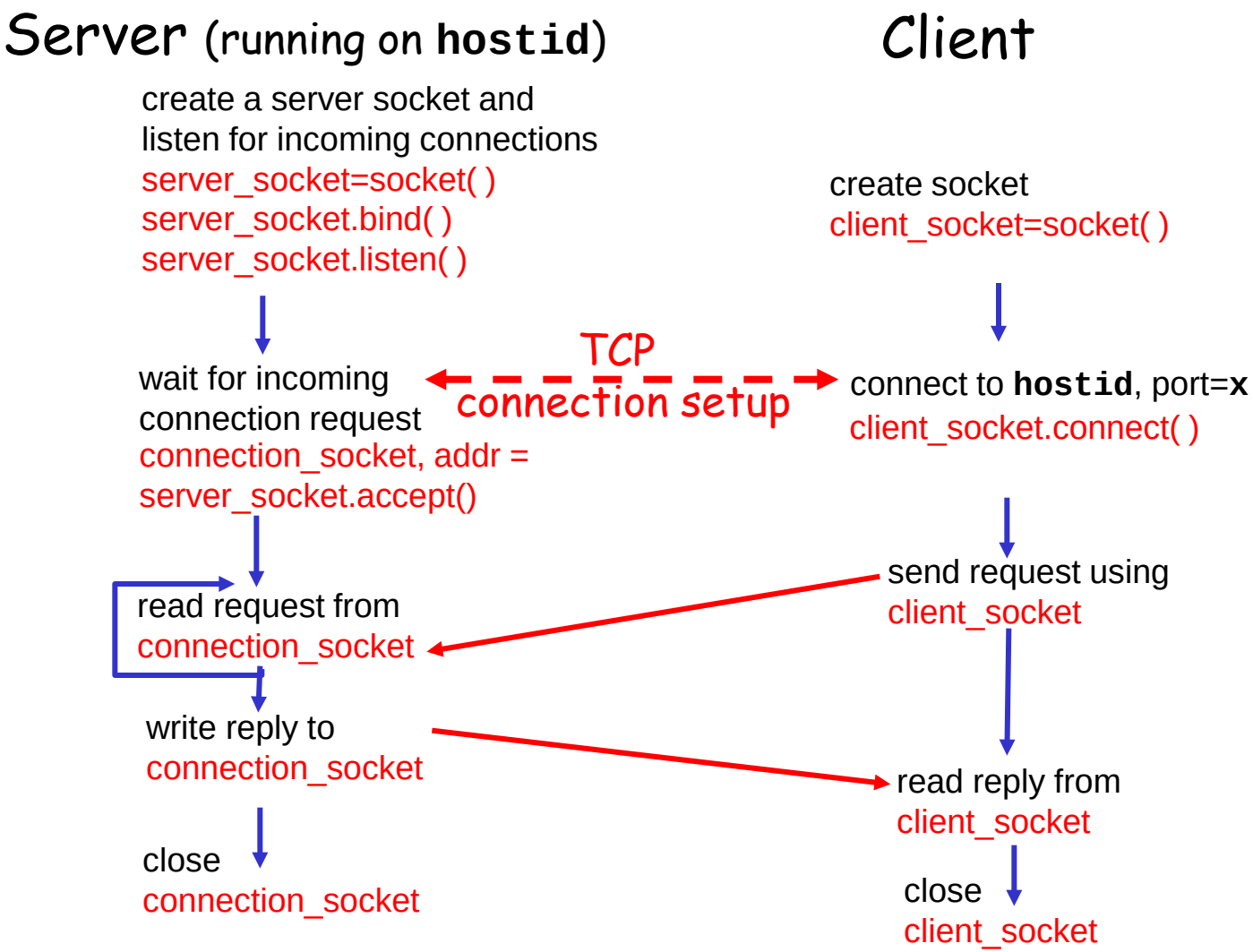
- creating client-local TCP socket
- specifying IP address, port number of server process
- when client creates socket: client TCP establishes connection to server TCP

- when contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client/server socket interaction: TCP



Socket programming with TCP

Example client-server app:

1) client reads line from standard input (using python **`raw_input()`**) , sends to server via socket
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket

➔ CODE

client1.py

server1.py

Client code

```
import socket

PORT=4444
HOST="127.0.0.1"

cli=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
cli.connect((HOST,PORT))

while True:
    cmd=raw_input("Input:  ")

    if len(cmd)==0: break
    cli.send(cmd)
    ans=cli.recv(1024)
    print "Output: %s"%ans

cli.close()
```

Creation of a socket of family AF_INET and type SOCK_STREAM : a TCP socket. The socket is connecting to 127.0.0.1 (your local machine) on port 4444.

The string cmd is sent over the socket to the server

In receiving data from a socket, you always need to specify the buffer size. The buffer size should always be a small power of 2 (1024 is OK). Recv will return a string of maximum length 1024. If the connection gets dropped, recv will return an empty string.

Server code

```
import socket

PORT=4444
HOST="127.0.0.1"

srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.bind((HOST,PORT))
srv.listen(1)

conn,addr=srv.accept()
print "Connected by %s:%d"%addr
while True:
    data=conn.recv(1024)
    if not data: break
    conn.send(data.upper())

conn.close()
```

Binding a socket to a specific host and port makes it become a server socket. HOST could be an empty string or "0.0.0.0" to listen on any interface

Accept() returns only once a client connected (end of 3 Way Handshake). It returns the connection socket and information on the client address/port.

The interaction with the client is performed over the connection socket.

If data is an empty string, the client has disconnected.

Socket programming *with UDP*

UDP: no "connection" between client and server

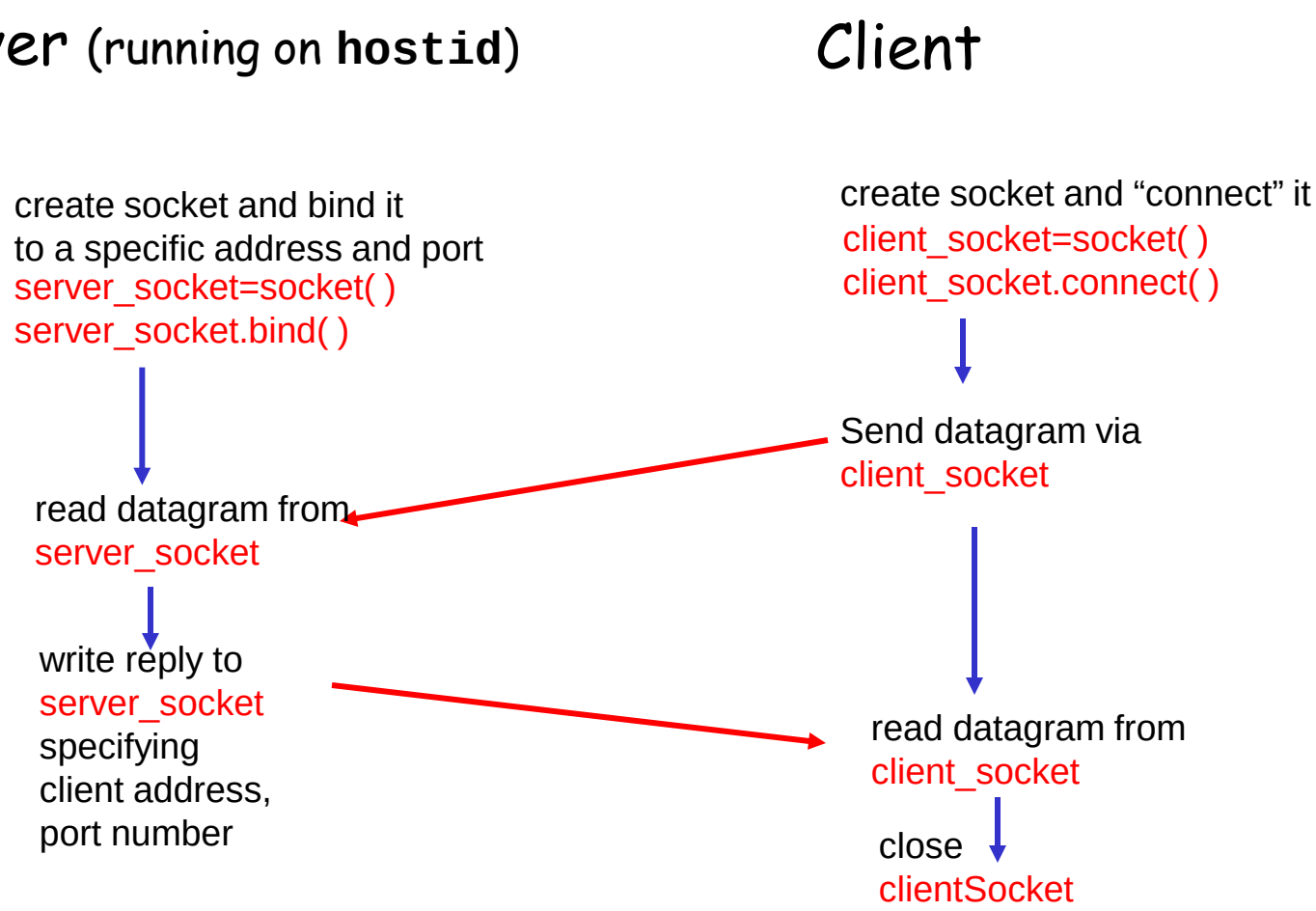
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint:

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Client code

```
#!/usr/bin/env python
import socket

PORT=4444
HOST="127.0.0.1"

cli=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
cli.connect((HOST,PORT))

while True:
    cmd=raw_input("Input:  ")

    if len(cmd)==0: break
    cli.send(cmd)
    ans=cli.recv(1024)
    print "Output: %s"%ans

#let the server know we are gone
cli.send("")
cli.close()
```

The connect() call has a very different behavior than in the TCP case. No packet is sent upon invocation of this function. To avoid specifying the server endpoint at every sendto() call, the API allows you to specify the server endpoint once with connect and use standard send() and recv() calls in the rest of the code.

Thanks to the use of the connect() call, this block is identical to the TCP case.

We send an empty datagram to let the server know the client is gone and replicate the behavior of the TCP case. Otherwise, the server would never know that the client closed the socket and terminated.

➔ CODE

client2.py

Server code

```
#!/usr/bin/env python
import socket

PORT=4444
HOST="127.0.0.1"

srv=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
srv.bind((HOST,PORT))

while True:
    data,addr=srv.recvfrom(1024)
    if not data: break
    srv.sendto(data.upper(),addr)

srv.close()
```

Creation of a socket of family AF_INET and type SOCK_DGRAM: a UDP socket. The socket is bound to 127.0.0.1 (your local machine) on port 4444.

No notion of stream exists for UDP sockets. The server socket provides endpoint information for any received datagram. Addr contains IP address of the client as well as its source port.

Similarly, whenever a datagram is sent the endpoint of the receiver must be provided.

➔ CODE

server2.py

socket.socket(family[,type[,protocol]])

- Opens a socket of a given type and returns a socket object
- Supported protocol families
  - AF_INET: IPv4 protocols
  - AF_INET6: IPv6 protocols
  - AF_LOCAL (historically AF_UNIX): Unix domain protocols
  - AF_ROUTE: Routing sockets
  - AF_KEY: Key sockets
- Supported protocol types
  - SOCK_STREAM: stream socket
  - SOCK_DGRAM: datagram socket
  - SOCK_RAW: raw socket
  - SOCK_PACKET: datalink socket (Linux only)

socket.socket(family[,type[,protocol]])

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

- Not all combinations of socket family and type are valid
- Protocol argument refers to the IP protocol field, used only for raw sockets

socket.socket.connect(address)

- In the case of a TCP socket, initiates the TCP three-way handshake. It returns only when the connection is established. In case of error (timeout, connection refused, …) an exception is raised.
- Arguments
  - address: a tuple (<IP>,<port>)

socket.socket.bind(address)

- Assigns a local protocol address to a socket.
- With TCP, calling bind lets us specify a port number, an IP address or both.
- Important: binding to privileged port numbers (0-1024) requires superuser privileges
- Arguments
  - address: a tuple (<IP>,<port>)
    - when <port> specified, the kernel replaces the ephemeral port otherwise chosen with the application-specified one.
    - if <IP> empty string or "0.0.0.0" the socket will be bound to all the currently active interfaces

socket.socket.bind(address)

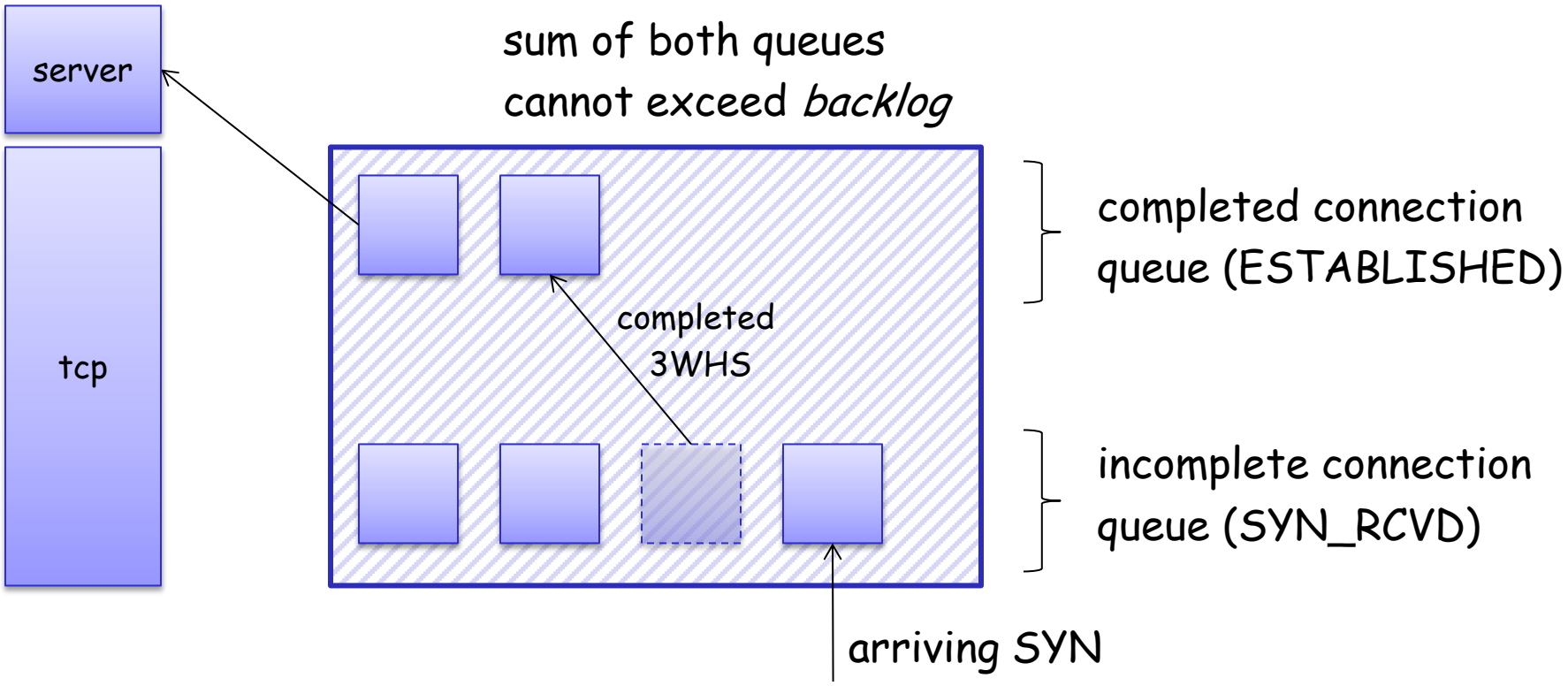
```
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.bind(("127.0.0.1",0))
sock.listen(15)
#discover the address/port we have just bound to
bind_address,bind_port=self.__sock.getsockname()
```

- Sample code that binds the socket on an IP address but without specifying a port. This will use as listening port the ephemeral port normally used for a client socket.
- Why?
  - Listening on a well defined port is not always required (P2P applications, …)

socket.socket.listen(backlog)

- Enables a server to accept connections
  - Converts an unconnected socket into a passive socket, that will be used only for accepting connections
- Arguments
  - backlog: an integer >= 1
    - Specifies the maximum number of connections that the kernel should queue for this socket

socket.socket.listen(backlog)



- The two queues maintained by TCP for a listening socket

Example: socket.socket.listen(3)

Event	SYN_RCVD	ESTABLISHED
cli1 sends SYN	1	0
cli2 sends SYN	2	0
cli1 completes 3WH	1	1
cli3 sends SYN	2	1
cli2 completes 3WH	1	2
cli4 sends SYN	1	2
application calls accept()	1	1

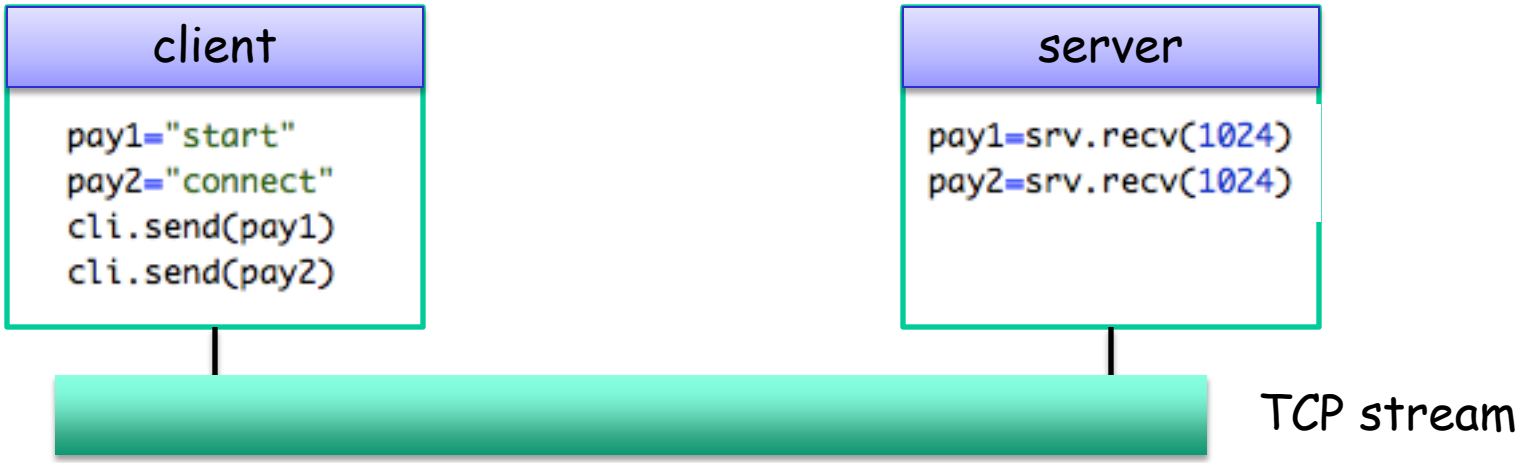
socket.socket.accept()

- Called by a TCP server to return the next completed connection from the front of the completed connection queue (see before)
- If the completed connection queue is empty, the process is put to sleep
- Returns:
  - (Socket object, address info)
    - The socket object is a connected socket associated to the specific client connection and will be used to transfer data
    - The address info is a tuple (client address, client port)

Sending and receiving data

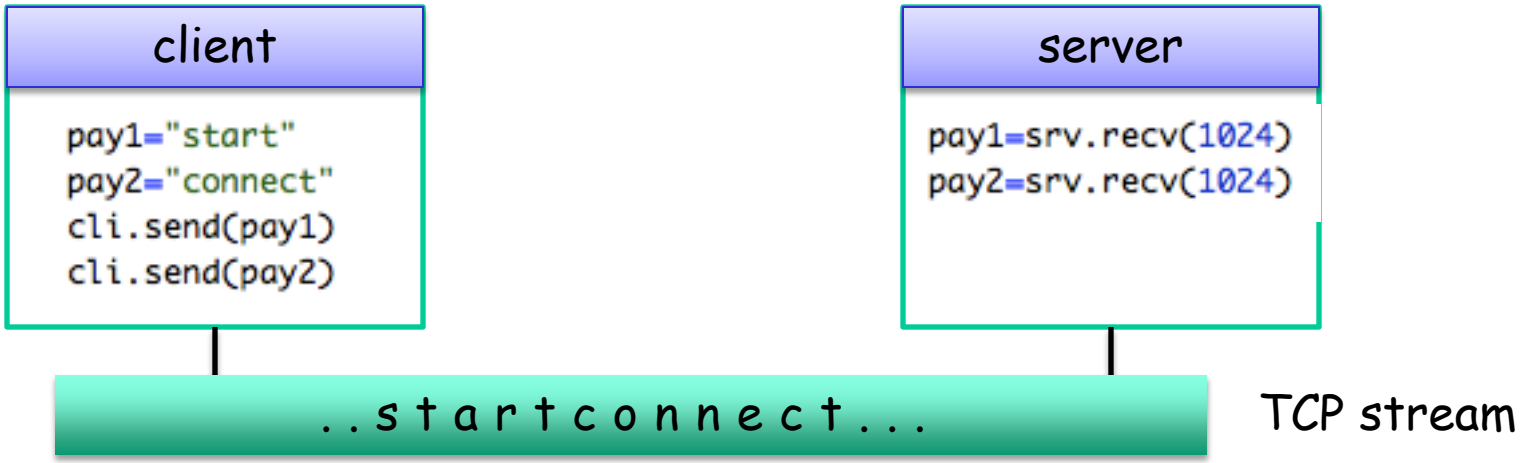
- send/recv functions operate on network buffers.
- Default behavior:
  - If the receive buffer is empty, recv() blocks until more data is received or until the connection is dropped
  - If the send buffer is full, send() blocks until the buffer is freed (TCP flow control)
  - send() always returns the number of bytes effectively pushed to the transmit buffer
- Setting the socket to non-blocking or setting a timeout changes this behavior

Sending and receiving data in TCP



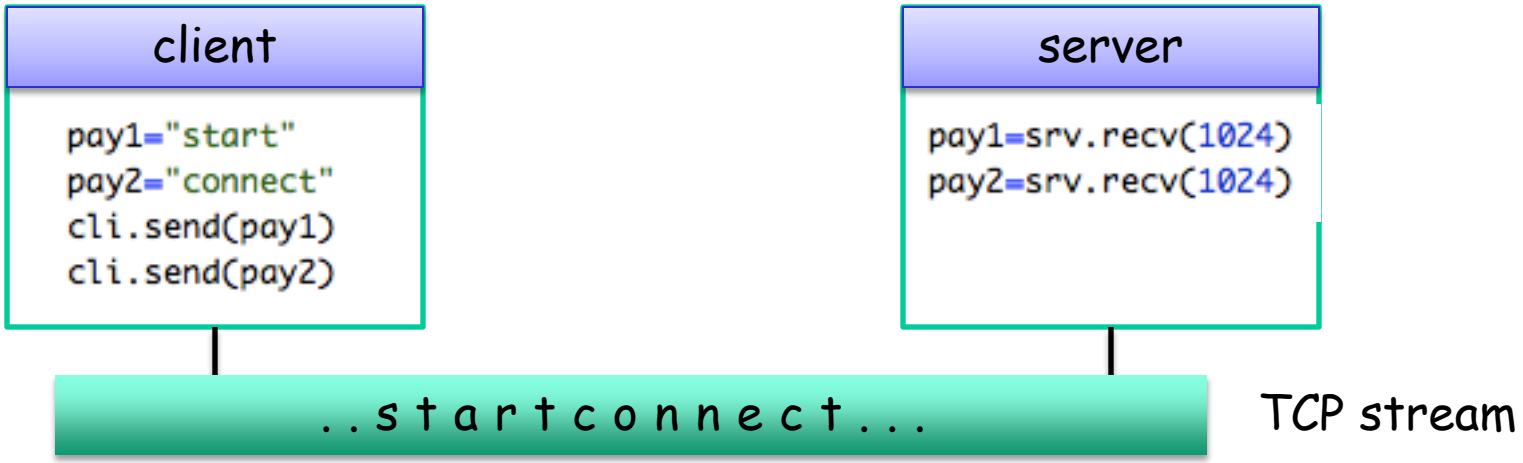
- Q: What will be the content of pay1 and pay2 on the server?

Sending and receiving data in TCP



- Q: What will be the content of pay1 and pay2 on the server?
- A: It depends!!!!
  - We may have pay1="start" and pay2="connect"
  - We may have pay1="startconnect" and the second recv may block
  - We may have pay1="startco" and pay2="nnect"

Sending and receiving data in TCP



- In designing your own protocol, you must design a way to unambiguously separate application-level messages. Options:
  - Use fixed length messages (generally inefficient)
  - Use separators (SMTP)
  - Explicitly advertise the length of each message (HTTP, DNS, …)
  - Send only one message per connection (time, whois)

socket.socket.close()

- Closes the socket. If the socket is a TCP one, calling this function will terminate the TCP connection.
- After close() has been called on a socket object, the object cannot be used any longer.

socket.socket.setsockopt(level,option,value)
socket.socket.getsockopt(level,option)

- Level SOL_SOCKET
  - SO_BROADCAST (UDP only): enables the process to send broadcast messages
  - SO_KEEPALIVE (TCP only): send a keepalive probe to the peer every 2 hours of inactivity
  - SO_RECVBUF/SO_SNDBUF: Control the size of the kernel send and receive buffers.
  - SO_REUSEADDR: allows a listening server to start and bind its well-known port even if previously established connections exist that use this port. Especially useful on server crashes while developing ☺ .Also allows multiple binds on same port but different IP addresses.
- Level IPPROTO_TCP
  - TCP_NODELAY: enables/disables Nagle algorithm

A few words on RAW sockets

- Three main features of interests
  - Ability to read and write ICMP and IGMP packets
  - Ability to read and write IPv4 datagrams with custom IP protocol field (different from 1:ICMP,2:IGMP,6:TCP or 17:UDP)
  - Ability to write your own IPv4 header (through socket option IP_HDRINCL at level IPPROTO_IP)
- Special behaviors
  - Bind and connect can be called for setting local/foreign addresses. No concept of port.
  - Protocol must be specified (IPPROTO_ICMP,IPPROTO_TCP,IPPROTO_UDP,…) and is used to set IP header "protocol" field
  - Superuser privileges are required to use them

Socket types and the TCP/IP stack

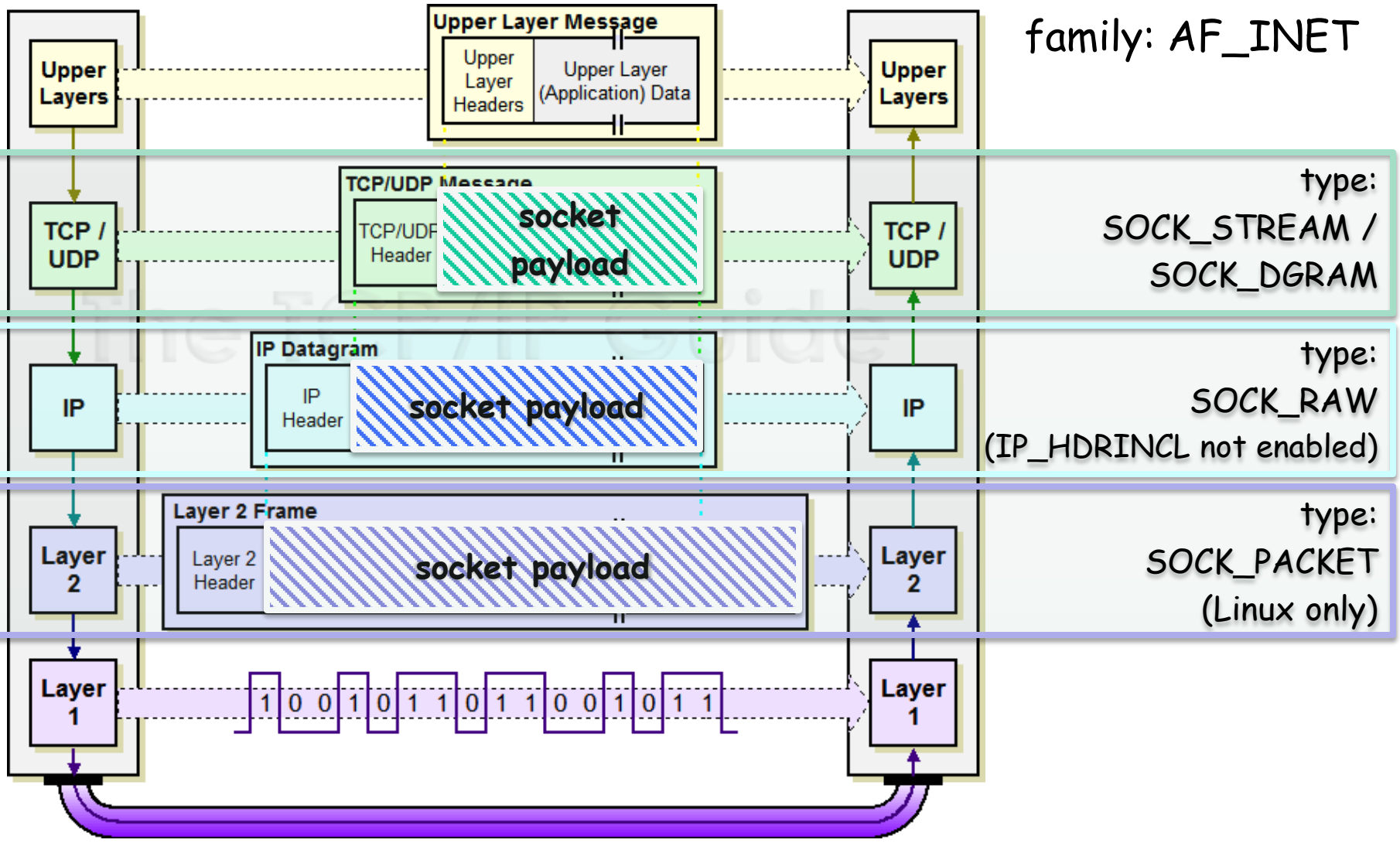
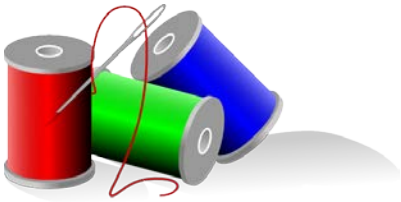


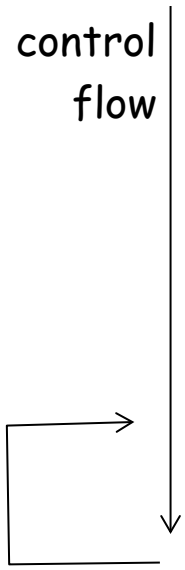
image source: tcpipguide.com

1. Introduction to python
2. Sockets in UNIX
3. Concurrent servers
4. Threads

CONCURRENT SERVERS



The problem




```
import socket

PORT=4444
HOST="127.0.0.1"

srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.bind((HOST,PORT))
srv.listen(1)

conn,addr=srv.accept()
print "Connected by %s:%d"%addr
while True:
    data=conn.recv(1024)
    if not data: break
    conn.send(data.upper())

conn.close()
```

Upon accept() invocation, the process sleeps until a client successfully completes the 3-way handshake

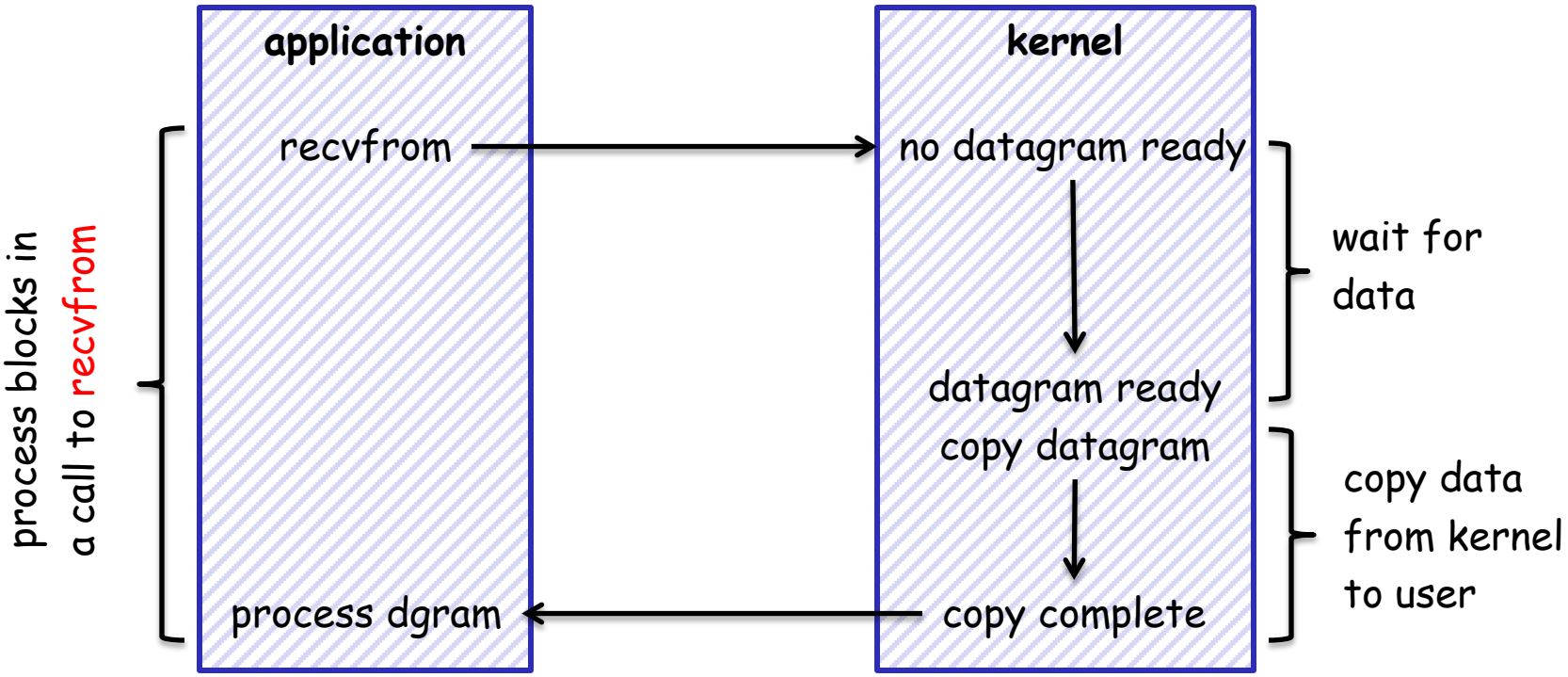
Upon recv() invocation, the process sleeps until a new payload is received from the client

- The server code written so far cannot support concurrent access of multiple clients

I/O models

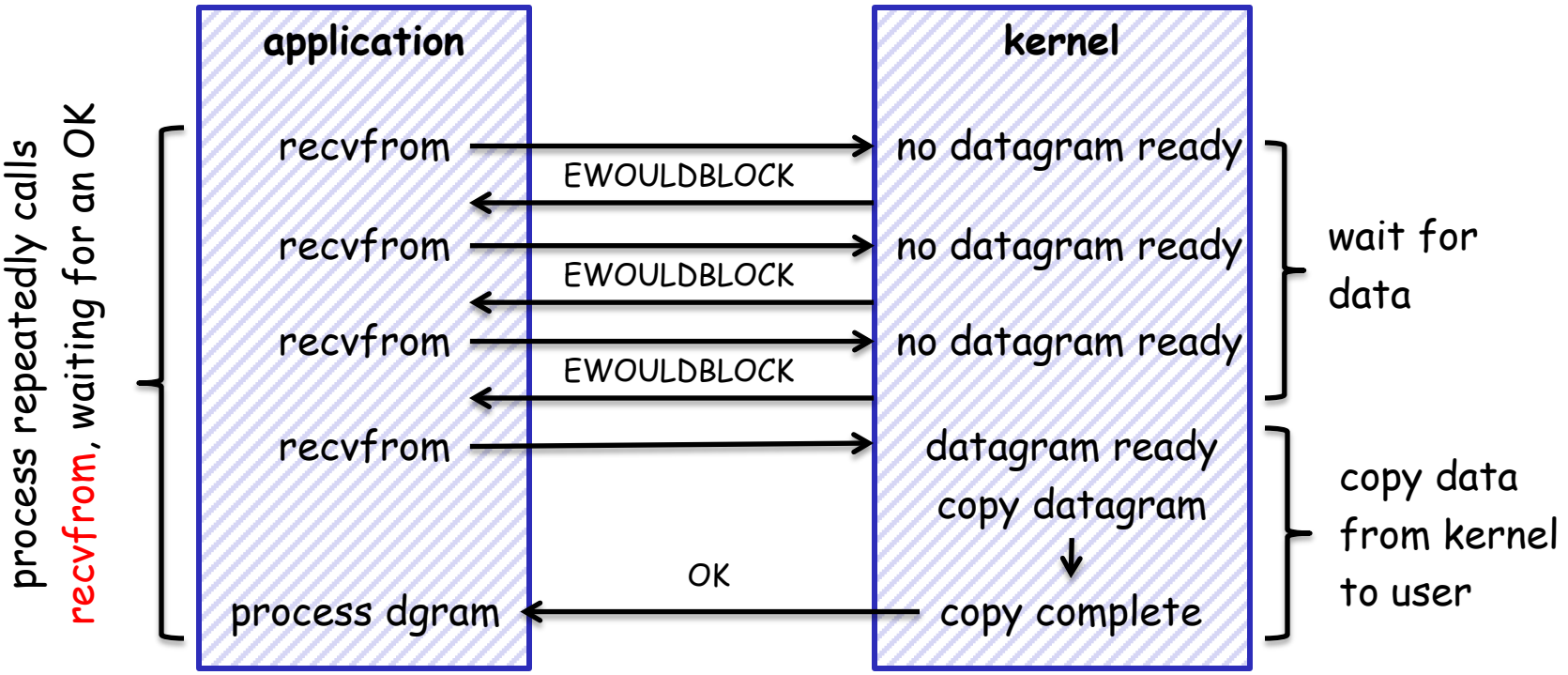
- 5 different I/O models can be identified
  - blocking I/O
  - nonblocking I/O
  - I/O multiplexing
  - asynchronous I/O
  - (signal-driven I/O)

Blocking I/O model



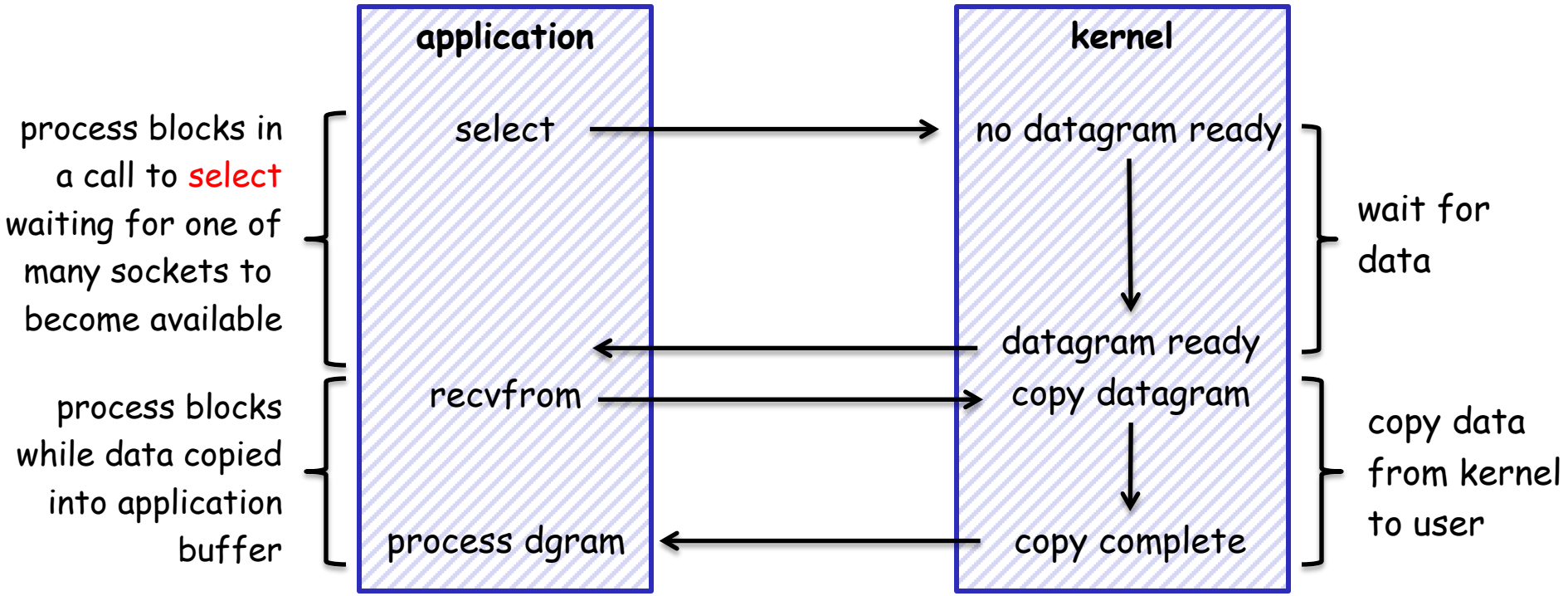
- The model we have used so far in our examples

Nonblocking I/O model



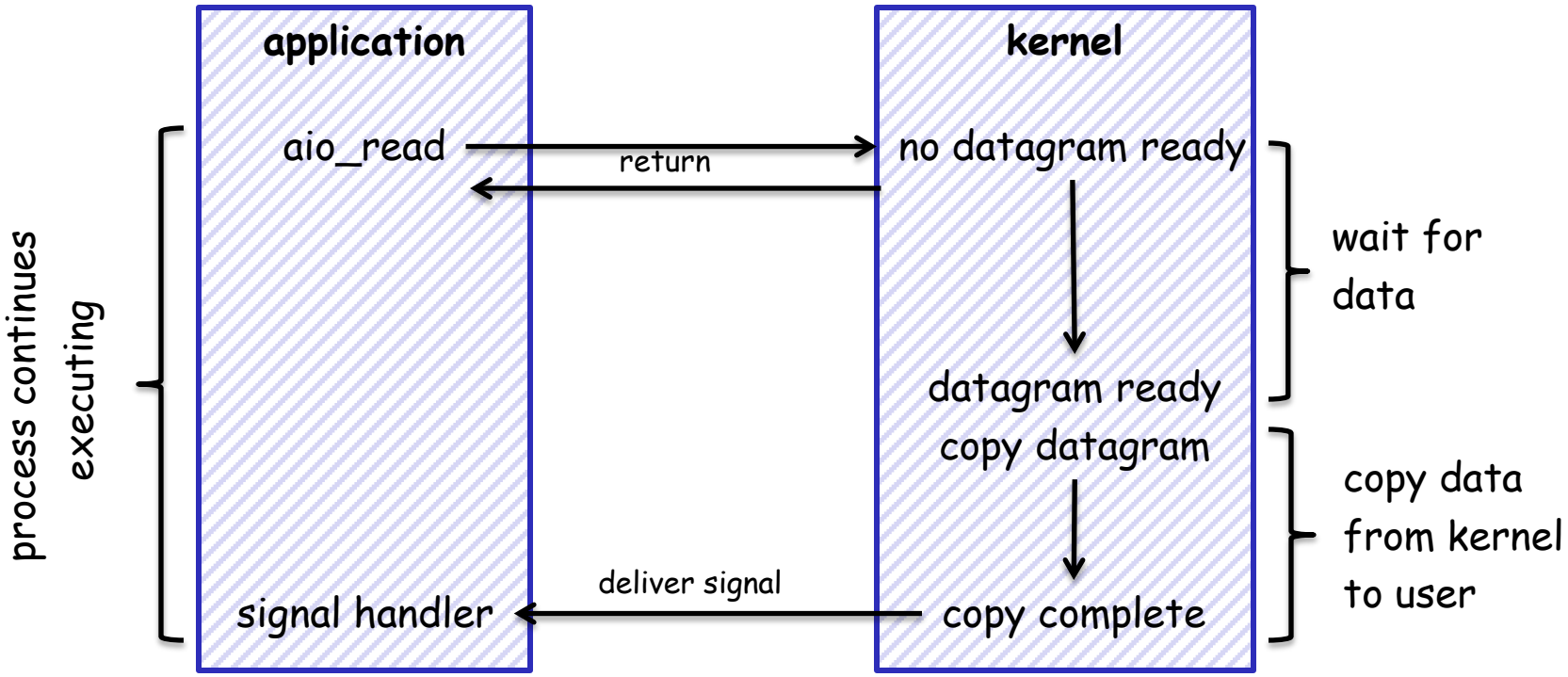
- A socket can be set as nonblocking: when an I/O operation cannot be completed without putting the process to sleep, raise an exception
- Polling is often a waste of CPU time

I/O multiplexing model



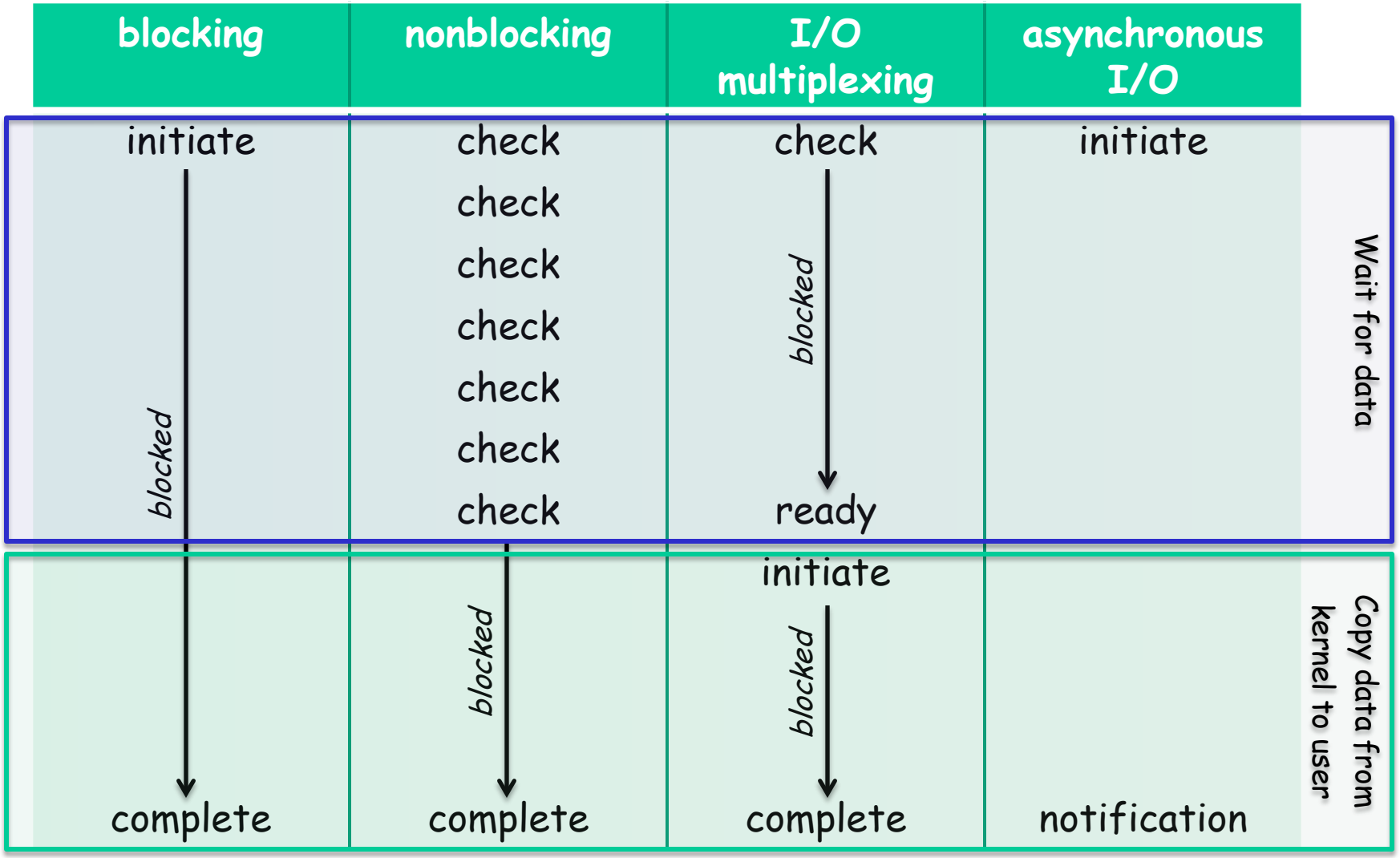
- Advantage: we can wait for multiple sockets or file descriptors to be ready

Asynchronous I/O model



- Programming pattern used by Twisted
- Full kernel support still work in progress

Summary of the I/O models



Blocking I/O

- Handling multiple clients in a server requires us to multiplex multiple sockets
  - server socket: to accept new connections
  - connection sockets: one socket per connected client
- How to achieve multiplexing with a blocking I/O model?
  - Any call to accept() or recv() may set the process to sleep
  - Solution: multiple processes! ☺

os.fork()

- The only function that is called once and returns twice
  - Once in the calling process (parent) with a return value that is the process ID of the newly created process (child)
  - Once in the child with a return value of 0 (parent process ID can be obtained using os.getppid())
- All descriptors (e.g. sockets) open in the parent are shared with the child
- In UNIX environments, fork() is the only way to create a new process.

os.fork(): an example

```
import os

if os.fork()==0:
    print "fork() exited in the child process"
    print "child: pid %d parent %d"%(os.getpid(),os.getppid())
else:
    print "fork() exited in the parent process"
    print "parent: pid %d parent %d"%(os.getpid(),os.getppid())
```

- When executing the script, both lines are printed.

```
corrado@daphne code $ python fork.py
fork() exited in the parent process
parent: pid 40718 parent 38712
fork() exited in the child process
child: pid 40719 parent 40718
```

➔ CODE

fork.py

os.fork(): an example

```
corrado@daphne code $ python fork.py
fork() exited in the parent process
parent: pid 40730 parent 38712
fork() exited in the child process
child: pid 40731 parent 1
```

- Q: How do you explain this output?

os.fork(): an example

```
corrado@daphne code $ python fork.py
fork() exited in the parent process
parent: pid 40730 parent 38712
fork() exited in the child process
child: pid 40731 parent 1
```

- Q: How do you explain this output?
- A: When the scheduler switched to the child process, the father had already died. In UNIX orphan processes get automatically adopted by the init system process (PID=1) (*automatic re-parenting*).

Fork-based server

```
#!/usr/bin/env python
import socket
import os

PORT=4444
HOST="127.0.0.1"

srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
srv.bind((HOST,PORT))
srv.listen(1)

while True:
    conn,addr=srv.accept()

    if os.fork()==0:     #child process
        print "connected by %s:%d"%addr
        while True:
            data=conn.recv(1024)
            if not data: break
            conn.send(data.upper())
        conn.close()
        print "child process exiting"
        break
```

Notice the SO_REUSEADDR option

Parent and child process share file descriptors. The conn socket is created by the parent, but used and closed by the child.

Code executed by the child process. The parent process will continue to loop over the accept() call and the fork().

→ CODE

server-fork.py

I/O multiplexing

- Alternative to the blocking I/O method
- Pros:
  - Does not require spawning multiple processes
- Cons:
  - Complex to use in complex servers without compromising server responsiveness: one single control flow in charge of data processing and network interaction

select.select(rlist,wlist,xlist[,timeout])

- Allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. More specifically, it waits for:
  - Any of the descriptors in *rlist* to be ready for reading
  - Any of the descriptors in *wlist* to be ready for writing
  - Any of the descriptors in *xlist* to have an exception condition pending
  - *Timeout* seconds to be elapsed

select.select(rlist,wlist,xlist[,timeout])

- Timeout values:
  - timeout==None: wait forever. Return only when one of the specified descriptors is ready for I/O.
  - timeout>0: wait up to a fixed amount of seconds. Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds specified by timeout.
  - timeout==0: do not wait at all. Return immediately after checking the descriptors. Used to implement polling (nonblocking I/O).

select.select(rlist,wlist,xlist[,timeout])

- A socket is ready for reading if any of the conditions is True:
  - Data is available in the receive buffer, and a read operation will not block and will return a non-empty string.
  - The read-half of the connection is closed (i.e. a FIN was received). A read operation on the socket will not block and will return an empty string.
  - The socket is a listening socket and the number of completed connections is nonzero. An accept on the listening socket will normally not block.
  - A socket error is pending. A read operation on the socket will raise an Exception.

select.select(rlist,wlist,xlist[,timeout])

- A socket is ready for writing if any of the conditions is True:
  - The send buffer is not full, and the socket is either connected (TCP) or not requiring a connection (UDP). A write operation will send a non-0 amount of bytes.
  - The write-half of the connection is closed. A write operation on the socket will generate an exception.
  - A socket error is pending. A write operation on the socket will raise an Exception.

- A socket has an exception condition pending if there exists out-of-band data for the socket or the socket is still at the out-of-band mark

Select-based server

➔ CODE

server-select.py

```
#!/usr/bin/env python
import socket,select

PORT=4444
HOST="127.0.0.1"

srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
srv.bind((HOST,PORT))
srv.listen(1)

#associate each socket to its address info
clisocks={}

while True:
    r,w,x=select.select([srv]+clisocks.keys(),[],[])

    if srv in r:
        #a new connection was completed
        conn,addr=srv.accept()
        print "new connection from %s:%d"%addr

        clisocks[conn]=addr

    for conn in [i for i in r if i!=srv]:
        data=conn.recv(1024)
        if data:
            conn.send(data.upper())
        else:
            print "dropped connection from %s:%d"%clisocks[conn]
            clisocks.pop(conn)

    print "%d clients connected"%len(clisocks)
```

Dictionary that we use to keep track of the currently open connected sockets, and associate them to client address information (for debugging).

Select is run over the currently open connected sockets and the server socket (to check whether new connections can be accepted)

If the listening socket is among the readable sockets, the queue of completed connections is not empty and accept will not block.

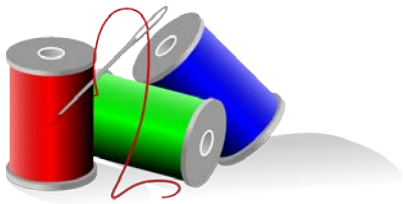
Read from any readable socket that is not the listening socket (list comprehension).

Multiplexed I/O and responsiveness

- In the previous server, one single control flow continuously handles in a "select loop" all the network interaction and processing
- All of the operations in the "select loop" must be fast
- What happens if they are not…?

1. Introduction to python
2. Sockets in UNIX
3. Concurrent servers
4. Threads

THREADS



Multithreaded programming

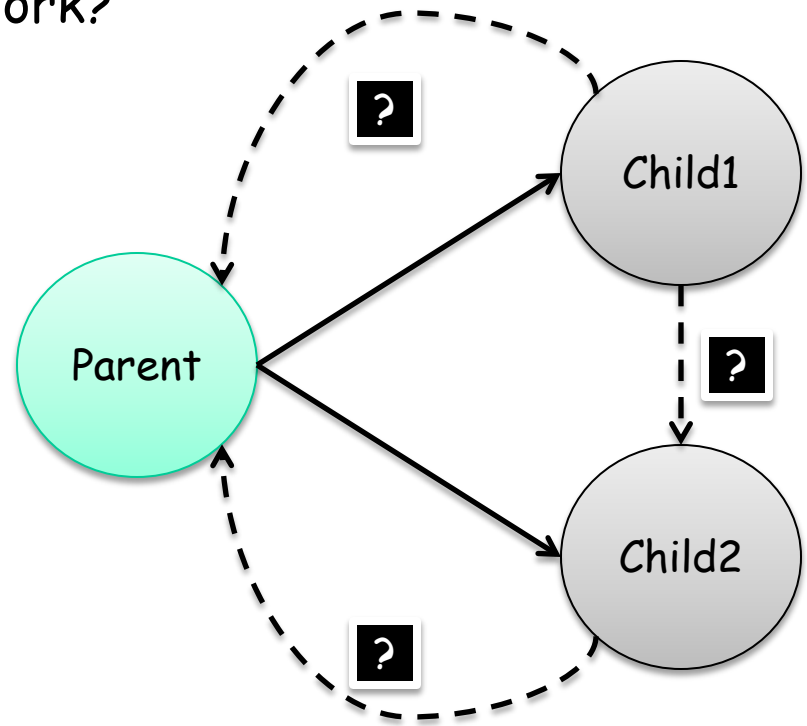
- Many UNIX servers are based on forks. The main process accepts the connection, and spawns a child in charge to serve it.
- Problems:
  - Cost
  - Inter-process communication

Fork: cost

- Whenever fork() is called, the entire process memory space needs to be duplicated
- Linux optimization: copy-on-write
  - Avoid copy of the parent's data space to the child until the child needs its own copy
- Every new child needs to be scheduled by the OS

Fork: IPC

- Sharing information between the parent and the child before the fork is easy: the child starts with a copy of the parent's data space.
- How to share information among the parent and its children after the fork?



Threads

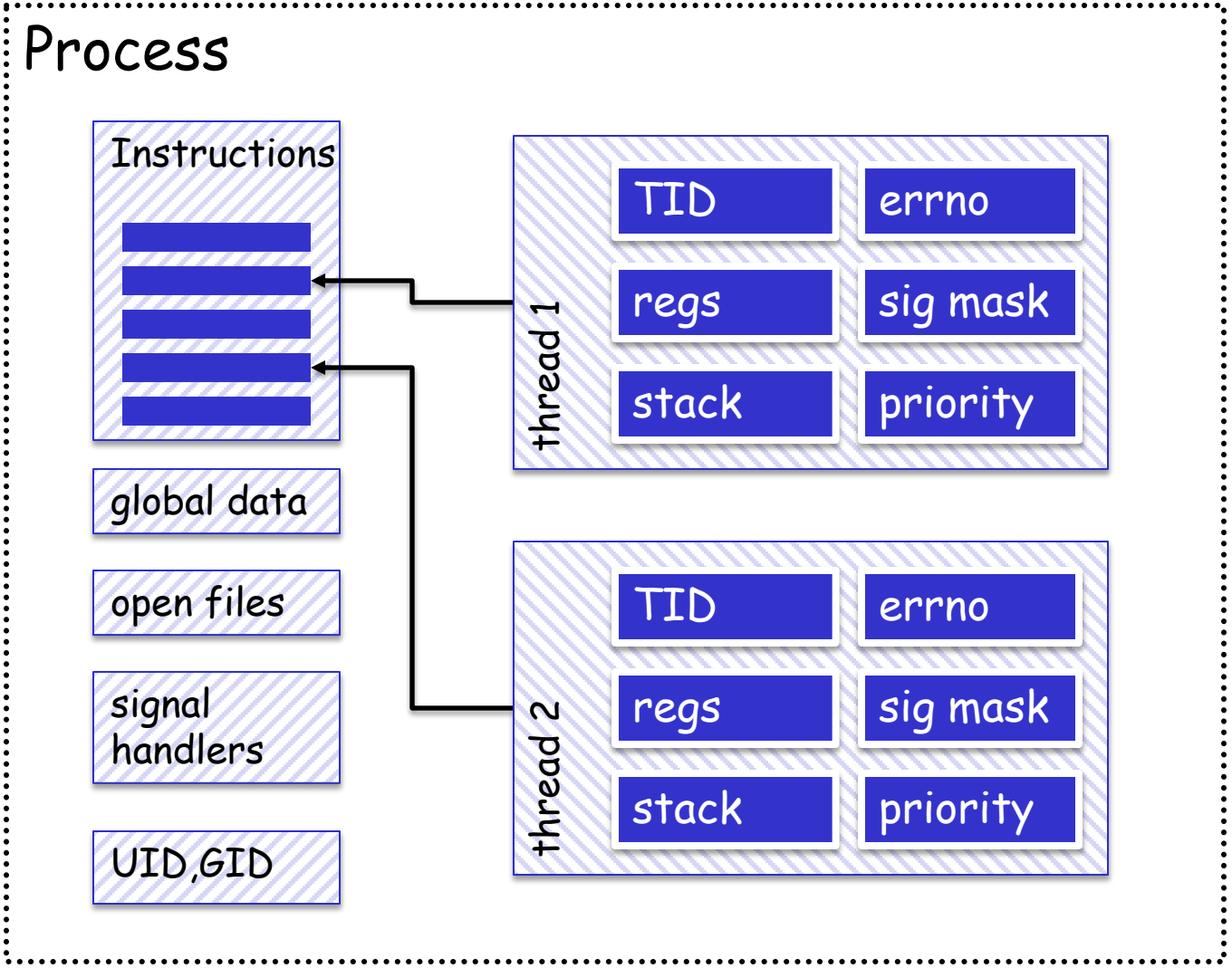
- Sometimes called *lightweight processes*.
  - Creation can be 10-100 times faster than process creation
- All threads share the same global memory
  - Sharing information is easy…
  - … but renders you vulnerable to programming errors and synchronization issues

"How to shoot yourself in both feet at once" (Alan Cox)

Shared information

- All threads share:
  - process instructions
  - most data
  - open files (e.g. descriptors)
  - signal handlers and signal dispositions
  - current working directory
  - user and group IDs
- But each thread has its own:
  - thread ID
  - set of registers, including program counter and stack pointer
  - stack (for local variables and return addresses)
  - errno
  - signal mask
  - priority

Shared information



Threads in Python

Thread executes the function code and then terminates.

```
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import thread
>>> def hello(arg):
...     print "Hello world! I'm a thread and arg is: %s"%arg
...
>>> thread.start_new(hello,("example",))
4488482816
>>> Hello world! I'm a thread and arg is: example

>>>
```

- Using threads in Python is straightforward.
- Two modules:
  - thread: low level API
  - threading: object-oriented interface similar to Java

Thread objects

➔ CODE

thread1.py

```
import threading,time

THREADS=10
INCREMENTS=50000

a=0

class IncrementingThread(threading.Thread):

    def __init__(self,increments):
        threading.Thread.__init__(self)
        self.increments=increments

    def run(self):
        global a
        print "%s starting processing"%self.getName()
        for i in range(0,self.increments):
            a=a+1
        print "%s completed processing"%self.getName()

running=[]
for i in range(0,THREADS):
    m=IncrementingThread(INCREMENTS)
    m.start()
    running.append(m)

[m.join() for m in running]
```

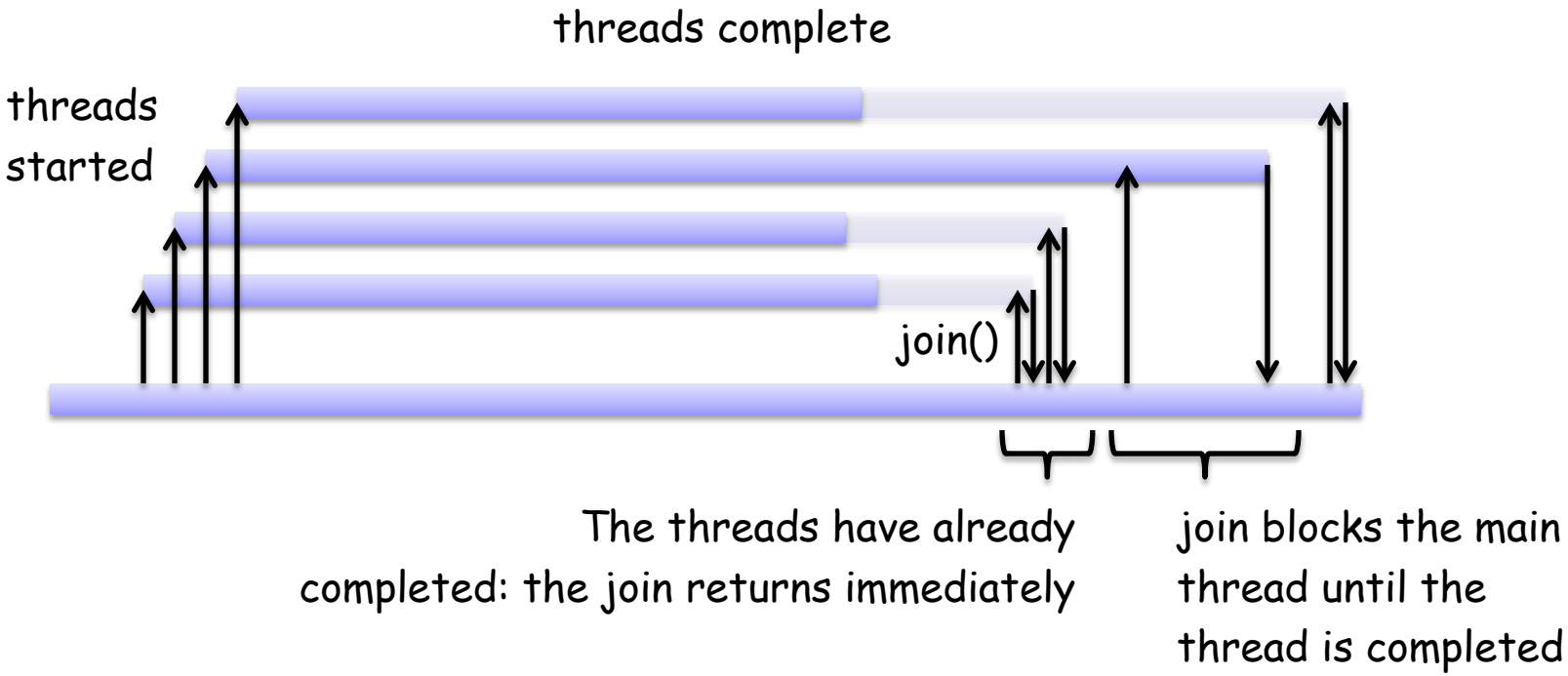
The thread operation and "local" data is wrapped within an object that subclasses threading.Thread.

Never forget to call the baseclass constructor

This method is executed in a new thread. Any thread can retrieve its identifier through threading.Thread.getName()

The main thread creates THREADS=10 threads and starts them immediately. The join call allows the main thread to wait for the completion of a subthread

threading.Thread.join(timeout)



- Wait for a thread to terminate. Timeout can be specified:
  - timeout==None: join waits indefinitely (default behavior)
  - timeout==0: join never blocks regardless of the thread status
  - timeout>0: join may block for at most timeout seconds
- threading.Thread.isAlive() should be called to understand if a timeout occurred

Daemon threads

- threading.Thread.daemon indicates whether a thread is a daemon thread or not. Can be set only before start() is called.
  - Joinable threads (daemon=False,default) can be joined. The thread retains ID information and exit status until join() is called by another thread.
  - Daemon/detached threads (daemon=True) release immediately all resources upon termination and cannot be joined.

Thread synchronization

➔ CODE

thread1.py

```
Thread-1 starting processingThread-2 starting processing

Thread-1 completed processing
Thread-2 completed processing
Thread-3 starting processing
Thread-4 starting processing
Thread-3 completed processing
 Thread-5 starting processing
Thread-5 completed processing
Thread-6 starting processing
Thread-4 completed processing
Thread-7 starting processing
Thread-7 completed processing
Thread-6 completed processing
Thread-8 starting processing
Thread-8 completed processingThread-9 starting process

Thread-9 completed processing
Thread-10 starting processing
Thread-10 completed processing
Excepted value: 500000
Obtained value: 381229
```

```
Thread-1 starting processingThread-2 starting processing

Thread-3 starting processing
Thread-4 starting processing
Thread-5 starting processing
Thread-1 completed processing
Thread-6 starting processing
 Thread-2 completed processing
Thread-7 starting processing
 Thread-4 completed processing
Thread-7 completed processing
Thread-8 starting processing
 Thread-9 starting processingThread-3 completed processing

 Thread-5 completed processing
Thread-10 starting processing
Thread-8 completed processing
Thread-6 completed processing
Thread-9 completed processing
 Thread-10 completed processing
Excepted value: 500000
Obtained value: 266816
```

- thread1.py contains a synchronization error
- Upon each execution the value of a is different!

Thread synchronization

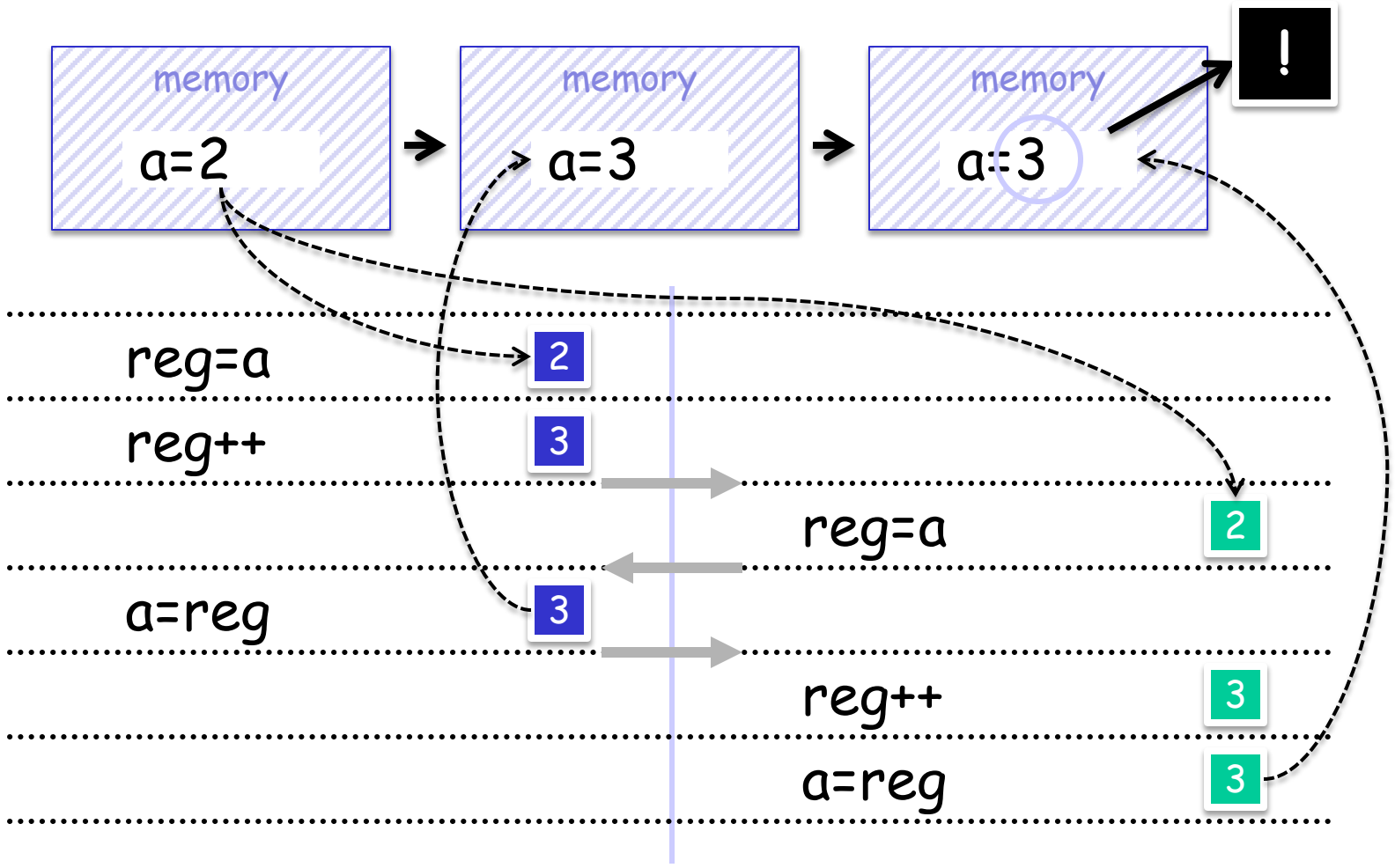
➔ CODE

thread1.py

```
import threading,time

THREADS=10
INCREMENTS=50000

a=0

class IncrementingThread(threading.Thread):

    def __init__(self,increments):
        threading.Thread.__init__(self)
        self.increments=increments

    def run(self):
        global a
        print "%s starting processing"%self.getName()
        for i in range(0,self.increments):
            a=a+1
        print "%s completed processing"%self.getName()

running=[]
for i in range(0,THREADS):
    m=IncrementingThread(INCREMENTS)
    m.start()
    running.append(m)

[m.join() for m in running]
```

This block is concurrently executed by multiple threads. The instruction "a=a+1" is split by the compiler in multiple bytecode instructions. A context switch may happen at the completion of any of these instructions!

Thread-unsafe functions



Thread safety in python

- The following operations are thread-safe
  - L.append(x)
  - L1.extend(L2)
  - x=L[i]
  - x.pop()
  - L1[i:j]=L2
  - L.sort()
  - x=y
  - x.field=y
  - D[x]=y
  - D1.update(D2)
  - D.keys()
- The following operations are not thread-safe
  - i=i+1
  - L.append(L[-1])
  - L[i]=L[j]
  - D[x]=D[x]+1

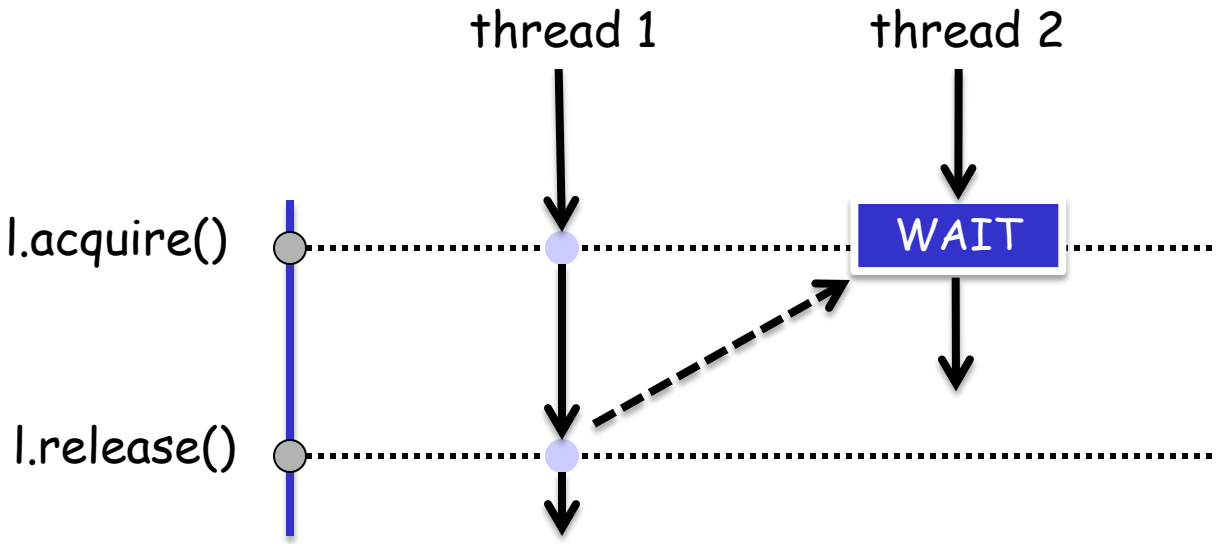
Concurrent programming errors

- Generally rare errors
  - The example code was specifically created to trigger the problem, but it hardly happens in practice
- Difficult to reproduce
  - They depend on non-deterministic timing of a large number of events
- Do not appear on all systems
  - The problem may appear only on certain architectures, with certain loads, with certain software configurations, …

Solution

- Protect thread-unsafe code with synchronization primitives
- Several synchronization primitives are implemented in the threading module:
  - threading.Lock: simple mutex objects
  - threading.RLock: reentrant locks
  - threading.Semaphore: semaphore objects
  - threading.Events: simple event notification
  - threading.Condition: condition variables

threading.Lock



- Two possible states:
  - locked: a call to acquire() will block until the lock is unlocked by calling release()
  - unlocked: default state, a call to acquire() will change the state to locked(). A call to release() will raise a RuntimeError

threading.Semaphore

```
poolsize=5
sem=threading.Semaphore(poolsize)

sem.acquire()
#use the resource
sem.release()
```

Every call to acquire() reduces the semaphore value of 1. Every call to release() increases the semaphore value. If the semaphore value becomes 0, acquire will block until a release() call increments the value again

- One of the oldest synchronization primitives in the history of computer science
- Used to guard resources with limited capacity
  - e.g. allow only n threads to use the same resource at a certain point in time

threading.Event

- Used to signal events among threads.
- Simplest mechanism for inter-thread communication:
  - Event.wait([timeout]): waits for the occurrence of the event. A timeout can be specified
  - Event.set(): sets the internal flag to true. All the threads waiting for the event will be awakened. Threads that will call wait() after the event has been set will not block at all.

threading.Condition

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

- Standard condition variables, useful to synchronize access to a shared state.
- Usually employed in typical producer-consumer problems

Synchronizing the code

➔ CODE

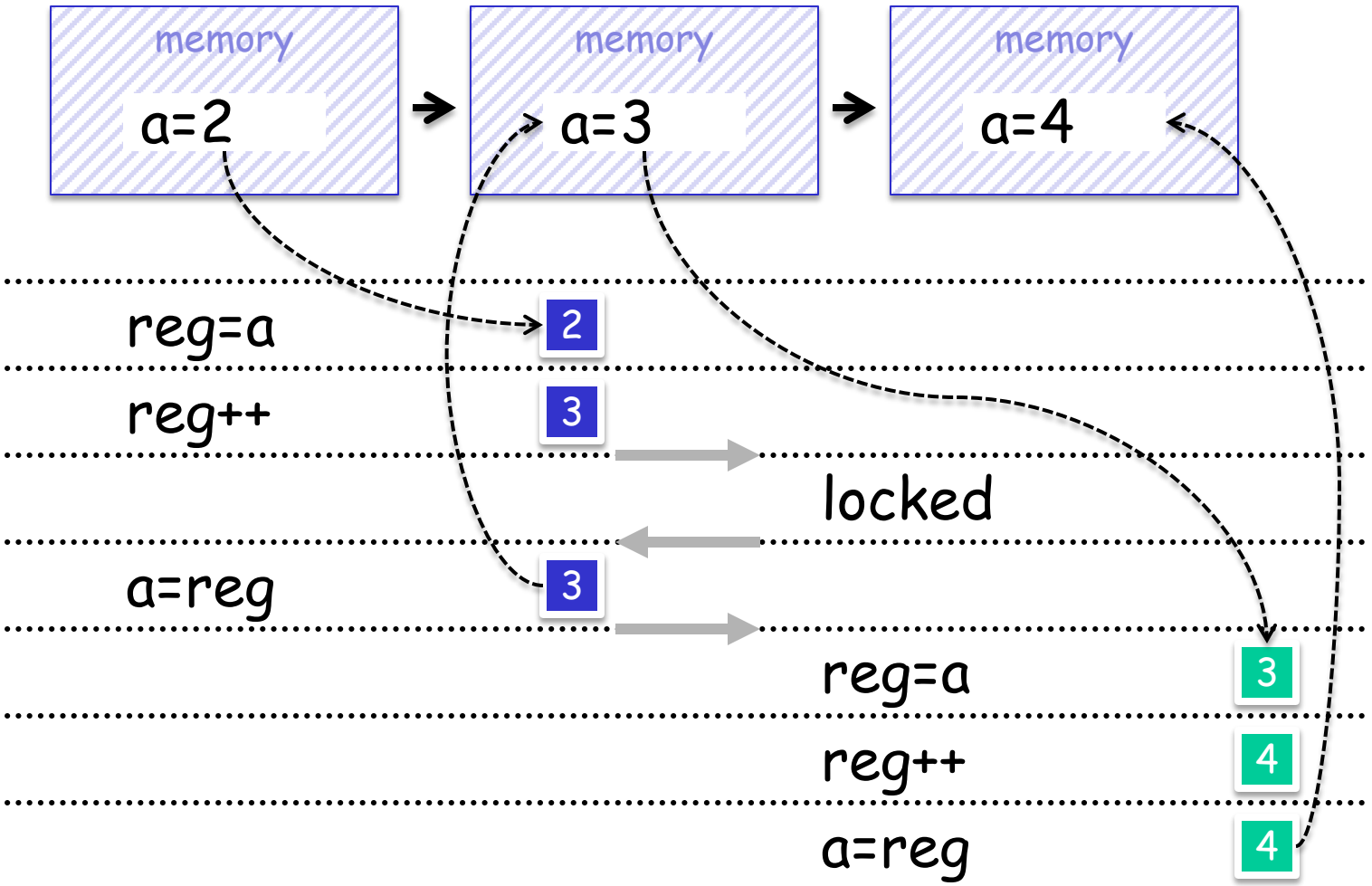
thread2.py

```
import threading,time

THREADS=10
INCREMENTS=50000

a_lock=threading.Lock()
a=0

class IncrementingThread(threading.Thread):

    def __init__(self,increments):
        threading.Thread.__init__(self)
        self.increments=increments

    def run(self):
        global a
        print "%s starting processing"%self.getName()
        for i in range(0,self.increments):
            a_lock.acquire()
            a=a+1
            a_lock.release()
        print "%s completed processing"%self.getName()

running=[]
for i in range(0,THREADS):
    m=IncrementingThread(INCREMENTS)
    m.start()
    running.append(m)

[m.join() for m in running]
```

A Lock object is created to protect the shared variable

The lock object is used to protect the critical region containing non-thread-safe code

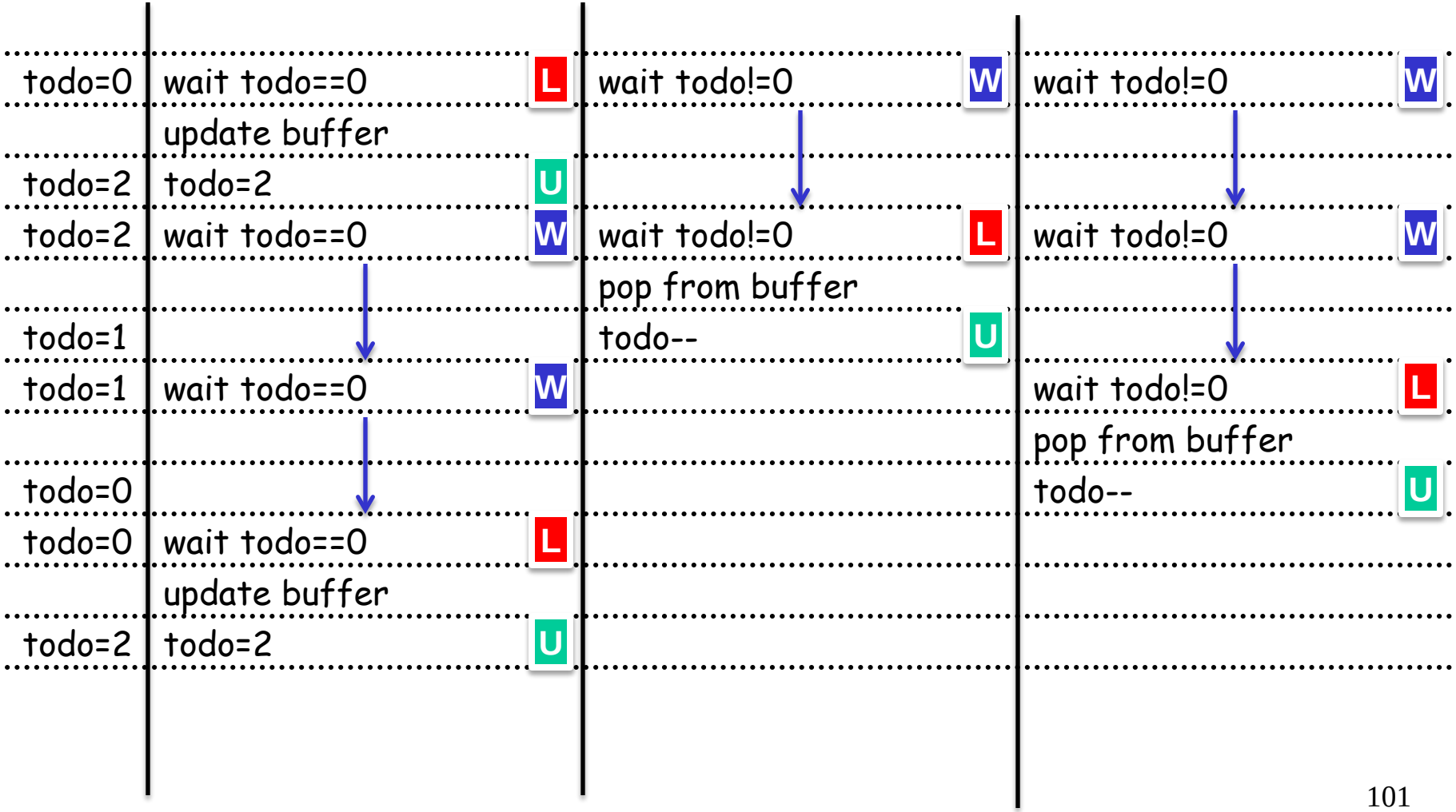
Synchronizing the code



Producer/Consumer problem

- Another classical "concurrency" example
- Producer thread
  - Produces data to be consumed by the application and pushes it to a buffer
  - e.g.: produce strings to be printed to standard output
- Consumer threads
  - Pull data from the shared buffer and process it
  - e.g.: pull data from buffer and print it to screen

Producer/Consumer problem



Reader

→ CODE

multiprint1.py

```
work_buffer=""
work_buffer_lock=threading.Lock()

class Reader(threading.Thread):

    def run(self):
        global work_buffer
        job=""
        while job!=None:
            #thread terminates when the producer sends None

            #wait for content
            work_buffer_lock.acquire()
            while work_buffer!=None and len(work_buffer)==0:
                work_buffer_lock.release()

                #we are out of the critical region, let's sleep
                #for a little bit
                time.sleep(1)

                #let's get in again
                work_buffer_lock.acquire()

            #remove the first character from the string buffer
            if work_buffer!=None:
                job=work_buffer[0]
                work_buffer=work_buffer[1:]
            else:
                job=None

            work_buffer_lock.release()

            if job!=None:
                print "%s: %s"%(self.getName(),job)
                #slow down things a bit
                time.sleep(1)

        print "%s: I'm done!"%self.getName()
```

Polling process: we acquire the Lock, check if the buffer contain anything, if not release the Lock and sleep for one second, then acquire it again.

Important: you must always remember to release the lock before sleeping. Otherwise, you would prevent access to all other objects!

This point is reached only when the buffer is not an empty string, or when the buffer has been set to None by the producer (to signal the fact that there is no more work to do). After the condition is verified **the Lock is not released**: no other thread will have access to the buffer, avoiding concurrency problems.

Reader

→ CODE

multiprint1.py

```
work_buffer=""
work_buffer_lock=threading.Lock()

class Reader(threading.Thread):

    def run(self):
        global work_buffer
        job=""
        while job!=None:
            #thread terminates when the producer sends None

            #wait for content
            work_buffer_lock.acquire()
            while work_buffer!=None and len(work_buffer)==0:
                work_buffer_lock.release()

                #we are out of the critical region, let's sleep
                #for a little bit
                time.sleep(1)

                #let's get in again
                work_buffer_lock.acquire()

            #remove the first character from the string buffer
            if work_buffer!=None:
                job=work_buffer[0]
                work_buffer=work_buffer[1:]
            else:
                job=None

            work_buffer_lock.release()

            if job!=None:
                print "%s: %s"%(self.getName(),job)
                #slow down things a bit
                time.sleep(1)

        print "%s: I'm done!"%self.getName()
```

Notice: the function redefines work_buffer to make it point to a new object (strings are immutable). work_buffer needs therefore to be declared as global.

If the buffer is not empty, the reader pulls the first character out of the buffer (a string object). That will be the "printing job" to be executed by the thread.

Print the job character to screen, and use a sleep(1) to "emulate" a more costly operation.

Producer

→ CODE

multiprint1.py

```
class Producer(threading.Thread):

    def run(self):
        global work_buffer
        job=""
        while job!=None:
            #fetch input from the command line
            job=raw_input("Input: ")

            #termination condition is empty string
            job=job if job!="" else None
```

Read from standard input using raw_input

Same pattern as the reader. Acquire the lock, check the condition and continue to poll on a regular basis until the condition is verified. **Always release the Lock before sleeping!!!**

```
#try to dispatch it to workers
work_buffer_lock.acquire()
while len(work_buffer)>0:
    work_buffer_lock.release()

    print "%s: waiting for buffer to empty"%self.getName()
    time.sleep(1)

    work_buffer_lock.acquire()

work_buffer=job
work_buffer_lock.release()
```

Condition variables

- Locks are not sufficient: locks simply prevent simultaneous access to a variable
- We need something else that let us go to sleep waiting for some condition to occur.
- Producer thread
  - Wait for the buffer to become empty
  - Fill the buffer
  - Let everybody know that something changed
- Consumer thread
  - Wait for the buffer to be filled
  - Pull data from buffer
  - Let everybody know that something changed

Reader

→ CODE

multiprint2.py

```
work_buffer=""
work_buffer_cond=threading.Condition()

class Reader(threading.Thread):

    def run(self):
        global work_buffer
        job=""

        while job!=None:
            #thread terminates when the producer sends None

            #wait for content
            work_buffer_cond.acquire()
            while work_buffer!=None and len(work_buffer)==0:
                work_buffer_cond.wait()

            #remove the first character from the string buffer
            if work_buffer!=None:
                job=work_buffer[0]
                work_buffer=work_buffer[1:]
            else:
                job=None
            #signal to everybody that the buffer has changed
            work_buffer_cond.notifyAll()
            work_buffer_cond.release()

            if job!=None:
                print "%s: %s"%(self.getName(),job)
                #slow down things a bit
                time.sleep(1)

        print "%s: I'm done!"%self.getName()
```

Let the thread go to sleep until a specific condition is verified. A Condition is coupled with a Lock, whose operation is hidden: before going to sleep, wait() releases the lock and reacquires it when the thread is woken up by another thread.

Notice the difference between notify() and notifyAll() . Notify will awake one and only one of the waiting threads. NotifyAll will awake all the threads waiting on the specific condition.

Here, the call will wake up all the other readers (that will check if there is more work to do) and the producer (that will check if the buffer is now empty).

Producer

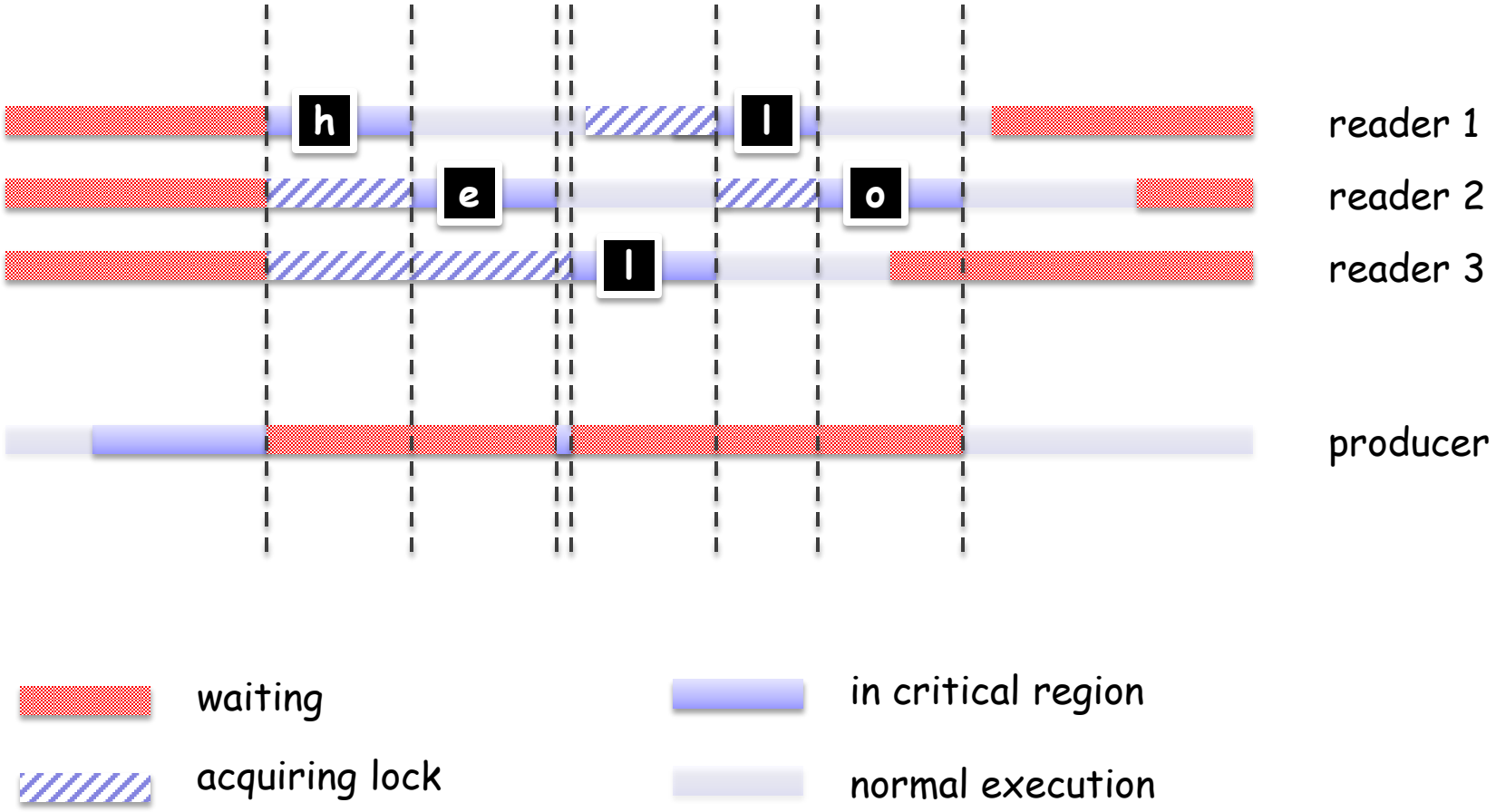
➔ CODE

multiprint2.py

```
class Producer(threading.Thread):

    def run(self):
        global work_buffer
        job=""
        while job!=None:
            #fetch input from the command line
            job=raw_input("Input: ")

            #termination condition is empty string
            job=job if job!="" else None

            #try to dispatch it to workers
            work_buffer_cond.acquire()
            while len(work_buffer)>0:
                work_buffer_cond.wait()

            work_buffer=job
            #signal to everybody that the buffer has changed
            work_buffer_cond.notifyAll()
            work_buffer_cond.release()
```

Whenever a reader consumes the buffer it executes a notifyAll() on the condition. This will awaken all the readers, as well as the producer that will check whether the buffer is empty. If the buffer is not empty, the producer continues to sleep.

Condition variables



Queue.Queue()

- Multi-producer, multi-consumer queues
- Fully thread-safe: all the locking semantics is transparently implemented
- Three types of queues:
  - Queue.Queue(maxsize=0): standard FIFO queue. maxsize is an integer that sets the upper limit on the number of items that can be placed in a queue. Insertion will block if maxsize is reached.
  - LifoQueue(maxsize=0): LIFO queue
  - PriorityQueue(maxsize=0): priority queue, the lowest valued entries are retrieved first. Typical pattern for entries is a tuple (<priority_number>,data).

Reader

➔ CODE

multiprint3.py

```
#just create a queue
q=Queue.Queue()

class Reader(threading.Thread):

    def run(self):
        job=""

        while job!=None:
            #thread terminates when the producer sends None

            #wait for content
            job=q.get()

            if job!=None:
                print "%s: %s"%(self.getName(),job)
                #slow down things a bit
                time.sleep(1)

            #notify the queue that you have done your job
            q.task_done()

        print "%s: I'm done!"%self.getName()
```

Standard queue methods are blocking. _nowait() methods are defined to allow non-blocking I/O. Each method also allows a blocking flag and a timeout value, to implement non blocking or timeout-based behavior similar to what we have seen before.

A queue needs to be notified when a queued element has been fully processed by the consumer. If the queue has limited size, only at that moment will a producer be allowed to insert a new element in the queue if the queue was previously full.

Producer

➔ CODE

multiprint3.py

```
class Producer(threading.Thread):

    def run(self):
        while True:
            #fetch input from the command line
            job=raw_input("Input: ")

            if job!="":
                #wait for the queue to be empty
                q.join()
                #insert the elements
                [q.put(c) for c in job]
            else:
                #send a None to each thread to let them know
                #we are done.
                [q.put(None) for i in range(THREADS)]
                break
```

Queue.join() returns only once the queue has been emptied.

Put each character in the queue. Each thread will receive a character by calling Queue.get()

Signal the completion of the work to all threads by pushing None values to the queue. Every thread will receive a None value and immediately exit. One None value per thread needs therefore to be pushed to the queue.

Threads in blocking I/O

➔ CODE

server-thread.py

```
class ClientSocket(threading.Thread):

    def __init__(self,sock,addrinfo):
        threading.Thread.__init__(self) #don't forget!

        self.conn=sock
        self.addr=addrinfo
        self.daemon=True #no need to join

    def run(self):
        print "client %s:%d connected"%addr
        while True:
            data=conn.recv(1024)
            if not data: break
            conn.send(data.upper())
        conn.close()
        print "client %s:%d disconnected"%addr


srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
srv.bind((HOST,PORT))
srv.listen(1)

while True:
    conn,addr=srv.accept()
    #create the thread and forget about it
    ClientSocket(conn,addr).start()
```

Don't forget to call the super constructor when subclassing Thread.

Child threads are daemonic: the main thread does not need to wait for their completion

Compare to the os.fork() case: the very same code that was was executed before in a separate process is now executed in the thread. In more complex scenarios, it will however be easier to share information among threads.

The main thread continues to accept new connections and assign them to new threads.

Thread pools

- While faster to create than processes, thread creation requires a non-null time
- Creating one thread per client may be expensive in certain scenarios
  - Total number of threads in a host is limited
  - → malicious behaviors…
- Thread pools
  - A limited number of threads is created at server startup
  - Any inbound connection is handled by a free thread of the pool (if any)

Thread pools

→ CODE

server-pool.py

```
srv=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
srv.bind((HOST,PORT))
srv.listen(1)
#couple the socket with a Lock
srv_lock=threading.Lock()

class ClientSocket(threading.Thread):

    def run(self):
        while True:
            srv_lock.acquire()
            conn,addr=srv.accept()
            srv_lock.release()


            print "client %s:%d connected"%addr

            while True:
                data=conn.recv(1024)
                if not data: break
                conn.send(data.upper())
            conn.close()
            print "client %s:%d disconnected"%addr
```

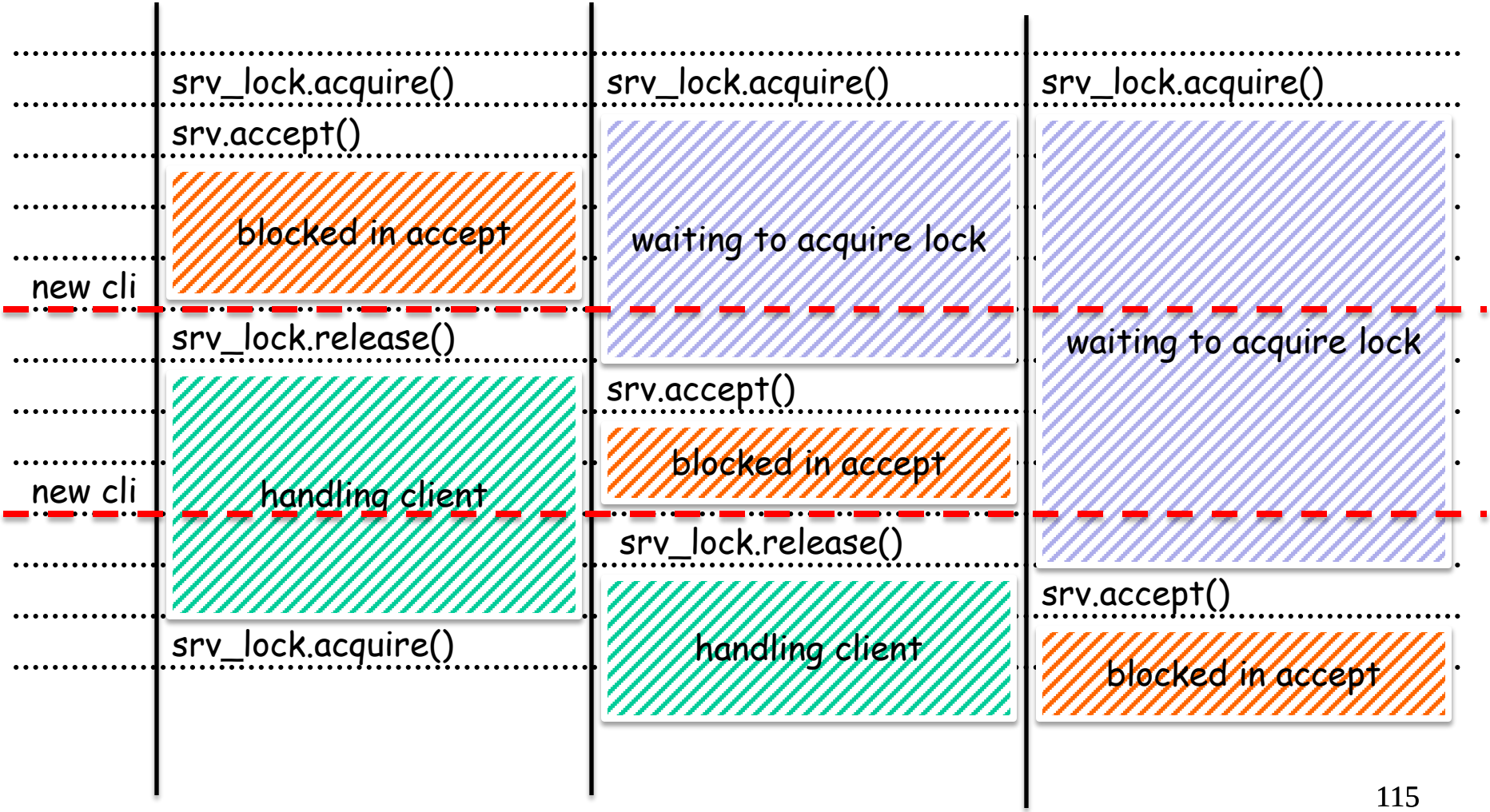
Differently from the previous implementation, the server socket is accessed concurrently by all threads to accept new clients.

All free threads will compete for the access to the critical region. Only one of the threads will run the accept() call, and will receive a client to handle. Once this will happen the thread will release the lock and let another free thread call the accept()

```
pool=[ClientSocket() for i in range(POOLSIZE)]
[t.start() for t in pool]
[t.join() for t in pool]
```

The main thread creates the thread pool, starts it and then waits for the threads of the thread pool to terminate.

Thread pools



Note on threads and GIL

- Common Python implementations of threads means that only one thread is ever running in the Python virtual machine at any point in time
  - Python threads are truly operating system threads
  - However, all threads must acquire a single shared lock (Global Interpreter Lock) when they are ready to run
- ➔ Python threads cannot today be distributed across multiple cores on a multi-core architecture
- Python threads are suitable for multiplexing blocking I/O, but not for parallel computation
  - Look at the multiprocessing library for that!