Chapter 3
Python tutorial +
Processes (part 2)
IPC Examples

CS 3423 Operating Systems

National Tsing Hua University

Python tutorial

CS 3423 Fall 2019

National Tsing Hua University

Python programming language

- What is Python?
  - High-level dynamic programming language
  - multi-paradigm: procedural, OO, functional, ...
  - highly readable, "executable pseudocode"
- What is Python good for?
  - General-purpose programming, machine learning
  - Quick validation of code ideas
  - Algorithms in OS, especially with concurrency
  - Friendlier way to try out system calls!! Just type directly

Installation

- Recommended:
  - preferably latest (3.7.*), but at least python 3.6, minimally python 3.*
- Several main ways
  - built-in to your system (may be named python3)
  - install from www.python.org
  - install for Cygwin (Windows only) - text mode
  - install Jupyter notebook, use Anaconda distribution

Interactive mode

```
$ python3
>>> print('hello')
hello
>>> x = 3
>>> x + 2
5
>>> y = 'hello'
>>> y + 'world'
'helloworld'
>>> y[0]
'h'
>>> y[4]
'o'
>>> y[1:]
'ello'
>>> y[1:3]
'el'
>>> y[::-1]
'olleh'
>>> 'H' > 'h'
False
>>> 'H' <= 'h'
True
```

Collection data types

- `list`
  - "dynamically array" of mixed types
- `tuple`
  - read-only (immutable) version of list
- `set`
  - unordered collection of immutable items, use union, intersection, subtraction operators
- `dict`
  - key-value pairs, think hash tables

Examples of Collection types

```
# Lists
>>> L = [1, 2, 3]
>>> L[0]
1
>>> L[1] = 'Hi'
>>> L
[1, 'Hi', 3]
>>> L + [6, 7, 8]
[1, 'Hi' 3, 6, 7, 8]
# Tuples
>>> T = (12, 34, 56)
>>> T[2]
56
```

```
# Sets
>>> A = {1,2,3}
>>> B = {1,3,5}
>>> A & B
{1, 3}
>>> A | B
{1, 2, 3, 5}
>>> A - B
{2}
>>> A ^ B
{2, 5}
```

Examples of dict: key-value pairs

```
>>> d = {'Jan': 1, 'Feb': 2, 'Mar': 3}
>>> d['Feb']
2
>>> d['Apr'] = 4
>>> d
{'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4}
>>> 'Mar' in d
True
>>> 'May' in d
False
>>> len(d)
4
```

Functions

```
>>> def Double(x):      # defines a function
...     return x + x
...
>>> Double(10)          # calling a function
20
>>> Double('10')        # on different types
'1010'
>>> dbl = Double        # copy "fn pointer"
>>> dbl(20)
40
```

Generator in Python

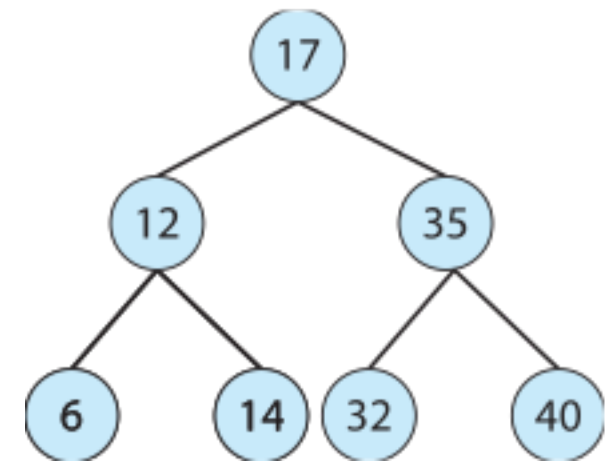
- like a function but yield instead of return
  - yield means it can resume after yield
- Usage: instantiate, then next()
  -
```
def numgen():
    i = 0
    while True:
        yield i
        i += 1
```

```
>>> g = numgen() # instantiate
>>> next(g) # run till yield
0
>>> next(g) # run till yield
1
>>> next(g)
2
>>>
```

  - generator can also be used in for-loop, which instantiates generator and calls next() automatically

Representation of data structures

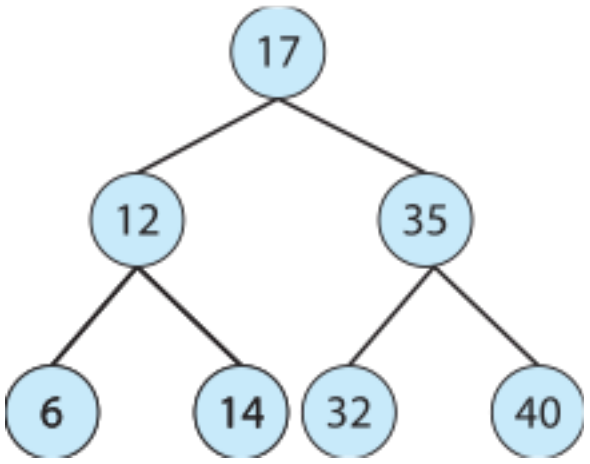
- Usually easier to use built-in data type
- Example: tree



  - could represent it with tuples (root, left, right) recursively
  - T = (17, (12, (6, None, None), (14, None, None)), (35, (32, None, None), (40, None, None)))
  - This is "pre-order" (root first)

Code for pre-order generator

```
def preorder(T):
    if not T:
        return
    yield T[0]
    for i in preorder(T[1]):
        yield i
    for i in preorder(T[2]):
        yield i

if __name__ == '__main__':
    T = (17, (12, (6, None, None), (14, None, None)),
             (35, (32, None, None), (40, None, None)))
    L = [i for i in preorder(T)] # list comprehension
    print(L)
```




```
$ python3 tree.py
[17, 12, 6, 14, 35, 32, 40]
```

yield from

- in case of recursive call or another generator, simply do "yield from" instead of a for-loop to yield each item!

```python
def preorder(T):
    if not T:
        return
    yield T[0]
    for i in preorder(T[1]):
        yield i
    for i in preorder(T[2]):
        yield i
```

```python
def preorder(T):
    if not T:
        return
    yield T[0]

    yield from preorder(T[1])

    yield from preorder(T[2])
```

IPC Examples

- Shared memory
  - POSIX
- Message Passing
  - Mach IPC
  - Pipes
  - Sockets
  - Remote procedure calls

POSIX Shared Memory IPC

- Include files
  - #include <sys/mman.h>
  - #include <fcntl.h>
- shm_open(*name*, *flag*)
  - open a shared-memory object with given name, similar to file
  - returns a file descriptor (nonnegative int)
- ptr=mmap(*addr*,*len*,*prot*,*flags*,*fd*,*offset*)
  - map the opened file descriptor for the shared memory object to the address region that you want

POSIX shared memory example (from textbook).. for producer

replace with
`#include <sys/mman.h>`
`#include <unistd.h>`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

/* create the shared memory object */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

/* configure the size of the shared memory object */
ftruncate(shm_fd, SIZE);

/* memory map the shared memory object */
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

/* write to the shared memory object */
sprintf(ptr,"%s",message_0);
ptr += strlen(message_0);
sprintf(ptr,"%s",message_1);
ptr += strlen(message_1);

return 0;
}
```

on Linux, compile with flag at end

```
$ cc shm_p.c -o shm_p -lrt
```

POSIX shared memory example (from textbook).. for consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory*/
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

replace with
```
#include <sys/mman.h>
#include <unistd.h>
```

on Linux, compile with flag `-lrt` at the end
```
$ cc shm_c.c -o shm_c -lrt
```

To run,
```
$ ./shm_p &
$ ./shm_c &
```

Producer-Consumer example using POSIX shared memory

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#define BUFFER_SIZE 10
const int SIZE = 4096;
const char *name = "OS";

typedef struct shm_struct {
    int in_p, out_p;
    char buffer[BUFFER_SIZE];
} shm_struct_type;

int shm_fd;
shm_struct_type *ptr;
char make_item() {
    static char c = 'A';
    if (c > 'Z') {
        c = 'A';
        printf("make newline\n");
        return '\n';
    }
    printf("make %c\n", c);
    return c++;
}
void use_item(char c) {
    printf("consume %c\n", c);
}
```

```c
void producer() {
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SIZE);
    ptr = (shm_struct_type*)mmap(0, SIZE, PROT_WRITE | PROT_READ,
                                 MAP_SHARED, shm_fd, 0);
    ptr->in_p = ptr->out_p = 0;
    while (1) {
        char next_produced = make_item();
        while (((ptr->in_p+1)% BUFFER_SIZE) == ptr->out_p) {/* yield*/ }
        ptr->buffer[ptr->in_p] = next_produced;
        ptr->in_p = (ptr->in_p + 1) % BUFFER_SIZE;
    }
}
void consumer() {
    shm_fd = shm_open(name, O_RDWR, 0666);
    ptr = (shm_struct_type*)mmap(0, SIZE, PROT_WRITE | PROT_READ,
                                 MAP_SHARED, shm_fd, 0);
    while (1) {
        while (ptr->in_p == ptr->out_p) { /* yield */ }
        char next_consumed = ptr->buffer[ptr->out_p];
        ptr->out_p = (ptr->out_p + 1) % BUFFER_SIZE;
        use_item(next_consumed);
    }
    shm_unlink(name);
}
```

```c
int main(int argc, char **argv) {
    if (fork()) { /* parent */
        printf("producer\n");
        producer();
    } else {
        printf("consumer\n");
        consumer();
    }
}
```

But... is shared memory between processes an overkill?

- Do you really need two processes (or threads) just to do producer-consumer?
  - seems very unstructured! very hard to trace
  - how would you debug?
- Or is there a more structured way?
  - some "factory" object that can be invoked repeatedly to give a series of data objects?

C: function with a static variable

```
char make_item() {
    static char c = 'A';
    if (c > 'Z') {
        c = 'A';
        printf("make newline\n");
        return '\n';
    }
    printf("make %c\n", c);
    return c++;
}
```

- static local variable
  - like global (one instance) except name is locally visible only
  - `static char c = 'A'` initialized only once at start of program, not every time the function is called!
  - c is the value to return next time; or if `c > 'Z'` then return a newline and wrap around to `'A'`
- Problems
  - only one instance of `make_item()` can be used! since there is only one static local `c`
  - hard to generalize to more complex items

Python solution: Generator

- generator in Python
  - a function that can yield and resume after the yield

```c
char make_item() {
    static char c = 'A';
    if (c > 'Z') {
        c = 'A';
        printf("make newline\n");
        return '\n';
    }
    printf("make %c\n", c);
    return c++;
}
```

C version

-

Python version

```python
def make_item():
    import string
    while True:
        for c in string.ascii_uppercase+'\n':
            yield c  # instead of return!!!
```

```python
>>> m = make_item() # instantiate generator
>>> next(m)         # run from beginning till yield
'A'
>>> [next(m) for i in range(27)] # keep resuming till yield
['B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
'Z', '\n', 'A']
```

Producer-Consumer with Python generator

- Consumer can use a `for`-loop
    - ```
      def consumer():
          for i in make_item():
              use_item(i)
      ```
    - That's it! very clean structure, easy to understand
    - `for`-loop instantiates the generator and calls `next()` for you!!
- Synchrony
    - this is a form of *rendezvous* synchronization
      => producer and consumer alternate till one cannot run any more (to wait for the other)
    - issue: no parallelism; not making and using item at same time!
      ```

two-way communication in Python generators

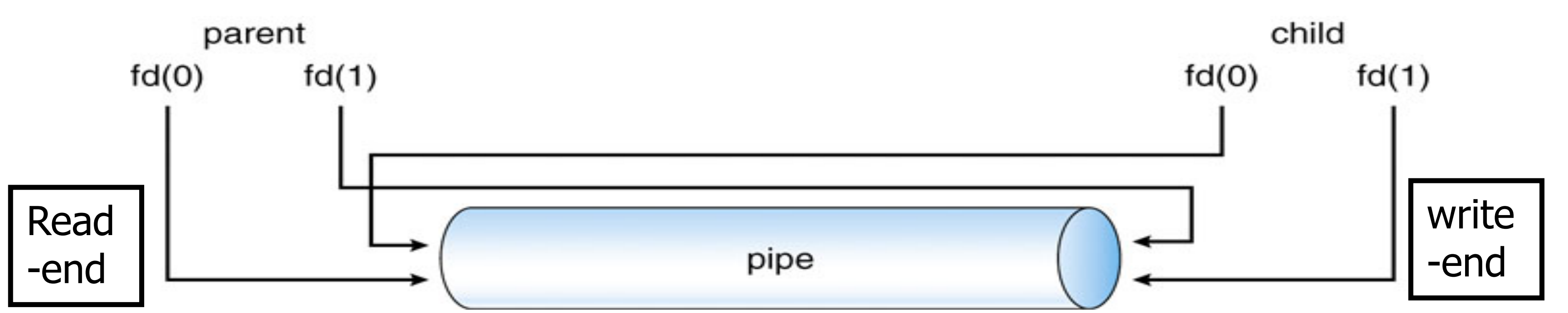
- caller can use `g.send(v)` to send a value
  - generator receives it as `yield` expression's value
  - `next(g)` must be called at least once initially

```python
def gennum(initval):
    while True:
        r = yield initval
        if type(r) == int:
            initval += r
        else:
            initval += 1
```

```python
>>> g = gennum(10) # instantiate
>>> next(g) # start g; can't send()
10
>>> next(g) # same as g.send(None)
11
>>> g.send(5) # received by yield
16
>>> next(g) # same as g.send(None)
17
>>> g.send(5)
22
```

Pipes

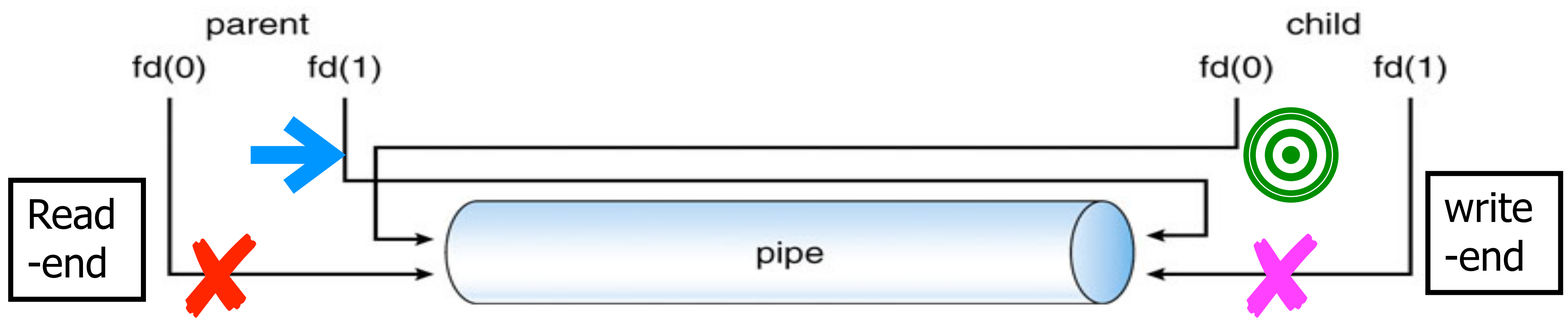
- one of the first IPC mechanisms in early Unix
- Pipe is accessed like a special type of file
  - Use file API for reading writing, but no random access
- Issues with implementation
  - unidirectional or bidirectional?
  - half duplex or full duplex?
  - is there a parent-child relationship?
  - over the network or reside on same machine?



Ordinary Pipes

- Also called anonymous pipes in Windows
- Requires parent-child relationship between communicating processes
  - implemented as a special file on Unix (via `fork()`)
  - child process inherits open files from parent
  - can only be used between processes **on same machine**
- Unidirectional (simplex)
  - two pipes must be used for two-way communication
  - Unix: `int fd[2]; pipe(fd);`
  - Windows: `CreatePipe(&ReadHandle, &WriteHandle, &sa, 0);`

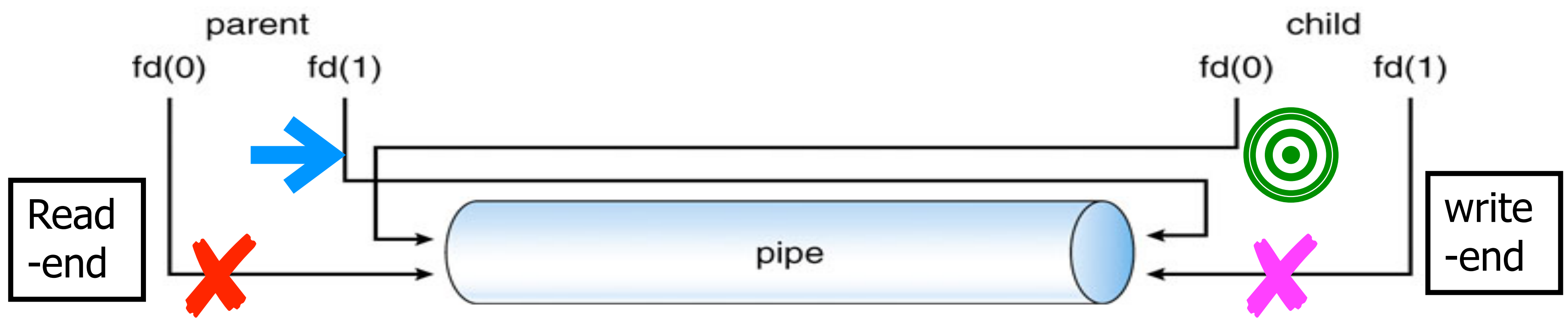
pipe example in C from textbook




```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void) {
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    /* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr, "Pipe failed\n");
        exit(1);
    }
```

```c
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        exit(2);
    }
    if (pid > 0) { // parent
        close(fd[READ_END]);
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
        close(fd[WRITE_END]);
    } else {
        close(fd[WRITE_END]);
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s", read_msg);
        close(fd[READ_END]);
    }
}
```

pipe example in Python




```python
import os

def make_item():
    import string
    while True:
        for c in string.ascii_uppercase + '\n':
            yield c

def use_item(c):
    print("consume %c" % c)

def producer():
    item_gen = make_item() # instantiate generator
    w = os.fdopen(write_fd, 'w')
    while True:
        next_item = next(item_gen)
        w.write(next_item)
    os.close(write_fd)
```

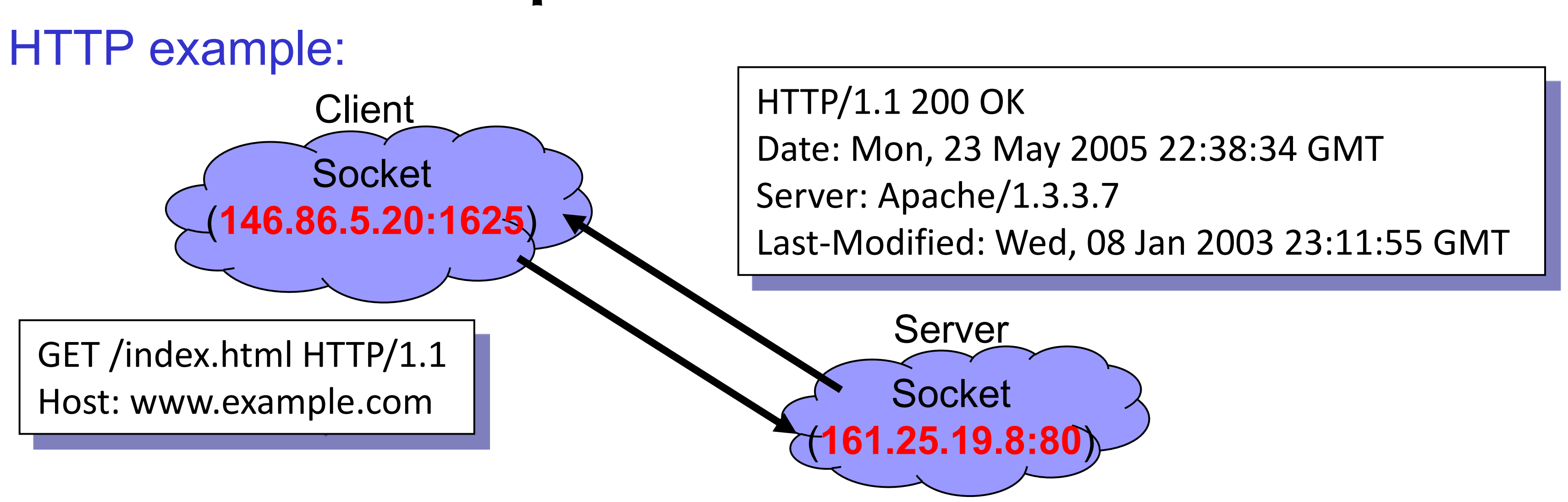
```python
def consumer():
    r = os.fdopen(read_fd, 'r')
    while True:
        next_item = r.read(1)
        print("read %c" % next_item)
    os.close(read_fd)

if __name__ == '__main__':
    # create the pipe
    read_fd, write_fd = os.pipe()
    pid = os.fork()
    if (pid > 0):           # parent
        os.close(read_fd)   # close the read
        producer()
    else:                   # child
        os.close(write_fd)  # close the write
        consumer()
```

Named Pipes

- No parent-child relationship is required
- Several processes can use it for communication
  - may have several writers
- Continue to exist after process terminates
- Unix
  - also called FIFO, must be on same machine
- Windows
  - bidirectional, can be on different machines

Sockets

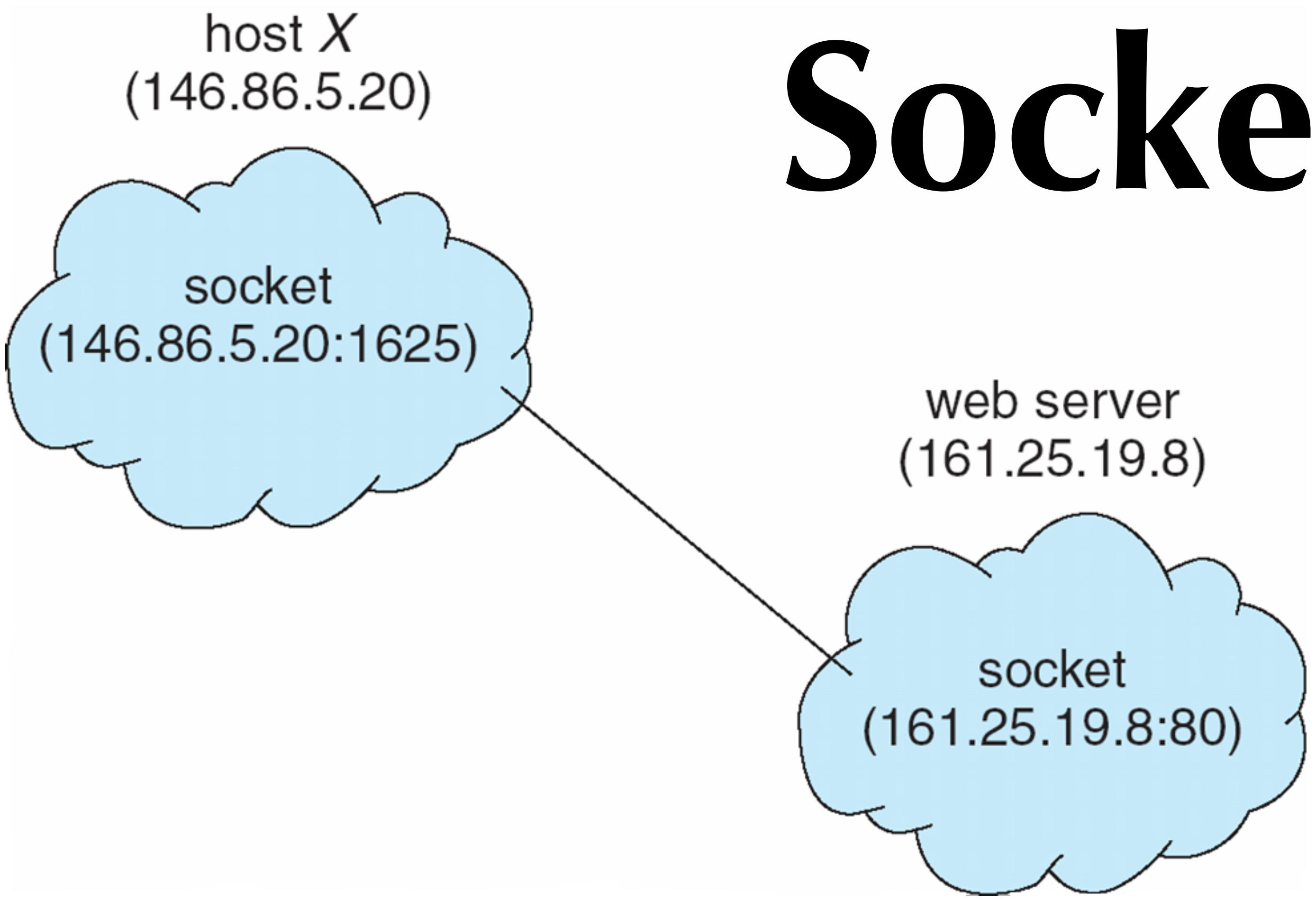
- unstructured stream of bytes
  - Low-level form of communication
  - as opposed to fixed-sized packets or struct or text with syntax
    => client and server need to agree on format
- HTTP example



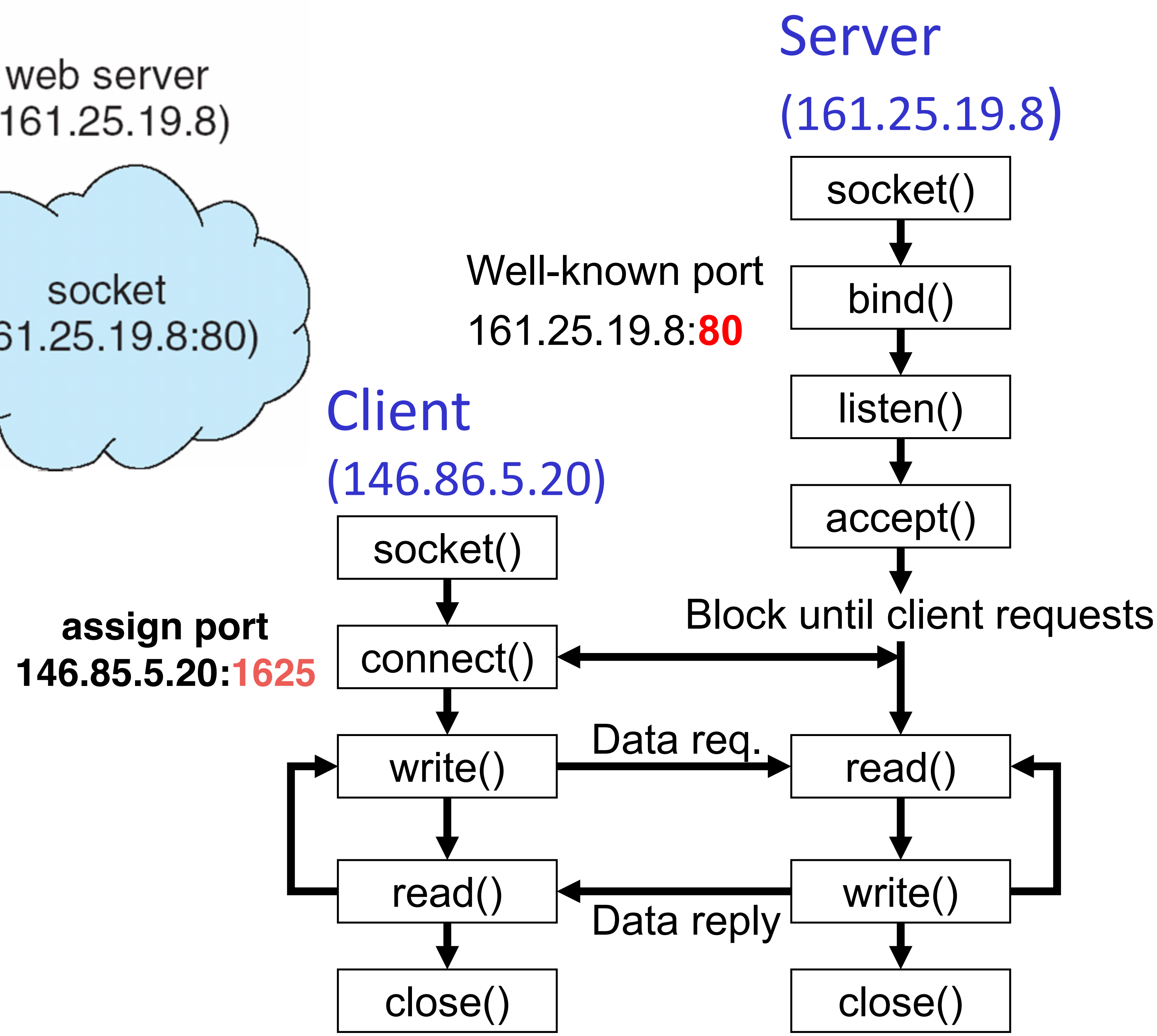
Sockets

- A socket is identified by concatenating
  - IP address : port number
  - e.g., `127.0.0.1:8080`
- Communication is between a pair of sockets
- Localhost
  - IPv4 address (32-bit) `127.0.0.1`
  - IPv6 address (64-bit) is `::1`, which is short for `0000:0000:0000:0000:0000:0000:0000:0001`

Sockets



host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)



Server
(161.25.19.8)
socket()
bind()
listen()
accept()
Well-known port
161.25.19.8:80
Block until client requests
Client
(146.86.5.20)
socket()
connect()
assign port
146.85.5.20:1625
write()
read()
close()
Data req.
Data reply
read()
write()
close()

Socket: client

- import socket # in Python
- s = socket.socket() #create a socket obj
- s.connect(*addr*, *port*) # connect to server
- s.recv(*nBytes*)
- s.send(*data*)
- s.close() # close the connection

Socket: server

- s = socket.`socket`() #create a socket object
- s.`bind`((*addr*, *port*)) # bind socket to addr, port
- s.`listen`(*nMaxConn*) # wait for client to connect
- Loop over multiple incoming connections c
  - c = s.`accept`() # get socket object and address
  - c.`send`(*data*)
  - c.`recv`(*nBytes*)
  - c.`close`() # close the connection

Python code to test sockets

client.py

```python
import socket

host = socket.gethostbyname('localhost')
port = 12345

s = socket.socket()
s.connect((host, port))
s.send(bytes('Hi!  I am the client \
making a request!', 'utf8'))
resp = s.recv(1024)
print('response from host: %s' % resp)
s.close()
```

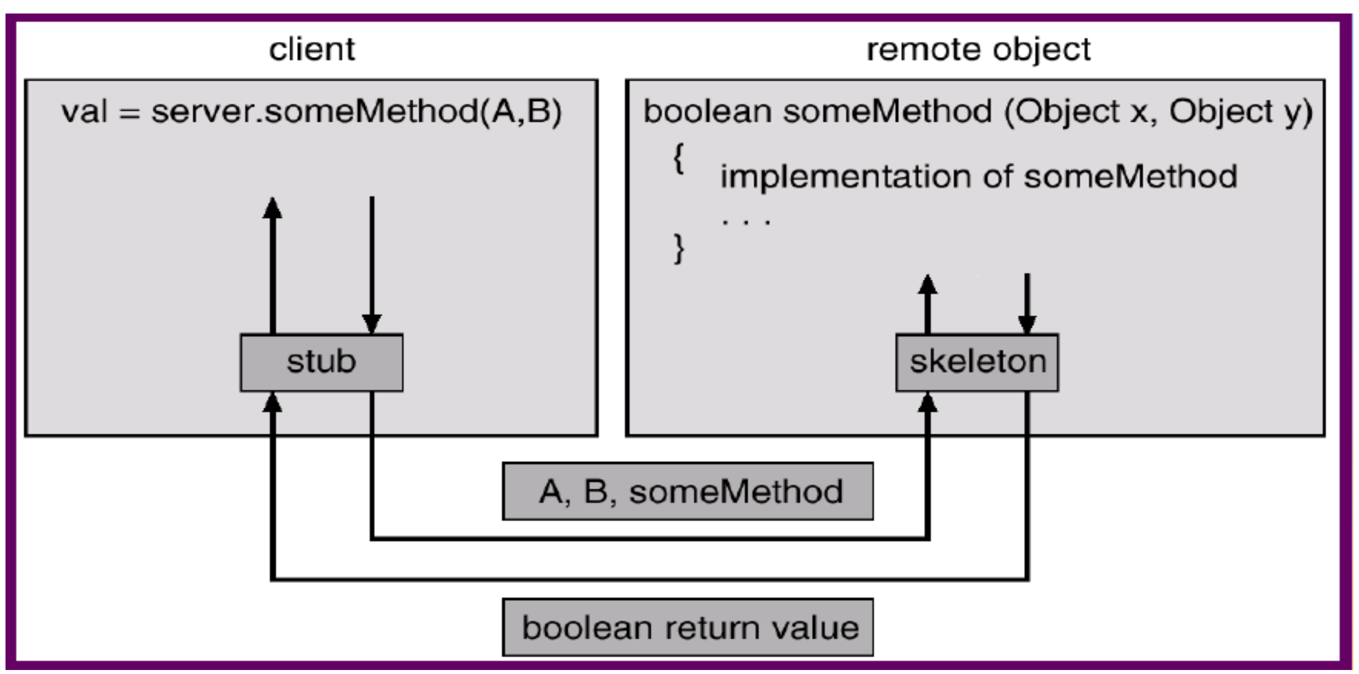
server.py

```python
import socket

host = socket.gethostbyname('localhost')
port = 12345

s = socket.socket()
s.bind((host, port))
s.listen(5)

while True:
    c, client = s.accept()
    req = c.recv(1024)
    print('from %s, requesting %s' % \
          (client, req))
    reply = 'response from server: your address \
is %s, your request %s' % (client, req)
    c.send(bytes(reply, 'utf8'))
    c.close()
```

Compare with Java version in textbook, Fig. 3.27-3.28

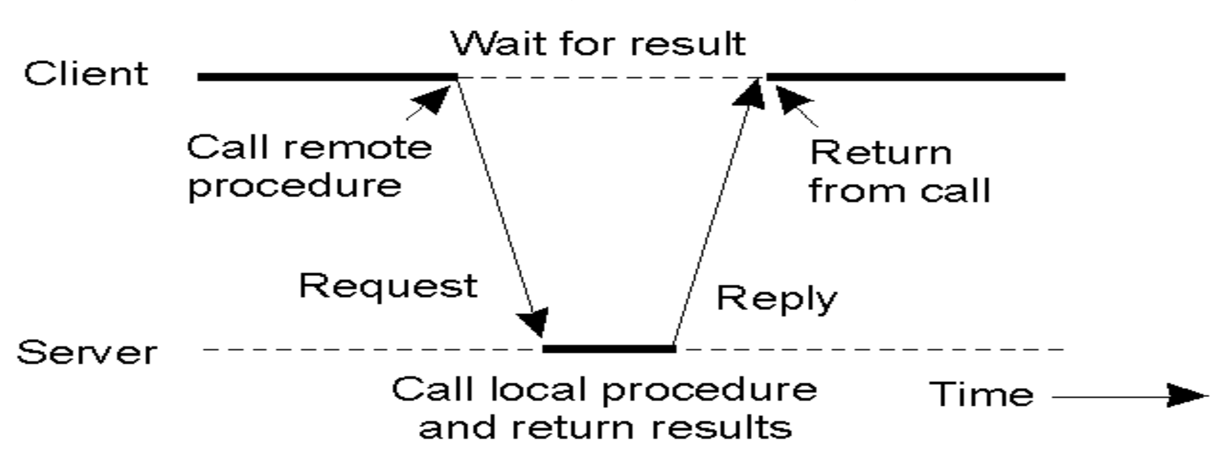
Remote Procedure Calls (RPC)

- Allows a program to **call procedures** on another machine
  - looks like a procedure call to a program
  - in reality, the call executes on another host
- Stub: proxy for the RPC on client and server



Client and Server Stubs

- Client stub
  - "**marshaling**": packs parameters into a message
  - calls OS to send directly to server (network)
  - waits for result to return from server (network)
- Server stub
  - receives call from a client, "**de-marshaling**": unpacks param
  - calls the corresponding procedure
  - returns results to the caller (network)



RPC Problems

- Data representation
  - integer, floating point?
- Different address spaces
  - what is the meaning of pointer?
- Communication error
  - duplicate or missing calls

RPC problems: data representation issue

- Problem
  - IBM mainframes use EBCDIC char encoding, but most others use ASCII
  - integer: one's complement vs 2's complement, little endian vs big endian
  - Floating-point numbers, sizes
- Solution
  - external data representation (XDR)

RPC problems: Address Space Issue

- A pointer is only meaningful in address space
- Solution
  - no pointer usage in RPC call
  - Copy the entire pointed area (arrays, strings)
  - only suitable for bounded and known areas

RPC Problems: communication issue

- RPC may fail or duplicate execution
  - due to problem in network
- At most once
  - attach timestamp (or sequence number) to each msg
  - server must keep a history large enough to ensure repeated msg
- Exact once
  - server must acknowledge to client RPC call received & executed
  - client must resend each RPC call periodically until server receives ACK