Thread

- Running several threads is similar to running several different programs concurrently, but with the following benefits:

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

- Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

Thread

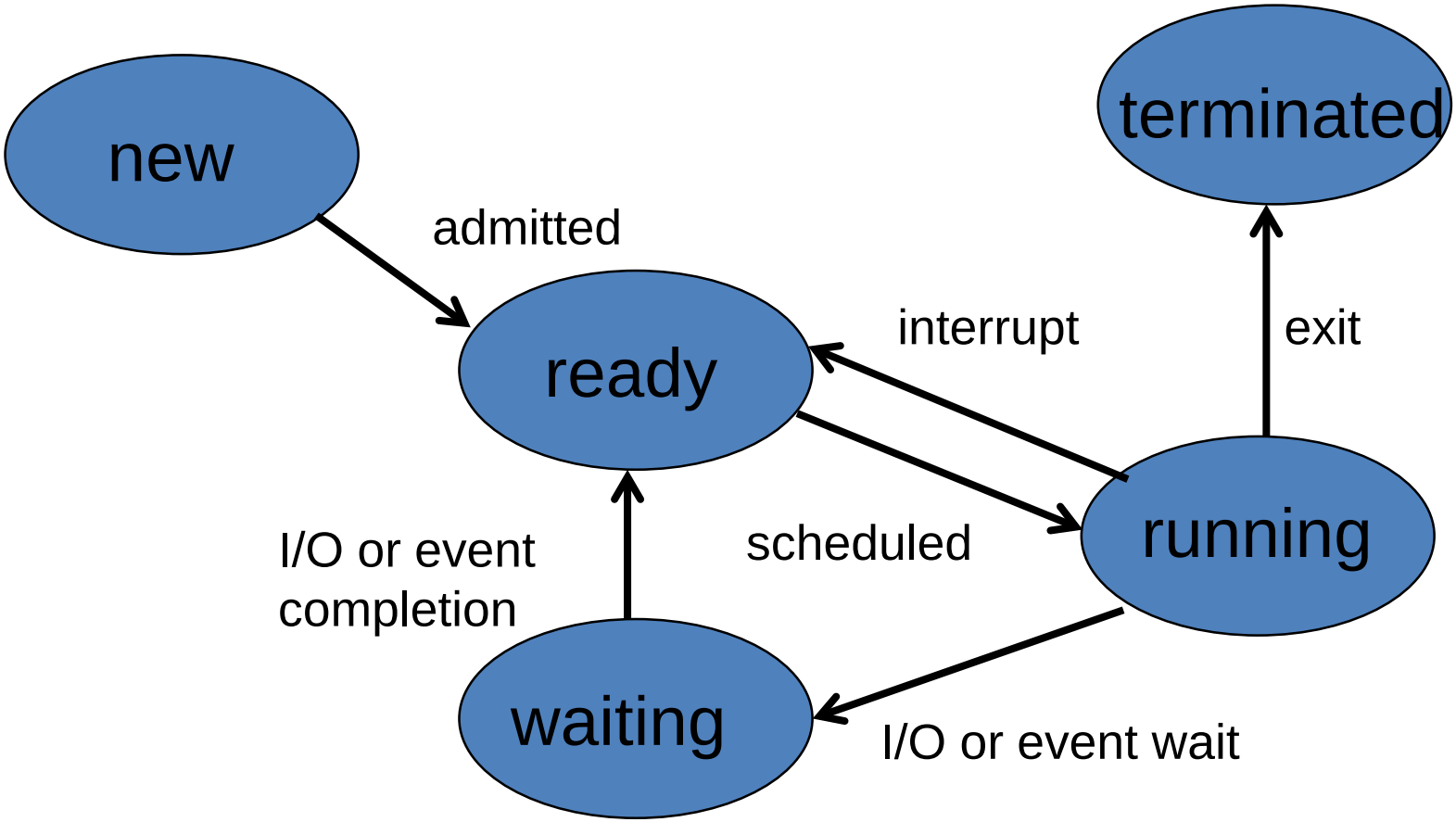
- A thread has a beginning, an execution sequence, and a conclusion.
- It has an instruction pointer that keeps track of where within its context it is currently running.
- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Processes

- Process
  - A basic unit of work from the viewpoint of OS
  - Types:
    - Sequential processes: an activity resulted from the execution of a program by a processor
    - Multi-thread processes
  - An Active Entity
    - Program Code – A Passive Entity
    - Stack and Data Segments
  - The Current Activity
    - PC, Registers, Contents in the Stack and Data Segments

Processes

- Process State

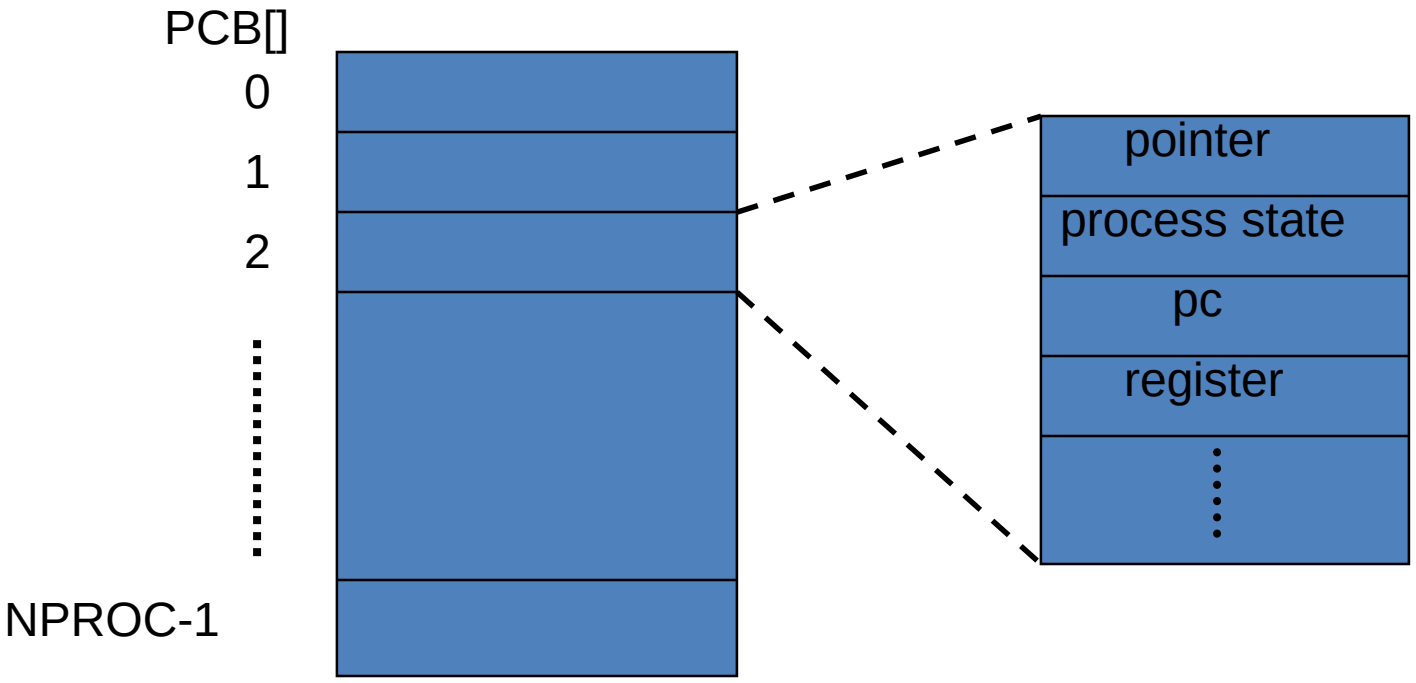


Processes

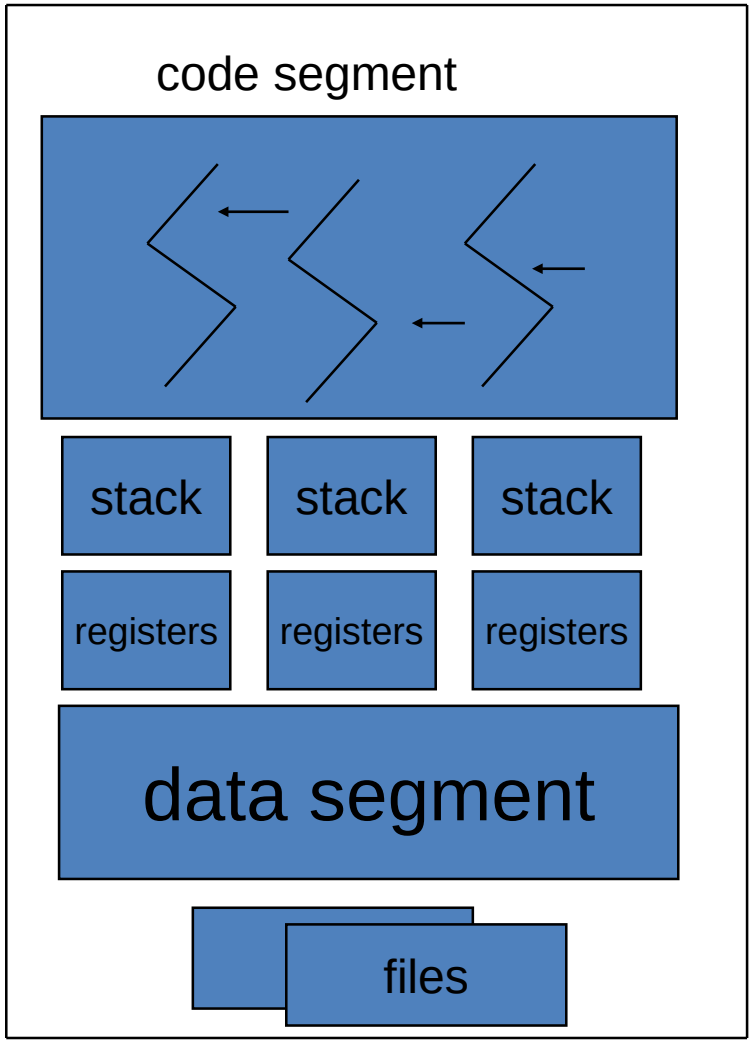
- Process Control Block (PCB)
  - Process State
  - Program Counter
  - CPU Registers
  - CPU Scheduling Information
  - Memory Management Information
  - Accounting Information
  - I/O Status Information

Processes

- PCB: The repository for any information that may vary from process to process



Threads

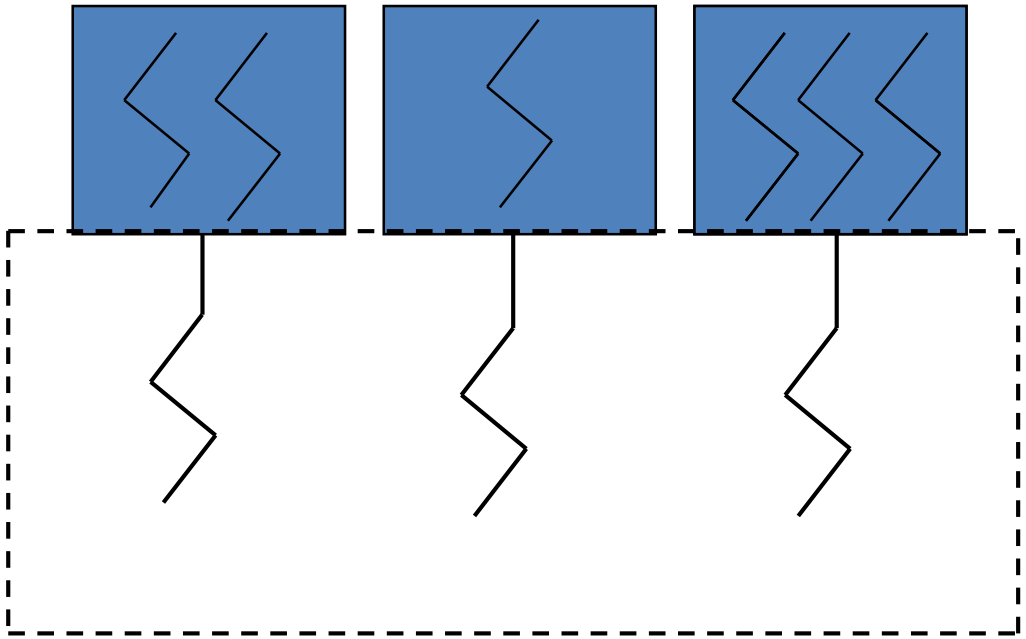


- Motivation
  - A web browser
    - Data retrieval
    - Text/image displaying
  - A word processor
    - Displaying
    - Keystroke reading
    - Spelling and grammar checking
  - A web server
    - Clients' services
    - Request listening

Threads

- Benefits
  - Responsiveness
  - Resource Sharing
  - Economy
    - Creation and context switching
      - 30 times slower in process creation in Solaris 2
      - 5 times slower in process context switching in Solaris 2
  - Utilization of Multiprocessor Architectures

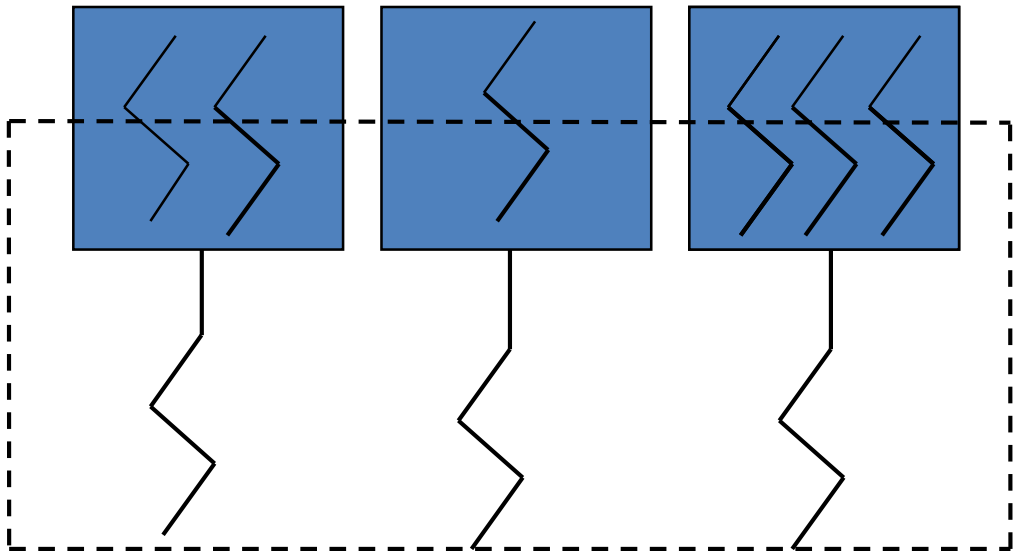
User-Level Threads



- User-level threads are implemented by a thread library at the user level.
- Examples:
  - POSIX Pthreads, Mach C-threads, Solaris 2 UI-threads

- Advantages
  - Context switching among them is extremely fast
- Disadvantages
  - Blocking of a thread in executing a system call can block the entire process.

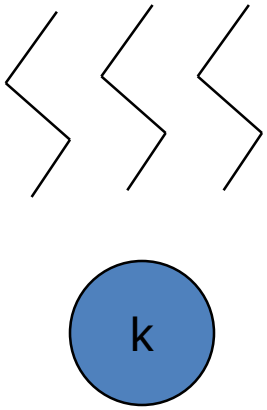
Kernel-Level Threads



- Kernel-level threads are provided a set of system calls similar to those of processes
- Examples
  - Windows 2000, Solaris 2, True64UNIX

- Advantage
  - Blocking of a thread will not block its entire task.
- Disadvantage
  - Context switching cost is a little bit higher because the kernel must do the switching.

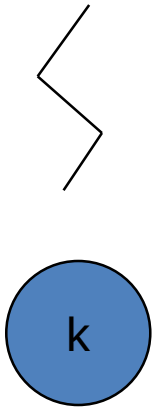
Multithreading Models



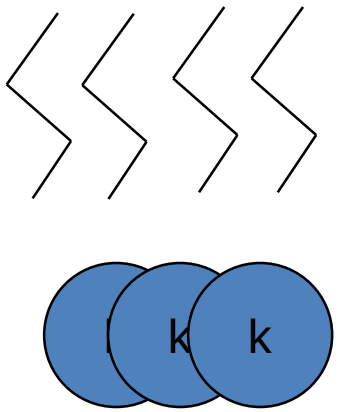
- Many-to-One Model
  - Many user-level threads to one kernel thread
  - Advantage:
    - Efficiency
  - Disadvantage:
    - One blocking system call blocks all.
    - No parallelism for multiple processors
  - Example: Green threads for Solaris 2

Multithreading Models

- One-to-One Model
  - One user-level thread to one kernel thread
  - Advantage: One system call blocks one thread.
  - Disadvantage: Overheads in creating a kernel thread.
  - Example: Windows NT, Windows 2000, OS/2



Multithreading Models



- Many-to-Many Model
  - Many user-level threads to many kernel threads
  - Advantage:
    - A combination of parallelism and efficiency
  - Example: Solaris 2, IRIX, HP-UX,Tru64 UNIX

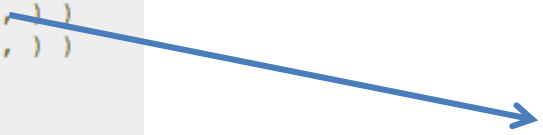
Starting a New Thread

- To spawn a thread, you need to call following method available in *thread* module:
  - thread.start_new_thread (function, args[, kwargs])
- This method call enables a fast and efficient way to create new threads in both Linux and Windows.
- The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*.
- When function returns, the thread terminates.
- Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments.
- *kwargs* is an optional dictionary of keyword arguments.

EXAMPLE

```
#!/usr/bin/python

import thread
import time

# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

```
Thread-1: Thu Aug 21 09:54:08 2014
Thread-2: Thu Aug 21 09:54:10 2014
Thread-1: Thu Aug 21 09:54:10 2014
Thread-1: Thu Aug 21 09:54:12 2014
Thread-2: Thu Aug 21 09:54:14 2014
Thread-1: Thu Aug 21 09:54:14 2014
Thread-1: Thu Aug 21 09:54:16 2014
Thread-2: Thu Aug 21 09:54:18 2014
Thread-2: Thu Aug 21 09:54:22 2014
Thread-2: Thu Aug 21 09:54:26 2014
```



- Although it is very effective for low-level threading, but the *thread* module is very limited compared to the newer threading module.

The *Threading* Module

- The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the *thread* module discussed in the previous section.
- The threading module exposes all the methods of the *thread* module and provides some additional methods:
- **threading.activeCount():** returns the number of thread objects that are active.
- **threading.currentThread():** returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** returns a list of all thread objects that are currently active.

The *Threading* Module

- The ***threading*** module includes the *Thread* class that implements threading.
- The methods provided by the *Thread* class are as follows:
  - **run():** is the entry point for a thread.
  - **start():** starts a thread by calling the run method.
  - **join([time]):** waits for threads to terminate.
  - **isAlive():** checks whether a thread is still executing.
  - **getName():** returns the name of a thread.
  - **setName():** sets the name of a thread.

Creating Thread using *Threading* Module

- To implement a new thread using the threading module, you have to do the following:

1. Define a new subclass of the *Thread* class.
2. Override the *__init__(self [,args])* method to add additional arguments.
3. Then, override the *run(self [,args])* method to implement what the thread should do when started.
4. Once you have created the new *Thread* subclass, you can create an instance and then start a new thread by invoking the *start()*, which will in turn call *run()* method.

EXAMPLE

```
#!/usr/bin/python

import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

```
Starting Thread-1Starting Thread-2Exiting Main Thread


>>> Thread-1: Thu Aug 21 09:52:58 2014
Thread-2: Thu Aug 21 09:52:59 2014
Thread-1: Thu Aug 21 09:52:59 2014
Thread-1: Thu Aug 21 09:53:00 2014
Thread-2: Thu Aug 21 09:53:01 2014
Thread-1: Thu Aug 21 09:53:01 2014
Thread-1: Thu Aug 21 09:53:02 2014
Exiting Thread-1
Thread-2: Thu Aug 21 09:53:03 2014
Thread-2: Thu Aug 21 09:53:05 2014
Thread-2: Thu Aug 21 09:53:07 2014
Exiting Thread-2
```

Process Synchronization

- Why Synchronization?
  - To ensure data consistency for concurrent access to shared data!
- Contents:
  - Various mechanisms to ensure the orderly execution of cooperating processes

Process Synchronization

– A Consumer-Producer Example

- Producer

```
while (1) {
    while (counter == BUFFER_SIZE);
    …
    produce an item in nextp;
    ….
    buffer[in] = nextp;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer:

```
while (1) {
    while (counter == 0)
    …
    nextc = buffer[out];
    out = (out +1) % BUFFER_SIZE;
    counter--;
    consume an item in nextc;
}
```

Process Synchronization

- counter++ vs counter—

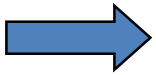
  r1 = counter r2 = counter

  r1 = r1 + 1 r2 = r2 - 1

  counter = r1 counter = r2

- Initially, let counter = 5.

1. P: r1 = counter
2. P: r1 = r1 + 1
3. C: r2 = counter
4. C: r2 = r2 – 1
5. P: counter = r1
6. C: counter = r2



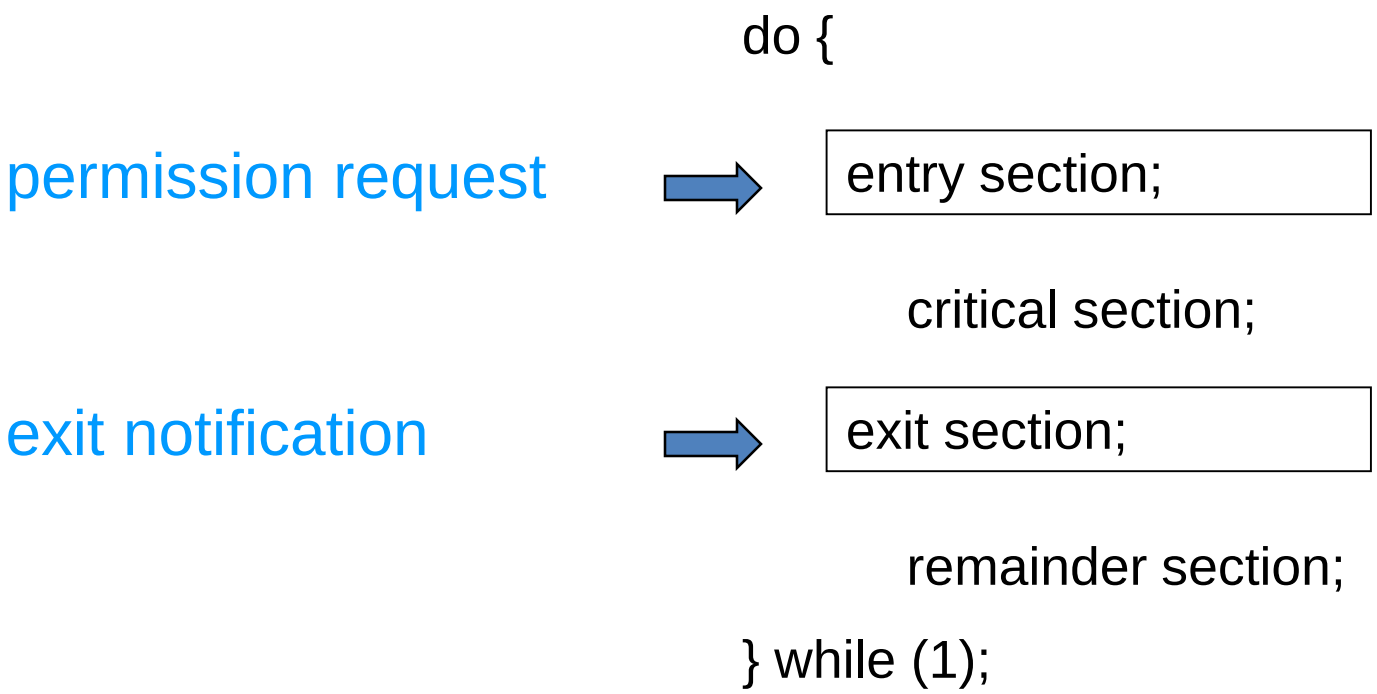
A Race Condition!

Process Synchronization

- A Race Condition:
  - A situation where the outcome of the execution depends on the particular order of process scheduling.
- The Critical-Section Problem:
  - Design a protocol that processes can use to cooperate.
    - Each process has a segment of code, called a critical section, whose execution must be mutually exclusive.

Process Synchronization

- A General Structure for the Critical-Section Problem



The Critical-Section Problem

- Three Requirements

1. Mutual Exclusion
   a. Only one process can be in its critical section.
2. Progress
   a. Only processes not in their remainder section can decide which will enter its critical section.
   b. The selection cannot be postponed indefinitely.
3. Bounded Waiting
   a. A waiting process only waits for a bounded number of processes to enter their critical sections.

Synchronizing Threads

- The threading module provided with Python includes a simple-to-implement locking mechanism that will allow you to synchronize threads.

- A new lock is created by calling the *Lock()* method, which returns the new lock.

- The *acquire(blocking)* method of the new lock object would be used to force threads to run synchronously.

- The optional *blocking* parameter enables you to control whether the thread will wait to acquire the lock.

Synchronizing Threads

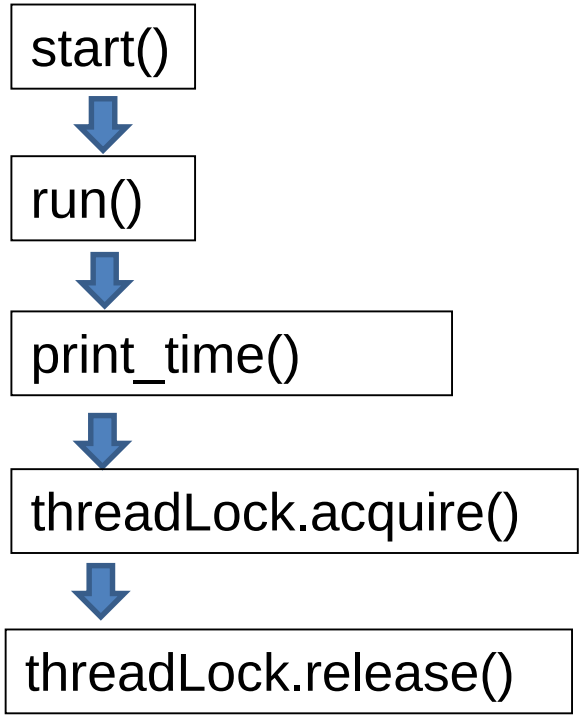
- If *blocking* is set to 0, the thread will return immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired.

- If blocking is set to 1, the thread will block and wait for the lock to be released.

- The *release()* method of the new lock object would be used to release the lock when it is no longer required.

```python
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

MPLE




```
Starting Thread-1Starting Thread-2

Thread-1: Thu Aug 21 09:50:43 2014
Thread-1: Thu Aug 21 09:50:44 2014
Thread-1: Thu Aug 21 09:50:45 2014
Thread-2: Thu Aug 21 09:50:47 2014
Thread-2: Thu Aug 21 09:50:49 2014
Thread-2: Thu Aug 21 09:50:51 2014
Exiting Main Thread
```

Asynchronous Request

- In the previous chapter, we used TCP Server which process requests synchronously.
- That means each request must be completed before the next request can be started.
- This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process.

Asynchronous Request handling Server code

```
# async.py

import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024))
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data))
        self.request.sendall(response)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(bytes(message))
        response = str(sock.recv(1024))
        print("Received: {}".format(response))
    finally:
        sock.close()
```

```
if __name__ == "__main__":
    # port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address

    # start a thread with the server.
    # the thread will then start one more thread for each request.
    server_thread = threading.Thread(target=server.serve_forever)

    # exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print("Server loop running in thread:", server_thread.name)

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()
```

Asynchronous Request handling Server

- The **ThreadingMixIn** class defines an attribute **daemon_threads**, which indicates whether or not the server should wait for thread termination.
- We should set the flag explicitly if we would like threads to behave autonomously.
  - The default value is **False**, meaning that Python will not exit until all threads created by **ThreadingMixIn** have exited.
- In the code, we set it **True**, which means Python will exit the server thread when the main thread terminates not waiting for other threads' exit.

- Forking and threading **TCPServer** can be created using the **ForkingMixIn** and **ThreadingMixIn** mix-in classes. For instance, a threading TCP server class is created as follows:

```
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
```

- The mix-in class must come first, since it overrides a method defined in TCPServer. Setting the various attributes also change the behavior of the underlying server mechanism.
- To implement a service, we must derive a class from **BaseRequestHandler** and redefine its **handle()** method:

```
class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024))
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data))
        self.request.sendall(response)
```

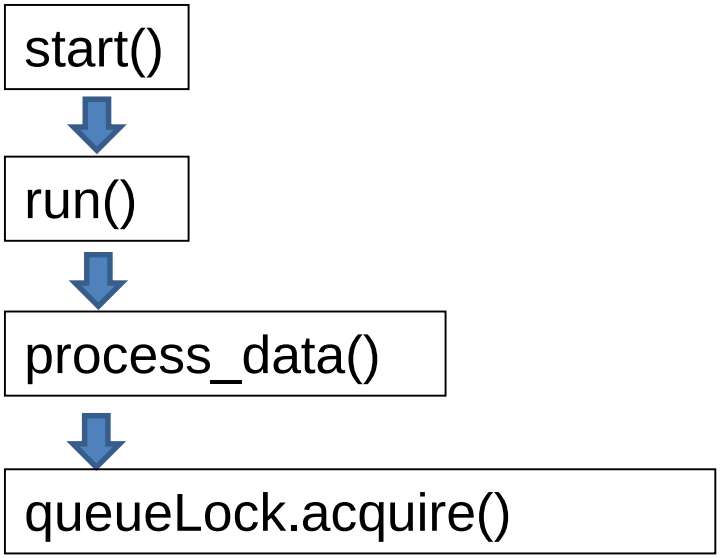
```
$ python async.py
('Server loop running in thread:', 'Thread-1')
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

Multithreaded Priority Queue

- The *Queue* module allows you to create a new queue object that can hold a specific number of items.
- There are following methods to control the Queue:
  - **get():** removes and returns an item from the queue.
  - **put():** adds item to a queue.
  - **qsize() :** returns the number of items that are currently in the queue.
  - **empty():** returns True if queue is empty; otherwise, False.
  - **full():** returns True if queue is full; otherwise, False.

Example

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Starting " + self.name
        process_data(self.name, self.q)
        print "Exiting " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# Fill the queue
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# Wait for queue to empty
```

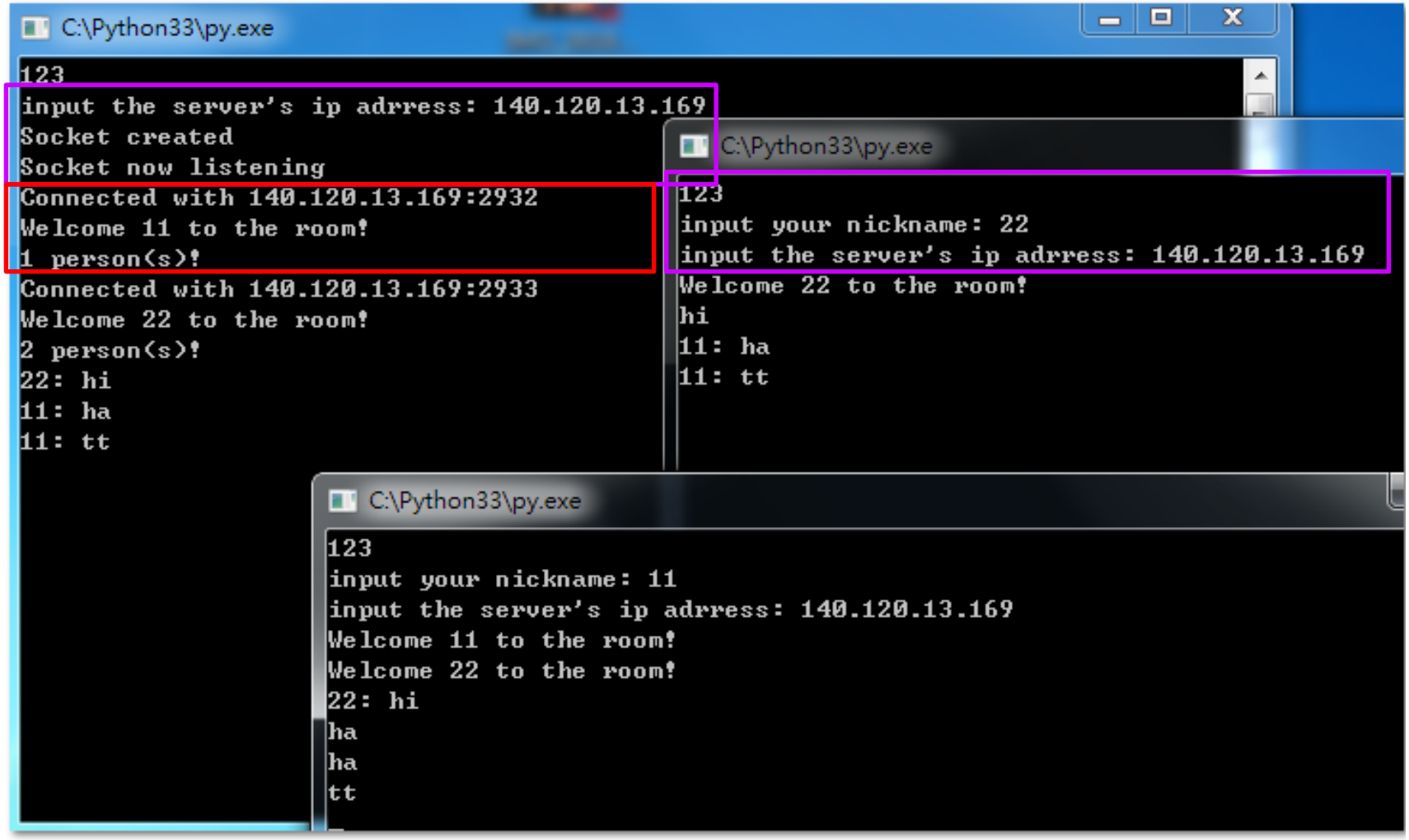



```
Starting Thread-1Starting Thread-2Starting Thread-3


Thread-1 processing OneThread-2 processing TwoThread-3 processing Three


Thread-1 processing FourThread-2 processing Five

Exiting Thread-3
 Exiting Thread-2Exiting Thread-1
```

Example

```
# Wait for queue to empty
while not workQueue.empty():
    pass

# Notify threads it's time to exit
exitFlag = 1

# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

Talking Room (Console)



Server (1)

```
import socket
import sys
import threading

con = threading.Condition()
HOST = raw_input("input the server's ip adrress: ")     # Symbolic name meaning all available interfaces
PORT = 8888     # Arbitrary non-privileged port
data = ''

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket created'
s.bind((HOST, PORT))
s.listen(10)
print 'Socket now listening'

#Function for handling connections. This will be used to create threads
def clientThreadIn(conn, nick):
    global data
#infinite loop so that function do not terminate and thread do not end.
    while True:
    #Receiving from client
        try:
            temp = conn.recv(1024)
            if not temp:
                conn.close()
                return
            NotifyAll(temp)
            print data
        except:
            NotifyAll(nick + " leaves the room!")
            print data
            return

        #came out of loop
```

Server (2)

```
def NotifyAll(sss):
    global data
    if con.acquire():
        data = sss
        con.notifyAll()
        con.release()

def ClientThreadOut(conn, nick):
    global data
    while True:
        if con.acquire():
            con.wait()
            if data:
                try:
                    conn.send(data)
                    con.release()
                except:
                    con.release()
                    return


while 1:
    #wait to accept a connection - blocking call
    conn, addr = s.accept()
    print 'Connected with ' + addr[0] + ':' + str(addr[1])
    nick = conn.recv(1024)
     #send only takes string
    #start new thread takes 1st argument as a function name to be run, second is the tuple of arguments to th
    NotifyAll('Welcome ' + nick + ' to the room!')
    print data
    print str((threading.activeCount() + 1) / 2) + ' person(s)!'
    conn.send(data)
    threading.Thread(target = clientThreadIn , args = (conn, nick)).start()
    threading.Thread(target = ClientThreadOut , args = (conn, nick)).start()

s.close()
raw_input( )
```

Multiple threads

threading.Condition

- This is a synchronization mechanism where a thread waits for a specific condition and another thread signals that this condition has happened.

- Once the condition happened, the thread acquires the lock to get exclusive access to the shared resource.

Client (1)

```
import socket
import threading


inString = ''
outString = ''
nick = ''

def DealOut(s):
    global nick, outString
    while True:
        outString = raw_input()
        outString = nick + ': ' + outString
        s.send(outString)

def DealIn(s):
    global inString
    while True:
        try:
            inString = s.recv(1024)
            if not inString:
                break
            if outString != inString:
                print inString
        except:
            break
```

Client (2)

```
nick = raw_input("input your nickname: ")
ip = raw_input("input the server's ip adrress: ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((ip, 8888))
sock.send(nick)
```

```
thin = threading.Thread(target = DealIn, args = (sock,))
thin.start()
thout = threading.Thread(target = DealOut, args = (sock,))
thout.start()
```

```
#sock.close()
raw_input( )
```

Multiple threads

Tic Tac Toe Game (Console)

- We uses **two for loops** to go through a list variable called map.
- This variable is a two dimensional **list** which will hold the info about what's in each position.

```
def print_board():
    for i in range(0,3):
        for j in range(0,3):
            print map[2-i][j],
            if j != 2:
                print "|",
        print ""

turn = "X"
map = [[" "," "," "],
       [" "," "," "],
       [" "," "," "]]
done = False
```

```
X 's turn

Please select position by typing in a number between 1 and 9, see below for whic
h number that is which position...
7|8|9
4|5|6
1|2|3

Select: 5
  |   |
  | X |
  |   |
O 's turn

Please select position by typing in a number between 1 and 9, see below for whic
h number that is which position...
7|8|9
4|5|6
1|2|3

Select: 4
  |   |
O | X |
  |   |
X 's turn

Please select position by typing in a number between 1 and 9, see below for whic
h number that is which position...
7|8|9
4|5|6
1|2|3
```

Tic Tac Toe Game (Console)

1. We check if all 3 squares in all **horizontal** and **vertical** lines are the same and not " ".
   - It won't think an completely empty line is a line with 3 in a row.
2. Then it checks the two **diagonally** lines in the same way.
3. If at least one of these 8 lines are a winning line we will print out turn, "won!!!" and also return the value True. The turn variable will hold which player who's in turn so the message will be either "X won!!!" or "O won!!!".

```
def check_done():
    for i in range(0,3):
        if map[i][0] == map[i][1] == map[i][2] != " " \
        or map[0][i] == map[1][i] == map[2][i] != " ":
            print turn, "won!!!"
            return True

    if map[0][0] == map[1][1] == map[2][2] != " " \
    or map[0][2] == map[1][1] == map[2][0] != " ":
        print turn, "won!!!"
        return True

    if " " not in map[0] and " " not in map[1] and " " not in map[2]:
        print "Draw"
        return True


    return False
```

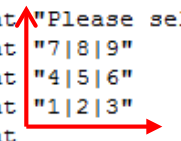
Tic Tac Toe Game (Console)

- We store the current users "name" at the right position (with the X and Y values), set move to True, check if we're done and stores that in done.
- If the game isn't over, change who's next to move and then we have two lines to print an error message if the try block failed in some way.

```
map[Y][X] = turn
                    moved = True
                    done = check_done()

                    if done == False:
                        if turn == "X":
                            turn = "O"
                        else:
                            turn = "X"


        except:
            print "You need to add a numeric value"
```

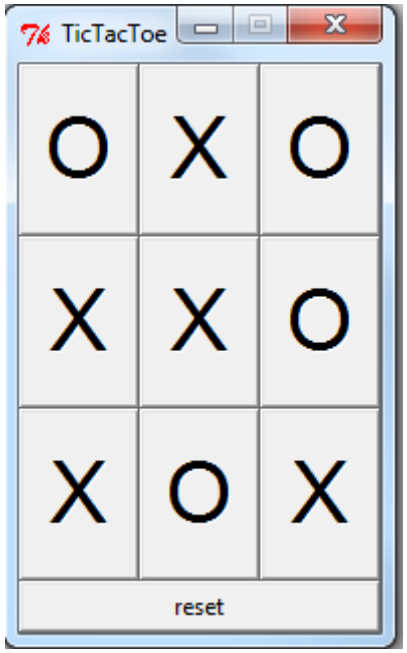
Tic Tac Toe Game (Console)

```
while done != True:
    print_board()

    print turn, "'s turn"
    print

    moved = False
    while moved != True:
        print "Please select position by typing in a number between 1 and 9, see below for which number that is which position..."
        print "7|8|9"
        print "4|5|6"
        print "1|2|3"
        print
        try:
            pos = input("Select: ")
            if pos <=9 and pos >=1:
                Y = pos/3
                X = pos%3
                if X != 0:
                    X -=1
                else:
                     X = 2
                     Y -=1

                if map[Y][X] == " ":
                    map[Y][X] = turn
                    moved = True
                    done = check_done()

                    if done == False:
                        if turn == "X":
                            turn = "O"
                        else:
                            turn = "X"


        except:
            print "You need to add a numeric value"
```



Tic Tac Toe Game (GUI)

```
from Tkinter import Tk, Button
from tkFont import Font
from copy import deepcopy

class Board:

  def __init__(self,other=None):
    self.player = 'X'
    self.opponent = 'O'
    self.empty = '.'
    self.size = 3
    self.fields = {}
    for y in range(self.size):
      for x in range(self.size):
        self.fields[x,y] = self.empty
    # copy constructor
    if other:
      self.__dict__ = deepcopy(other.__dict__)

  def move(self,x,y):
    board = Board(self)
    board.fields[x,y] = board.player
    (board.player,board.opponent) = (board.opponent,board.player)
    return board
```



Tic Tac Toe Game (GUI)

```
def __minimax(self, player):
  if self.won():
    if player:
      return (-1,None)
    else:
      return (+1,None)
  elif self.tied():
    return (0,None)
  elif player:
    best = (-2,None)
    for x,y in self.fields:
      if self.fields[x,y]==self.empty:
        value = self.move(x,y).__minimax(not player)[0]
        if value>best[0]:
          best = (value,(x,y))
    return best
  else:
    best = (+2,None)
    for x,y in self.fields:
      if self.fields[x,y]==self.empty:
        value = self.move(x,y).__minimax(not player)[0]
        if value<best[0]:
          best = (value,(x,y))
    return best
```

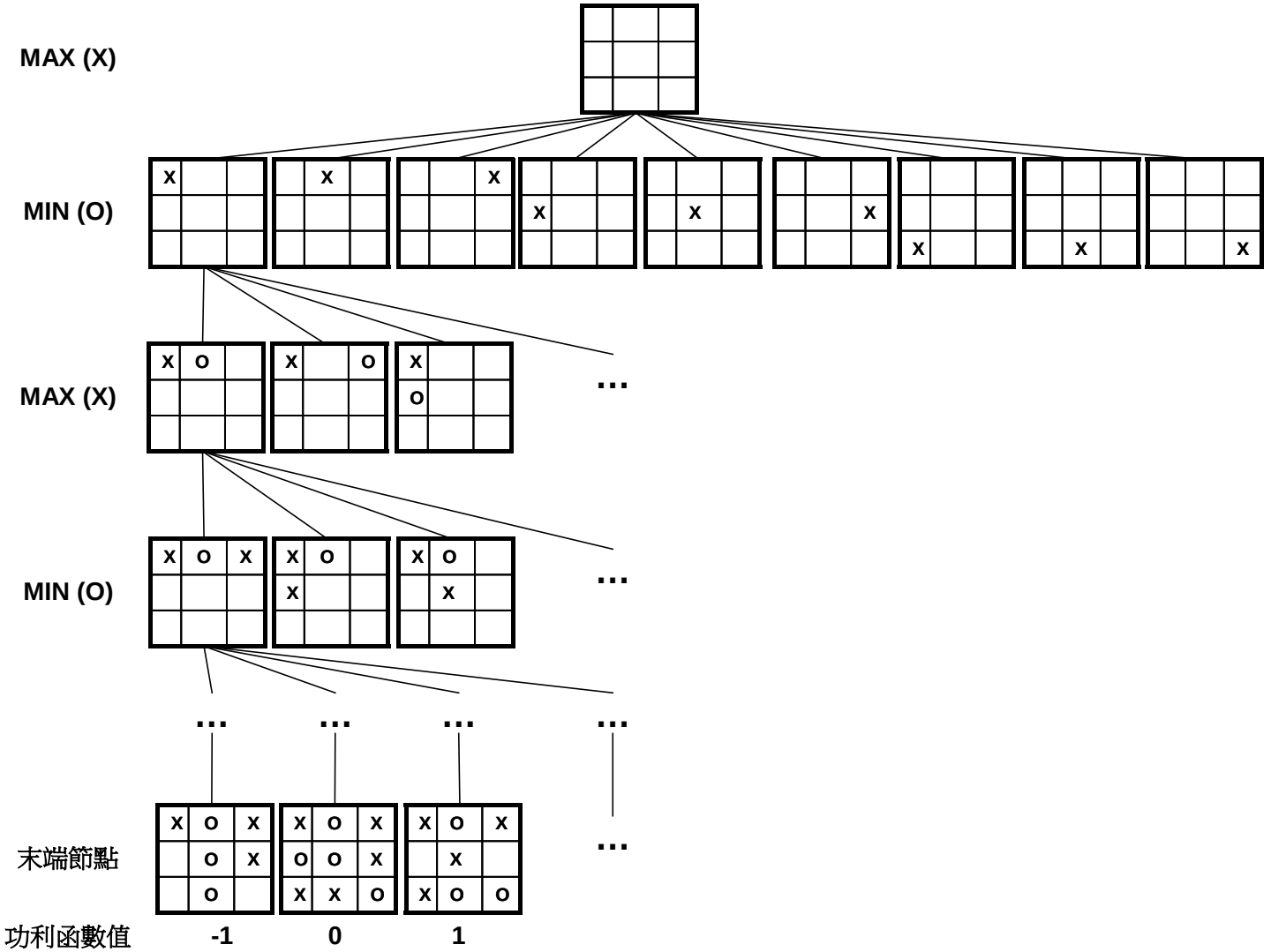
Minimax Game Decision

- 假設一個遊戲中有兩位參與者，為方便起見我們稱他們為MAX和MIN。
- 由MAX先移動，且兩人輪流移動直到遊戲結束。
- 贏的人會得到獎賞(或輸的一方得到處罰)。

一個遊戲若具備以下條件，則可以被轉換成為一個搜尋的問題：

- 初始狀態(initial state)，包含初始位置及哪一方先移動等。
- 運算元集合(a set of operators)，指一個玩家在遊戲中可作的動作。
- 結束測試(terminal test)，決定何時遊戲會結束。遊戲結束的地方稱為結束狀態(terminal states)。
- 功利函數(utility function)，或稱回報函數(payoff function)，對於遊戲的結果給予一數值。

井字遊戲的搜尋樹(search tree)

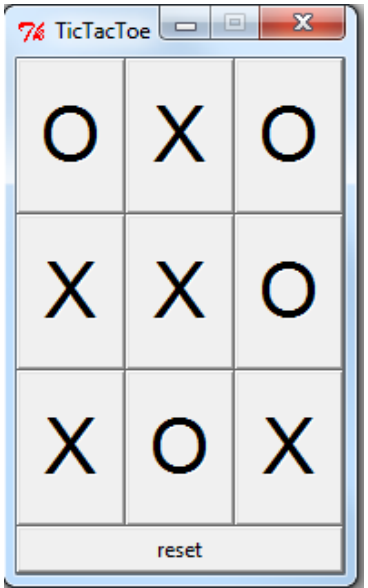


Tic Tac Toe Game (GUI)

```
  def won(self):
    # horizontal
    for y in range(self.size):
      winning = []
      for x in range(self.size):
        if self.fields[x,y] == self.opponent:
          winning.append((x,y))
      if len(winning) == self.size:
        return winning
    # vertical
    for x in range(self.size):
      winning = []
      for y in range(self.size):
        if self.fields[x,y] == self.opponent:
          winning.append((x,y))
      if len(winning) == self.size:
        return winning
    # diagonal
    winning = []
    for y in range(self.size):
      x = y
      if self.fields[x,y] == self.opponent:
        winning.append((x,y))
    if len(winning) == self.size:
      return winning
    # other diagonal
    winning = []
    for y in range(self.size):
      x = self.size-1-y
      if self.fields[x,y] == self.opponent:
        winning.append((x,y))
    if len(winning) == self.size:
      return winning
    # default
    return None
```

Tic Tac Toe Game (GUI)

```
class GUI:

  def __init__(self):
    self.app = Tk()
    self.app.title('TicTacToe')
    self.app.resizable(width=False, height=False)
    self.board = Board()
    self.font = Font(family="Helvetica", size=32)
    self.buttons = {}
    for x,y in self.board.fields:
      handler = lambda x=x,y=y: self.move(x,y)
      button = Button(self.app, command=handler, font=self.font, width=2, height=1)
      button.grid(row=y, column=x)
      self.buttons[x,y] = button
    handler = lambda: self.reset()
    button = Button(self.app, text='reset', command=handler)
    button.grid(row=self.board.size+1, column=0, columnspan=self.board.size, sticky="WE")
    self.update()

  def reset(self):
    self.board = Board()
    self.update()

  def move(self,x,y):
    self.app.config(cursor="watch")
    self.app.update()
    self.board = self.board.move(x,y)
    self.update()
    move = self.board.best()
    if move:
      self.board = self.board.move(*move)
      self.update()
    self.app.config(cursor="")
```



Tic Tac Toe Game (GUI)

```
def update(self):
  for (x,y) in self.board.fields:
    text = self.board.fields[x,y]
    self.buttons[x,y]['text'] = text
    self.buttons[x,y]['disabledforeground'] = 'black'
    if text==self.board.empty:
      self.buttons[x,y]['state'] = 'normal'
    else:
      self.buttons[x,y]['state'] = 'disabled'
  winning = self.board.won()
  if winning:
    for x,y in winning:
      self.buttons[x,y]['disabledforeground'] = 'red'
    for x,y in self.buttons:
      self.buttons[x,y]['state'] = 'disabled'
  for (x,y) in self.board.fields:
    self.buttons[x,y].update()

def mainloop(self):
  self.app.mainloop()
```