English Edition

6. Inter Process Communication (1)

6. IPC
6.1 Introduction

Synchronization with Python (1)

- 100 Threads manipulate a variable
- Result of calculation should be 0
- Problem: critical section when accessing the global variable

```
$ ./test.py
Result: 200
$ ./test.py
Result: 800
$ ./test.py
Result: 0
$ ./test.py
Result: 100
$ ./test.py
Result: 0
```

```
from threading import Thread

class testthread(Thread):
   def __init__ (self):
      Thread.__init__(self)
   def run(self):
      global globalcount
      # start of critical section
      for j in range(0,99999):
         globalcount += 100
         globalcount -= 100
      # end of critical section

globalcount=0            # glob. variable
threads = []
for i in range(0,100):   # start threads
  t = testthread()
  threads.append(t)
  t.start()
for t in threads: t.join()   # clean-up

print "Result:",globalcount
```


Addendum to chapter 5: Synchronization with Python

Synchronization with Python (2)

1st solution: **Lock**

- acquire: lock
- release: unlock

```
$ ./lock.py
Result: 0
$ ./lock.py
Result: 0
$ ./lock.py
Result: 0
$ ./lock.py
Result: 0

$ time ./test.py
real    0m5.560s
$ time ./lock.py
real    0m5.560s
```

(no speed difference)

```
from threading import Thread, Lock

class testthread(Thread):
   def __init__ (self):
      Thread.__init__(self)
   def run(self):
      global globalcount
      # start critical section
      mylock.acquire()
      for j in range(0,99999):
         globalcount += 100
         globalcount -= 100
      mylock.release()
      # end of critical section

globalcount=0            # glob. variable
threads = []
mylock = Lock()          # global lock
for i in range(0,100):   # start threads
  t = testthread()
  threads.append(t)
  t.start()
for t in threads: t.join()   # clean-up

print "Result:",globalcount
```

Synchronization with Python (3)

- **Lock**
  - acquire can also be used in a non-blocking mode: **mylock.acquire(0)**
    - when the lock is available, **acquire()** returns **True** (the calling thread now holds the lock)
    - when another process already holds the lock, **acquire(0)** will not wait, but immediately return **False**.
- **RLock** (Reentrant **Lock**)
  - like **Lock**, but can be used recursively
  - lock must be **release()**d as often as it was **acquire()**d

Synchronization with Python (4)

2nd solution: **BoundedSemaphore**

- acquire: count down
- release: count up

```
$ ./sem.py
Result: 0
$ ./sem.py
Result: 0
$ ./sem.py
Result: 0
$ ./sem.py
Result: 0
```

```
from threading import Thread, BoundedSemaphore

class testthread(Thread):
   def __init__ (self):
      Thread.__init__(self)
   def run(self):
      global globalcount
      # start critical section
      mysem.acquire()
      for j in range(0,99999):
         globalcount += 100
         globalcount -= 100
      mysem.release()
      # end of critical section

globalcount=0               # glob. variable
threads = []
mysem = BoundedSemaphore(1) # init: 1
for i in range(0,100):      # start threads
  t = testthread()
  threads.append(t)
  t.start()
for t in threads: t.join()  # clean-up

print "Result:",globalcount
```


Synchronization with Python (5)

- **BoundedSemaphore**
  - for granting to access to several (identical) resources: initialize with bigger counter value
  - **acquire** = **Wait** operation
    - with argument 0: non-blocking
  - **release** = **Signal** operation

Synchronization with Python (6)

- **Condition:** condition variables
  - cf. Java / Monitor
  - condition variable is always protected by a lock
  - functions

cv.acquire ()	acquire corresponding lock (or block)
cv.release ()	release corresponding lock
cv.wait ()	release lock and block until signal arrives
cv.notify ()	wake up one of the threads waiting for **cv**
cv.notifyAll ()	wake up all threads waiting for **cv**

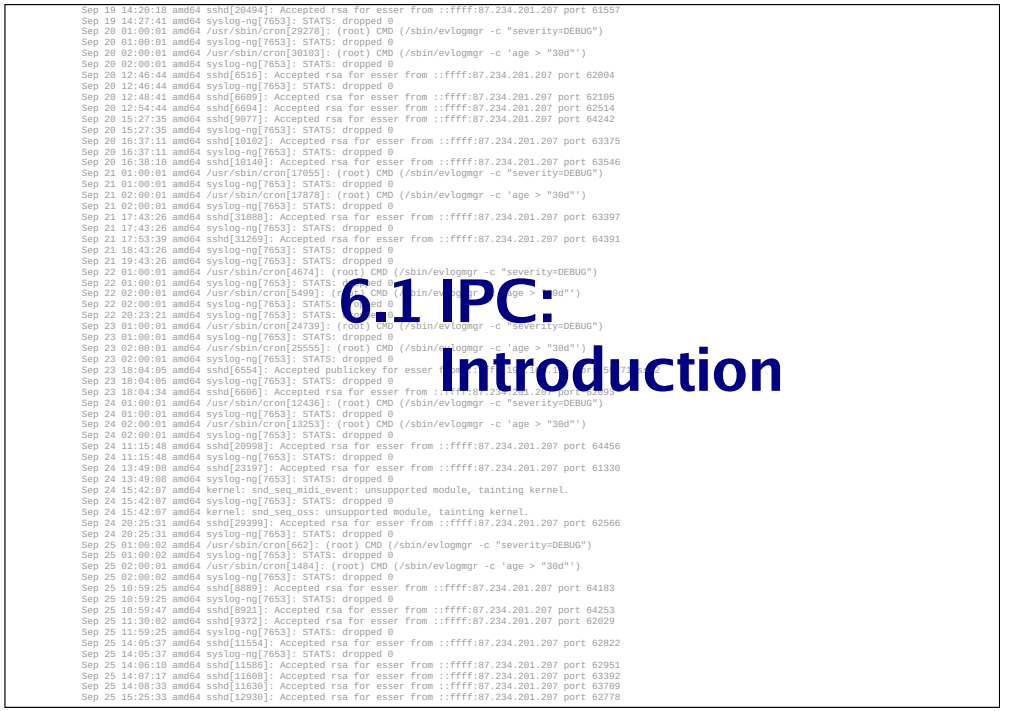
Synchronization with Python (7)

Condition: Producer Consumer Problem

```
from threading import Thread, Condition
cv = 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()
```

Synchronization with Python (8)

Event: corresponds to „manual events“ (Windows)

```
from threading import Thread, Event
ev = Event()

ev.set()                # set event
ev.clear()              # reset/clear event
ev.isSet()              # query event status
ev.wait()               # block until status changes to set
```

longer description in the Python documentation:
http://docs.python.org/lib/module-threading.html

Inter Process Communication

- message exchange between several processes or threads
- connection through communication system
- sender and receiver
- necessary when there is no shared memory

- yet another synchronization method

IPC Characterization (1)

- communication model:
  - point-to-point communication
  - publish-subscribe communication
  - broadcast communication
- transmission direction:
  - simplex / uni-directional
  - duplex / bi-directional
- Synchronicity
  - synchronous / blocking
  - asynchronous / non-blocking

IPC Characterization (2)

- message type
  - messages
  - stream
- platform (in-)dependence
- portability
- locality
  - system-bound (local) or
  - allows network communication (across machine borders)

Communication Model

- point-to-point
  - exactly one sender and one receiver
- broadcast
  - one sender and several receivers
- publish & subscribe
  - peer-to-peer communication
  - publisher: threads which change some system state
  - subscriber: threads which are notified when such changes occur

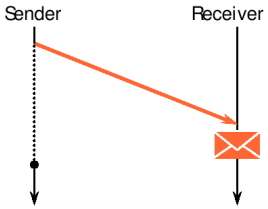
Transmission Direction

- simplex message from sender to receiver
  - beginning: sender posts message,
    end: delivery at the receiver
  - no reply expected
- duplex message
  - beginning: sender posts message,
    end: acknowledgement / receipt is delivered at the sender
  - in between: request is processed by the receiver
  - in the simplest case: two messages (there and back)
  - detect missing receipts via timeouts (negative receipts)

Synchronicity

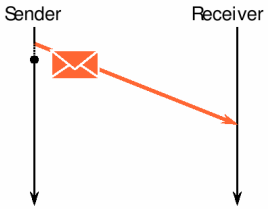
Synchronous / blocking communication

- sender blocks until message arrives (at receiver)
- needs almost no buffer capacity
- restricted parallelism



Asynchronous / non-blocking communication

- sender only blocks until message has been copied into the buffer
- needs larger buffer capacity
- sender can send several messages in succession



Picture: (c)Peter Sturm, Uni Trier, „Episodes on Operating Systems"
http://strathisla.uni-trier.de/lectblog/wp-content/Syssoft1_13_Communication.pdf

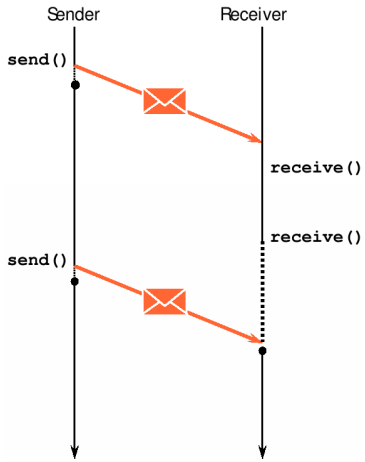
Four Elementary Communication Types

combining two transmission directions and the choice of synchronous vs. asynchronous leads to

	asynchronous	synchronous
simplex: message	-1- **asynchronous message**	-2- **synchronous message**
duplex: request	-3- **asynchronous request**	-4- **synchronous request**

-1- asynchronous message

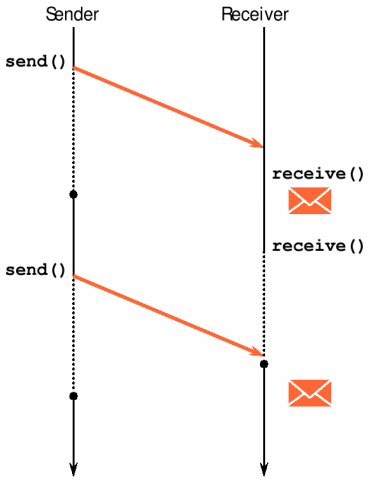
- sender and receiver are uncoupled -> parallelism
- **UDP:** User Datagram Protocol (IP-based)
- signals: Linux software interrupts



graphics on the following four slides:
(c) Peter Sturm, Uni Trier, „Episodes on Operating Systems",
http://strathisla.uni-trier.de/lectblog/wp-content/Syssoft1_13_Communication.pdf

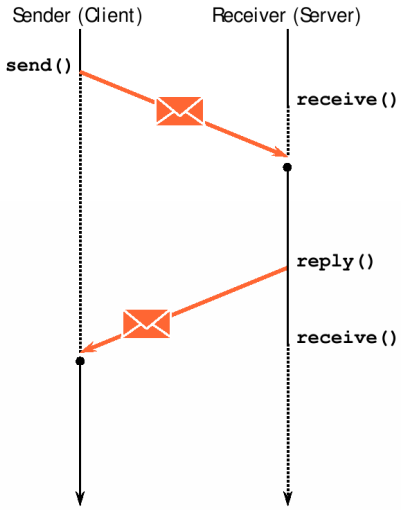
-2- synchronous message

- limited parallelism: sender must wait for a receipt from the receiver
- no buffers needed
- sender knows that message arrived at destination



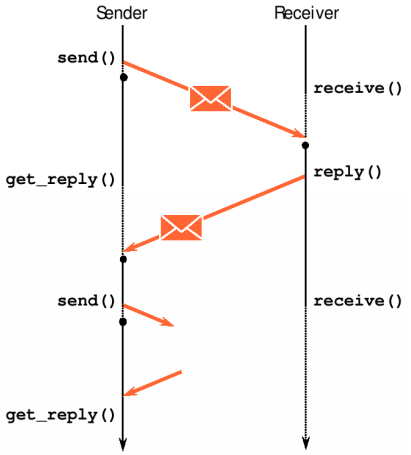
-4- synchronous request

- **RPC:** Remote Procedure Call
- sender waits until he has received a reply to his request [not for a receipt]
- e.g. databases



-3- asynchronous request

- client sends request, but does not wait for the result
- at a later point in time he accepts the server's reply
- thus: fewer blocking times, more parallelism



Sockets (1)

- sockets are a network communication tool that offers a communication channel with the following properties:
  - they allow IPC between processes
  - either local or over the network
  - can work with several (low-level) protocols
  - platform-independent
- bi-directional communication

Sockets (2)

- addressing, e.g.
  - local (AF_UNIX): path name
  - network address (AF_INET): host + port, (local via AF_INET: host = localhost, 127.0.0.1)
- reliability:
  - reliable, connection oriented (e.g. **TCP**): error-free, no packet loss, no duplicates, message order is preserved
  - unreliable, connection-less (e.g. **UDP** / datagram)

Sockets (3)

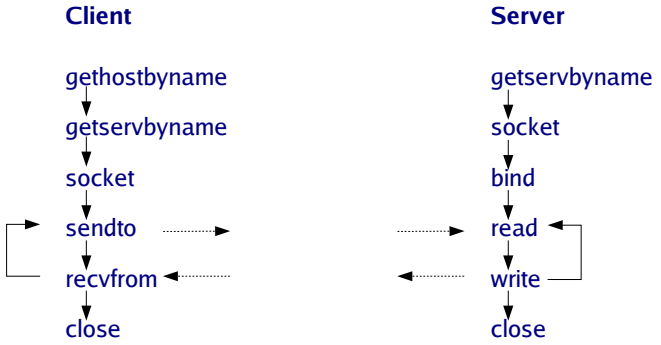
- communication types:
  - message-oriented: **recvmsg**() or **sendmsg**()
  - stream-oriented: **sendto**(), **recfrom**(), **read**(), **write**()
  - typically with an 8 KByte buffer
  - sending of larger data will block until the other side has read the data
  - optionally choose non-blocking behavior
  - Unix/Linux maps sockets to file descriptors (accessing them is similar as with regular files)

Sockets (4)

- Application
  - possible problems:
    - port busy --> **lsof** („list open files“ command)
    - port is still busy --> using „linger“ option may help
    - message boundaries might not remain intact:
      e.g. send(170 bytes) + send(230 bytes)
      ➞ receive(400 bytes)
  - API functions allow optional choice of non-blocking behavior
  - for experiments: C examples for Client/Server

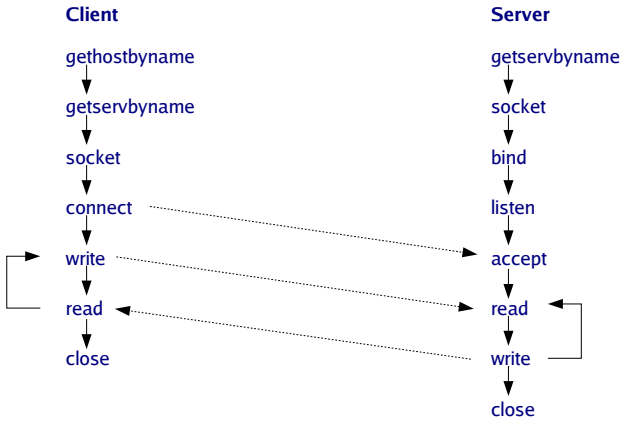
Connectionless Sockets

Connectionless communication via datagrams / UDP



Connection-oriented Sockets

Connection-oriented communication via streams / TCP



Datagram Server

```
#include <unistd.h>        // read(), close()
#include <arpa/inet.h>     // sockaddr_in, INADDR_ANY
#include <sys/socket.h>    // SOCK_DGRAM, socket(), bind()

const short port = 5242;
int         n, sockfd;
char        buf[256];
struct sockaddr_in serv_addr;

int main() {    // UDP_Socket
    if ((sockfd = socket( AF_INET, SOCK_DGRAM, 0 )) < 0)  perror("opening datagram");

    // Create name with wildcards
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons( port );

    if ( bind( sockfd, (sockaddr *)&serv_addr, sizeof(serv_addr) ) != 0 )
        perror( "binding to address" );

    n = read( sockfd, buf, sizeof(buf) ); printf( "Received: %s\n", buf );

    close( sockfd ); return  0;
}
```


Datagram Client

```
#include <unistd.h>        // read(), close()
#include <arpa/inet.h>     // sockaddr_in, AF_INET
#include <sys/socket.h>    // SOCK_DGRAM, socket(), bind()
#include <sys/param.h>     // MAXHOSTNAMELEN
#include <netdb.h>         // gethostbyname()

const short port = 5242;
char        hostname[MAXHOSTNAMELEN+1] = "server";
int         sockfd;
struct sockaddr_in peer_addr;

int main() {    // UDP_Socket
    if ((sockfd = socket( AF_INET, SOCK_DGRAM, 0 )) < 0)  perror("opening datagram");

    struct hostent * hp = gethostbyname( hostname );
    bcopy( hp->h_addr, (char *)&peer_addr.sin_addr, hp->h_length );
    peer_addr.sin_family = AF_INET;
    peer_addr.sin_port = htons( port );

    if ( sendto( sockfd, "Hello World", 11, 0, (sockaddr *)&peer_addr,
        sizeof(peer_addr) ) < 0 )  perror( "sending data" );

    close( sockfd ); return  0;
}
```


Connection-oriented Server

```
#include <unistd.h>        // read(), close()
#include <arpa/inet.h>     // sockaddr_in, INADDR_ANY
#include <sys/socket.h>    // SOCK_DGRAM, socket(), bind()

const short port = 5242, waitqueuelen = 1;
int         n, sockfd, con;
char        buf[256];
struct sockaddr_in serv_addr;

int main() {    // TCP_Socket
    if ((sockfd = socket( AF_INET, SOCK_STREAM, 0 )) < 0)  perror("opening stream");
    serv_addr...;
    if ( bind( sockfd, (sockaddr *)&serv_addr, sizeof(serv_addr) ) != 0 )
        perror( "binding to address" );

    if ( listen( sockfd, waitqueuelen ) != 0 )  perror( "listening to address" );

    if ( ( con = accept( sockfd, (sockaddr *)&peer_addr, sizeof(serv_addr) ) < 0 )
        perror( "accepting client" );

    n = read(con, buf, sizeof(buf)); printf("Received: %s\n", buf); write(con, buf, n);

    close( con ); close( sockfd ); return  0;
}
```


Connection-oriented Client

```
#include <unistd.h>        // read(), close()
#include <arpa/inet.h>     // sockaddr_in, AF_INET
#include <sys/socket.h>    // SOCK_DGRAM, socket(), bind()
#include <sys/param.h>     // MAXHOSTNAMELEN
#include <netdb.h>         // gethostbyname()

const short port = 5242;
char        hostname[MAXHOSTNAMELEN+1] = "server";
int         sockfd;
struct sockaddr_in peer_addr;

int main() {    // TCP_Socket
    if ((sockfd = socket( AF_INET, SOCK_STREAM, 0 )) < 0)  perror("opening stream");
    peer_addr...;
    if ( connect( sockfd, (sochaddr *)&peer_addr, sizeof(peer) ) != 0 )
        perror( "connecting server" );

    write( sockfd, "Hello World", 11 );
    read( sockfd, buf, sizeof(buf)); printf("Received: %s\n", buf);

    close( sockfd ); return  0;
}
```