Byte Data Communication

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)

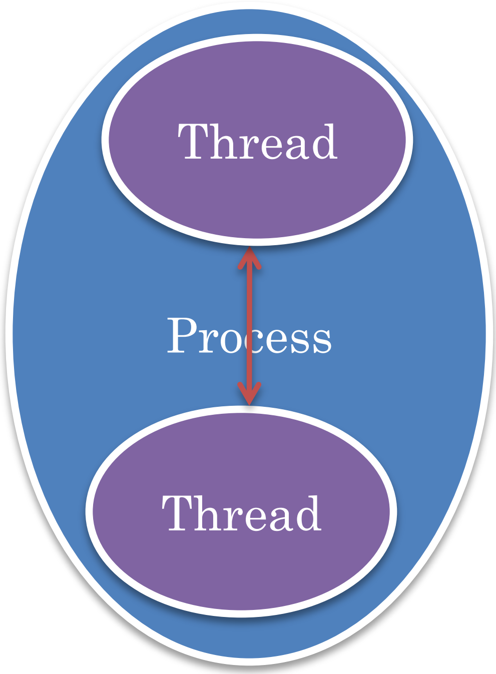
Case Studies

- XINU IPC: Design and Implementation
- Unix Pipes
- Java Sockets
- Java Non-blocking IO

XINU Low-Level Message Passing

Focus is on simplicity of design and use

Location of Communicating Threads



Multiple address spaces not supported in XINU

Also true in Several PC OS's

No notion of a separate process; process = thread

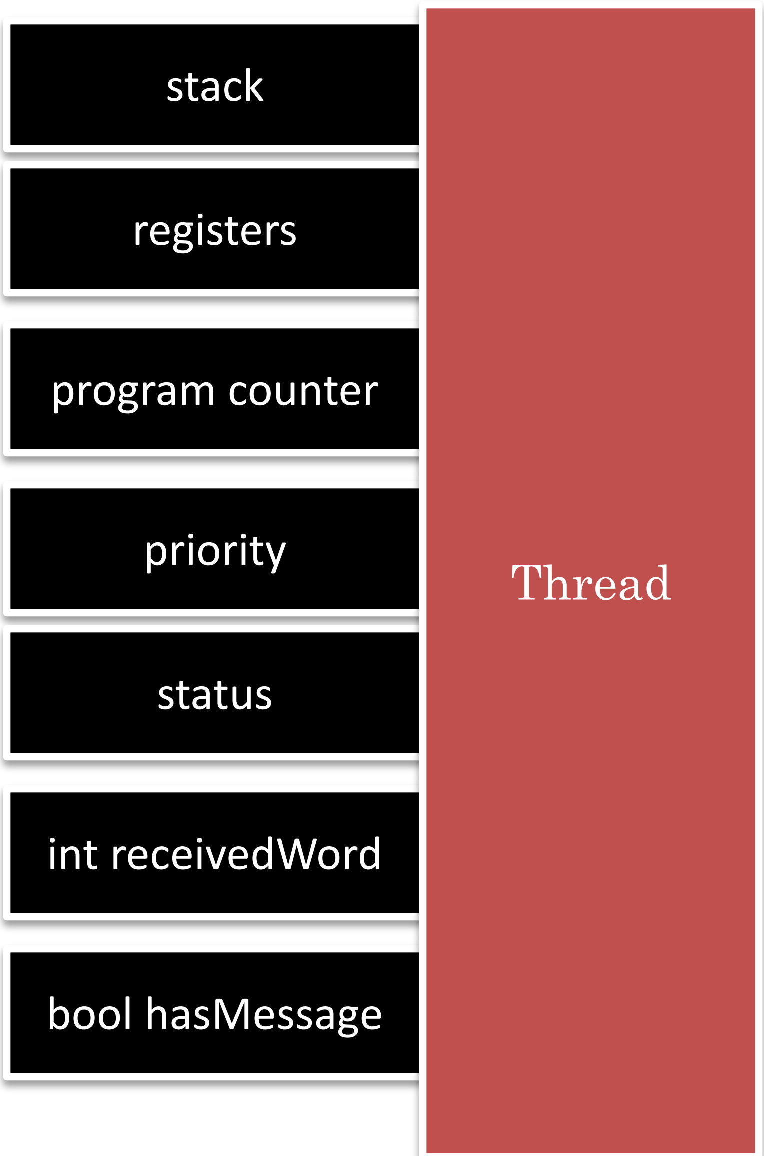
Intra-address communication

OPERATIONS

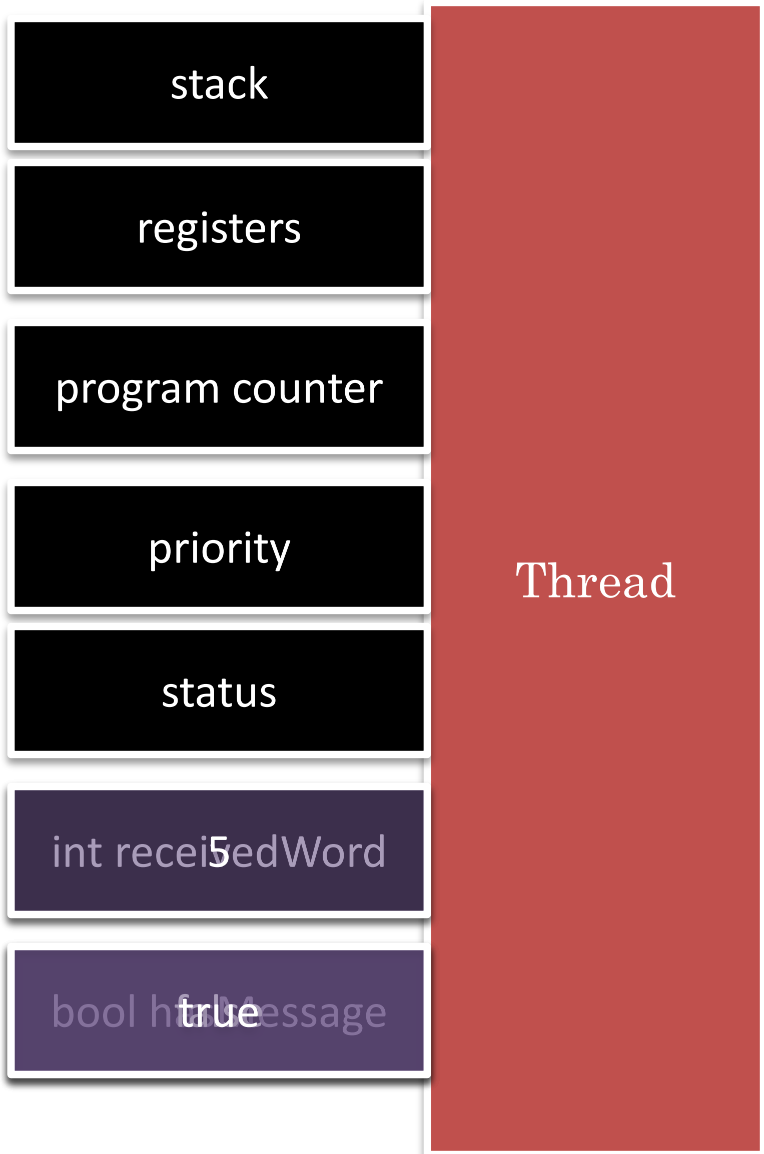
send (<thread_id>, <int expression>)	Non blocking
int receive ()	Synchronous
int recvclr ()	Non blocking, polling, returns either message if it exists, otherwise a special value

Implementation?

Data Structure



SEND OPERATION




```
send (tid, 5 )
```

```
send (tid, 6 )
```

```
send (tid,  intExpression>)

If  (! wordReceived) {
    receivedWord = intExpression;
    hasMessage = true;
    if (waiting(tid)) {
        ready(tid))
}
```

Receive Operation

stack

registers

program counter

priority

READY

int receivedWord

5

bool hasMessage

false

Thread

receive( )	5

```
int receive ()
```

```
If  (!wordReceived) {
     status = RECEIVE;
     blockAndResched(currentPid);
}
```

```
hasMessage= false;
return receivedWord;
```

RECVCLR OPERATION

stack

registers

program counter

priority

CURRENT

status

Int received word

false

bool hasMessage

Thread

receive( )	NO_VAL

int receive ()

If (!word received) {
    return NO_VAL
}

hasMesage= **false**;
return receivedWord;

UNIX Pipes

```
bluetang(65)% man 2 pipe | more
PIPE(2)                    Linux Programmerâs Manual                   PIPE(2)



NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates  a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used to return two
       file  descriptors  referring to the ends of the pipe.  pipefd[0] refers
       to the read end of the pipe.  pipefd[1] refers to the write end of  the
       pipe.   Data  written  to  the write end of the pipe is buffered by the
       kernel until it is read from the read end of  the  pipe.   For  further
       details, see pipe(7).
```

User command

API call

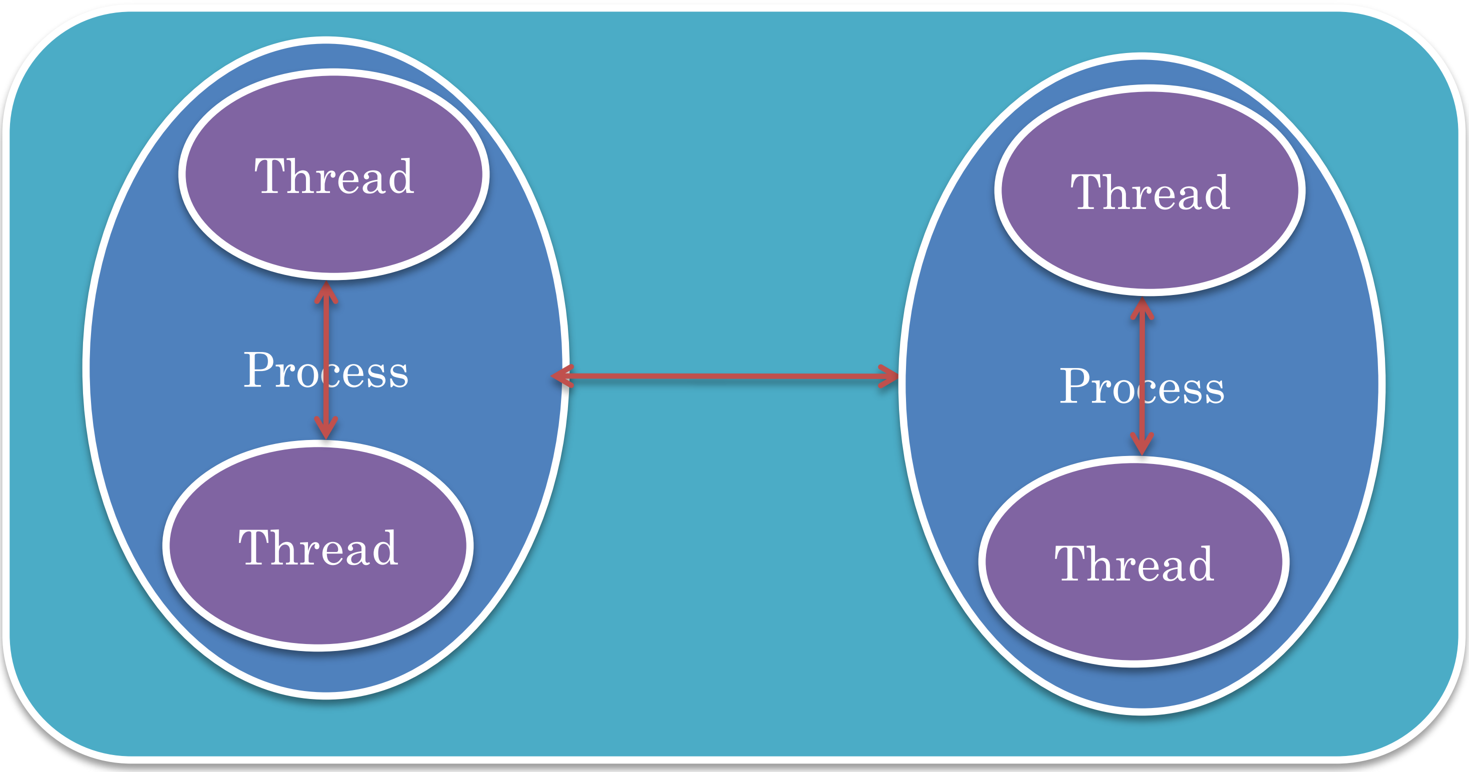
User command exposes part of the functionality of API call

UNIX Pipes

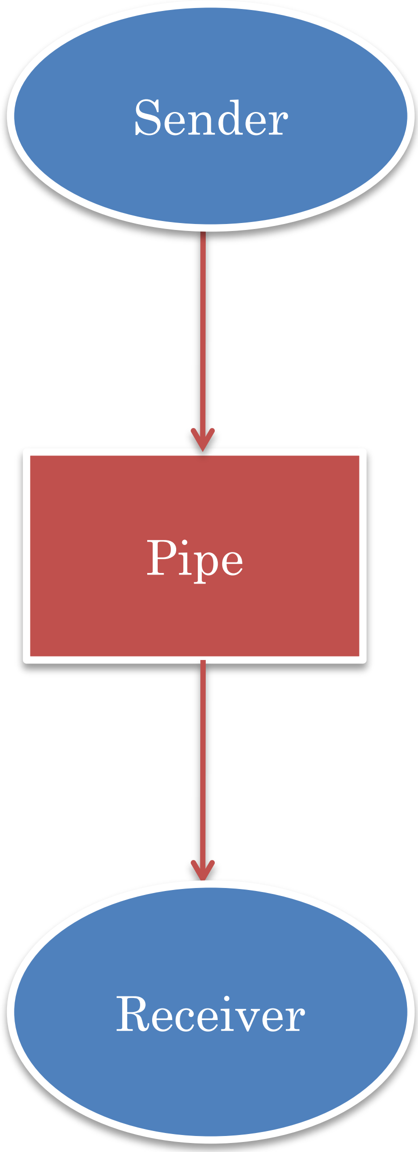
Focus is on Late Binding of Teletype I/O Source and Sink

Stream-based communication

Intra-Computer Message Passing



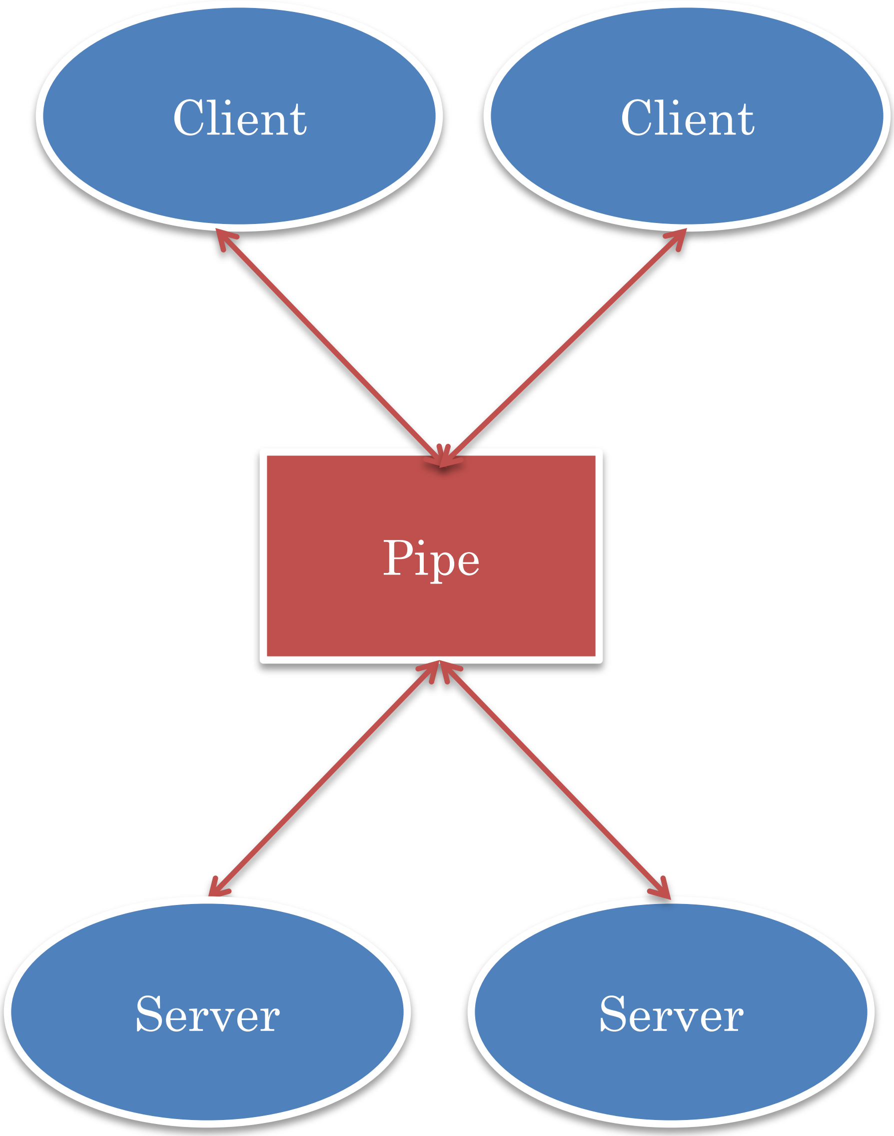
Simplex Pipe?



man 2 pipe | more

Command interpreter connects output of a process to input of another process

Naming?



How do writers and readers name the pipe?

Memory is (usually) not shared by processes

Child process inherits file descriptors (and environment variables) from parent process

File and Environment Table

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR

Copy

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR

Parent Process

create (fork() + exec())

Child Process

Environment

Variable	Value
CLASSPATH	/usr/java/lib;/lib
PATH	/usr/java/bin;/bin

Copy

Variable	Value
CLASSPATH	/usr/java/lib;/lib
PATH	/usr/java/bin;/bin

Sharing Pipe Descriptors

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Copy

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Parent Process

create (fork() + exec())

Child Process

pipe(int[2])

How I/O Redirection?

Copying File Table Entries

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Copy

Index	Resource
0	STD_IN
1	PIPE_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Parent Process

create (Fork + Exec)

Child Process

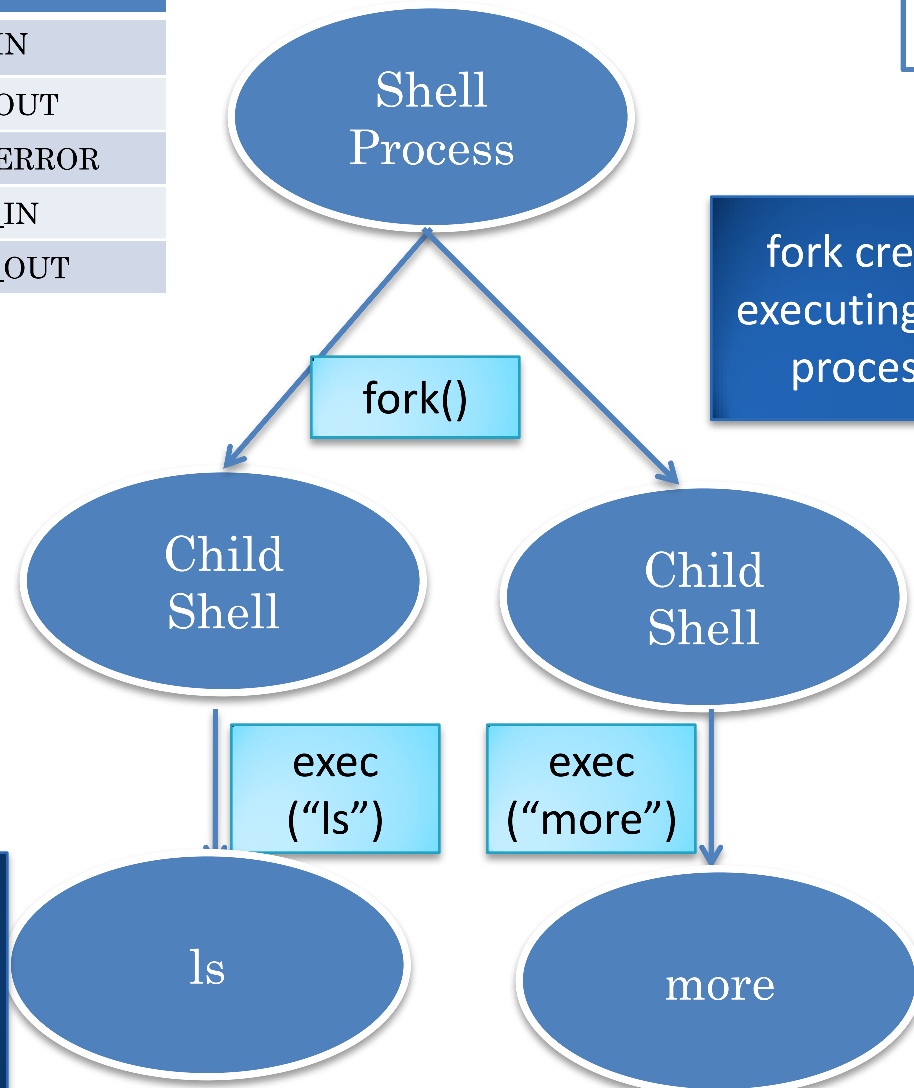
pipe(int[2])

Can copy file table entry to another position (standard input/output) and can close it.

I/O Redirection Details

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

cat file | more



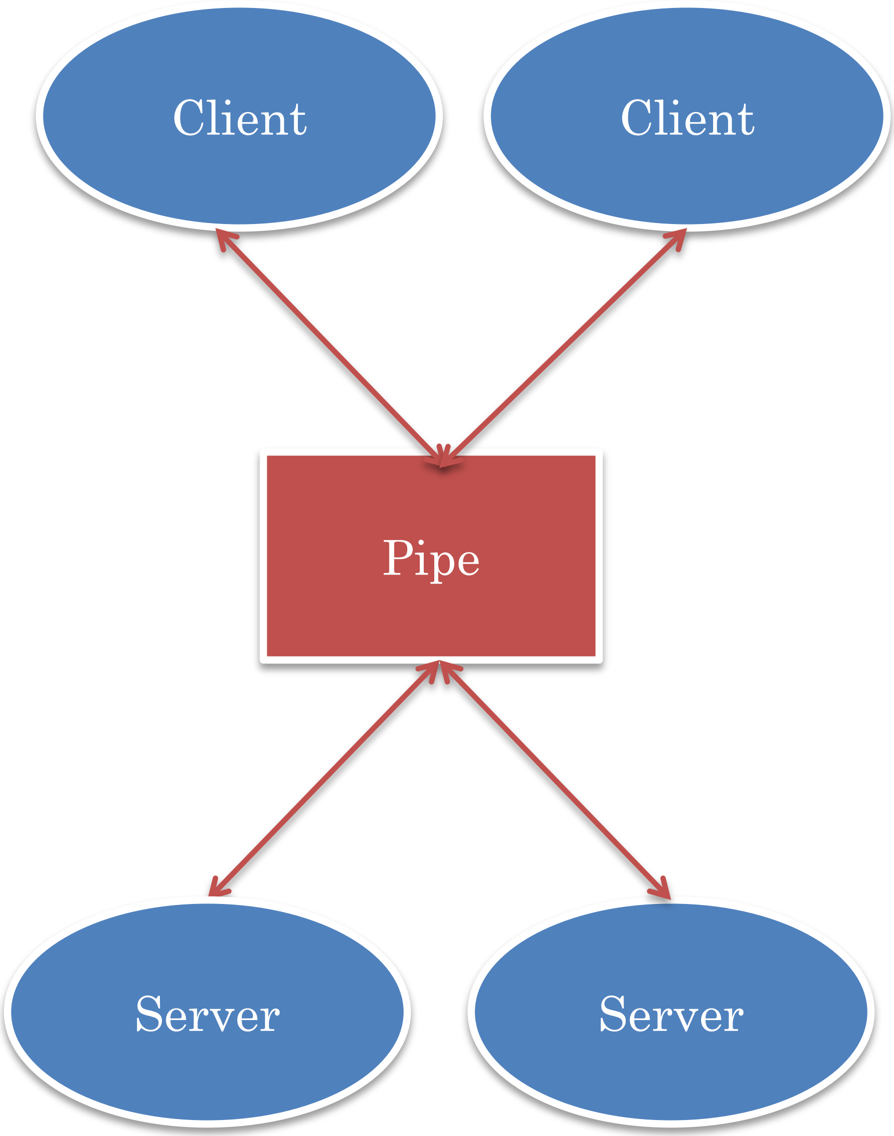
fork creates new process executing same program as process executing fork

Index	Resource
0	STD_IN
1	PIPE_OUT
2	STD_ERROR

Index	Resource
0	PIPE_IN
1	STD_OUT
2	STD_ERROR

exec makes same process execute another program, keeping the file table

Pipe (Review)



How do writers and readers name the pipe?

Memory is (usually) not shared by processes

Child process inherits file descriptors (and environment variables) from parent process

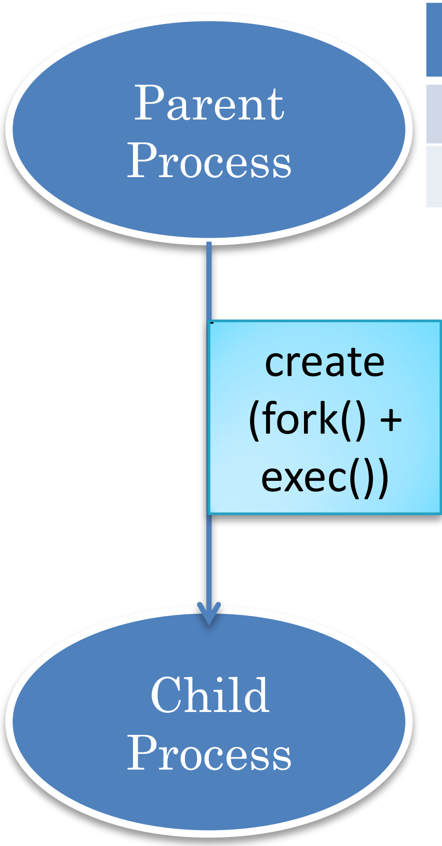
File and Environment Table (Review)

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR

Copy

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR



Environment

Variable	Value
CLASSPATH	/usr/java/lib;/lib
PATH	/usr/java/bin;/bin

Copy

Variable	Value
CLASSPATH	/usr/java/lib;/lib
PATH	/usr/java/bin;/bin

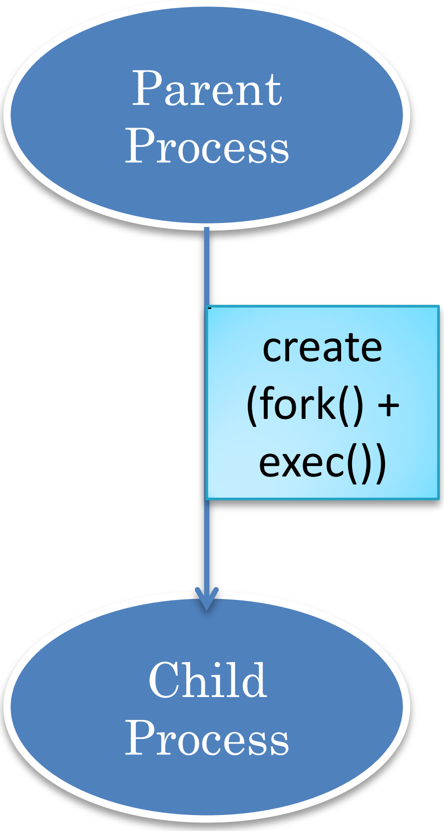
Sharing Pipe Descriptors (Review)

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Copy

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT



pipe(int[2])

How I/O Redirection?

Copying File Table Entries (Review)

File Table

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Copy

Index	Resource
0	STD_IN
1	PIPE_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

Parent Process

create (Fork + Exec)

Child Process

pipe(int[2])

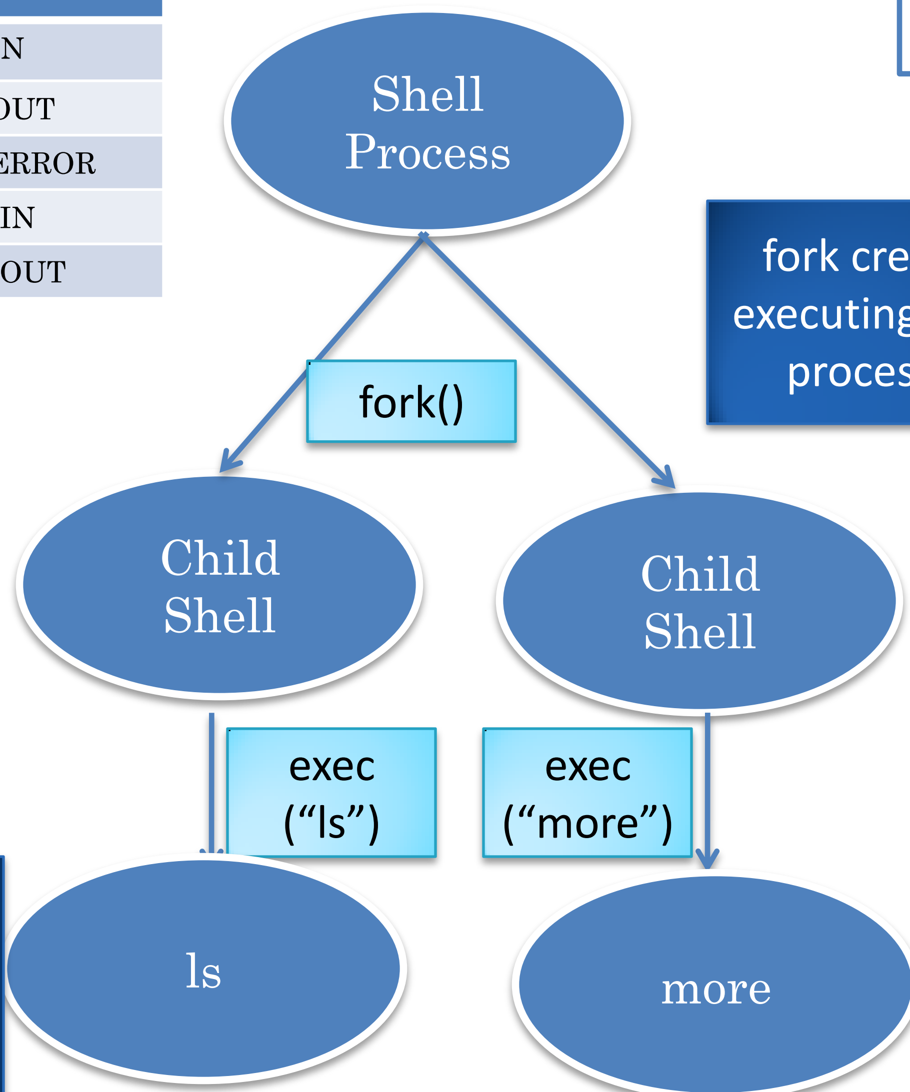
Can copy file table entry to another position (standard input/output) and can close it.

I/O Redirection Details (Review)

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

cat file | more

fork creates new process executing same program as process executing fork

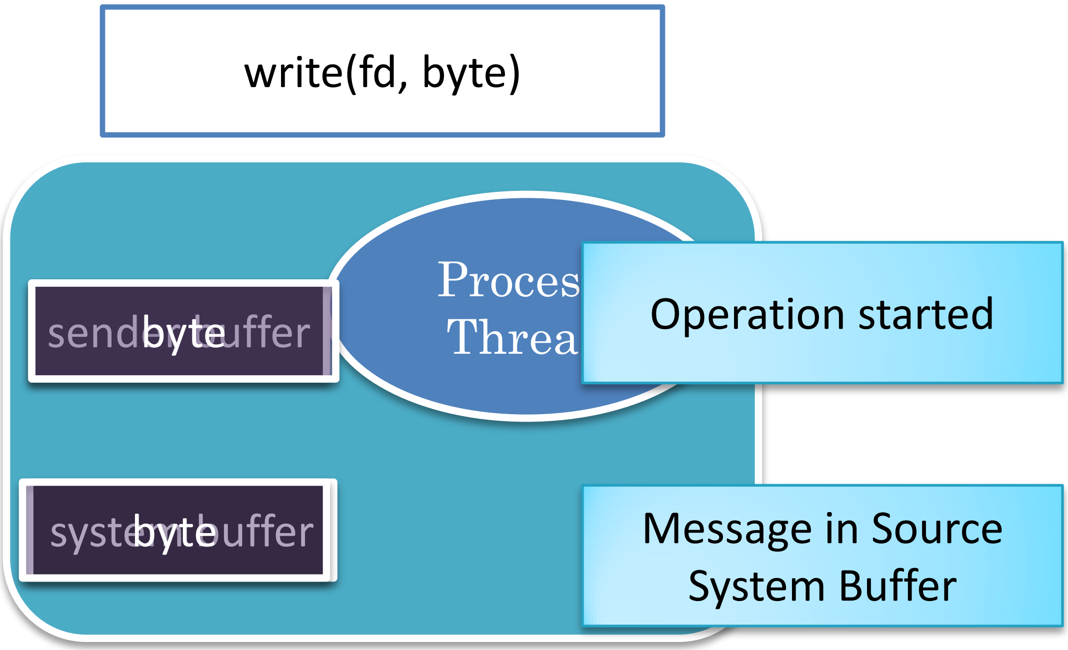


Index	Resource
0	STD_IN
1	PIPE_OUT
2	STD_ERROR

Index	Resource
0	PIPE_IN
1	STD_OUT
2	STD_ERROR

exec makes same process execute another program, keeping the file table

Send/Receive Blocking Times?: Message Pipe Line



Bounded buffer semantics

- Sender waits for non full buffer
- Receiver waits for non empty buffer

Allows lazy evaluation

- Do not to computation that is not needed

Blocking Times?: Message Pipe Line

infinite_output_producer

```
static void main (String args[] {
      while (true) {
            printf("infinite output");
      }
}
```

infinite_output_producer | head - 2

infinite_output_producer blocks after filling buffer

head grabs first two lines from buffer, closes pipe, and terminates

Parent shell process waiting for head unblocks and kills infinite_output_producer

Pipes: Implementation

Pipes: Pros and Cons

I/O Redirection to Different Processes

Processes on same computer with common ancestor

Sockets

Introduced by Berkeley Unix (4.2 BSD)

All OS’s seem to have them in their basic form

Languages such as Java provide a layer above them

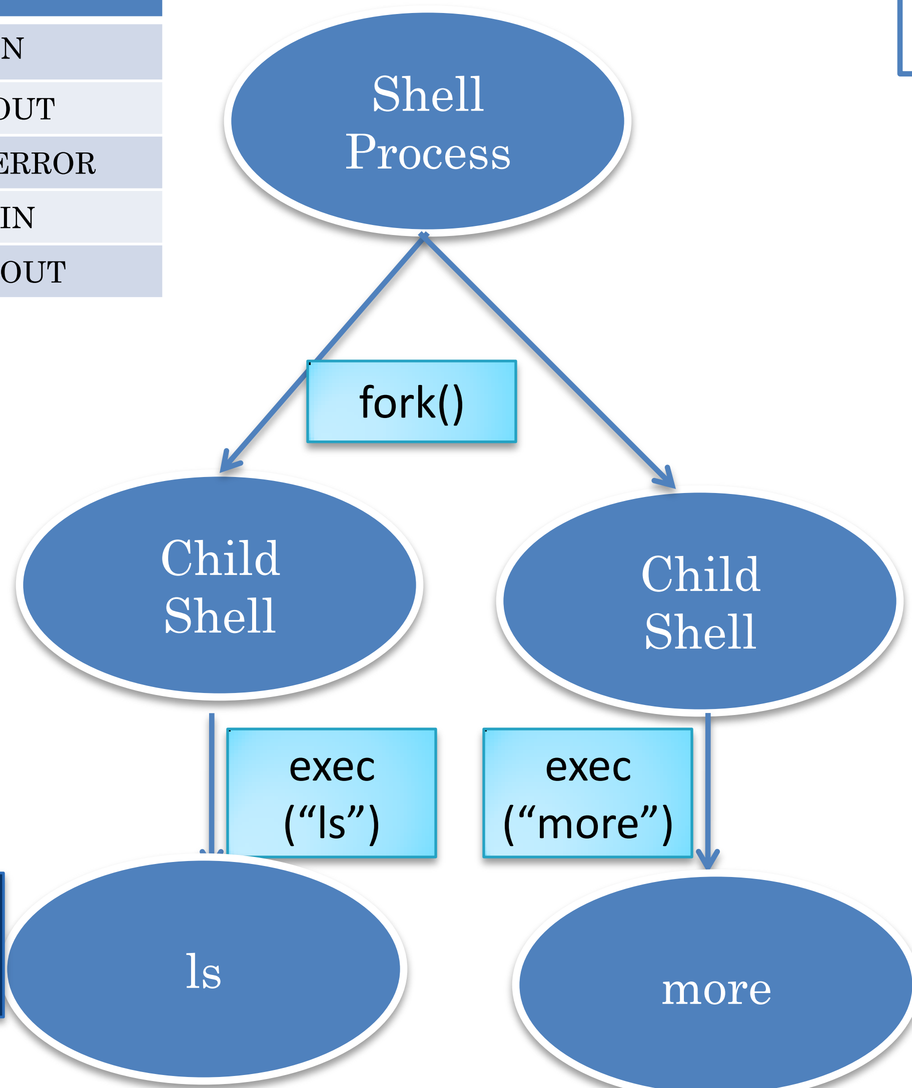
Sockets

Focus is on generality and integration with
File and Teletype I/O

Naming and Sharing in Pipes

Index	Resource
0	STD_IN
1	STD_OUT
2	STD_ERROR
3	PIPE_IN
4	PIPE_OUT

cat file | more

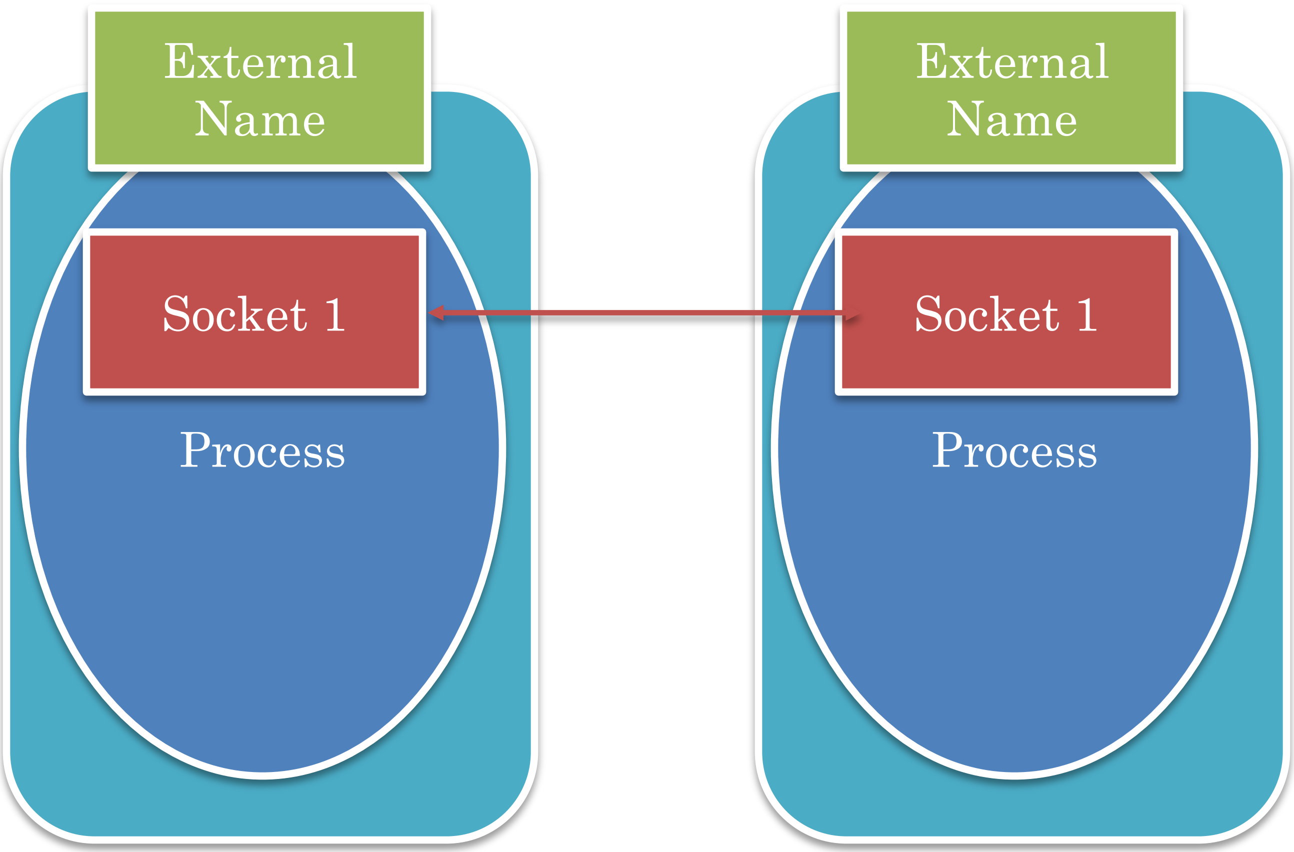


Index	Resource
0	STD_IN
1	PIPE_OUT
2	STD_ERROR

Index	Resource
0	PIPE_IN
1	STD_OUT
2	STD_ERROR

Different hosts?

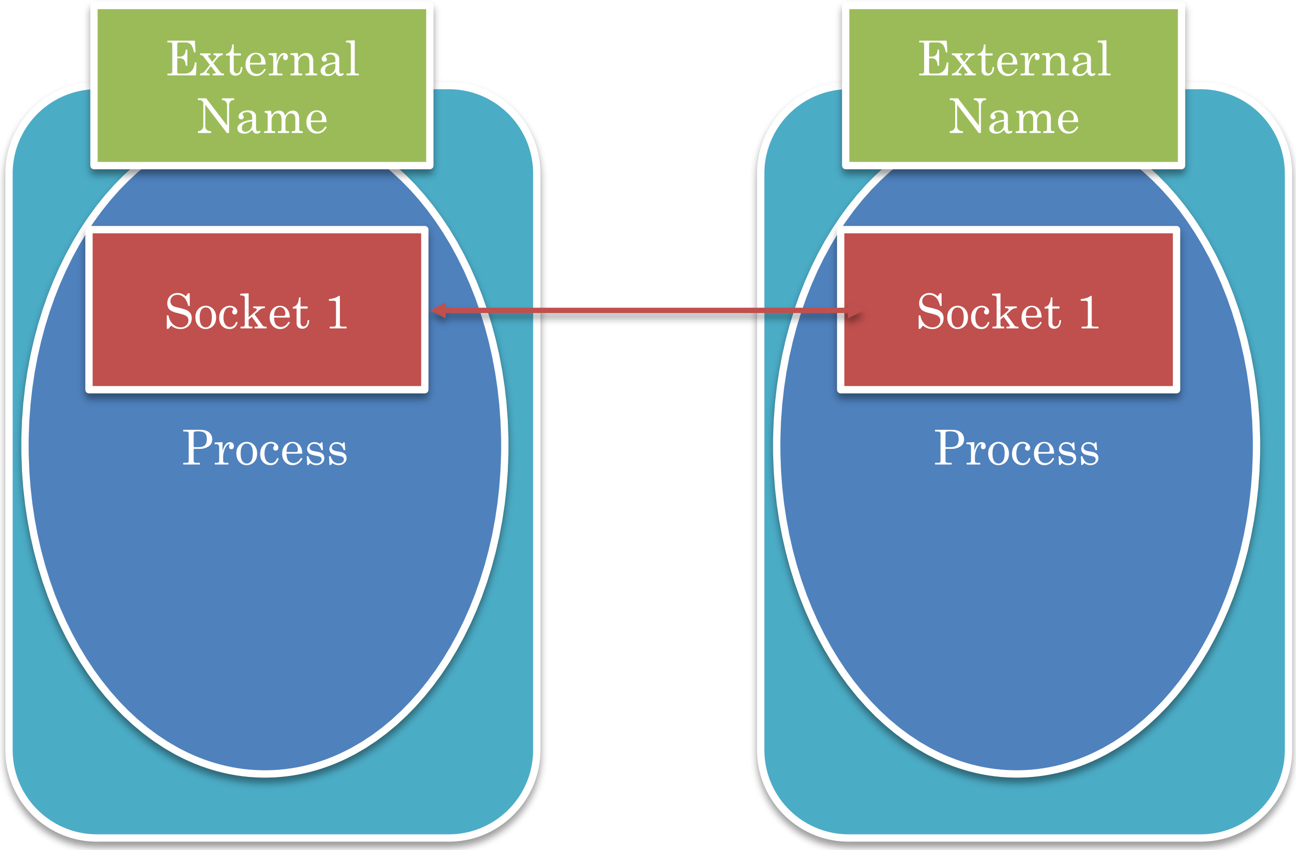
Need to Connect Descriptors



Some how message sent to socket in one process must be received at socket in a another process with no common ancestor

Need external names

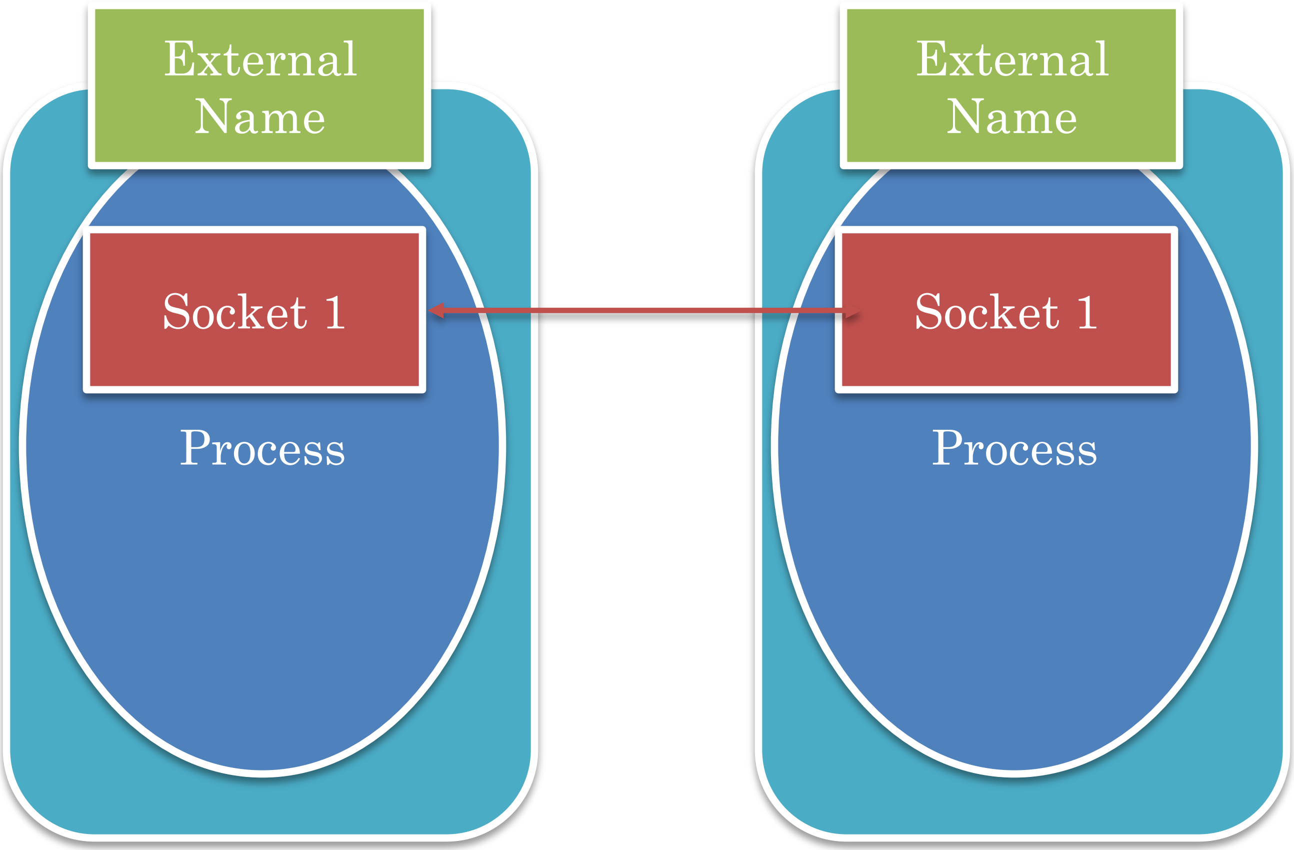
External Name



AF_INET address family: host, port number (Java, Unix)

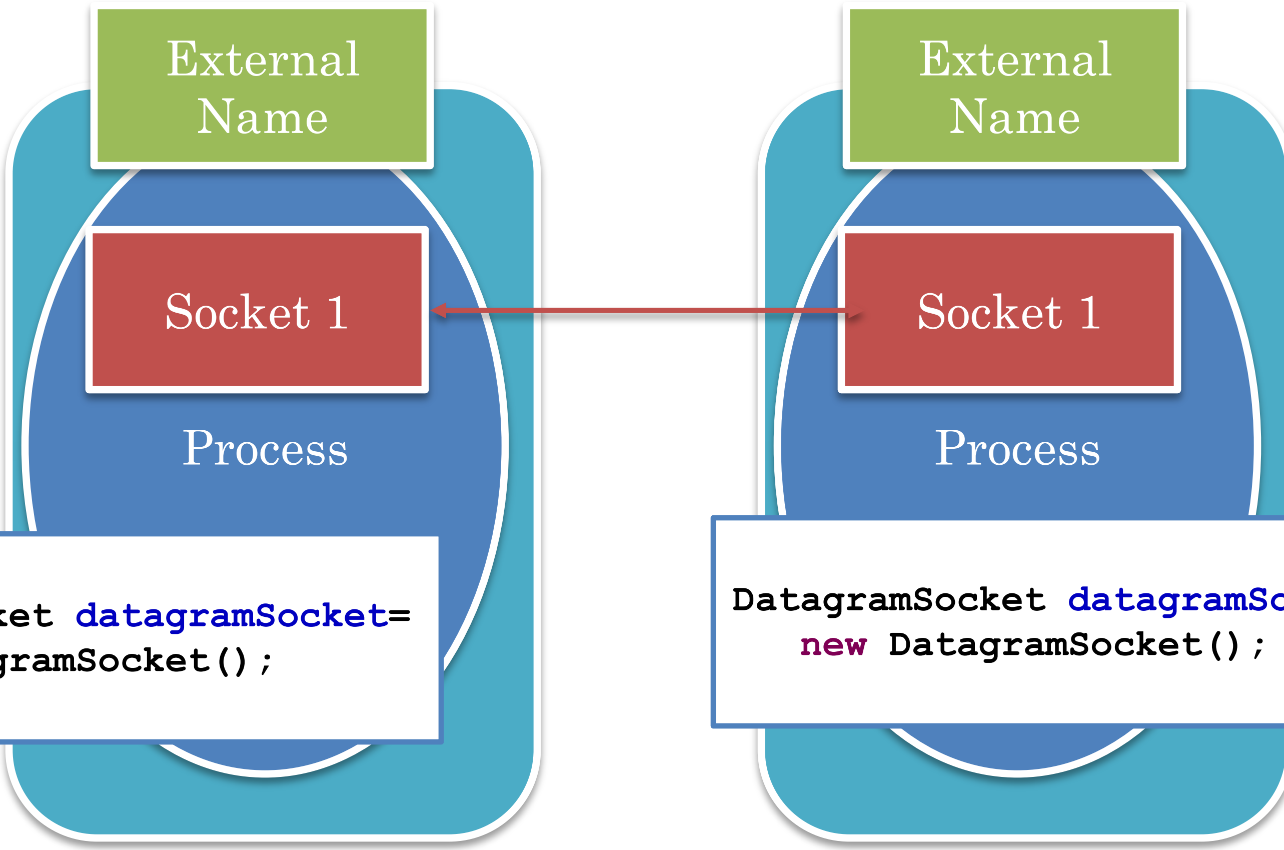
AF_UNIX address family: file name (Unix)

External Name



Somehow message sent to socket 1 must be received at socket 2

Datagram Socket: Send gives Destination



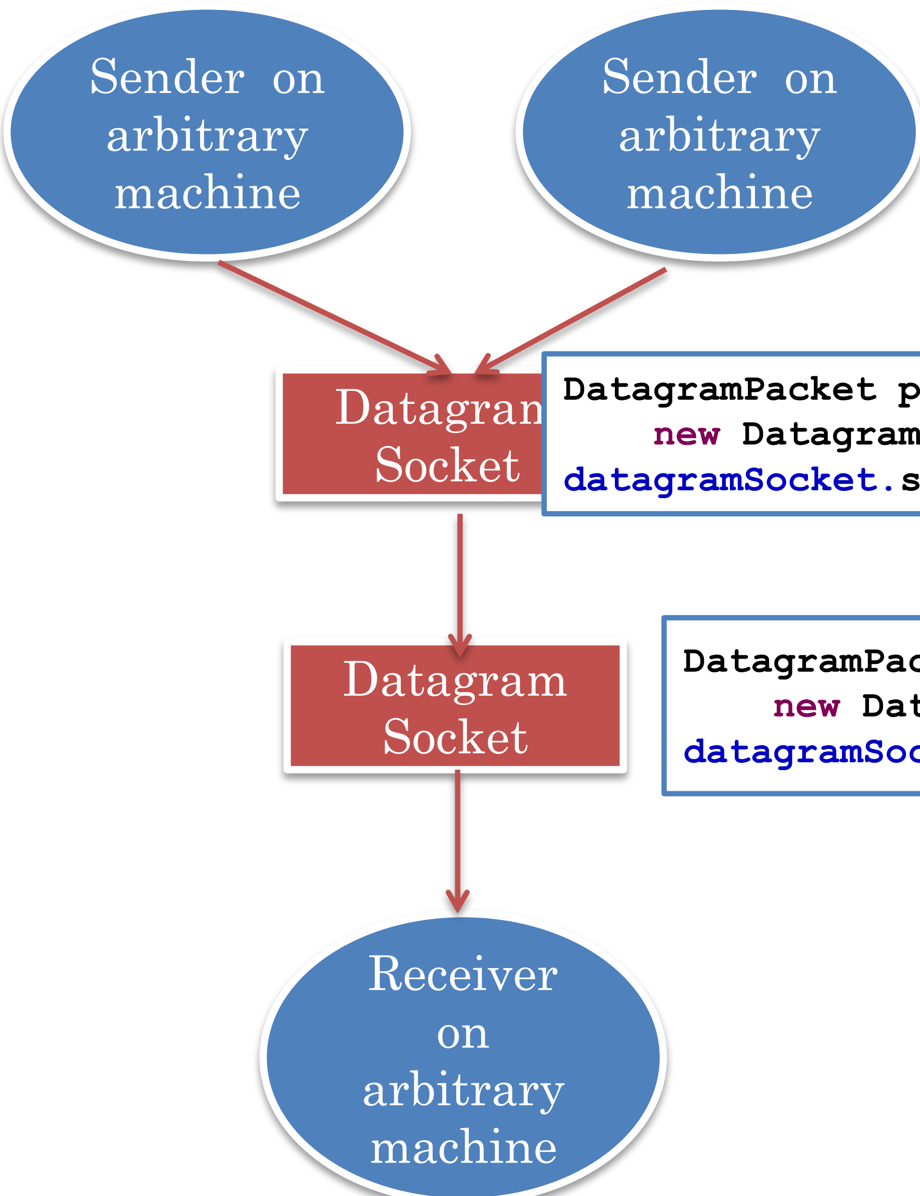

```
DatagramSocket datagramSocket=
   new DatagramSocket();
```

```
DatagramSocket datagramSocket=
   new DatagramSocket();
```

```
DatagramPacket packet= new DatagramPacket(buf, offset,
length);
datagramSocket.receive(packet);
```

```
DatagramPacket packet= new DatagramPacket(buf, offset,
length host, port);
datagramSocket.send(packet);
```

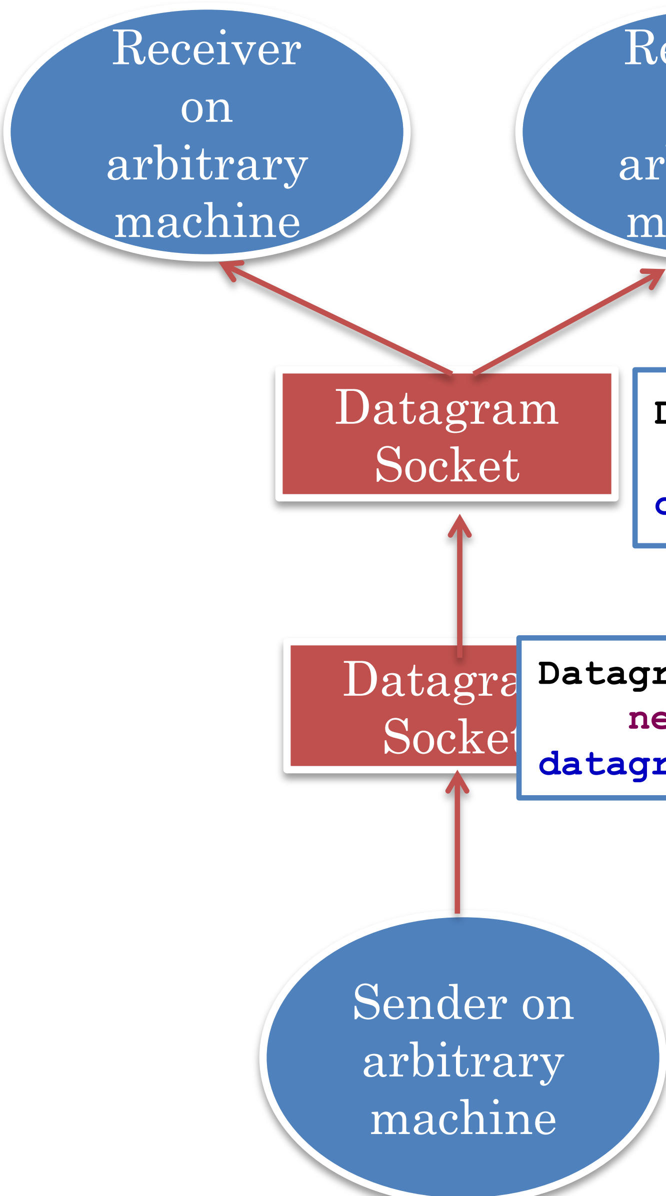
Datagram Socket Sharing




```
DatagramPacket packet=
    new DatagramPacket(buf, offset, length host, port);
datagramSocket.send(packet);
```

```
DatagramPacket packet=
    new DatagramPacket(buf, offset, length);
datagramSocket.receive(packet);
```

Datagram Socket for Datagrams

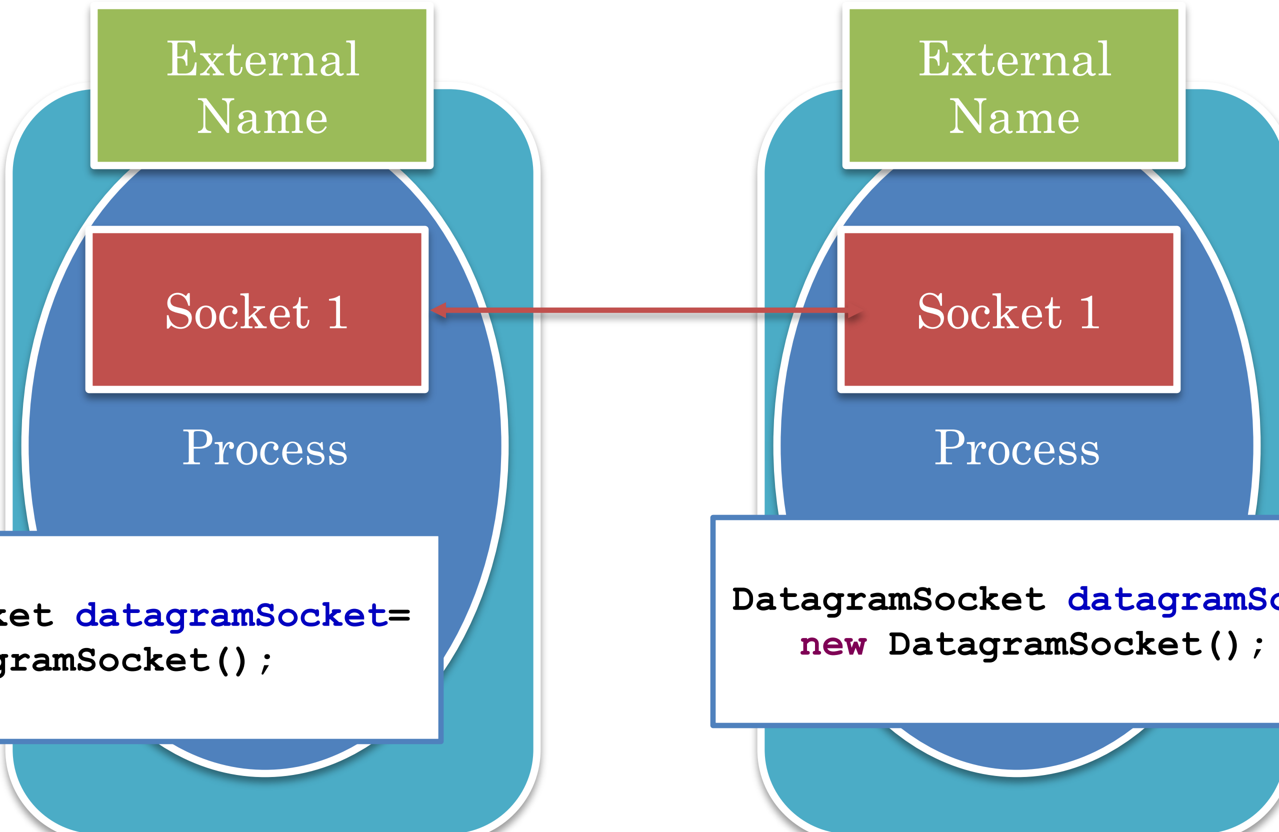



```
DatagramPacket packet=
    new DatagramPacket(buf, offset, length);
datagramSocket.receive(packet);
```

```
DatagramPacket packet=
    new DatagramPacket(buf, offset, length host, port);
datagramSocket.send(packet);
```

Convenience method when destination host and port are repeated?

Datagrams: Datagram Socket and Special Calls



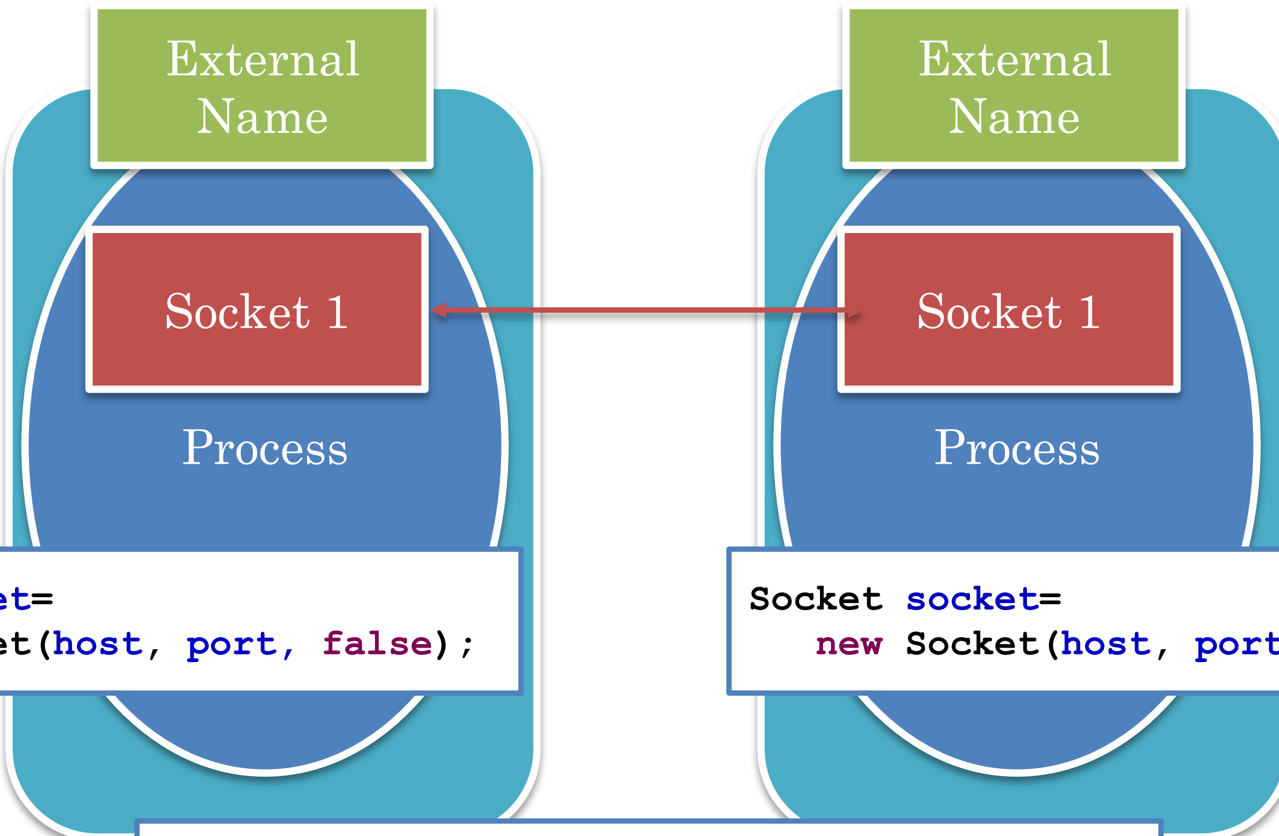

```
DatagramSocket datagramSocket=
   new DatagramSocket();
```

```
DatagramSocket datagramSocket=
   new DatagramSocket();
```

```
DatagramPacket packet= new DatagramPacket(buf, offset,
length);
datagramSocket.receive(packet);
```

```
DatagramPacket packet= new DatagramPacket(buf, offset,
length host, port);
datagramSocket.send(packet);
```

Datagrams: Socket and Special Calls



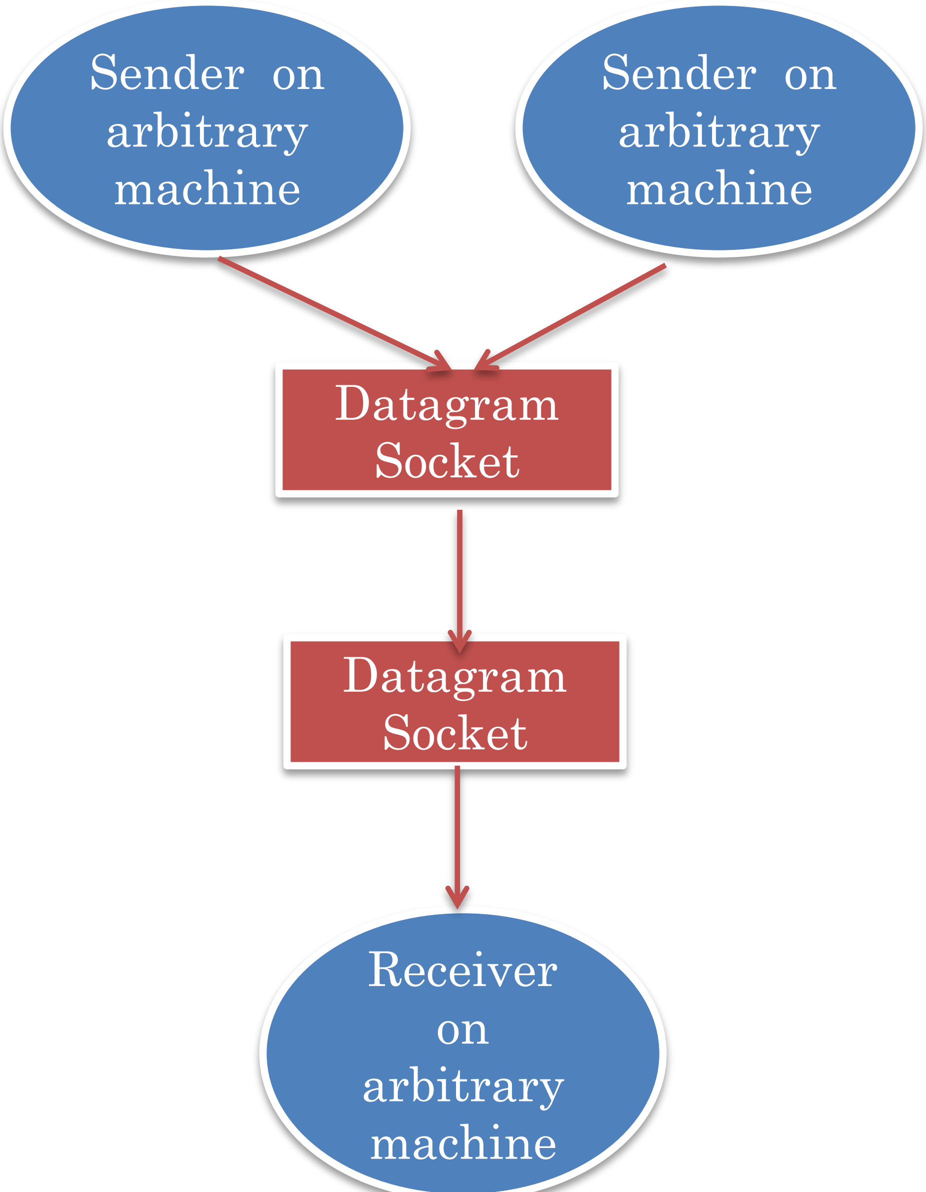

```
Socket socket=
   new Socket(host, port, false);
```

```
Socket socket=
   new Socket(host, port, false);
```

```
inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);
```

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

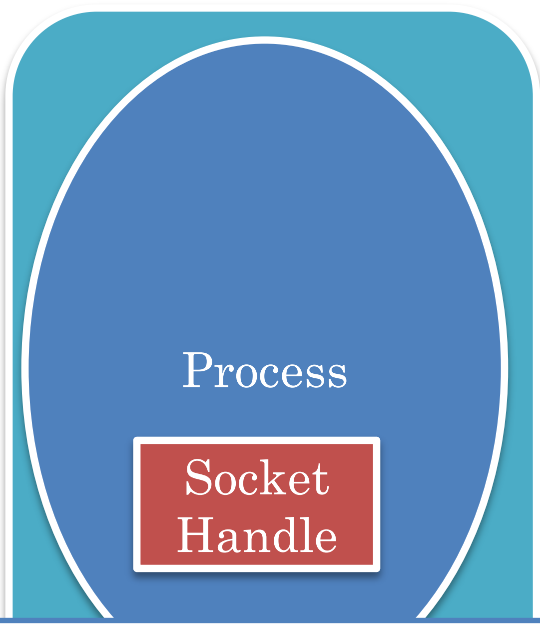
Datagram Socket Sharing

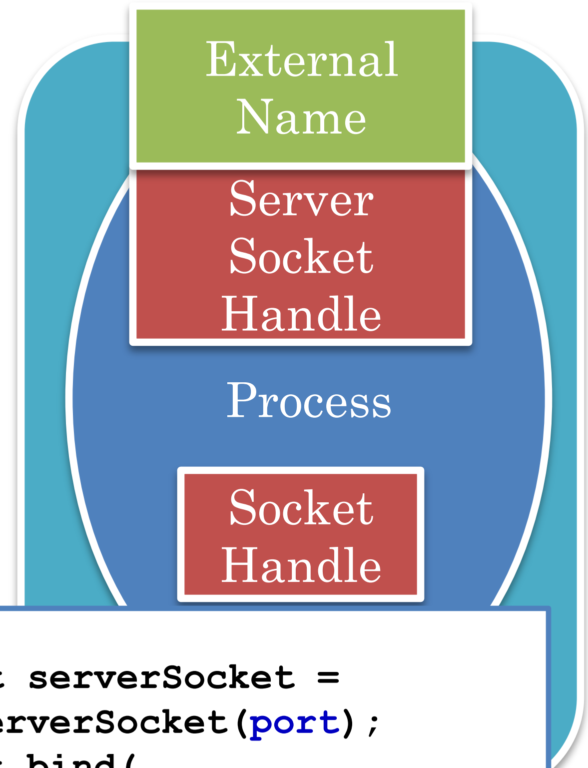


No private channels

E.g. open file

From Datagram to Stream Socket





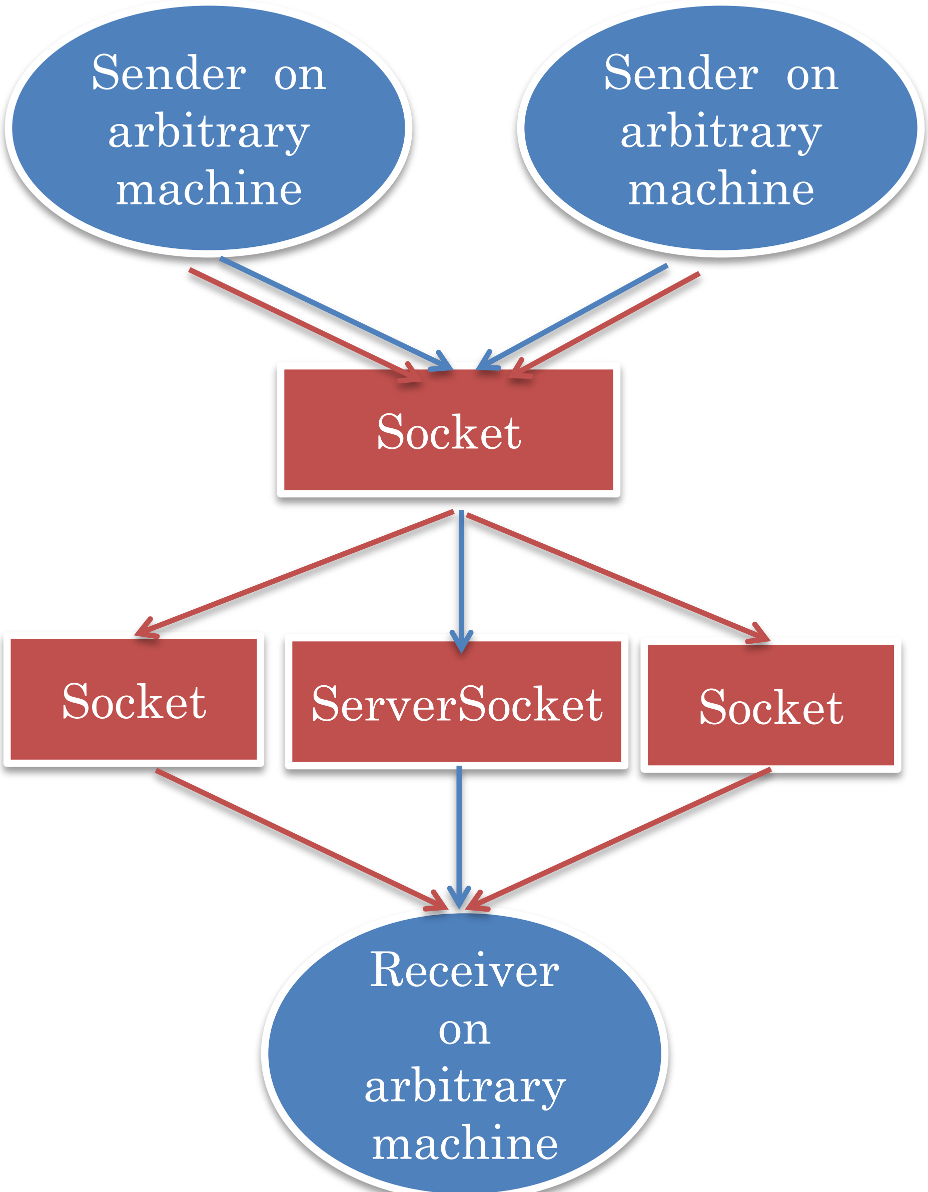

```
Socket socket = new Socket();
socket.connect(
 new InetSocketAddress(host, port));
```

```
ServerSocket serverSocket =
      new ServerSocket(port);
serverSocket.bind(
      new InetSocketAddress(port));
```

```
Socket socket =
   serverSocket.accept();
```

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

Stream Socket Sharing



Server socket is used to create stream-based socket

Each client connects to it to create a dedicated connection

A data (file) server would create single server socket

“Open” data source operation would connect to server socket

Stream-based socket would represent opened source, which can be read and written

Reliability and In-Order?

Datagram sockets: no guarantee, built on top of UDP, message size limit

Stream sockets: in-order reliable on top of TCP/IP

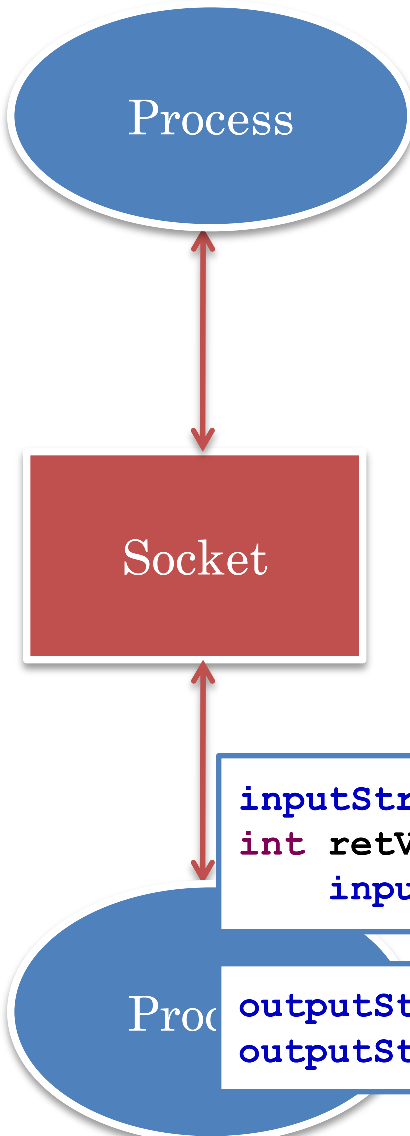
Do not have to change IPC mechanism to change guarantee

Makes sockets complex, Java separating them

```
socket = new Socket(host, port, isStream);
```

```
socket = new DatagramSocket();
```

Read/Write



Explicit operations to send and receive information

Integrated with File and Standard I/O

```
inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);
```

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

Implicit vs Non Implicit

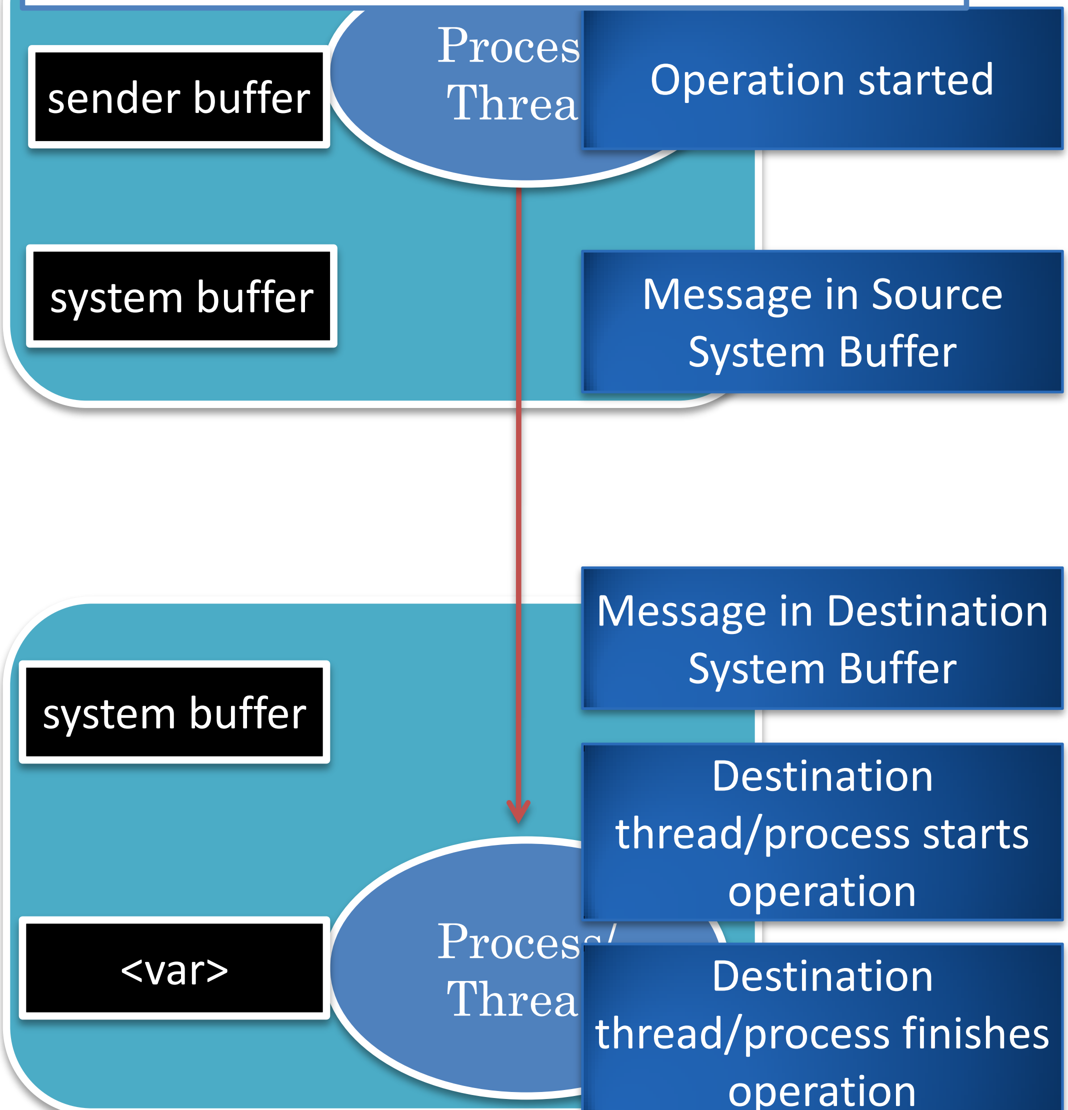
Non connection based ipc can use both implicit and explicit addressing

Connection based ipc uses implcicit addressing of destination streams and offset within stream

May also define access rights with which connection was opened

Send Blocking Times?: Message Pipe Line

```
outputStream.write(buf, offset, length);
```



Semantics should be like file and terminal I/O

In older systems, file I/O blocked until data on disk (synchronous)

Inefficient, specially if stream I/O

Block until in system buffer

If stream socket, then message will get through

Receive Blocking Times

```
inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);
```

Block until <= length >=1 bytes received

retVal indicates actual length

Idea is to not block beyond next network message arrival

Give max value so buffer not overwritten

If expecting message of certain size, must loop

Blocking Operations

```
socket.connect(
 new InetSocketAddress(host,port));
```

Block until server accepts connection to server socket

```
Socket socket =
    serverSocket.accept();
```

Block until next client tries to contact the server socket

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

Block until in system buffer

```
inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);
```

Block until <= length >=1 bytes received

Socket Blocking

All operations involve some blocking

What if we want no blocking?

In Java, heavyweight threads can be created

In Unix several primitives for single thread to not block

In Java special NIO layer for blocking and non blocking   for sockets (and other I/O resources)

NIO

NIO (blocking and non blocking)

Sockets (blocking)

Even more flexibility than sockets

How do add non blocking

Blocking Operations

```
socket.connect(
 new InetSocketAddress(host,port));
```

Block until server accepts connection to server socket

```
Socket socket =
    serverSocket.accept();
```

Block until next client tries to contact the server socket

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

Block until in system buffer

```
inputStream = socket.getInputStream();
int retVal =
    inputStream.read(buf, offset, length);
```

Block until <= length >=1 bytes received

Xinu vs Ada

int recvclr ()

Non blocking, polling, returns either message if it exists, otherwise a special value

select

receive … <port^{1}> …

receive … <port^{n}> …

end

Each arm statically registers and interest in an operation on a port, and provides variables and code for completing the operation

Select chooses which of the enabled operations is executed

Combining the Two Ideas

int recvclr ()

Like rcvclr, no operation will ever block

But will not poll, instead will execute operation that is ready - guaranteed to succeed at least partly

```
select
    receive … <port¹> …
    receive … <portⁿ> …
end
```

Select-like concept to register interest in receive and other operations

Select will not choose operation to execute, it will tell programmers which operations are ready

Non blocking ready operation such as rcvclr can then be used without blocking

Form of Select in NIO

select

receive … <port^{1}> …

receive … <port^{n}> …

end

Objects for arm and selector

Interest in a port and operation, and execution of operation decoupled

Can dynamically register with a selector interest in an operation (e.g. receive, , send, accept) on a resource (e.g. port, file)

Selector blocks and on unblocking tells its user which of the interested (operation, resource)pairs are enabled and thus ready for execution

Selector

Resource on which async operation may be executed (replaces Socket, ServerSocket, File)

Selectable Channel

- SelectionKey register(Selector s, int ops)
- configureBlocking(boolean)

Registers interest in (resource, operation) pair referenced by a key id

```
SelectionKey key = selectableChannel.register( selector, SelectionKey.OP_ACCEPT );
```

Single selector for all operations

```
Selector selector = Selector.open();
```

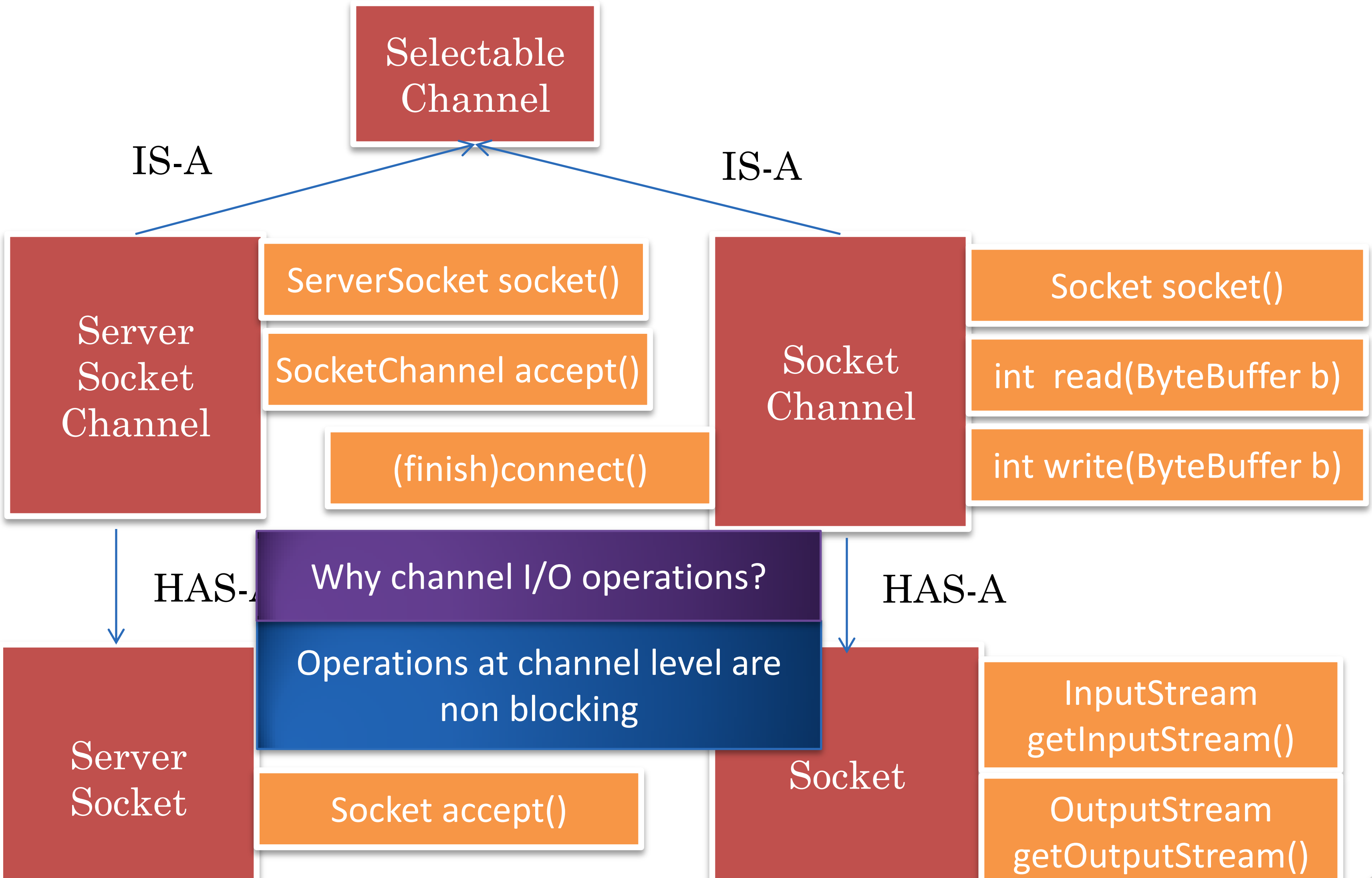
Selector

- **int** select() — Blocking call waiting until at least one registered pair is enabled by some event, and returning # of such pairs
- Set<SelectionKey> selectedKeys() — Keys of actual enabled operations
- Selector wakeup() — Unblock select, usually after a new registration

Selection Key

- SelectableChannel channel () — Key to selectable channel, which can be used to execute enabled operation immediately

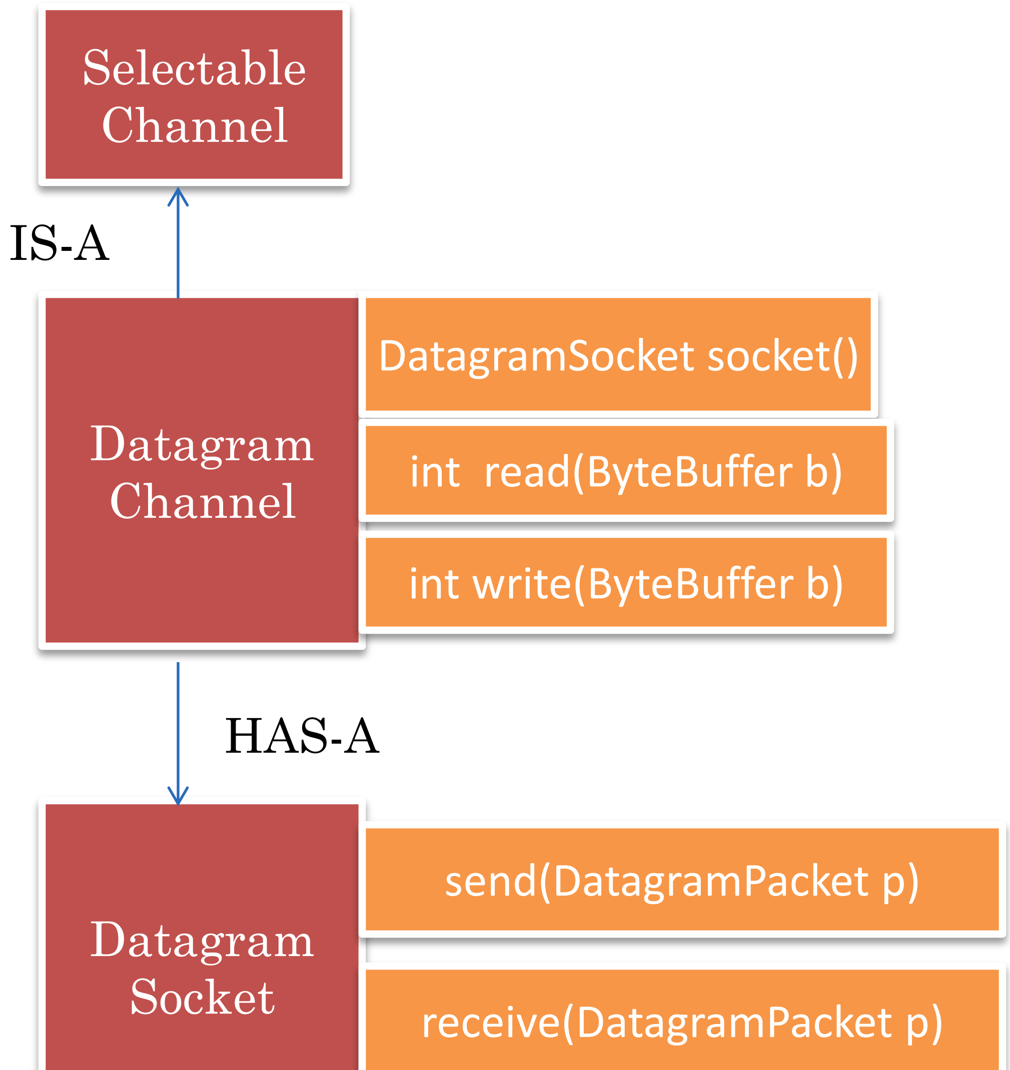
ServerSocket and Socket Channels




```
ServerSocketChannel serverSocketChannel =
    ServerSocketChannel.open();
```

```
SocketChannel  socketChannel =
    SocketChannel.open();
```

DataGram Channel




```
DatafromChannel datagramSocketChannel =
    DatagramChannel.open();
```

Channel vs. Stream I/O

```
inputStream = socket.getInputStream();
int retVal = inputStream.read(buf, offset, length);
```

```
outputStream = socket.getOutputStream();
outputStream.write(buf, offset, length);
```

```
int retVal = socketChannel.read(byteBuffer);
```

```
int retVal = socketChannel.write(byteBuffer)
```

ByteBuffer, like packet, encapsulates buf, offset, and length

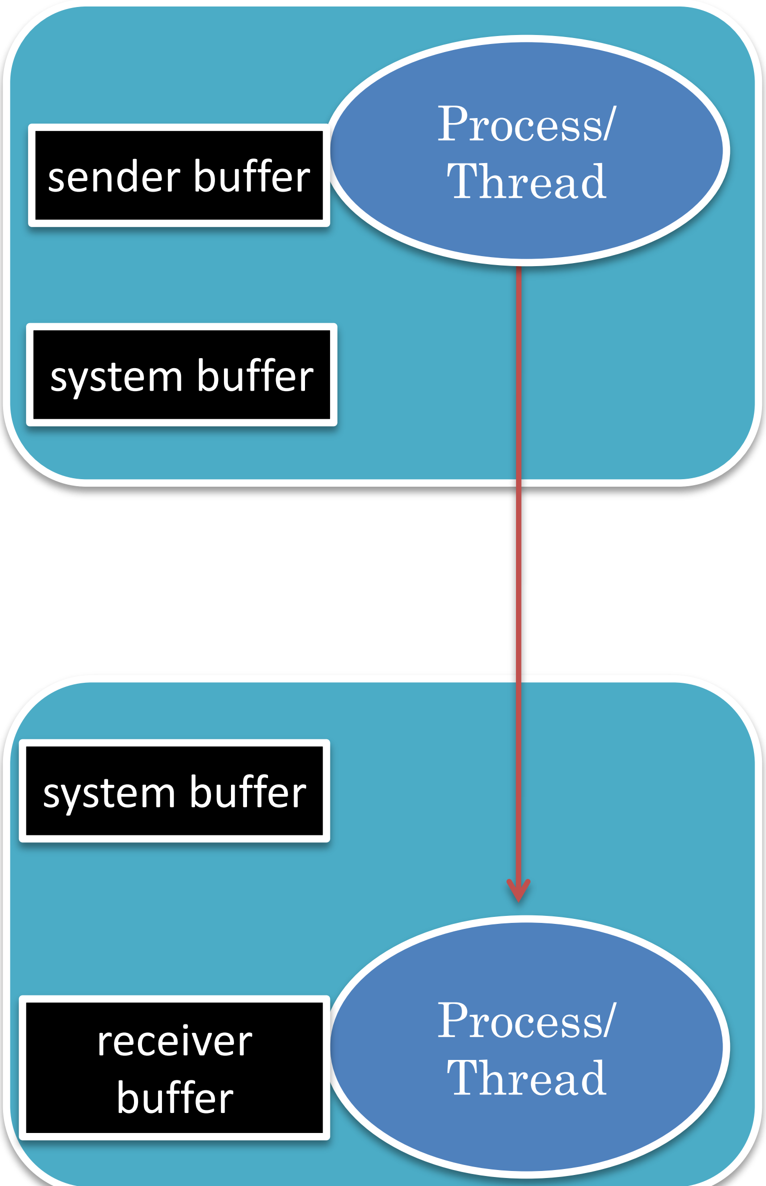
Write writes only as many bytes as available in source buffer when in async mode

Channel unlike stream and like Unix file/socket can be read and written

IPC mechanism may not complete operation and same buffer may be used for multiple batch operations

System can use the buffer directly instead of creating own source or destination buffer

Direct vs. Non Direct



Direct Buffer: System tries to use sender and receiver buffer directly without creating intermediate source or system non direct buffer

Buffer copying is an expensive operation

In synchronous sends safe to avoid copying. In asynchronous, requires careful programming

Direct buffer allocation from kernel space so more costly

Use direct buffer only when performance is an issue and buffer is long lived

Allocating Direct vs. Non Direct

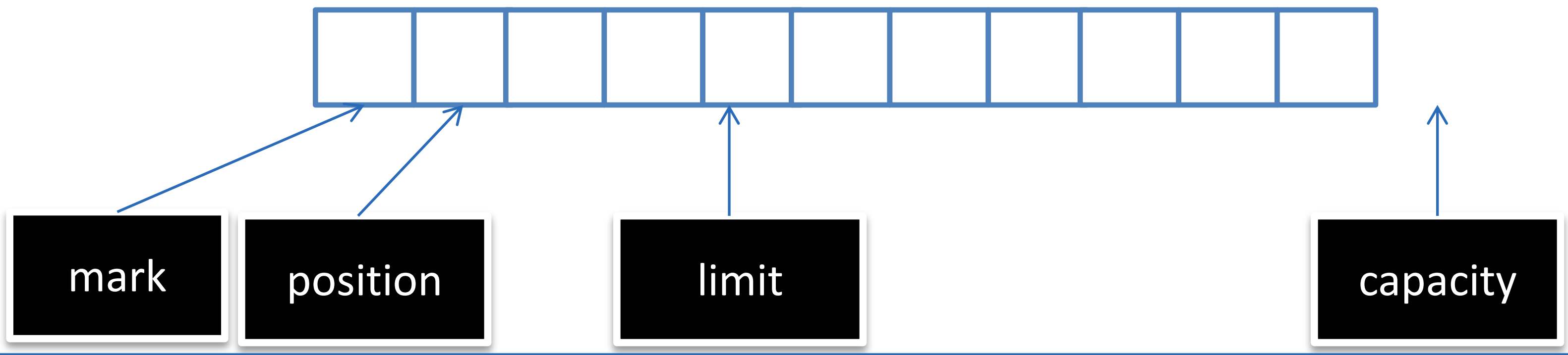
Direct

```
ByteBuffer ByteBuffer.allocateDirect(capacity)
```

Indirect

```
ByteBuffer ByteBuffer.allocate(capacity)
ByteBuffer ByteBuffer.wrap(bytes[])
```

Data Structure



Storage for contents

Size of (available) contents

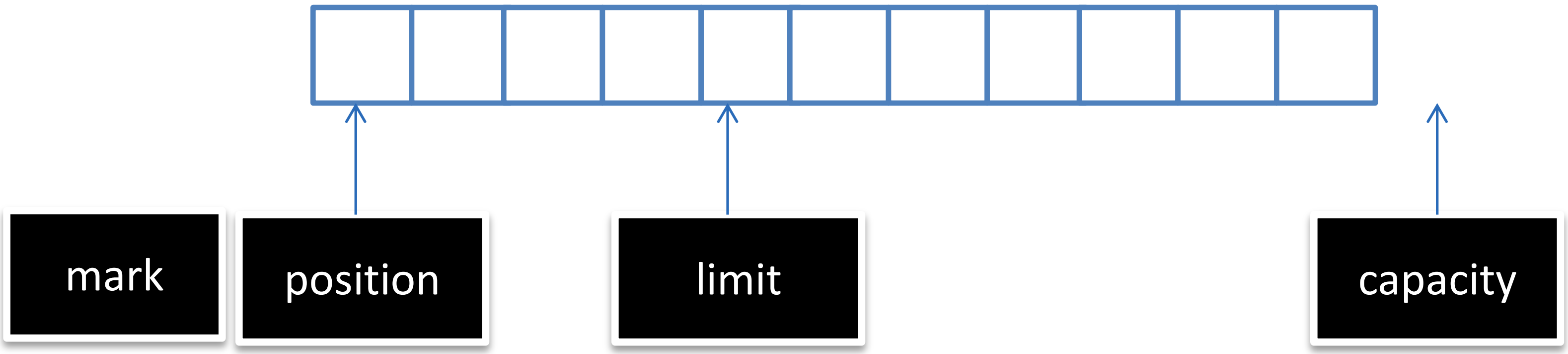
Position of next element to be read or written

Read may not yield expected bytes, write may not empty all bytes

May use the same buffer for multiple serial operations
or batch operations, need to mark position of first unconsumed byte

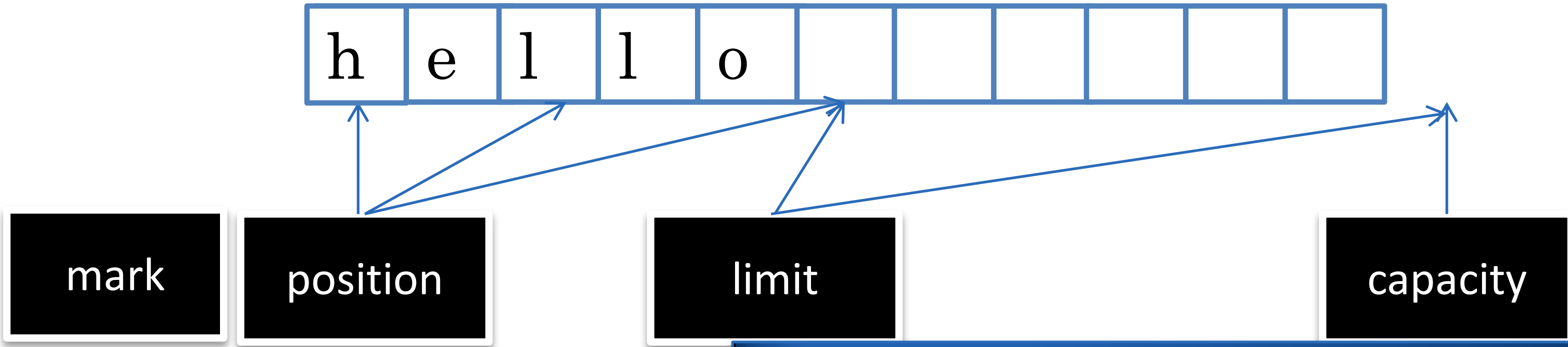
mark <= position <= limit <= capacity

Data Structure



	ByeBuffer	
int position		next element to be read or written by ipc/app
int limit		first element not be read or written by ipc / app
int capacity		size of buffer
byte[] contents		
mark		Position of first unaccessed byte
isMarked		

Writing



ByeBuffer

- clear()
- put(byte[])
- flip()

Application makes buffer ready for writing by it

byteBuffer.clear()

Puts data in it

byteBuffer.put(“hello”.getBytes())

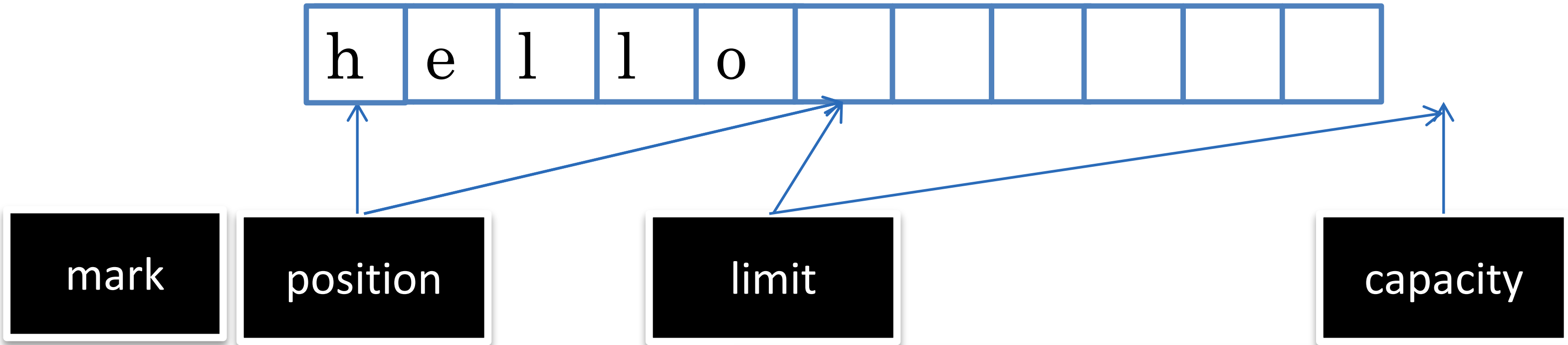
Makes buffer ready for reading by IPC mechanism

byteBuffer.flip()

Invokes channel write

channel.write(buf) → 2

Reading



ByeBuffer

get(byte[])

Application makes buffer ready for writing by IPC

byteBuffer.clear()

Application invokes channel read, channel writes

byteBuffer.read(byteBuffer)

Makes buffer ready for reading by it

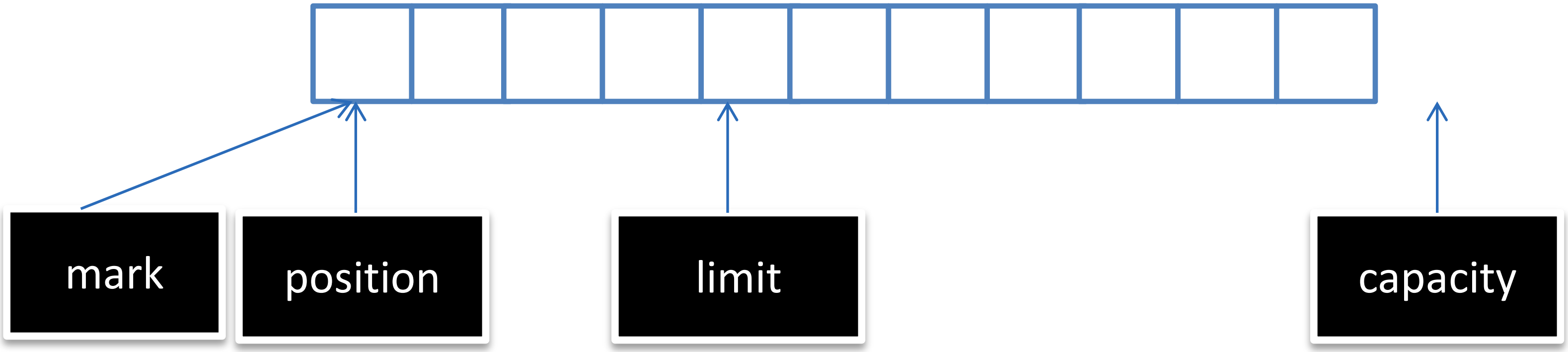
byteBuffer.flip()

Reads buffer

byte[] bytes = **new** byte[byteBuffer.remaining()]

byteBuffer.get(bytes)

Marking



int position

int limit

int capacity

byte[] contents

mark

marked

ByeBuffer

mark()

mark position of first unconsumed op

byteBuffer.mark()

Same buffer may be used for multiple batch operations before result is consumed

Receiving same logical data in multiple physical chunks

Operations

ByeBuffer

Int remaining()	limit – position
flip()	limit = position, position = 0, marked = false
clear()	position = 0, limit = capacity, marked = false
rewind()	position = 0, marked = false
reset()	position = mark, if marked
mark()	mark = position, marked = true
put(bytes[] b)	For each byte, write next byte at pos, pos++
get(bytes[] b)	For each byte, read next byte at pos, and pos++
position(int p)	position = p
limit(int l)	limit = l

NIO Evaluation

More flexible than even Java sockets

Hence more complex

Even the normal case requires tutorials describing normal patterns of user

Selection must be done in a single complex thread