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

- stack
- registers
- program counter
- priority
- status
- int receivedWord
- bool hasMessage

Thread
SEND OPERATION

```
send (tid, 5 )

send (tid, 6 )

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

```java
int receive () {
    hasMessage= false;
    int receivedWord;

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

    return receivedWord;
}
```
**RECVCLR Operation**

```
int receive ()

If (!word received) {
    return NO_VAL
}

hasMessage = false;
return receivedWord;
```
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?

Sender

Pipe

Receiver

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

man 2 pipe | more
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

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
</tbody>
</table>

### Environment

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH</td>
<td>/usr/java/lib;/lib</td>
</tr>
<tr>
<td>PATH</td>
<td>/usr/java/bin;/bin</td>
</tr>
</tbody>
</table>

### Parent Process

create (fork() + exec())

### Child Process

Copy

### Copy

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH</td>
<td>/usr/java/lib;/lib</td>
</tr>
<tr>
<td>PATH</td>
<td>/usr/java/bin;/bin</td>
</tr>
</tbody>
</table>
Sharing Pipe Descriptors

File Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

Parent Process

Create (fork() + exec())

Child Process

pipe(int[2])

How I/O Redirection?
COPYING FILE TABLE ENTRIES

File Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

fork creates new process executing same program as process executing fork

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

cat file | more

exec ("ls")

fork()

exec ("more")
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

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
</tbody>
</table>

#### Environment

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH</td>
<td>/usr/java/lib;/lib</td>
</tr>
<tr>
<td>PATH</td>
<td>/usr/java/bin;/bin</td>
</tr>
</tbody>
</table>

#### Parent Process

- Create (fork() + exec())

#### Child Process

- Variable    | Value                |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH</td>
<td>/usr/java/lib;/lib</td>
</tr>
<tr>
<td>PATH</td>
<td>/usr/java/bin;/bin</td>
</tr>
</tbody>
</table>

#### Copy
SHARING PIPE DESCRIPTORS (REVIEW)

File Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

Parent Process

create (fork() + exec())

Child Process

pipe(int[2])

How I/O Redirection?
COPYING FILE TABLE ENTRIES (REVIEW)

File Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

Parent Process

create (Fork + Exec)

Copy

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)

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

fork creates new process executing same program as process executing fork

cat file | more

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

exec ("ls")

exec ("more")
send/receive blocking times?: message pipe line

write(fd, byte)

- operation started
- message in source system buffer
- 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
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
  - Pre 4.2 Unix BSD

- Sockets
  - 4.2 Unix BSD
PIPEC: 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

<table>
<thead>
<tr>
<th>Index</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STD_IN</td>
</tr>
<tr>
<td>1</td>
<td>STD_OUT</td>
</tr>
<tr>
<td>2</td>
<td>STD_ERROR</td>
</tr>
<tr>
<td>3</td>
<td>PIPE_IN</td>
</tr>
<tr>
<td>4</td>
<td>PIPE_OUT</td>
</tr>
</tbody>
</table>

```
cat file | more
```

```
fork()
```

```
exec ("ls")
```

```
exec ("more")
```

```
ls
```

```
more
```

Different hosts?
Some how message sent to socket in one process must be received at socket in 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)
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

Sender on arbitrary machine

Sender on arbitrary machine

Datagram Socket

Datagram Packet

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

Datagram Socket

Datagram Packet

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

Receiver on arbitrary machine
**Datagram Socket for Datagrams**

- **Sender on arbitrary machine**
- **Datagram Socket**
  - `DatagramPacket packet = new DatagramPacket(buf, offset, length);`
  - `datagramSocket.send(packet);`
- **Datagram Socket**
  - `DatagramPacket packet = new DatagramPacket(buf, offset, length host, port);`
  - `datagramSocket.receive(packet);`
- **Receiver on arbitrary machine**
- **Receiver on arbitrary machine**

Convenience method when destination host and port are repeated?
Datagrams: Datagram Socket and Special Calls

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

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

datagramSocket.send(packet);
```
**Datagrams: Socket and Special Calls**

```java
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**

Sender on arbitrary machine

Datagram Socket

Datagram Socket

Receiver on arbitrary machine

No private channels

E.g. open file

Sender on arbitrary machine
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**

- **Sender on arbitrary machine**
- **Receiver on arbitrary machine**
- **Socket**
  - **ServerSocket**
  - **Socket**

**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();
Explicit operations to send and receive information

Integrated with File and Standard I/O

```java
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 implicit addressing of destination streams and offset within stream

May also define access rights with which connection was opened
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**

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

- Block until $<= \text{length} >=1$ bytes received
- $\text{retVal}$ 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**

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

Socket socket =
    serverSocket.accept();

Block until server accepts connection to server socket

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));

Socket socket =
    serverSocket.accept();

Block until server accepts connection to 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 ()

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

Non blocking, polling, returns either message if it exists, otherwise a special value
Combining the Two Ideas

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-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

<table>
<thead>
<tr>
<th><strong>Selectable Channel</strong></th>
<th><strong>Register interest in (resource, operation) pair referenced by a key id</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionKey register(Selector s, int ops)</td>
<td>SelectionKey key = selectableChannel.register(selector, SelectionKey.OP_ACCEPT);</td>
</tr>
<tr>
<td>configureBlocking(boolean)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Selector</strong></th>
<th><strong>Single selector for all operations</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Selector selector = Selector.open();</td>
<td></td>
</tr>
<tr>
<td>int select()</td>
<td></td>
</tr>
<tr>
<td>Set&lt;SelectionKey&gt; selectedKeys()</td>
<td></td>
</tr>
<tr>
<td>Selector wakeup()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Selection Key</strong></th>
<th><strong>Key to selectable channel, which can be used to execute enabled operation immediately</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectableChannel channel ()</td>
<td></td>
</tr>
</tbody>
</table>

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

- Blocking call waiting until at least one registered pair is enabled by some event, and returning # of such pairs

- Keys of actual enabled operations

- Unblock select, usually after a new registration
ServerSocket and Socket Channels

ServerSocketChannel

- ServerSocket socket()
- SocketChannel accept()
- (finish)connect()

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

SocketChannel

- Socket socket()
- int read(ByteBuffer b)
- int write(ByteBuffer b)

SocketChannel socketChannel = SocketChannel.open();

Why channel I/O operations?

Operations at channel level are non blocking

InputStream

- getInputStream()

OutputStream

- getOutputStream()
**DataGram Channel**

Selectable Channel

**IS-A**

Datagram Channel

DatagramSocket socket()

int read(ByteBuffer b)

int write(ByteBuffer b)

**HAS-A**

Datagram Socket

send(DatagramPacket p)

receive(DatagramPacket p)

DataframChannel datagramSocketChannel = DatagramChannel.open();
**Channel vs. Stream I/O**

```java
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

- **mark**: Position of next element to be read or written
- **position**: Storage for contents
- **limit**: Size of (available) contents
- **capacity**: Position of first unconsumed byte

- 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**

```
| mark | position | limit | capacity |
```

- **int position**
- **int limit**
- **int capacity**
- **byte[] contents**
- **mark**
- **isMarked**

- **next element to be read or written by ipc/app**
- **first element not be read or written by ipc/app**
- **size of buffer**
- **Position of first unaccessed byte**
**Writing**

![Diagram showing the process of writing in a buffer]

- **mark**
- **position**
- **limit**
- **capacity**

**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) \to 2
```
**Reading**

ByeBuffer

- mark
- position
- limit
- capacity

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

```java
byte[] bytes = new byte[byteBuffer.remaining()]
byteBuffer.get(bytes)
```
Marking

`byteBuffer.mark()`

Mark position of first unconsumed op

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