System Architecture

10 Message Passing

IPC Design & Implementation, IPC Application, IPC Examples

December 01 2008
Winter Term 2008/09
Gerd Liefländer
Agenda

- Motivation/Introduction
- Message Passing Model
- Elementary IPC
  - Design Parameters for IPC
    - Synchronization
    - Addressing Modes
    - Lifetime
    - Data Transfer
    - Types of Activations (your work)
- High Level IPC
- IPC Applications
- IPC Examples
Motivation
Yet another Concept?

1. Previous mechanisms relied on shared memory ⇒ all these solutions do not work in distributed systems

2. Threads of different applications need protection ⇒ even in systems with a common RAM, because you do not want to open your protected AS for another cooperating non trusted piece of software

3. To minimize kernels some architects only offer IPC to solve
   - communication problems as well as all
   - synchronization problems

4. However, the opposite way also works, i.e. you can use semaphores to implement a message passing system
Message Passing

- Used for Inter-“Process” Communication (IPC)
  - Interacting threads within a distributed system
  - Interacting threads within the same computer
  - Interacting threads within the same address space
  - We expect a decreasing complexity and vice versa an increasing speedup when implementing IPC

- Application:
  - Exchange information in form of messages

- At least two primitives:
  - send (destination, message)
  - receive (source, message)
IPC Model
Basic Principle of Message Passing

Thread T1
send

Explicit Data Transfer

Thread T2
receive
Problems with Message Passing

- *Data inconsistency still a problem?*

- Yes, because messages can be
  - *out of order* \(\Rightarrow\) even if each message is consistent, the sequence of messages is not
  - *incomplete*, because receiver has not enough buffer space
  - *lost*
  - *outdated*, i.e. they no longer reflect the current state of the sender

- Each message can be an *information leak*, that’s why we must control whether messages should be transferred or not
Elementary IPC
Design of Message Passing

- Elementary communication (two threads)
  - 1 Sender and 1 Receiver

- Later on:
  - Higher communication level
  - Typical applications
  - IPC examples of current operating systems
**Orthogonal Design Parameters**

- Connection of communicators
- Synchronization
- Addressing
- Docking of IPC objects
- Ownership
- Organization of data transfer
  - Ordering of messages
  - Format of messages (size)
  - Buffering
  - Internal scheduling
- ...

---

2008 Universität Karlsruhe (TH), System Architecture Group
Connection

- Connection oriented

openConnection(address)
  Tests whether receiver exists and whether he/she wants a connection with the caller

send(message)

receive(message)

closeConnection()
  Empty message buffer and deletes connection
Connection

- Connectionless

send(target_address, message)
receive(source_address, message)

- Target is often a server
- Source is often a client
Synchronization of Sender

- Unsynchronized Send (Non Blocking)
  If receiver does not wait for message, skip message, continue

- Asynchronous Send (Non Blocking)
  If receiver does not wait for message, deposit message (if enough buffer place), continue

- Synchronous Send (Blocking)
  If no receiver waits for a message, deposit message, wait for receiver

In all cases:
If receiver already waits for message, transfer message, continue
Synchronization of Receiver

- **Non-Blocking Receive**  \textit{Void} if there is no message (test for arrival)

- **Blocking Receive**  \textit{Waits} if there is no message available

- In both cases, if message has been buffered, transfer message to receiver’s AS, \textit{continue}
## Combinations of Senders/Receivers

<table>
<thead>
<tr>
<th></th>
<th>Non-blocking Receive</th>
<th>Blocking Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsynchronized Send</td>
<td>- <em>bogus</em> -</td>
<td>Sender polling</td>
</tr>
<tr>
<td>Asynchronous Send</td>
<td>Receiver polling</td>
<td>Asynchronous communication</td>
</tr>
<tr>
<td>Synchronous Send</td>
<td>Receiver polling</td>
<td>Rendezvous</td>
</tr>
</tbody>
</table>

**Observation:**
As long as asynchronous sending is used we have to provide *message buffers* (in the communication link)
Enhanced Message Passing

Sender S synchronously sends message to receiver R

*What might happen to S due to many reasons?*

Receiver R can be down or has already finished

⇒ Sender S would wait forever

(another example for starvation ≠ deadlock)

*What to do?*

Enhance communication with a timeout mechanism
Timeout

With a timeout you specify how long you want to wait until a certain event should have taken place.

Assume: Even under heavy load your partner thread should have accepted your messages within $xyz$ ms.

⇒

Enhance your synchronous send operation as follows:

```plaintext
syncSend(receiver', message, xyz, result)
```

If receiver does not receive message within $xyz$ ms, sender can be informed via `result: "missing receiver"`. So it’s up to the sender how to proceed (if after all).
Addressing the Communication

- Direct Addressing
  - send (TID, message)
  - \textit{send}\((\text{filter}\{TID\}, \text{message})\)
  - receive(TID, message)
  - \textit{receive}\((\text{filter}\{TIDs\}, \text{message})\)

- Indirect Addressing
  - send (channel identifier, message)
  - send (port identifier, message)
  - ...

2008 Universität Karlsruhe (TH), System Architecture Group
Direct Addressing

- caller names partner thread explicitly or per wildcard:
  - \texttt{send(T, message)} send message to \texttt{T}
  - \texttt{receive(Q, message)} receive message from \texttt{Q}
  - \texttt{send(T1 \land T2, message)}
  - \texttt{send(T1 \lor T2, message)} different semantics
  - \texttt{receive(*, message)} receives message from any thread
Direct Addressing

- Properties of temporary communication link
  - Links are established automatically
  - Link is associated with exactly one pair of communicating threads
  - Link can be *unidirectional* or *bi-directional* depending on the communication pattern:
    - Notification
    - Request
Notification

- Notification is a *one way message* from sender to receiver
- *Message transaction* ends with message delivery to receiver
- Interpreting of the message in the receiver does no longer belong to IPC
Request

- Request is a *two way message* from sender to receiver
  - It starts by sending the request to the receiver and ends with an acknowledge (+ result of service) from receiver to sender
  - In the mean time receiver (~server) has delivered the required service
Problem with Asynchronous Send

Start denial of service attack

message queue within the kernel (e.g. as part of a mailbox)
Asynchronous via Synchronous Send

Question: Is there a correlation between # buffer slots and # of message deliverers?
Indirect Addressing

- Messages are sent to (and received from) mailboxes, ports, channels
  - Each mailbox has a unique id (e.g. MBID)
  - Threads can communicate only if sharing a mailbox

- Properties of communication link
  - A link established only if threads share a common mailbox
  - A link can be associated with many threads
  - Each thread can share several communication links
  - A link can be both unidirectional or bi-directional.
Indirect Addressing

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - attach and detach mailbox members
  - delete a mailbox

- Interface primitives
  - `send(MB,message)`: sends message to mailbox MB
  - `receive(MB,message)`: receives message from MB
  - `attach(MB,T)`: attaches thread T to MB
  - `detach(MB,T)`: detaches T from MB

You can enhance attach by additional access rights etc.
Indirect Addressing

- Mailbox sharing
  - $T_1, T_2,$ and $T_3$ can share a mailbox A.
  - Suppose: $T_1$ sends; $T_2$ and $T_3$ have previously invoked a receive at A.
  - Who will get the message?

- Possible Solutions
  - Type the message with an additional thread ID
  - Allow the system to select arbitrarily the receiver. Sender can be notified to which receiver the message has been delivered
  - High level communication patterns often build upon mailboxes
Advantages:

- Still easy to understand, but more flexible than direct addressing.
- Suited for arbitrary partnerships (s ≥ 1 sender, r ≥ 1 receiver).
- Each mailbox may provide an individual security policy.
- Mailboxes can survive threads.

Disadvantages:

- More spatial overhead due to extra data structure.
- Potentially one additional copy of the message.
- What to do with attached threads if mailbox owner deletes it? (Dangling thread problem)
- If a thread currently attached to a mailbox has to be aborted ⇒ problem of dangling messages.
Docking IPC Objects

Docking = relationship of the communicating threads with the communication facility, i.e. IPC-object:

- **Dynamically**, i.e. a thread can
  - Create a new mailbox
  - Attach to and detach from a mailbox
  - Delete its mailbox

- **Statically**, i.e. thread has its IPC-object (e.g. port) only during its life
Organization of Data Transfer

**Question:** Do we need to copy the message in each case?
Data Transfer of Messages

- Register (short messages, 0 copy)
  - Implications how to synchronize
- Shared memory (long message 0 copy)
  - Implications how to synchronize
  - Registers or kernel memory only used to transmit address(es) of message(s)
- Temporal mapping of message (1 copy)
  - Implications ...???
- Kernel Buffer (2 copies)
Data Transfer via Register

IPC Data Transfer

Hint: Discuss this proposal

Does it work for all variations?

Main advantages?

Main constraints?
Data Transfer via Shared Memory

sender
thread 1

receiver
thread 2

Only transfer the start address
Data Transfer via Mapping

thread 1

copy

reserved kernel region

map

thread 2

IPC Data Transfer
Data Transfer Via Kernel Buffer

Thread 1
- send
- copy_in

Thread 2
- receive
- copy_out

Kernel Buffer
Potential Formats of Messages

- **Message ID**
- **Message Type**
- **Message Length**
- **Control Information**
  - buffer overflow reaction
  - sequence numbers
  - priorities
  - queueing discipline: usually FCFS
  - ...
- **Sender and/or Source ID**
- **Receiver and/or Destination ID**
- **Message**
- **Message Body**
Non Contiguous Messages

Problem:
Message to be sent is scattered

Approach 1:
copy m1 … m4 into a buffer
send buffer to target R

Solution:
send(R, <a1, a2, a3, a4>)
Types of Communicating Activities

Homogeneous Communication
Heterogeneous Communication
Evaluate for your own
High Level IPC

Local Systems

Distributed Systems (see ST 2008)
Client Server Communication

- Local server
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Topic of the course DS

2008 Universitätkarlsruhe (TH), System Architecture Group
Synchronous IPC with a Server

Client

Request a service from a server

\{
  \text{SynchSend}(\text{server}, \text{request})
  \text{SynchReceive}(\text{server}, \text{result})
\}

Pro: No additional feature

Con: 2 system calls \(\Rightarrow\) more overhead

If dispatching takes place between these calls \(\Rightarrow\) server cannot deliver its result, it is delayed
Remote Procedure Call (RPC)*

Request a service from a server \( \text{RPC}(\text{Server}, \text{Request}) \)

**Pro:** Only 1 system call, requesting sender has to wait, needed in distributed systems anyway

**Con:** Additional feature

*In local systems this IPC is called LPC*
Indirect Communication

Different Semantics can be implemented:

- via mailbox, channel, port

Sender

Receiver
Indirect Communication (2)

1. Arbitrary Sender and Receiver

Sender

Mailbox

Receiver
Indirect Communication

2. Arbitrary Sender, all Receiver (Broadcasting)
Indirect Communication

3. All Sender, arbitrary Receiver (combined message)
Indirect Communication

4. All Sender, all Receiver (Broadcasting combined message)
IPC Applications
Mutual Exclusion with RPC

**Trick:** Use a specific thread to execute the critical section!

- **Client Ti:**
  ```
  var msg: message;
  repeat
    rpc(CSthread);
    RS
  forever.
  ```

- **CSthread:**
  ```
  repeat
    client := receive(any);
    CS;
    send (client, "done")
  forever.
  ```

**Hint:** Discuss the pros and cons of this solution.
Producer/Consumer with RPC

Producer:
repeat
  rpc (Buffer, produce())
forever.

Consumer:
repeat
  msg = rpc(Buffer);
  consume(msg)
forever.
Producer/Consumer with RPC

Buffer:

state = normal:
repeat
  (client, msg) = receive(any);
  if (client == Producer)
    then put
  elseif client = Consumer
    then get
  fi
forever.

put:
  insert (msg) ;
  if BufferFull()
    then state := ProducerPending
  else send (client,“ok”);
    if state = ConsumerPending
      then send (Consumer, msg);
        dummy := delete();
        state := normal
    fi
  fi.

get:
  if BufferNotEmpty()
    then msg := delete();
      send (client,msg);
      if state = ProducerPending
        then send (Producer,”ok”)
          state := normal;
      fi
    else state := ConsumerPending
      fi.
Mutex Emulation with IPC

create a mailbox `mutex` shared by n threads

receive() blocks if `mutex` empty

send() is non blocking

Initialization: send(`mutex, "go"`) 

The first Ti executing receive() will enter its CS.

Others will be blocked until Ti sends back msg.

```
thread Ti:
  var msg: message;
  repeat
    receive(mutex, msg);
    CSi
    send(mutex, msg);
    RSi
  forever
```
IPC Examples
Unix V IPC Mechanisms

To communicate data across tasks(processes):

- Pipes
  - Anonymous pipe
  - Named pipe
- Messages
- Shared memory

To trigger actions by other tasks(processes):

- Signals
- Semaphores
Pipes

- Two processes can transfer a byte stream in FCFS order
- Pipes are “kernel objects” of size 4KB or 64 KB (cyclic buffer) depending on the Linux Version
- Pipes can be used at the
  - kernel API within application programs
  - user interface level via the “|” pipe operator
- Implicit synchronization is done in case of a full respectively empty pipe, i.e. the producer will automatically stop writing to the pipe, when the pipe has become full
- Writing into a pipe without any reader raises an exception
Anonymous Pipes

- Can only be used by processes of the same family (e.g. parent and child)
- Typically they are used only in a unidirectional way
Example Pipe

Pipe operator at CLI:

$ more test.txt | lpr -kycera

The content of file \textit{test.txt} is sent via a pipe to the printer that will print out using printer kycera

$ set | grep PATH

The output of the \texttt{set} command will act as the input of the command \texttt{grep}. In this case all lines of the environment containing the character string \texttt{PATH} will be printed to standard output
Example Pipe at Kernel API (1)

```c
main()
{
    char buffer[5]; // buffer for received data
    int pp[2];     // descriptor for write end
                    // versus read-end of the pipe
    pipe(pp);      // create a new pipe pp
    if (fork()==0) { // child process as writer
        close(pp[0]); // close the read end of pp
        write(pp[1], "TEST", 5); // write to pp
        ...             // end of child process
        exit(0);        // end of child process
    }
}
```
Example Pipe Kernel API (2)

// now within the parent process
// acting as the reader
close(pp[1]);  // close write end of pp

read(pp[0], buffer, 5);
printf(" having read an item from pp: \%s\n, buffer);
...
}

Remark:
The close system calls are not necessary, but very helpful to prevent a consumer from writing to the pipe and vice versa
Named Pipes

- Can be used by non related process and in a bidirectional way (full duplex)
- Usual pipes are not persistent, i.e. they are deleted as soon as the last reader or writer is terminating
- A named pipe is an object of the file system and remains persistent, however its content is lost whenever the last writer terminates and there is no reader
- It can be reused in the future by any process that is authorized to access this named pipe
Example Named Pipe

For example, one can create a pipe `my_npipe` and set up gzip to compress things piped to it:

```bash
$ mkfifo my_npipe
$ gzip -9 -c < my_npipe > out.gz
$ rm rm my_npipe
```

In a separate process shell, independently, one could send the data to be compressed:

```bash
$ cat file > my_npipe
```

- Name pipes are often used to establish client-server relations

See: [http://developers.sun.com/solaris/articles/named_pipes.html](http://developers.sun.com/solaris/articles/named_pipes.html)
int mkinfo(const char *path, mode_t mode)

The system call function takes the *pathname* to establish at the related directory a "pipe file object" with all the access rights that can be defined according to mode_t.

A named pipe is used as a usual file, i.e. after having opend you can read or write to the named pipe.

Every write and read to a named pipe is atomic.
Overview: System V IPC Resource

- Processes request IPC Resources that will
  - be created dynamically
  - be persistent
  - be used by any process (who knows the key)
  - have a 32-bit **IPC key** that can be selected by the programmer
  - be identified unambiguously by a 32-bit **IPC identifier** determined by the kernel
Semantics IPC

- IPC messages are sent asynchronously
- No FCFS order within a message queue
- IPC messages are deleted, once they have been received, i.e. only one process can read a message
Send & Receive

- **msgsnd()**
  - IPC identifier of the target message queue
  - Size of message
  - Address of a user mode buffer

- **msgrcv()**
  - IPC identifier of the source queue
  - Pointer to a user mode buffer as the target
  - Size of buffer
  - Type t determines the message type, the caller is interested in
MSGGET

msgget() resource

IPC Key

Resource with key associated?

all ok?

yes

no

create new resource

IPC identifier

error code

no

yes
How to share a Message Queue?

1. Fixed, predefined IPC key
   - Simple case, works also for complicated applications
   - IPC key might be used by any process

2. Set IPC key = IPC_PRIVATE
   - IPC resource can not be used by another process
   - IPC identifier has to be sent to another process before it can use the IPC resource
System V IPC

- Kernel manages a message queue
- Sender processes can send messages to it
- Receiver processes can receive messages from it

```c
int msgget(key_t key, int msgflag)
```

**key** is used to identify unambiguously the related message queue.

The return value is either -1 in case of an error, or the message queue id.

**msgflag** is used to specify what to do in case the message queue already exists.
MSGflags

- **IPC_CREAT**
  
  resource *msgqueue* must be created, if not yet done

  if not set msgget simply returns the msg identifier

- **IPC_EXCL**
  
  `typeget()` schlägt fehl, wenn die Resource bereits existiert und **IPC_CREAT** gesetzt ist.
Implementing Message Queues

```c
struct msg_queue
    msg_ids.entries

struct msg_msg
    n_list
    q_messages
    Message text

struct msg_msgseg
    next
    Message text
```

2008 Universität Karlsruhe (TH), System Architecture Group
Unix Typed Messages

- In `receive()` receiver specifies that it is only interested in a message of specific type.

- The message type is either defined in the message at a specific location or it is a parameter of `send()`.

Example: `receive(mb1, blue_letter)`
IPC of L4*

Characteristics of L4 IPCs:

- Synchronous
- Direct addressing
  - `send(tid, message)`
  - `receive(tid, message)`
  - `receive(from any, message)`
  - `call(tid, request)`
  - `reply&wait(tid, answer)`