Sistemas Operativos: Concurrency (Part 4)

Pedro F. Souto (pfs@fe.up.pt)

April 9, 2012

Agenda

Agenda

Synchronization Barriers

Messages

Remote Procedure Call (RPC)

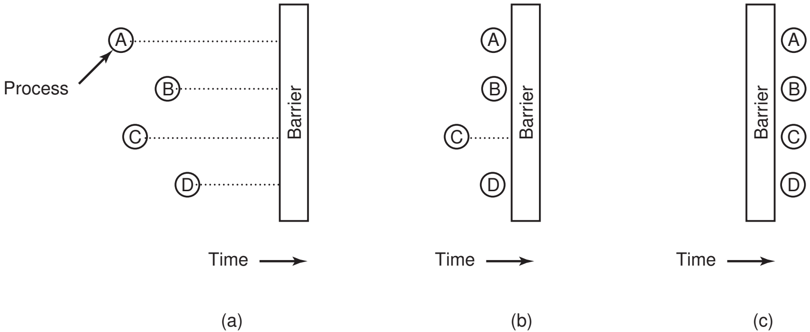
Communication Channel Properties

Message-based Communication in Unix/Linux

Additional Reading

Synchronization Barrier

- This allows the synchronization of *n* processes/threads rather than just 2 threads



source: Modern Operating Systems, 2nd. Ed.

- Processes that arrive at the barrier are made to wait until all other processes of the group arrive there
- Unblocking is performed automatically, when the last process arrives at the barrier

- Particularly useful in parallel programs that proceed in phases

Synchronization Barrier: POSIX API

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
            const pthread_barrierattr_t *restrict attr,
            unsigned count); // number of threads in group
int pthread_barrier_destroy(pthread_barrier_t *barrier);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

- `count` "specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call."
- Apparently not available in Linux

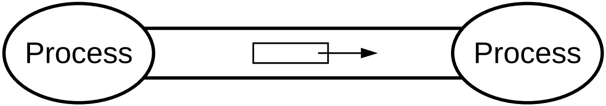
Agenda

Comnunication (IPC) with *Messages*

Problem The synchronization mechanisms considered so far (locks, semaphores, monitors and condition variables) cannot be used in systems where threads/processes do not share memory – for example, when they run on different computers

Solution Use *messages*:

processes/threads send/receive messages via a communication channel:

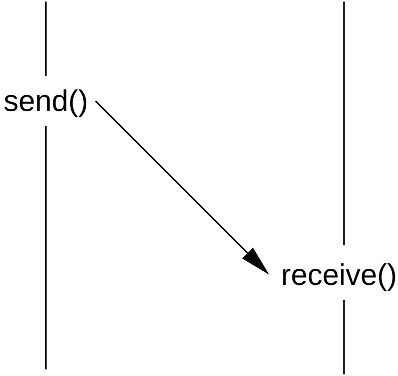


- Uma *mensagem* é uma sequência de bits **indivísivel**:
  - O formato e o significado duma mensagem são especificados pelo *protocolo de comunicação*.

Communication Primitives with Messages

- Message-based IPC uses mainly 2 primitives:

```
send(destination, &message)
receive(source, &message)
```



- Often, `source` can be a special value, say `ANY`, which allows a process to receive a message from any process

Message-based IPC: Semantics and Application

- Surprisingly, there are many variations on this theme:

  naming: what are `destination` and `source`? Process names? Names of channel endpoints?

  sinchronization: do the sender/receiver synchronize/block on message send/receive?

  buffering: does the channel store the messages?

  channel properties: loss of messages? duplication of messages? order?

- Message-based IPC is typically studied in computer networks and distributed systems courses
- Message-based IPC can also be used in systems that share memory. E.g.:
  - in parallel systems: MPI (Message Passing Interface);
  - between processes/threads executing in the same computer, or even processor

Message Passing: Naming

Direct Naming i.e. `send()` and `receive()` specify processes identifiers

- To simplify the management of process identifiers, often these are local to each computer
  - In this case, one needs also to specify the computer on which the process runs, e.g. the computer IP address
- Not a very flexible solution:
  - If processes are not always assigned the same identifier, applications must find the identifier the process has

Indirect Naming i.e. `send()` and `receive()` specify the "name/address" of the channel end

- Known either as **mailbox** or **port**
- In this case, the processes may have to associate themselves with the name/address of the port
  - If this is not known *a priori*, some way to look it up must be provided

Message Passing: Synchronization/Blocking

Asynchronous communication The sender does **not** synchronize with the receiver

- `send()` never blocks, and the sender can proceed immediately after it returns from `send()`
- The receiver blocks on `receive()` if the channel has no message

Synchronous communication The sender **synchronizes** with the receiver

- `send()` blocks, if the receiver is not ready to receive
- The receiver blocks on `receive()`, if the sender is not ready to send

Message Passing: Buffering

- Buffering refers to the ability of the channel to store messages in transit
  - It is related to the capacity of the "communication channel"
- Not an issue in synchronous communication
  - Conceptually, in synchronous communication there is no need for "buffering"
  - In practice, it is usually not possible to copy a message directly from the sender to the receiver, thus there is usually some buffering
- In the case of asynchronous communication there must be some buffering. Conceptually, the buffer can be

  Unbounded

  Bounded In this case, if the buffer is full, when the sender tries to send a message, either:

  1. The sender must block, or
  2. The message will be lost

Agenda

- Synchronization Barriers
- Messages
- Remote Procedure Call (RPC)
- Communication Channel Properties
- Message-based Communication in Unix/Linux
- Additional Reading

Remote Procedure Call (RPC)

Problem Message based programming with `send()/receive()` is not very convenient

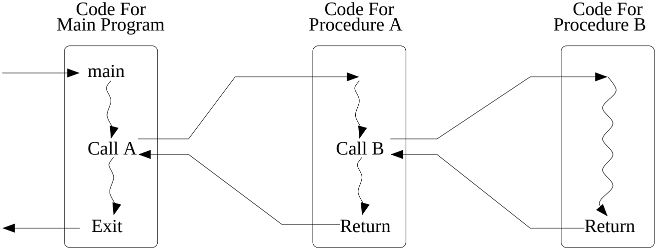
- depends on the properties of the communication channel
- requires the specification of an application protocol
- resembles I/O programming

Idea invoke functions to be executed in remote computers

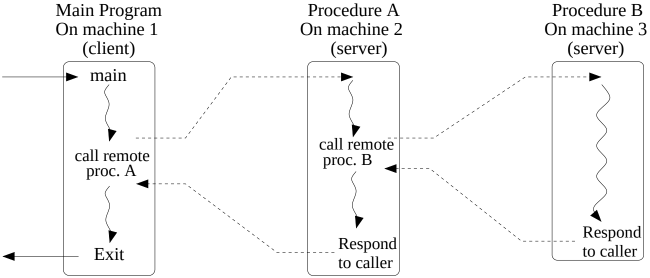
- familiar paradigm
- facilitates transparency
- particularly convenient for client-server applications

RPC: The Idea

Local Calls



Remote Calls



RPC: Some Remarks

- RPC is a higher-level of abstraction primitive than pure message passing
  - It is language-based (like monitors) rather than OS-based
  - It encapsulates a request message and a response message
- RPC is a kind of synchronous communication
  - The caller blocks until the callee returns
- Some variations on the same theme:

  Rendezvous: RPC in which there is no concurrent execution of calls
  - Remote calls are performed sequentially, hence in mutual exclusion

  Remote Method Invocation (RMI): RPC for objects
  - Invocation of methods (an object's operation) of objects that run in a different address space
  - Very common since Java RMI (and C# Remote)

Agenda

Properties of a Communication Channel

- Connection-oriented vs. connectionless
- Reliablility: loss and duplication
- Order
- Abstraction: message-based vs stream-based
- Flow control
- Directionality and number of channel ends
- Identification of the communication entities

Connection-oriented *vs.* Connectionless

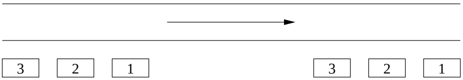
Connection-oriented: the processes must set up the communication channel before exchanging data – like phone communication;

Connectionless: the processes may exchange data immediately without previously setting up a communication channel – like standard (and electronic) mail communication

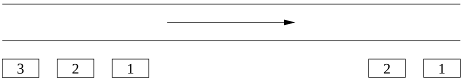
Reliability (loss)

Reliable: ensures that the data sent is received by the receiver

- under certain assumptions
- otherwise, it notifies the communicating processes

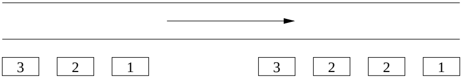


Unreliable: it is up to the communicating processes to detect any communication problem and to take the appropriate actions, if any

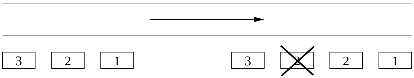


Reliability (duplication)

“Generates” duplicate: the channel may deliver duplicate messages to the destination process(es) – it is up to the latter to detect and discard them (if relevant)

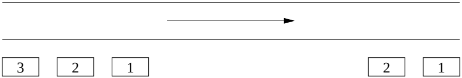


No duplicates: the channel never delivers duplicate messages to the destination process(es)



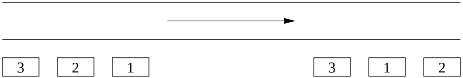
Order

Ordered: ensures that the data is delivered to the destination in the order in which it was sent



Unordered: does not ensure data is delivered in the order it was sent

- If maintaing the order is important it is up to the application to ensure it

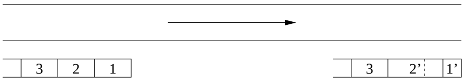


IMP order and reliability are orthogonal properties

Communication Abstraction

Message (datagram): the channel preserves message boundaries – sequence of *bytes* processed as atomic entities: analogous to standard (and electronic) mail

Stream: the channel does not preserve message boundaries. Essentially it operates as a pipe for bytes: analogous to other I/O streams.



Other Charateristics of the Communication Channel

Flow control: prevents “fast” senders from flooding “slow” receivers

- Senders do not necessarily have to be more powerful than the receivers to be “faster”

Number of processes on the receiving end

unicast only one receiver
broadcast all processes in a “universe”
multicast a subset of processes in a “universe”

Directionality whether it can be used to exchange data in a single direction or bidirectionally

Agenda

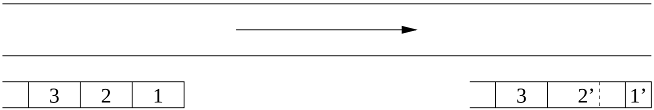
BSD Sockets API

- A socket can be seen as an end of a communication channel
  - An object to which one can plug several channels
- It specifies functions to `send()`/`receive()` messages to/from sockets, among other functions
- It is very flexible, hence somewhat complex
  - It was designed in the early 1980's
  - The Java API, designed about 10 years later, is much simpler
- It is used virtually by all Internet applications
  - Windows' Winsock API is based on BSD sockets
- It is usually studied in computer networks and distributed system courses
  - But it can also be used for communication among processes on the same computer

Pipes

Pipe Unidirectional channel used for the communication of byte streams between processes

- Can be thought of as a FIFO buffer to wich the sender process may send bytes and from which the receiver process may receive bytes
- Pipes do not preserve message boundaries



Pipes: API

```
#include <unistd.h>
```

`int pipe(int fd[2])` creates a pipe

- Upon success, `fd[]` is initialized with the file descriptors of the two ends of the pipe:

  `fd[0]` is the receive/read end of the pipe

  - Receiving is via the `read()` system call

  `fd[1]` is the send/write end of the pipe

  - Sending is via the `write()` system call

- Pipes can be accessed only via these descriptors.
- Pipes can be used only for data exchange among:
  - The process that created it, by invoking `pipe()`
  - Its descendats, that inherit the descriptors via `fork()`
- A process should close the end of the pipe that it does not use

Pipes: Code Fragment

```
/* IMPORTANT: No error checking */
#include <unistd.h>
void main(void) {
    int fd[2];  // For the pipe's end

    pipe(fd);
    if( fork() == 0 ) {  // child is sender
        close(fd[0]);    // close read end
        ...              // use write(fd[1],...) to send
    } else {             // parent is reader
        close(fd[1]);    // close write end
        ...              // use read(fd[0],...) to receive
    }
}
```

- Not sure which is the read and which is the write end? Use macros:

  ```
  #define PIPE_RD_END 0
  #define PIPE_WR_END 1
  ```

- How can we use pipes for bidirectional communication?

Pipes and Filters

- Pipes allow the development of powerful transformations by pipelining **filters**

  Filter Program that reads from its standard input, performs some transformation, and outputs to its standard output

  - Many useful Unix programs, such as `head`, `sort`, `grep`, `sed` and `awk` are designed as filters to support their use in pipelines
- For example:

  ```
  ps -ax | wc -l
  ```

  counts the number of processes running on the system
- Implementing filter pipelines requires:
  - Creating pipes for communication between filters
  - Redirecting the standard input/output to the appropriate end of the pipes

Pipes: Filter Example

```c
#include <unistd.h>

int main(void) {
   int fd[2];
   char *ps_args[] = {"ps", "ax", NULL};
   char *wc_args[] = {"wc", "-l", NULL};

   pipe(fd);

   if ( fork() != 0 ) { // ps is parent, but need not be
       close(1);         // redirect stdout to
       dup(fd[1]);       //  the write end of the pipe
       close(fd[0]);     // first filter does not read pipe
       execve("/bin/ps", ps_args, NULL);
   } else {              // wc is child, but need not be
       close(0);         // redirect stdin to
       dup(fd[0]);       //  the read end of the pipe
       close(fd[1]);     // second filter does not write pipe
       execve("/usr/bin/wc", wc_args, NULL);
   }
   return 0;
}
```

FIFOs (Named Pipes)

- The pipes API can be used for communication only between processes that are descendants of the process that created it
  - Pipes can be accessed only using the file descriptors returned by `pipe()`
  - File descriptors can be passed to other processes only via `fork()`
- FIFOs are like pipes but are named/identified by a pathname in the file system
  - Can be used for communication between any pair of processes with the appropriate permissions (OS dependent)
- Access to FIFOs is done as for any file:
  - Using `open()`/`close()`, and `read()`/`write()` syscalls
  - But data exchange with FIFOs needs not be via any permanent storage media
- Creation of a FIFO uses a function different from `open()`:

  ```
  int mkfifo(const char *pathname, mode_t mode);
  ```

  Alternatively, you can use the `mknod()` syscall

Agenda

Additional Reading

Sistemas Operativos

- Subsection 6.5.5 – Barriers
- Sections 10.1, 10.2, 10.3.1 – Messages

Modern Operating Systems, 2nd. Ed.

- Subsections 2.3.8 and 2.3.9 – Message Passing & Barriers
- Section 8.2 – Multicomputers (Messages and RPCs)

Operating Systems Concepts, 7th. Ed.

- Subsection 3.4.2 – Message Passing Systems
- Subsection 3.6.2 – Remote Procedure Calls