Karlsruhe Institute of Technology

Operating Systems 2016/17
Tutorial-Assignment 6

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 6.1: Interprocess Communication (IPC)

a. How can concurrent activities interact in a (local) system?

Solution:
*The two models of IPC are **shared memory** and **message passing**. Concurrent activities can also communicate by other means. Some examples of communication facilities are:*

- *Shared memory*
  implicitly: *Same address space (e.g., another thread of the same process)*
  explicitly: *Shared memory area (e.g., between processes with different address spaces)*
- *OS communication facilities (messages (IPC), pipes, sockets)*
- *High-level abstractions (files, database entries)*

b. Why does it make sense to define a timeout for IPC operations?

Solution:
Servers may define a timeout to react to clients which died or maliciously never answer. These clients would otherwise be hogging resources in the server infinitely.

c. Asynchronous `send` operations require a buffer for sent but not yet received messages. Discuss possible locations for this message buffer and evaluate them.

Solution:
Message buffers can be located in the receiver's address space, in the kernel, or in the sender's address space.

receiver AS: *The receiver can designate a finite region of its address space as a message queue. However, if the number of clients rises, the number of messages sent to this receiver might increase, thus requiring an ever larger receive area. If the receive area is not large enough, many IPCs will either fail and be repeated (doubling the IPC overhead) or block the sender, turning the asynchronous send undesirably into a synchronous one. Receive buffers do not scale well.*

kernel: *The kernel can allocate message buffers on demand for each asynchronous message that cannot be delivered instantly. However, malicious or bogus threads might flood the kernel with messages, thus we would need to impose limits on the message buffer size with consequences similar to the receiver-based buffers.*

Kernel-based buffers also scale poorly and can provide opportunities for denial-of-service attacks if not implemented properly.

sender AS: *Keeping the message in the sender's AS is a viable option. The sender has stored the to-be-sent message anyways, so we can use its current location as a message buffer. If we use the original message memory as message buffer, the application needs to be notified once the message has been consumed by the receiver so that the memory can be released or reused.*

Alternatively, we might provide a separate sent-message buffer and copy the message there iff an asynchronous send cannot deliver the message instantaneously. This approach would allow the application to overwrite the original message memory at the cost of requiring an (additional) copy of the message.

Sender-based buffering scales well, as only the sending processes pay for their communication requirements; applications that seldom communicate with other threads will never block due to insufficient buffer space.

d. Consider a system that uses synchronous message passing and timeouts to detect/recover from non-responding communication partners. Discuss why the system designers might choose to provide an atomic `send-and-receive` system call in addition to separate `send` and `receive` calls.

Solution:
In a common remote procedure call (RPC) scenario, where a client invokes a procedure in the server, the single send-and-receive scheme requires only one system call invocation for the client.

More importantly, since the server does not necessarily trust the clients, it will have to protect itself from malicious or faulty clients by replying to the RPC with a timeout. Otherwise, (1) the server might block forever, waiting for the client to accept the response and (2) the server has to buffer the reply message for the whole time, which possibly depletes server resources (remember we concluded in the previous question that we want to use sender-based buffering–for the reply the server is the sender).

Now, if the server knows that there is an atomic send-and-receive system call available to the client, it can assume that a correctly behaving client used this call and will as such be instantly waiting to receive the server's reply. The server can therefore specify a 0-timeout for the reply.

This both removes the risk of denial-of-service *attacks against the server and relieves the system and server of the burden to setup timeouts and buffer the reply message.*

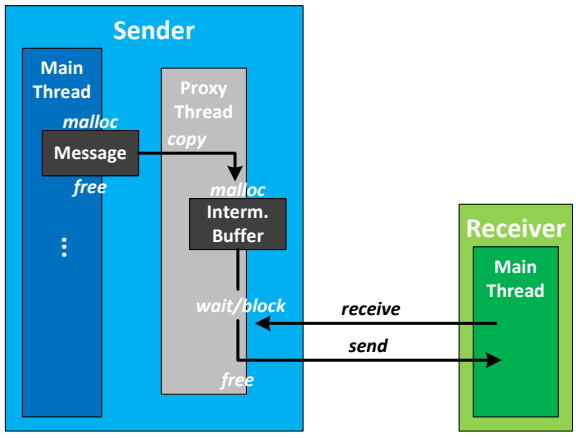
Question 6.2: Emulation using IPC

a. How can you perform asynchronous interprocess communication if your operating system only provides synchronous IPC mechanisms?

Solution:

The idea is to use a separate proxy thread for delivering the asynchronous message via synchronous IPC. If the receiver is not waiting for reception, the proxy will block on the send operation. In any case, the original thread can continue execution, because it is not responsible for transmitting the message.

The message may be pending for a long period of time. Therefore, it can be stored in an intermediate buffer in the sender's address space, which is managed by the proxy thread.



The thread receiving an asynchronous message queries for pending messages. The receiver can potentially block on the proxy thread if no message is to be transmitted.

This scheme requires one proxy thread per outstanding asynchronous message. It is therefore a good idea to limit the number of outstanding messages and/or combine the approach with timeouts to protect against malicious receivers.

b. How can you provide synchronous IPC if your operating system only offers asynchronous IPC mechanisms?

Solution:
Asynchronous IPC includes asynchronous `send` *and non-blocking* `receive` *operations. If you want the sender to stop execution until the receiver has accepted the message, you could come up with a solution such as:*

```
do {
    res = async_send(msg, receiver);
} while(res != SUCCESS);
```

However, this would keep the sender busy only until the messaging system accepts the message. However, the sender does not know if the message arrived at the receiver, already—`send` *does not return status information of the receiver. As a consequence, the sender would be released too early using the above scheme: after the message system accepted the message but most likely before the intended receiver got the message.*

The correct solution establishes a software protocol. The receiver must acknowledge each message with a message back to the original sender. Of course, ACK messages must not be acknowledged to avoid endless message loops. Note that this approach requires bidirectional communication!

```
function sync_send(in msg, in receiver) {
    async_send(msg, receiver);

    do { /* wait for the ACK message */
        res = async_receive(msg, receiver);
    } while((res != SUCCESS) || !isAck(msg));
}

function async_receive_with_ack(out msg, out sender) {
    res = async_receive(msg, sender);    // receive a (pending) message

    if((res == SUCCESS) && (!isAck(msg)) {
        async_send("ACK", sender);       // acknowledge reception (if any)
    }
}
```

Emulating synchronous receive using asynchronous receive is more intuitive:

```
function sync_receive(out msg, out sender)  {
    do { /* wait for the message */
        res = async_receive(msg, sender);
    } while(res != SUCCESS);

    if (!isAck(msg)) {
        async_send("ACK", sender);       // acknowledge reception
    }
}
```

Question 6.3: Pipes

a. Write a C-program for Linux that creates two child processes, `ls` and `less` and uses an ordinary pipe to redirect the standard output of `ls` to the standard input of `less`.

Solution:

```
#define READ_END    0
#define WRITE_END   1

int main( void )
{
    int pid, pipefd[2];
    /* Create pipe. Parent has write and read end open*/
    int p = pipe(pipefd);

    if((pid = fork()) == 0) {
        /* first child, gets write end of pipe */
        dup2(pipefd[WRITE_END], STDOUT_FILENO); // Close stdout and replace
                                                //  with write end.
        close(pipefd[WRITE_END]);               // One duplicated, one not needed
        close(pipefd[READ_END]);

        execlp("/bin/ls", "ls", NULL);          // Execute ls, which writes to
                                                //  (replaced) stdout
    } else {
        /* parent, forks a second child */
        if((pid = fork()) == 0) {
            /* second child, gets read end of pipe */
            dup2(pipefd[READ_END], STDIN_FILENO); // Close stdin and replace
                                                  //  with read end
            close(pipefd[WRITE_END]);             // One duplicated, one not needed
            close(pipefd[READ_END]);

            execlp("/bin/less", "less", NULL);    // Execute less, which reads
                                                  //  from (replaced) stdin
        } else {
            /* parent, pipe fds must be closed so that 'less' gets an EOF */
            close(pipefd[READ_END]);
            close(pipefd[WRITE_END]);

            /* wait for both children to exit */
            wait(NULL);
            wait(NULL);
        }
    }
}
```