Appl. Math. and Comp. Sci., 1995, vol.5, No.1, 189–208

STRUCTURING THE MESSAGE-PASSING CHANNELS[†]

STANISLAW CHROBOT[*], AGATA STRAS[*], ROBERT STRAS[*]

Low-level primitives for a high-level language and protocols in this language are proposed to implement synchronous message passing for a set of distributed processes running on a set of single-processor multiprocessing nodes. This implementation is meant to explicitly express process scheduling and message dispatching algorithms in the high-level language. The link is proposed as a low-level one-to-one asynchronous communication channel able to buffer a simple data entity. Protocols are presented for using links in synchronous message passing. These protocols use interrupts as a main synchronisation tool. This decreases the context switching and process scheduling overhead considerably. On the top of these protocols, many-to-one message channels are implemented.

1. Introduction

The art of programming with interrupts has its own specifics. It differs from sequential and concurrent programming. Accepting an interrupt invokes an action (interrupt handler) asynchronous to the running process. The interrupt handler seems like a regular subroutine, but the programmer cannot foresee when such a subroutine will be invoked in the course of process execution. This is why the interrupts are usually hidden inside the operating system kernel which provides the user with safe and convenient synchronous IO operations and message-passing primitives.

Recently, interrupts have become available at the user level in the form of signals or exceptions generated by the kernel. *Signals*, implemented in (*SunOS*, 1990), can be sent to heavyweight (UNIX-like) processes. *Exceptions*, implemented by Jagannatan and Philbin (1994), can be sent to lightweight threads. Such signals give the user the possibility of designing reactions for events like arithmetic exceptions, interrupts generated by user requests, or completion of non-blocking IO operation.

A close relationship between interrupts and message passing primitives is reflected in many distributed languages which allow high-level interrupt handling (*Occam*, 1988; *ADA*, 1983; Gehani and Roome, 1986; Andrews *et al.*, 1988). Using Ada terminology, "an interrupt acts as an entry call issued by a hardware task" (*ADA*, 1983). Interrupt handling is considered a rendezvous between the device process and the server process.

Rendezvous is widely used in the client-server model of distributed programming. The server process defines a set of operations, called *entries* or *transactions*, which can be called remotely by client processes. A transaction is executed after it has been

[†] This work was supported by Kuwait University under the grant number SM 087

[*] Kuwait University, Department of Mathematics, P.O. Box 5969 Safat, 13060 Kuwait

called by a client and *accepted* by the server. The server can wait for more than one transaction call at a time using the *select* statement. When more than one transaction is acceptable, the server selects one of them in an arbitrary way. To be always ready for accepting the client call stream, the servers usually execute the *select* statement in a loop.

Such a model is a source of significant process management overhead. To analyse it, let us assume that the server is waiting for a client. When the client's message arrives, the client is blocked, and the server is resumed. *This requires a context switch and process rescheduling.* When the transaction is completed, the server is blocked, and the client is resumed. *This requires another context switch and process rescheduling.* It is worth stressing that this kind of overhead is extremely expensive on RISC processors where the context switching needs reloading the register windows.

Let us note that such overheads do not appear when the buffer is implemented by a monitor. If a process calls a monitor when its critical region is not occupied, neither context switch nor process rescheduling is needed.

Another source of overhead is the guard evaluation. The server has to re-evaluate the guards every time the *select* statement is started. This leads to a semi-busy form of waiting. In monitor, the synchronisation conditions are evaluated when there is a need, before a potential waiting or signalling.

In this paper we suggest that both the process management and the guard evaluation overheads can be reduced considerably, if the interrupt feature is used as a regular low-level synchronisation mechanism for message passing IPC. Ada entries or Concurrent C transactions can be considered interrupt-driven not only when called by a hardware process, but when called by any software process as well. On top of the interrupt feature, other high-level message passing primitives can be built in a structured way, much the same as the shared memory primitives are built on top of the spin lock (Chrobot, 1994; Wirth, 1985).

Both the interrupt feature and the spin lock are hardware supported features. Both of them are one-out-of-many selection mechanisms. In case of *spin lock*, many processes compete, in a loop, for a signal deposited in a shared bit. The process that finds the signal set, resets it, and passes the selection operation (traditionally called Test-and-set). The hardware support for this kind of selection is called *memory arbiter.*

In case of *interrupt*, one process checks, in a loop, many signal bits. Each of them is associated with a process to be selected. A process is selected when it has sent a signal to its signal bit and when the selecting process has accepted this signal. The hardware support for this kind of selection is called an *interrupt system.* The processor plays usually the role of the selecting process. The signal bits are called *interrupt sources*, and the processes associated with the signals are called *interrupt handlers.*

The spin lock can be considered a prototype of the *mutual exclusion* mechanism, while the interrupt a prototype of the *rendezvous* mode of execution. Two aspects of our approach are worth stressing. We suggest that

- high-level inter-process communication (IPC) primitives can be built on top of low-level concepts in a systematic way,
- the low-level message passing IPC concepts are related to the interrupt feature; which means that neither spin lock nor monitor need to be used to structure the high-level message passing primitives.

The advantages of our approach follow from both the above mentioned aspects. As far as the first aspect is concerned, including the low-level primitives into a high-level programming language allows the user to

- express both the process scheduling and message dispatching algorithms at the application level, so queuing disciplines in these algorithms can be tailored to the application needs,
- choose different kinds of message passing primitives for different applications according to the nature of his application.

As far as the second aspect is concerned, using the interrupt feature to structure the message passing mechanism yields significant efficiency gains. It reduces

- the process management overhead related to context switching and process scheduling,
- the overhead related to the evaluation of guards (synchronisation conditions) which control the acceptance of the messages incoming to the server.

In this paper we will focus on the rendezvous type of message passing and will illustrate our discussion with the bounded buffer example. The paper consists of 7 sections. In the two main sections, Section 2 and Section 3, we introduce the low-level concepts: link, coroutines, and needles. In Section 4 we present how they can be used to build structured channels. Our main interest is in the mailbox as a part of a multi-client server. In Section 5 we present the directions for further work. Section 6 describes related research and Section 7 is the conclusion.

2. Links and Related Protocols

2.1. The Link Concept

We introduce the link as an elementary message passing concept. The *link* is a type of asynchronous, uni-directional, point-to-point communication entity for transmitting values between processes. The link can store a simple data entity (word or pointer) and a signal bit. The link, however, is not a one-capacity bounded buffer. The stored entity can be overwritten if it has not been read early enough. Processes access a link through its ports: *link output port* and *link input port*. A link l associated with an output port op and an input port ip will be denoted by $l(op, ip)$.

A link output port op is accessible through its three output operations: LPUTW(op, w) or LPUTP(op, p) and LSIGNAL(op). The LPUTW(op, w) operation writes the word (*int*) value w to the link l associated with the port op, and LPUTP(op, p) writes the pointer (*void**) value p to the link l associated with the port op. The value w or p is stored in the link l until the next LPUTW( ) or LPUTP( ) operation to the same link overwrites it. LSIGNAL(op) sets a signal in the

link l (assigns value 1 to the signal bit). The signal is stored in the link l until it is reset (value 0 is assigned to the bit signal) by the LWAIT(ip) operation accessing the input port ip associated with the link l.

A link input port ip is accessible through four operations: LWAIT(ip), LTEST(ip), LGETW(ip, x) and LGETP(ip, px). LWAIT(ip) busy-waits until the signal bit in the link l associated with ip is set and then resets it. We will say that a process has accepted a signal from the port ip when it has reset its signal bit. LTEST(ip) is a Boolean function that returns *true* when the signal bit in the link l is set; otherwise it returns false. LGETW(ip, x) reads the word value stored in the link l and assigns it to the variable x. LGETP(ip, px) reads the pointer value stored in the link l and assigns it to the variable px. To avoid undefined results, we assume that a given link is used either to transmit words (*word link*) or to transmit addresses (*pointer link*).

Links can be used to pass words or addresses between processes running on the same processor (local links) or on different processors (remote links). The sender and the receiver of a local link can be the same process.

It is obvious that the link is not a typical asynchronous channel. Such a channel, to be safe and reliable, usually includes unbounded buffering and transmission protocols. From our point of view, such a channel is too heavy to be used as a primitive concept to build other channels; we need a simpler concept to express both the buffering and the protocols on top of it.

2.2. Simple Transmission Protocols

Well-defined results of the transmission down a link are achieved if the sender and the receiver apply the same *transmission protocol*. In this section we analyse protocols that transmit simple values of the word or pointer type. We call such protocols *simple transmission protocols*. Later on, we will analyse protocols to transmit structured values – messages.

Transmitting more than one word requires engaging two links: data link $d(opd, ipd)$ and signal link $s(ops, ips)$ and using them according to the protocol:

a) for each word transmitted, the sender puts the word w into link d, signals down this link and waits for a signal from the link s

$$\text{LPUTW}(opd, w); \quad \text{LSIGNAL}(opd); \quad \text{LWAIT}(ips)$$

b) for each word received, the receiver waits for a signal from the link d, gets a word from this link and sends a signal down the link s

$$\text{LWAIT}(ipd); \quad \text{LGETW}(ipd, x); \quad \text{LSIGNAL}(ops)$$

The above protocol will be called the *simple acknowledgement protocol*. The link d will be called *data link*, while the link s will be called *signal link*.

There is an alternative protocol where the link s is used by the receiver to send a request signal. We will call this protocol the *simple request protocol*:

a) for each word sent, the sender waits for a request signal from the link s, and then puts the word w into the link d and signals down the link d

$$\text{LWAIT}(ips);\quad \text{LPUTW}(opd, w);\quad \text{LSIGNAL}(opd)$$

b) for each word received, the receiver sends a request signal down the link s, waits for a signal from the link d and gets a word from d into variable x

$$\text{LSIGNAL}ops);\quad \text{LWAIT}(ipd);\quad \text{LGETW}(ipd, x)$$

In the above protocols the processes use a data link and a signal link. The signal link is used for strict synchronisation purposes. We call such protocols *simple synchronous protocols.*

Another kind of protocol uses two data links: $l1(opl1, ipl1)$ and $l2(opl2, ipl2)$. To exchange two words (request and replay) between the sender and the receiver, we use the following protocol:

a) for each request, the sender puts the request word w into link $l1$, signals down this link, waits for a signal from the link $l2$ and gets a replay word from this link into variable $x1$

$$\text{LPUTW}(opl1, w);\quad \text{LSIGNAL}(opl1);\quad \text{LWAIT}(ipl2);\quad \text{LGETW}(ipl2, x1)$$

b) for each request, the receiver waits for a signal from the link $l1$, gets a request word from this link to the variable $x2$, puts the replay word r in the link $l2$ and sends a signal down this link

$$\text{LWAIT}(ipl1);\quad \text{LGETW}(ipl1, x2);\quad \text{LPUTW}(opl2, r);\quad \text{LSIGNAL}(opl2)$$

The receiver can execute any operation R after getting the request $x2$ and before putting the replay r. If the operation R takes $x2$ as its input parameter and returns r as its output parameter, it is called a *simple rendezvous operation*, and the protocol is called a *simple rendezvous protocol.*

$$\text{LWAIT}(ipl1);\quad \text{LGETW}(ipl1, x2);\quad R(x2, r);\quad \text{LPUTW}(opl2, r);\quad \text{LSIGNAL}(opl2)$$

The simple synchronous protocols can be considered specific cases of the simple rendezvous protocol if

- the receiver using the acknowledged protocol executes the operation $R(x)$ (R takes x as its input parameter) after getting the value x from its data link and before sending the acknowledgement down the signal link or
- the sender using the request protocol executes the operation $R(w)$ (R produces w as its output parameter) after getting the request signal and before putting the value w into its data link.

For the above three protocols, the process which executes the rendezvous operation is called the *server process.* Its partner at the other end of the channel is called the *client process.* A pair of links working in a simple protocol will be called a *simple channel.*

Using the LWAIT operation binds the server to one link. The server can, however, transmit data entities through many channels alternately using the LTEST operation to monitor their input ports in a selective loop. Such protocol applied by the server will be called *selective acceptance protocol.*

3. Using Interrupts in the Simple Protocols

3.1. Busy and Non-busy Waiting in the Protocols

The LWAIT( ) operation and the selective acceptance protocol presented above use the busy form of waiting. The server continuously tests its input ports until the link signal is set in any of them. Such a form of waiting is acceptable if each server runs on its own processor. If the server and other processes are multiplexed on one processor, this solution is not efficient. The non-busy form of waiting is usually used for such multiplexed processes.

Typically, the non-busy form of waiting for the processes which share common memory is implemented by blocking the waiting process inside a monitor. The monitor solution for the selective acceptance protocol is as follows:

- the server is blocked on a monitor condition if none of the input ports is ready for acceptance; the first client which sends an entity to an unmasked channel resumes the blocked server which executes the rendezvous operation and replays to the client,
- the client is blocked on a monitor condition associated with its channel if the server is not ready to accept its message; when the server is ready, it selects one of the clients blocked on one of the unmasked channels, processes its message, returns the replay, and resumes this client.

The monitor implementations of the synchronous channel and of the selective acceptance protocol are both very common. Shared memory implementations for one-to-one, many-to-one and many-to-many channels are well-known (Brinch, 1987; Gehani and Roome, 1986; *Occam*, 1988). Unfortunately, the shared memory implementation does not work for the processes running on different computers interconnected with a network. However, even in such configurations the monitor implementation usually dominates. The operating system kernel treats the network as a peripheral device using interrupt signals for synchronisation purposes and input/output registers to transmit data to/from one device at a time. This way of handling devices corresponds roughly to our simple one-to-one channels. The acceptance protocol and other types of channels and protocols, however, are implemented on top of such simple one-to-one channels using monitors.

Below we will show that this transformation from the interrupt paradigm to the monitor paradigm is not necessarily needed. The higher level protocols and channels can be implemented in the interrupt paradigm only. As a presentation language we have chosen Modula-2 since its low level concurrency primitives are a very good starting point for further extension.

3.2. The Modula-2 Low-Level Concurrency Concepts

In Modula-2, Wirth (1995) introduces a kind of process called *coroutine*. A set of coroutines can run on a single processor in an interleaving mode one at a time. The coroutines pass control explicitly from one to another using the TRANSFER($p1, p2$) operation. It suspends the current coroutine, assigns its identifier to $p1$, and resumes the coroutine designated by $p2$ previously suspended by another TRANSFER operation.

The coroutines can be synchronised with the processes running in the environment (e.g. IO devices) using interrupt signals. The operation IOTRANSFER($p1, p2, i$), much like the TRANSFER, suspends the current coroutine, assigns its identifier to $p1$, and resumes the coroutine $p2$. Later, on the arrival of the interrupt signal i, the current coroutine is suspended, its identifier is assigned to $p2$, and the coroutine $p1$ is resumed (just after the IOTRANSFER operation).

3.3. Non-busy Waiting in the Simple Transmission Protocols

The first crucial step on our way to using the interrupts to implement message passing protocols is to identify the link signals with the interrupt signals. One could say in technical terms that we "connect the link input ports to the interrupt system". In this way the link signals have become a subset of the interrupt signals. This can raise some naming problems. Naming, however, is out of the scope of this paper, and for the sake of simplicity we assume the set of link signals and the set of interrupt signals are identical, so the interrupt signals can be well-identified by the names of the link ports connected to them.

On top of the low-level primitives, we are going to build a module which maintains the queue of ready processes. It also implements the process management and the non-busy form of waiting for monitor signals and interrupt signals. For the non-busy waiting for interrupt signals we are going to design the IOWait(ip) procedure which delays the running process until the interrupt signal from the link input port ip arrives. The delay is implemented by suspending the calling process and resuming one of the ready processes by the IOTRANSFER( ) operation. When the signal from the link input port ip is accepted, the current process (providing it is not the idle process) becomes ready, and the delayed process becomes running.

```
COROUTINE run, idle;
ProcessQueue Ready;

void IOWait(LINPORT: ip);
{ COROUTINE delayed;
  delayed = run;
  if (Empty(Ready)) run = idle else Remove(Ready, run);
  IOTRANSFER(delayed, run, ip);
  if (run <> idle) Insert(Ready, run);
  run:= delayed;
}
```

Our IOWait( ) operation is functionally equivalent to the LWAIT( ) operation but uses an efficient non-busy form of waiting. Thus IOWait(ip) can replace LWAIT(ip) in the simple transmission protocols presented in Section 2.2.

It is worth noting that each IOWait( ) operation switches the context and reschedules processes twice: first before the calling process is suspended in the IOTRANSFER operation, and then after it is resumed by this operation.

3.4. Non-busy Waiting in the Selective Acceptance Protocol

IOTRANSFER( ) and IOWait( ), unfortunately, cannot be used in the selective acceptance protocol. The process which calls IOTRANSFER( ) or IOWait( ) can wait for one interrupt signal at a time only. To overcome this problem we could introduce new primitives: IATTACH( ) and ITRANSFER( ). The IATTACH(*ip*) operation attaches the link input port *ip* to the calling coroutine. More than one link input port can be attached to a given coroutine at a time. ITRANSFER($p1, p2$) suspends the calling coroutine until an interrupt signal from any of the attached input ports is accepted; after resumption, the calling coroutine detaches all the input ports attached to it.

On top of ITRANSFER( ), the IWait( ) operation is defined. It corresponds to the IOWait( ) operation. IWait( ) has no parameters since it waits for any of the attached interrupt signals. Its algorithm is very similar to the IOWait( ) algorithm; IOTRANSFER( ) is simply replaced with ITRANSFER( ).

3.5. Needle Concept

The operation which follows IWait( ) in the above protocol can be considered a handler of the interrupt signal which completes this operation. A full context switch (inside ITRANSFER( )) and a process rescheduling (inside IWait( )) have to be done before the handler starts. As a result, the handler is executed as a part of the coroutine body. We will call such a handler *coroutine* (or *process*) *level handler.*

Many distributed languages like ADA (*ADA*, 1983) or Concurrent C (Gehani and Roome, 1986) use the process level handlers to handle the interrupt signals. The advantage of such handlers is that they can be designed like any other part of the server. Among others, the server can be suspended in a selective acceptance protocol nested in the process level handler. The cost of this facility is, however, too high in many situations. Generally, real-time systems do not accept the overhead introduced by the process level handler. They usually do not nest the selective protocols, and they require a fast interrupt reaction time.

If the server accepts the client requests in a loop, additional overhead can be observed. After accepting one request, the server has to start the select loop again to re-evaluate all the channel guards, reattach all the input ports, and invoke the rescheduling operation and context switching before accepting the next request.

An alternative for the process level interrupt handler is a handler which is executed on the stack of the interrupted process. Before invoking such a handler, the context of the interrupted process must be saved to be restored when the interrupt handler is completed. In this way the interrupted process can continue its run without any stack switching or rescheduling of the running process. The handler has, however, no identity of its own, which is why no delay operation can be invoked in it.

In terms of context switch and process scheduling overhead, such an interrupt handler is "lighter" than lightweight threads implemented in the same memory space. This is why we call it *needle handler* or just *needle.*

3.6. Using Needles in the Selective Acceptance Protocol

In order to use the needle handlers in the selective acceptance protocol, we have introduced five primitives: ATTACH( ), MASK( ), UNMASK( ), MTRANSFER( ), and CONTINUE( ).

Formally, the *needle handler* is a procedure H associated with a signal bit of the link input port ip which is attached to the calling coroutine by the primitive ATTACH(ip, H). The calling coroutine will be called *native coroutine* for this handler. Many input ports can be attached to one coroutine at the same time. Each input port is also associated with a *mask flag.* The mask flag indicates the *masked* or *unmasked* state of the port signal bit and is set or reset by two primitives MASK(ip) and UNMASK(ip).

The *interrupt flag* (to indicate the *enabled* or *disabled* state of the interrupt system) is an element of the coroutine context. This means that each coroutine saves the interrupt flag when it is suspended and restores it when it is resumed (by the TRANSFER, IOTRANSFER( ), ITRANSFER( ) or MTRANSFER( ) primitives). A coroutine disables and enables the interrupt system on entry and on exit, respectively, from the monitor procedure. Each coroutine is provided with a *continuation flag.* It is set to *false* when MTRANSFER( ) starts and can be turned to *true* by any handler associated with this coroutine by calling the CONTINUE( ) primitive.

The MTRANSFER($p1$, $p2$) operation suspends the *current* coroutine, assigns its identifier to $p1$, sets the current coroutine *continuation* flag to false, makes the coroutine designated by $p2$ *current*, and resumes it. From now on, if the current coroutine interrupt flag is *enabled*, any *attached* and *unmasked* interrupt signals will be accepted. On each acceptance the current coroutine is interrupted, the handler H of the accepted signal is executed, and the control returns to the interrupted coroutine. Such interrupt handling will be repeated until the handler H sets the calling coroutine's *continuation* flag to *true.* In this case, after the completion of the handler, the current (interrupted) coroutine is suspended, its identifier is assigned to $p2$, all the input ports attached to the coroutine $p1$ are detached, and $p1$ is made *current* and resumed (just after its MTRANSFER( ) operation).

During a needle execution, the interrupt system is disabled. Thus, at most one needle can be executed at a time. When the needle is completed, the MTRANSFER( ) operation is still pending and other signals can be accepted. It is worth stressing that the sequence in which the needles are selected for execution is not defined by the foreground process program (and its data). It depends on the sequence of arrival of interrupt source signals. If more than one signal has arrived at the same time, the selection is done in an arbitrary way by the interrupt system. Such a mode of selection of operations for execution in a process will be called the *dynamic selection mode.* It is an alternative to the *static selection mode* in which the next operation to be executed by a process is selected (defined) by the program (and the data) of the process.

The foreground process enters the dynamic selection mode by executing a MTRANSFER( ) operation and stays in it until one of the needle handlers closes it explicitly by executing the CONTINUE operation. When the needle calling CONTINUE( ) is completed, the control is transferred back from the current process $p2$ to the native process $p1$. The native process enters the static selection mode and stays in it until the next MTRANSFER( ) operation is executed. This process handles the interrupts in the static selection mode by using IOTRANSFER( ) to wait for interrupts. In the static selection mode, interrupts are handled by process level handler on the stack of the native process. In the dynamic selection mode, interrupts are handled by needle handlers on the stack of interrupted process.

On top of MTRANSFER( ), the IOMWait( ) operation is defined. Like IWait( ), it is similar to IOWait( ) with the only difference that IOTRANSFER( ) is replaced with MTRANSFER( ) in IOMWait( ).

Introducing the needle handlers removes the two deficiencies of the process level handlers:

- the needle handler starts as soon as possible (after saving the state of the interrupted process);
- a set of input ports once attached is not detached after accepting one interrupt signal; the next signals can be accepted repetitively without any need to re-attach the ports and re-evaluate their guards; the state of the guards can be represented by the port signal masks; the handlers switch the masks only when the states of the guards have been changed.

3.7. The Producer-Consumer Example

To illustrate the low-level concepts, we present a Producer-Consumer system. Links are used to pass messages, and needles are used to implement transactions in the system. Two coroutines, Producer (P) and Consumer (C), use the Buffer Server coroutine (S) to exchange word messages between them. The Producer and Server are connected with the channel *send* consisting of the links: *send_data* and *send_signal*. The channel uses the acknowledge protocol. The Producer is a sender, while the Server is a receiver for this channel.

On the other hand, the Consumer and the Server use the channel *receive* working in the request protocol and consisting of the *receive_data* and *receive_signal* links. The Server is a sender for this channel, and the Consumer is a receiver for it.

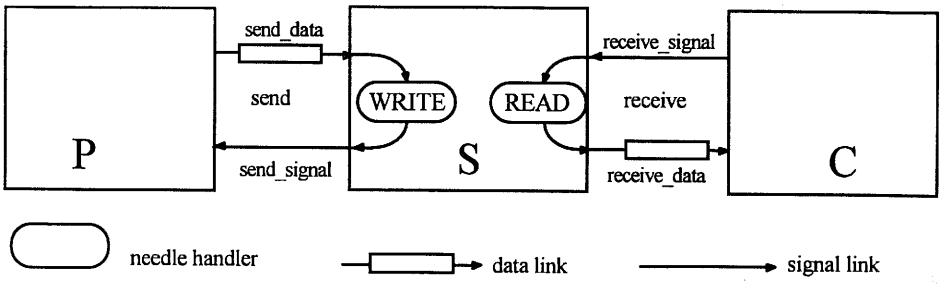

Fig. 1. Coroutines, links and needles in a distributed Producer-Consumer system.

The Buffer Server coroutine implements a circular buffer and accesses the ports:

```
#define max ...                    /* size of the buffer */
int buf[max];
int n, in, ou;
LINPORT send_data_input, receive_signal input;
LOUTPORT receive_data_output, send_signal_output;
```

Attached to the input ports are two needles: WRITE( ) and READ( ):

```
void WRITE( );
{ LGETW(send_data_input, buf[in]); LSIGNAL(send_signal_output);
  in:= (in + 1) % max; n + +;
  if (n == max) MASK(send_data_input);
  if (n == 1) UNMASK(receive_signal_input);
}
void READ( );
{ LPUTW(receive_data_output, buf[ou]); LSIGNAL(receive_data_output);
  ou = (ou + 1) % max; n − −;
  if (n == 0) MASK(receive_signal_input);
  if (n == max −1) UNMASK(send_data_input);
}
```

The Buffer Server's body initialises the interrupt sources and enters the dynamic selection mode by the IOMWait operation. The Server body and the needles maintain the invariant: $0 \leq n \leq N$ by masking and unmasking the interrupt signals depending on its internal state.

```
void BufferServer( );
{ n=0; in=0; ou=0;
  ATTACH(receive_signal_input, READ); ATTACH(send_data_input, WRITE);
  MASK(receive_signal_input); UNMASK(send_data_input);
  IOMWait( );
}
```

The Producer accesses two ports. It produces a word and sends it down the *send_data* link. To wait for the acknowledge signal, it uses IOWait(send_signal_input):

```
LOUTPORT send_data_output ;
LINPORT send_signal_input ;
void Producer( )
{ int w;
  while (1)
  { produce(& w ); LPUTW(send_data_output, w); LSIGNAL(send_data_output);
    IOWait(send_signal_input); }
}
```

The Consumer also accesses two ports. It signals readiness to consume the next portion of data, waits for its reception using IOWait, and consumes it.

```
LOUTPORT receive_signal_output;
LINPORT receive_data_input;
void Consumer( )
{ int w;
  while (1)
  { LSIGNAL(receive_signal_output); IOWait(receive_data_input);
    LGETW(receive_data_input, & w); consume(w); }
}
```

Since two different protocols are employed, the BufferServer structure is symmetrical with one channel on each side. Such symmetry cannot be obtained in languages like Occam (*Occam*, 1988) which use one kind of communication protocol and require that an input operation can only be used in the alternation statement.

An important thing is that our Buffer Server solution reduces the context switching and eliminates the process scheduling overhead almost completely. After initialisation, the BufferServer coroutine stays in the IOMWait, and the whole job is done by the needles. There are also no guards; the synchronisation is enforced by interrupt masks.

4. Structured Channels

Presented above are the simple one-to-one channels and the related protocols: they transmit the values of the word or pointer type between one sender and one receiver. On top of them, more structured channels and protocols can be built. Below we will discuss how the messages containing one or more words can be transmitted and how many-to-one channels can be built.

4.1. Message Channels

Message channels are used to transmit structured values (messages) between processes. The way the messages are actually transmitted depends on the architecture of the memory accessed by the processes.

In the shared memory with uniform access (UMA), the messages are sent by pointers. The receiver accesses the message contents directly in the sender's memory location and then acknowledges it. In the shared memory with non-uniform access (NUMA), a message pointer is transmitted to the receiver, and then the message contents are copied from the server's to the receiver's location before the message is acknowledged.

In the distributed memory, the message contents are transmitted down a so called *message channel*, word by word. Below, we present such a channel for the remote transmission of messages between two processes running on different computers. The message is represented as an array of words with its length defined at the first position. On each computer the channel is accessible as a *message port*. The message port is an agent process interconnected with the master processes by a simple pointer channel. The message port agents (on different computers) are interconnected with a simple word channel.

The sender process sends the message to its local agent by pointer and waits for acknowledgement. The agent reads the message word by word from the sender's area and sends them to its counterpart on the other computer using the simple acknowledgement protocol. The other agent collects the message words in its own area. When the transmission is completed, the receiver agent sends the message by pointer to the receiver process. The receiver process accesses its agent copy of the message as if it were accessing the remote sender's copy.

When the message is processed (possibly by modifying some of its contents), the receiver acknowledges to its agent. The receiver agent sends back the (modified part of the) message contents again word by word in the acknowledgement protocol. The sender agent receives the contents, updates the sender's message area, and sends acknowledgement to the sender. Let us note that the sender and the receiver work with their local agents in the simple pointer acknowledgement protocols. The remote processes do not have to be aware that their agents send the message word by word by the remote link. In this way, the sender and the receiver communicate, while actually not aware of whether they are local or remote to each other.

Employing one agent for each sender and one for each receiver is not the most optimal solution. We have chosen it, however, since it can be generalised to the case where one remote simple channel is used to pass messages between many senders and many receivers. The agents simply need to implement classical multiplexing/demultiplexing algorithms. The agents use, in such a case, the selective acceptance protocol to communicate with their master processes.

4.2. Message Dispatcher

In many distributed programming languages, servers use many-to-one channels to accept transactions from many clients. Such channels can also be implemented using our primitives and protocols.

Let us assume that C client processes are interconnected with a server process which provides T types of synchronous transactions. The server is attached to a *mailbox* which allows each client process to send messages to each transaction. Such a mailbox can be implemented by a process called *message dispatcher* and *simple pointer channels*. All simple channels work in the simple acknowledgement protocol.

The clients are interconnected with the message dispatcher by *client channels*. Remote clients are represented by their local agents. The message dispatcher is a receiver for these channels. The client channels are seen at the message dispatcher side as two arrays:

```
LINPORT ccip[C];        /* client call input ports */
LOUTPORT caop[C]        /* client ack output ports */
```

The server is interconnected with the message dispatcher by T simple pointer channels called transaction channels. The message dispatcher is a sender for these channels. For each transaction type t the dispatcher maintains also

- pointer tm to the client message which is currently processed by the transaction t

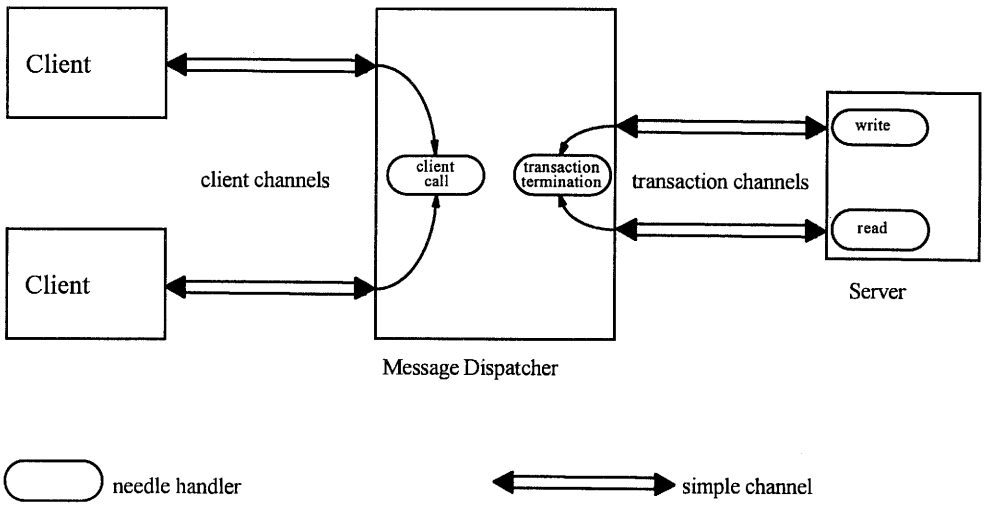

Fig. 2. Needle handlers and simple channels used by the message dispatcher.

- index c of the client's input port from which the message pointer tm has been received; if there is no such message, tm takes the value NULL;
- queue pen of the transaction invocations (tm, c) which are waiting to be processed by this transaction; they are called *pending invocations*;

The above entities constitute a transaction description structure called $TransDsc$. The descriptions for all transactions are collected in the transaction table $TT[T]$:

```
struct TransDsc
{ LINPORT taip;        /* transaction ack input port */
  LOUTPORT tcop;       /* transaction call output port */
  void * tm;           /* current message */
  int ccip;            /* current client input port */
  InvocQueue pen;      /* pending invocation queue */
} TT[T];
```

After initialisation, the message dispatcher works in the dynamic selection mode. All the client link input ports are handled by the *client_call*( ) needle, and all the transaction link input ports are handled by the *transaction_termination*( ) needle.

```
void MeassageDispatcher( )
{ int i;
  for (i = 0; i < C; i++) ATTACH(ccip[i], client_call);
  for (i = 0; i < T; i++) { ATTACH(TT[i].taip, transaction_termination);
                            TT[i].tm = NULL; }
  IOMWait( );
}
```

The *client_call* needle receives a message pointer m from the client input port ip, and dispatches it to appropriate transaction. The number t of the transaction is sent in the message contents. If the transaction t is occupied by another invocation, the incoming invocation (ip, m) is inserted into the *pen* queue. Otherwise, the incoming invocation occupies the transaction, and the message pointer m is sent to the server down the link output port $TT[t].tcop$.

```
void client_call( )
{ LINPORT ip;                                   /* client input port */
  void *m;                                      /* client message pointer*/
  int t;                                        /* transaction number */
  ip = THISIP( );                               /* get input port for calling needle*/
  LGET(ip, m);                                  /* get message pointer*/
  t = get_transaction_number(m);                /* and transaction number */
  if (TT[t].tm == NULL)                         /* is transaction occupied? */
  { TT[t].tm = m; TT[t].ccip = ip;              /* save current invocation */
    LPUT(TT[t].tcop, m); LSIGNAL(TT[t].tcop); } /* invoke server */
  else insert(TT[t].pen, ip, m);                /* transaction occupied - queue
                                                   the invocation */
}
```

Having completed the transaction, the server sends an acknowledgement signal back to the message dispatcher. As a result of accepting this signal, the *transaction_termination* needle is invoked. The needle recognises the number t of the transactions which has been terminated and forwards the acknowledgement to the client which has called the transaction. If there are some pending invocations for the transaction t, one of them is selected and sent to the server. Otherwise, the transaction is freed by assigning NULL value to tm.

```
void transaction_termination( )
{ LINPORT ip;                                   /* transaction input port */
  LOUTPORT cop;                                 /* client output port */
  int t;                                        /* transaction number */
  ip = THISIP( );                               /* get input port for calling
                                                   needle */
  t = transaction_terminated(ip);               /* get transaction number */
  cop = MYOP(TT[t].ccip);                       /* get client output port */
  LSIGNAL(cop);                                 /* acknowledge client */
  if (empty(TT[t].pen) TT[t].tm = NULL;         /* no pending invocation */
  else { remove(TT[t].pen, TT[t].ccip, TT[t].tm);  /* get new current
                                                   invocation */
  LPUTP(TT[t].tcop, TT[t].tm); LSIGNAL(TT[t].tcop);} /* invoke server */
}
```

As we can see, the many-to-one channel can be built using the one-to-one channels. With this approach, the message dispatching algorithm (a queue of pending

invocations) which is a part of such a channel, can be designed at application level. The message dispatcher works in the dynamic selection mode thereby avoiding superfluous context switching and process rescheduling overhead.

4.3. The Producer-Consumer Example Revisited

In this example we present a many-producer-many-consumer system. Since the clients and the server are interconnected via the mailbox, data is sent and received as message contents. All the simple channels used in the system are now pointer channels. Presented below, is a modified solution of the system from Section 3.7.

The transaction channels are seen at the Buffer Server side as two arrays of ports (index 0 for WRITE and index 1 for READ transaction).

```
LINPORT tcip[2];          /*transaction call input ports */
LOUTPORT taop[2];         /* transaction ack output ports */

# define max ...
int buf[max];
int n, in, ou;

void WRITE( );                                  /* needle handler */
{ void *m;                                      /* transaction message pointer */
  int w;                                        /* character to write */
  LGETP(tcip[0], m); LSIGNAL(taop[0]);          /* get message pointer and acknowledge*/
  w = get_input_parameter(m); buf[in] = w;      /* retrieve and buffer data */
  in:= (in + 1) % max; n++;                     /* update buffer status */
  if (n == max) MASK(tcip[0]);                  /* update link masks */
  if (n == 1) UNMASK(tcip[1]);
}
void READ( );                                   /* needle handler */
{ void *m;                                      /* transaction message pointer */
  int w;                                        /* character to read */
  LGETP(tcip[1], m);                            /* get message pointer */
  w = buf[ou]; put_output_parameter(m, w);      /* read char and put it into message
                                                   contents */
  LSIGNAL(taop) ;                               /* acknowledge */
  ou = (ou + 1) % max; n − −;                   /* update buffer state */
  if (n == 0) MASK(tcip[1]);                    /* update link masks */
  if (n == max −1) UNMASK(tcip[0]);
}

void BufferServer( );
{ n = 0; in = 0; ou = 0;
  ATTACH(tcip[1], READ); ATTACH(tcip[0], WRITE);
  MASK(tcip[1]); UNMASK(tcip[0]);
  IOMWait( );
}
```

The client channels are seen at the client side as two arrays of ports. Each Client uses its own unique index i to access the ports.

```
LINPORT ccop[C];        /* client call output ports */
LOUTPORT caip[C];       /* client ack input ports */
```

Each Producer has to pack the WRITE transaction number and data to be buffered in the message.

```
void Producer(int i)
{ MsgBuffer mb;                          /* message contents buffer */
  int w;                                 /* data to be sent */
  while (1)
  { produce(&w);
    put_input_parameter(mb, w);
    put_transaction_number(mb, 0)
    LPUTP(ccop[i], mb); LSIGNAL(ccop[i]); /* send message pointer to message
                                             dispatcher */
    IOWait(caip[i]);                     /* wait for acknowledgement */
  }
}
```

Each Consumer has to send the pointer of its message with the READ transaction number packed in it. After the transaction is acknowledged, it has to unpack the data.

```
void Consumer(int i)
{ MsgBuffer mb;                          /* message contents buffer */
  int w;                                 /* data to be received */
  while (1)
  { put_transaction_number(mb, 1)
    LPUTP(ccop[i], m); LSIGNAL(ccop[i]); /* send message pointer to message
                                            dispatcher */
    IOWait(caip[i]);                     /* wait for acknowledgement */
    get_output_parameter(mb, &w);
    consume(w);
  }
}
```

As we can see, the Server meant for many Producers and many Consumers can be built as a collection of two processes: Message Dispatcher and Buffer Server interconnected with links and using interrupts for their synchronisation.

5. Future Works

We have implemented both the message channel and the message dispatcher on top of the UNIX operating system. The low-level library defines coroutines in a UNIX-process memory space and links connecting the coroutines in the same or different memory spaces.

We are continuing our work in two directions:
- performance evaluation of structured channels, and
- introduction of multi-grain transactions.

The grain is meant to be a part of a process which does not contain any synchronisation delay and is separated from other grains by synchronisation delay operations. Thus, a grain can be executed as a needle. We propose to represent a transaction as a chain of grains executed in the interrupt-driven mode. A grain execution can be delayed by masking associated interrupts. A multi-grain transaction can be nested in a non-blocking way. Thus, many transactions can run in parallel in one server, in the same way as many operations can run in parallel in a monitor (Hoare, 1974).

As a result, the expressiveness of the multi-grain transaction can be compared with that of the monitor call.

6. Related Works

Silberschatz (1979) has presented a set of a system level primitives and protocols for an abstract implementation of the CSP IO commands and guarded statements (Hoare, 1978). The primitives and protocols are similar to our primitives and busy-waiting protocols. We, however, have related our primitives to a simpler communication concept, link, and shown that it can be used to implement not only the CSP constructs but more complex channels as well. Moreover, we have shown that an efficient implementation of the non-busy form of waiting in such protocols can be based on the interrupt feature.

An extension of Modula-2 low-level concurrency features similar to our IATTACH and ITRANSFER primitives is given in Modula-2 Standard Draft (1992). It facilitates implementation of selective acceptance protocol and supports the process level interrupt handlers. The Draft, however, considers the interrupt to be an IO device synchronisation feature. It does not define any mechanism to either generate the interrupt signals by the processes or to transmit data between the processes.

Hills has gone further with his proposal for Structured Interrupts (Hills, 1993). He has added the facility to send interrupt signals by processes. In this way, the interrupts are considered as an interprocess synchronisation mechanism. The primitives proposed by Hills are of higher level than our primitives; they include the ready process queue management. His interrupt handlers are the process-level handlers.

Bjorkman (1994) has pointed out that substantial efficiency gains can be achieved if interrupt handlers are used instead of threads (process-level handlers) for network protocol processing. Typical message delivery includes one interrupt and two context switches on the receiving side. The context switching is extremely costly in many modern computers, such as RISC processors with large register sets. When the protocol processing is included in the interrupt handler, the context switching can be avoided.

An approach similar to ours, in building a high-level message-passing mechanism on the bases of low-level primitives incorporated into programming language, is very popular in functional languages, which support the first-class continuation concept. As an example, there is the SML/NJ language (Morrisett and Tolmach, 1993), where

quite powerful many-senders-to-many-receivers message passing functions are implemented. This is a 'classic' shared-memory based implementation for local threads. Interrupts are not used in this implementation.

7. Conclusion

In this paper we have presented a set of low-level primitives and related protocols for the implementation of a high-level synchronous message passing mechanism. This approach allows us to explicitly express both the process scheduling and the message dispatching algorithms in a high-level language program. Such algorithms are hidden from the user inside the high-level primitives offered by most of the distributing programming languages. Our approach gives the user direct control over these algorithms. The user not only can tailor these algorithms according to his application needs, but can design a new transmission mechanism which best suits his application as well.

We propose the link as a major elementary message passing concept. It is a tool to transmit a simple data value and a synchronisation signal between processes. The link, however, is not a regular asynchronous channel. We abandoned the idea of unbounded buffering in the link. It makes the asynchronous message passing too heavy as a tool for structuring other kinds of channels. At the same time we did not choose a simple synchronous channel as the elementary tool because that would involve bi-directional transmissions, not always efficient in building highly organised channels. Whenever bi-directional transmission is needed, using two independent links has proven to be more flexible.

The real innovation in our approach, however, is in using the interrupt feature in order to achieve uniform protocols for remote and local message passing. We consider the interrupt feature a synchronisation tool not only between the environment (IO devices, users) and the processes, but between the processes themselves as well.

If the interrupt signals are sent between processes running in the same memory space, they can be used to support the non-busy form of waiting in the selective acceptance protocol. With the interrupts, the communication channels can be built in a hierarchical way, very similar to the way the shared memory tools are built using monitors.

The efficiency of such protocols can be improved considerably when we allow the interrupt handlers to execute on the stack of the interrupted process. In this case the overhead involved in the context switching and process scheduling can be reduced. This reduces the interrupt response time, which is of special importance for real-time applications. Further optimisation can be achieved for the servers which accept the clients' requests in a loop. If the request operations are implemented as needle handlers, the server does not have to reattach the same entries and re-evaluate the same guards repetitively.

Using interrupts as a basic synchronisation tool leads to the concept of a multi-grain transaction. The multi-grain transaction improves the expressiveness of synchronous transactions. Such transactions can be executed in parallel and nested in a non-blocking way similar to monitor operations.

We believe that the uniform implementation of both local and remote message passing IPC on the basis of interrupts can guarantee expressiveness and efficiency comparable with that achieved by monitors and other shared memory tools for lightweight processes. In this way the message passing paradigm will become as efficient as the shared memory paradigm. The paradigms can become equivalent tools for fine grained multiprocessor parallel computations.

Acknowledgement

Many ideas presented in this paper emerged during fruitful discussions with Prof. Zbigniew Banaszak.

References

ADA (1983): *United States Department of Defense Reference Manual for the ADA Programming Language.* — ANSI/MIL-STD 1815A.

Andrews G.R., Ollson R.A., Coffin M., Elshof I., Nilsen K., Purdin T. and Townsen G. (1988): *An Overview of the SR language and implementation.* — ACM Trans. Programming Languages and Systems, v.10, No.1, pp.51–86.

Bjorkman M. (1994): *Interrupt protocol processing in the x-kernel.* — Technical Report 94–14, Computer Science Dept., University of Arizona.

Brinch Hansen P. (1987): *A joyce implementation.* — Software – Practice and Experience, v.17, No.4, pp.267–276.

Chrobot S. (1994): *Where concurrent processes originate.* — Proc. Conf. *Programming Languages and System Architecture*, ETH, Zurich, March, Lecture Notes on Computer Science 782, pp.151–170.

Gehani N.H. and Roome W.D. (1986): *Concurrent C.* — Software – Practice and Experience, v.16, No.9, pp.821–844.

Hills T. (1993): *Structured interrupts.* — Operating System Review, v.27, No.1, pp.51–69.

Hoare C.A.R. (1974): *Monitors: An operating system structuring concept.* — Comm. ACM, v.17, No.10. pp.549–557.

Hoare C.A.R. (1978): *Communicating sequential processes.* — Comm. ACM, v.21, pp.66–677.

Jagannatan S. and Philbin J. (1994): *High-level abstraction for efficient concurrent systems.* — Proc. Conf. *Programming Languages and System Architecture*, ETH, Zurich, March, Lecture Notes on Computer Science 782, pp.171–190.

Modula-2 (1992): *2nd Committee Draft of the Modula-2 Standard.* — CD 105114.

Morrisett J.G. and Tolmach A. (1993): *Procs and Locks: A portable multiprocessor platform for standard ML of New Jersey.* — 4th ACM PPOPP, SIGPLAN Notices, v.28. No.7.

Occam (1988): INMOS Limited, occam2 Reference Manual, New York: Prentice-Hall.

Silberschatz A. (1979): *Communication and synchronisation in distributed systems.* — IEEE Trans. Software Engineering, v.Se-5, No.6, pp.542–546.

SunOS (1990): *Sun Microsystems, system services overview.* — Kernel Interface, pp.20–24.

Wirth N. (1985): *Programming in MODULA-2.* — Berlin: Springer-Verlag.

Received: June 28, 1994
Revised: November 10, 1994