This is Chapter 2 from the second edition of :

Networks, Routers and Transputers:

Function, Performance and applications

Edited by M.D. May, P.W. Thompson, and P.H. Welch

© INMOS Limited 1993

This edition has been made available electronically so that it may be freely copied and distributed. Permission to modify the text or to use excerpts must be obtained from INMOS Limited. Copies of this edition may not be sold. A hardbound book edition may be obtained from IOS Press:

IOS Press
Van Diemenstraat 94
1013 CN Amsterdam
Netherlands

IOS Press, Inc.
P.O. Box 10558
Burke, VA 22009–0558
U.S.A.

IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England

Kaigai Publications, Ltd.
21 Kanda Tsukasa–Cho 2–Chome
Chiyoda–Ka
Tokyo 101
Japan

This chapter was written by M.D. May, R.M. Shepherd and P.W. Thompson

2 The T9000 Communications Architecture

2.1 Introduction

This chapter describes the communications capabilities implemented in the IMS T9000 transputer, and supported by the IMS C104 packet router, which is discussed in chapter 3. The T9000 retains the point-to-point synchronised message passing model implemented in first generation of transputers but extends it in two significant ways. The most important innovation of the T9000 is the virtualization of external communication. This allows any number of *virtual* links to be established over a single hardware link between two directly connected T9000s, and for virtual links to be established between T9000s connected by a routing network constructed from C104 routers. A second important innovation is the introduction of a many-one communication mechanism, the resource. This provides, amongst other things, an efficient distributed implementation of servers.

2.2 The IMS T9000

The IMS T9000 is a second–generation transputer; it has a superscalar processor, a hardware scheduler, 16K bytes of on-chip cache memory, and an autonomous communications processor.

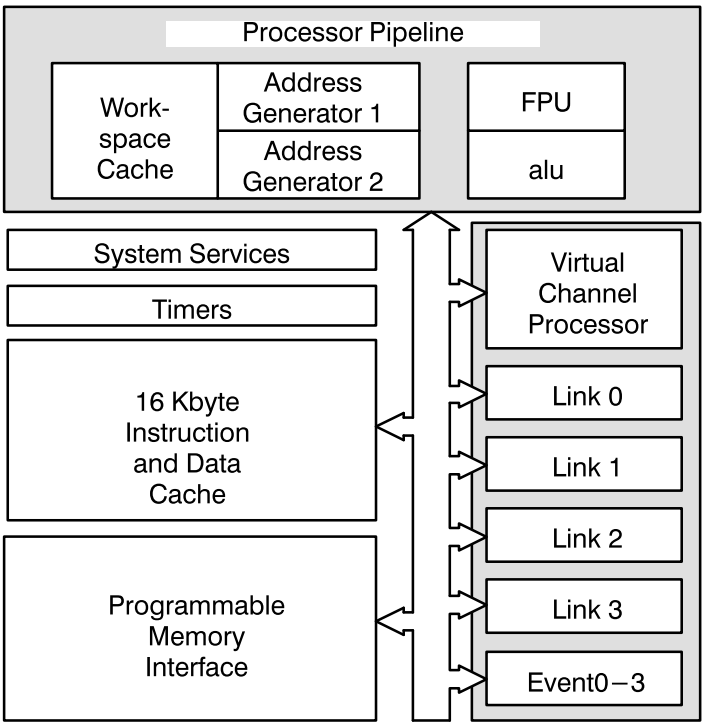

Figure 2.1 The IMS T9000 Transputer

The T9000's scheduler allows the creation and execution of any number of concurrent processes. The processes communicate by passing messages over point-to-point *channels*. Channels are unidirectional, and message passing is synchronised and unbuffered; the sending process must wait until the receiving process is ready, and the receiving process must wait until the sending process is ready. Once both processes are ready the message can be copied directly from one process to the other. The use of this type of message passing removes the need for message queues

and message buffers in the implementation, and prevents accidental loss of data due to variations in the order in which processes happen to be executed. The T9000's scheduler also provides each process with its own timer, and the means for a process to deschedule until its timer reaches a specified alarm time.

The T9000's processor and scheduler implement communication between processes executing on the same processor. The T9000's communication system allows processes executing on different transputers to communicate in the same manner as processes on the same transputer. The communication system has four link interfaces, each of which may be directly connected to a link interface of another transputer, or may be connected via a network of routing devices to other transputers. Messages are passed over these links by the autonomous communications processor, the *virtual channel processor* (VCP).

2.3 Instruction set basics and processes

2.3.1 Sequential processes

The T9000 has a small set of registers which support the execution of sequential processes:

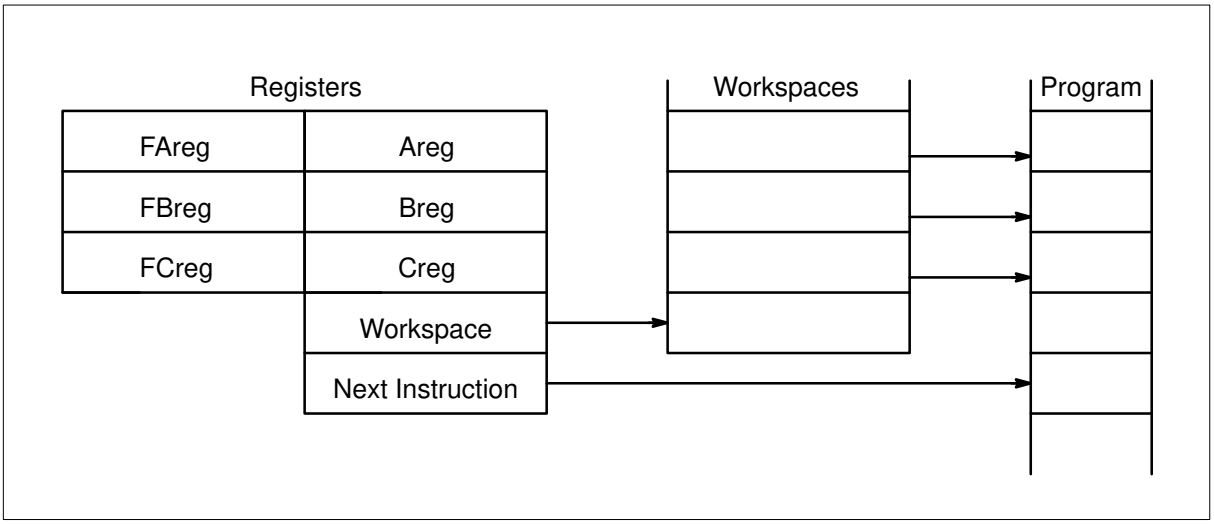

Figure 2.2 IMS T9000 Registers

The workspace pointer (Wptr) points to the workspace of the currently executing process. This workspace, which is typically organized as a falling stack, contains the local variables and temporaries of the process. When a process is not executing, for example while it is waiting for a communication, its workspace also contains other information, such as the process' instruction pointer.

The instruction pointer (Iptr) points at the next instruction to be executed by the current process.

The Areg, Breg and Creg are organized as stack. The stack is used for the evaluation of integer and address calculations, and as the operands of more complex instructions, such as the communication instructions. The FAreg, FBreg and FCreg form another stack, used for floating point arithmetic.

2.3.2 Concurrent processes

The T9000 provides efficient support of concurrency and communication. It has a hardware scheduler which enables any number of processes to be executed together, sharing the processor time. This removes the need for a software kernel.

At any time, a concurrent process may be:

active	being executed
	on a list waiting for execution
inactive	ready to input
	ready to output
	waiting until a specified time
	waiting for a semaphore

The T9000's scheduler operates in such a way that inactive processes do not consume any processor time.

The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented using two registers, one of which points to the first process on the list, the other to the last.

In figure 2.3, S is executing, and P, Q and R are active, awaiting execution.

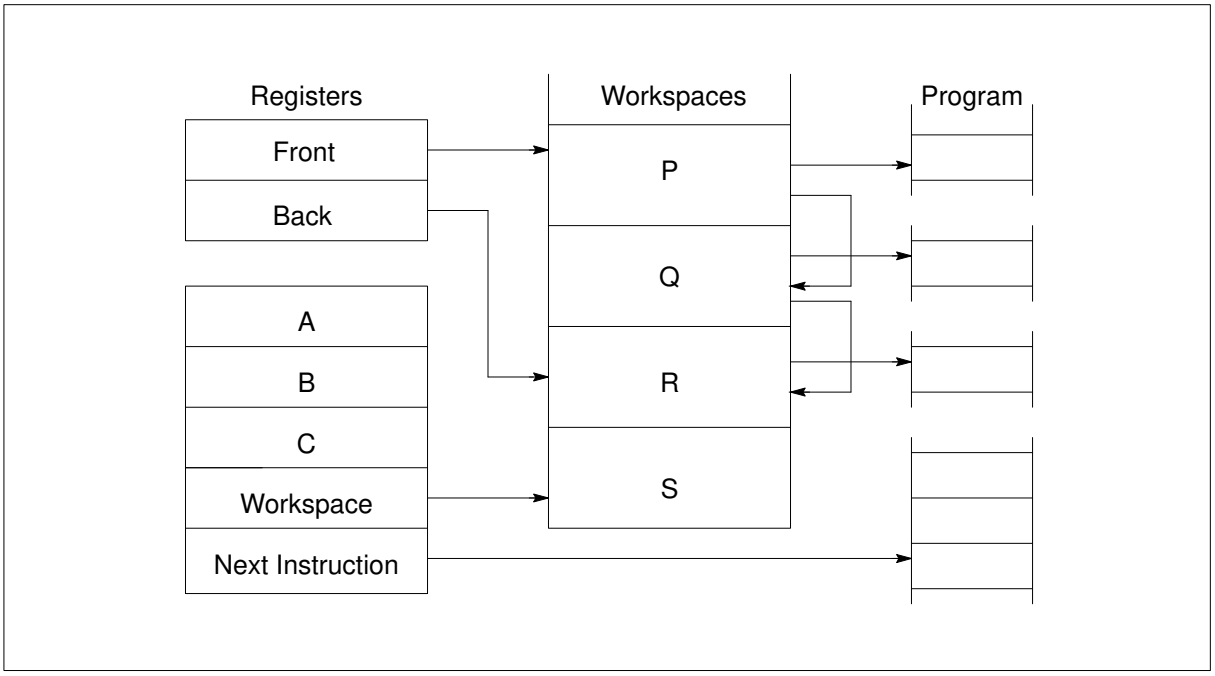

Figure 2.3 Active processes

The T9000 provides a number of instructions to support the process model. These include *start process*, and *end process*. The *start process* instruction creates a new concurrent process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. The *end process* instruction allows a number of concurrent processes to join together, so that a successor process is executed when, and only when, all of its predecessors have terminated with an *end process* instruction.

Priority scheduling

The T9000 scheduler is actually more complex than described above. It provides two scheduling queues, one for each of two priorities. Whenever a process of high priority (priority 0) is able to proceed, it will do so in preference to a low priority (priority 1) process. If a high priority process becomes active whilst a low priority process is executing, the high priority process preempts the low priority process.

To identify a process entirely, it is necessary to identify both the process' workspace and its priority. These can be encoded in a single word by or-ing the priority of the process into the bottom bit of the workspace address; the resulting value is known as the *process id*.

2.4 Implementation of Communications

The T9000 provides a number of instructions which implement communication over channels. These instructions use the address of the channel to determine whether the channel is internal or is a virtual channel. This means that the same instruction sequence can be used, allowing a process to be written and compiled without knowledge of where its channels are connected.

Since channels are distinct objects from the processes which communicate over them, they serve to hide the internal structure of such processes from each other. A process which interacts with others only via channels thus has a very clean and simple interface, which facilitates the application of structured programming principles.

Before a channel can be used it must be allocated and initialized. The details depend on whether the channel is to connect two processes on the same transputer, or two processes on different transputers.

2.4.1 Variable length input and output

The *variable input message (vin)*, *variable output message (vout)* and *load count* instructions provide the basic message passing mechanism of the T9000. They convey a message and its length, from an sending process to an receiving process. The receiver specifies the maximum length of message that it is prepared to receive, and the sender the actual length of the message to be sent. If the actual length is longer than the receiver is prepared to receive than an error is signalled.

A sending process performs an output by loading the evaluation stack with a pointer to the message, the length of the message and the address of the channel. It then executes a *vout* instruction. A receiving process performs an input by loading the evaluation stack with a pointer to the variable, the maximum length of message and the address of the channel. It then executes a *vin* instruction followed by a *load count* instruction. The *load count* instruction either loads the length of the message received onto the evaluation stack, or signals an error, if the length specified by the sender was too long.

2.4.2 Internal channel communication

A channel between two processes on the same transputer is implemented by a single word of memory. Before the channel is used it must be initialized to the special value `NotProcess` (=80000000_{16}) which cannot be the address of the workspace of any process.

At any time, an internal channel (a single word in memory) either holds the identity of a process, or holds the special value `NotProcess`, which indicates that the channel is empty. The channel is initialized to `NotProcess` before it is used.

When a message is passed using the channel, the identity of the first process to become ready is stored in the channel, and the processor starts to execute the next process from the scheduling list. When the second process to use the channel becomes ready, the message is copied, the waiting process is added to the scheduling list, and the channel is reset to its initial state. It does not matter whether the receiving or the sending process becomes ready first.

In figure 2.4, a process *P* is about to execute an output instruction on an ‘empty’ channel *C*. The evaluation stack holds a pointer to a message, the address of channel *C* and a count of the number of bytes in the message.

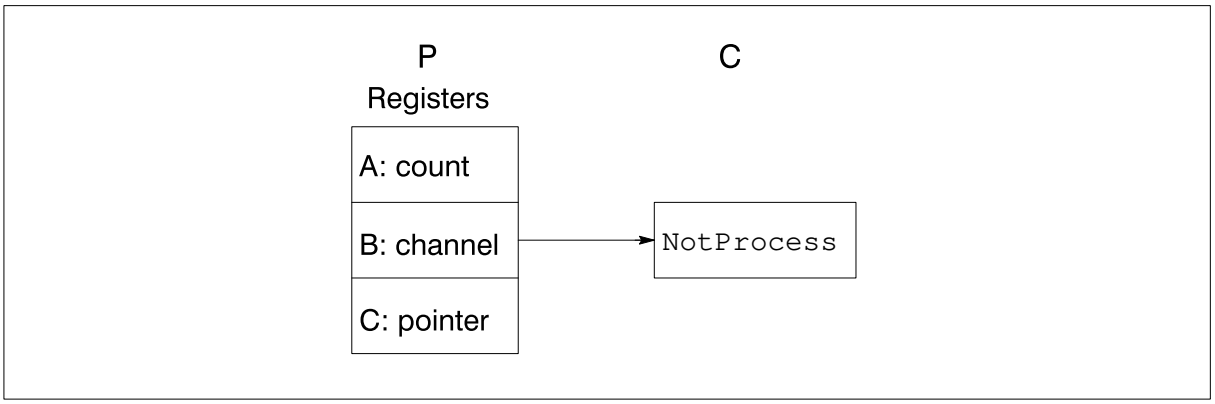

Figure 2.4 Output to empty channel

After executing the variable output instruction, the channel *C* holds the address of the workspace of *P,* and the address and length of the message to be transferred are stored in the workspace, as shown in figure 2.5. *P* is descheduled, and the processor starts to execute the next process from the scheduling list.

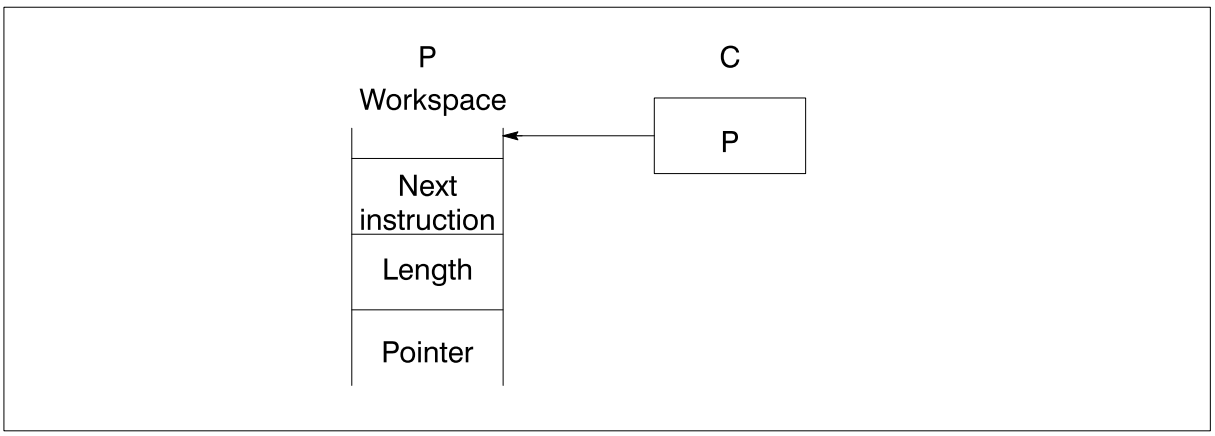

Figure 2.5 Outputting Process Descheduled

The channel *C* and the process *P* remain in this state until a second process, *Q* executes a variable input instruction on the channel, as shown in figure 2.6.

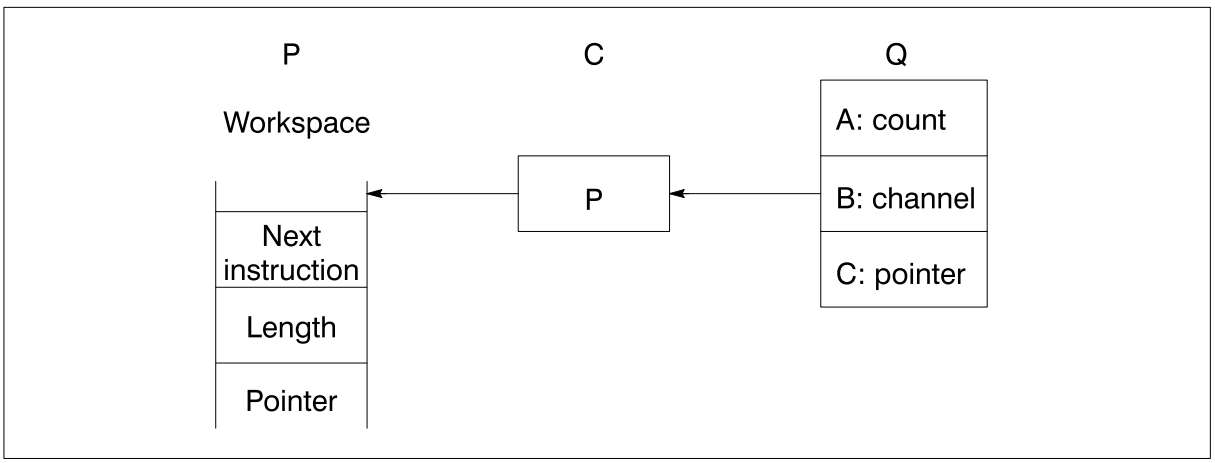

Figure 2.6 Input on a Ready Channel

Since the channel is not empty, the message is copied and the waiting process *P* is added to the scheduling list. The channel *C* is reset to its initial 'empty' state, as shown in figure 2.7. The length of the message (as specified by *P*) is recorded in the workspace of *Q* so that it can be put onto the stack by the *load count* instruction.

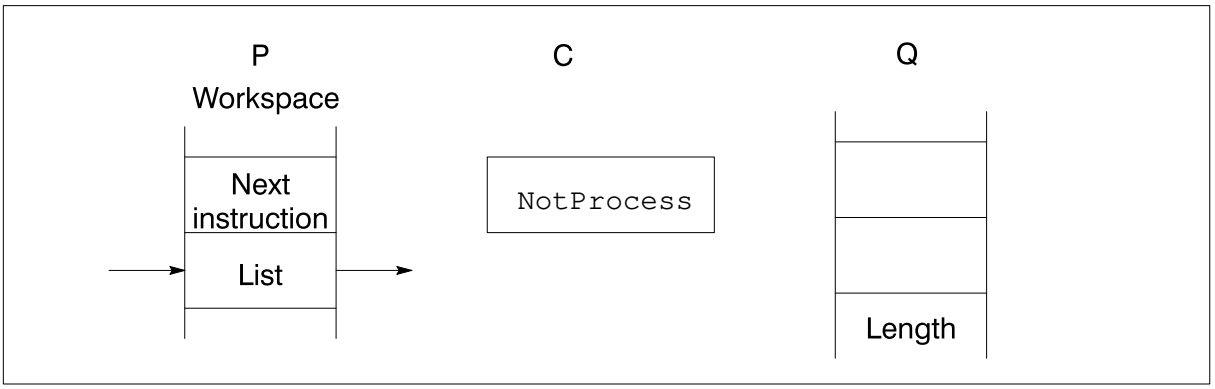

Figure 2.7 Communication completed, output ready first

If *P* is the receiving process and *Q* the sending one, the same set of pictures apply, except that the final state is as shown in figure 2.8.

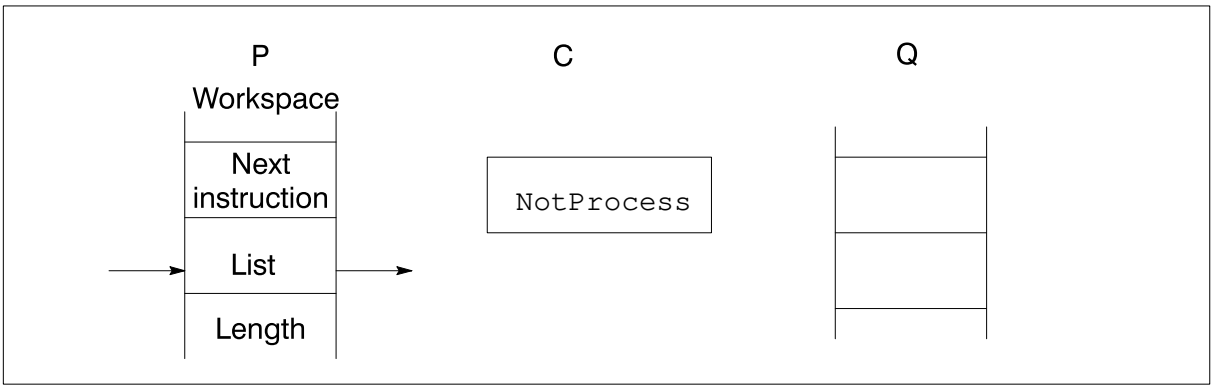

Figure 2.8 Communication completed, input ready first

2.4.3 External channel communication

The synchronised message passing of the transputer requires that data be copied from the sending process to the receiving process, and that the sending process continue execution only after the receiving process has input the data. Where the processes communicating reside on different transputers, it is necessary to transfer the data from one transputer to the other, and to signal in the other direction that an input has occurred. Thus the connection between the processes must convey information in both directions.

Virtual links

In the first–generation transputers, each point-to-point physical link between transputers provides two communication channels, one in each direction. In the new transputers, each physical link provides an *arbitrary* number of point-to-point *virtual links*. Each virtual link provides two channels, one in each direction. Hardware within the transputer multiplexes virtual links onto the physical links. At any moment, each physical link has an associated list of virtual links waiting to use it.

Each virtual link is represented by a pair of *virtual link control blocks* (VLCBs), one on each transputer. When a process executes an input or output instruction to send or receive a message on a virtual link, the process is descheduled and its identity is stored in the control block. At the same time the control block is used to determine the physical link to be used for the communication, and is added to the associated list of waiting virtual links. An example of how the lists might look at one moment is illustrated in figure 2.9.

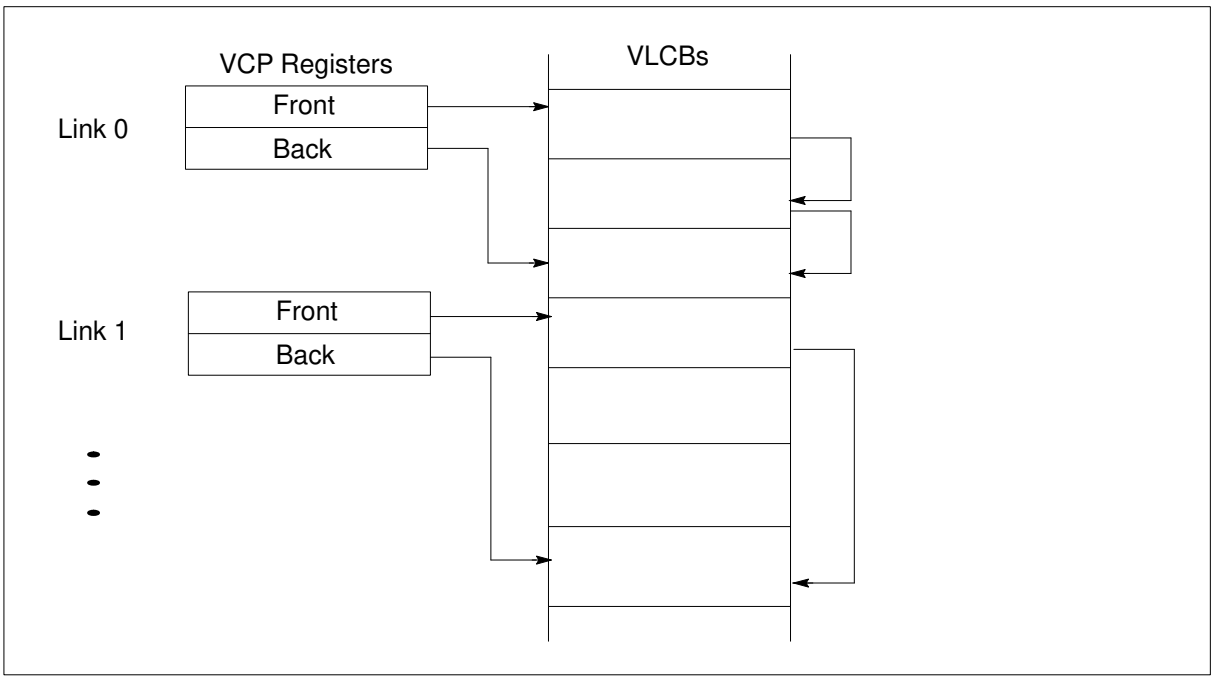

Figure 2.9 Queues of VLCBs

Message–passing Protocol

When an output is performed, the message is transmitted as a sequence of packets, each of which is restricted in length to a maximum of 32 data bytes. There are several reasons for this which are explained below. Each packet of the message starts with a *header*, which is used to route the packet to an receiving process on a remote transputer. The header also identifies the control block of the virtual link used by the remote receiving process. Thus a virtual link is established by setting up a control block in each of two transputers such that the header in each control block is set to cause packets to address the other control block.

Each packet of a message is transferred directly from the sending process to the physical link and is transferred directly from the physical link to the receiving process, provided that a process is waiting when the packet arrives. An acknowledgement packet is dispatched back along the virtual link as soon as each packet starts to arrive (thus transmission of acknowledge packets can overlap transmission of message packets). At the outputting end of the virtual link, the process will be rescheduled after the last acknowledgement packet has been received.

When the first packet of a message starts to arrive on a virtual link, it is possible that no process is waiting to input the message. In this case, it is essential that the packet is stored temporarily so that communication via other virtual links sharing the same physical link is not delayed. A single packet buffer associated with each virtual link control block is sufficient for this purpose, since the outputter will not send any further packets until an acknowledgement packet is received.

The splitting of messages into packets of limited size, each of which is acknowledged before the next is sent, has several important consequences:

- It prevents any single virtual link from hogging a physical link
- It prevents a single virtual link from hogging a path through a network
- It provides flow-control of message communication and provides the end-to-end synchronization needed for synchronised process communication
- It requires only a small buffer to be used to avoid blocking in the case that a message arrives before a process is ready to receive it

Each VLCB must be initialized with the address of the packet buffer for the input channel, the header to be used for outgoing packets, and which physical link is to be used by the virtual link.

The implementation of message–passing

When a message is passed via a virtual channel the processor of the T9000 delegates the job of transferring the message to the VCP and deschedules the process. Once a message has been transferred the VCP causes the waiting process to be rescheduled. This allows the processor to continue the execution of other processes whilst the external message transfer takes place.

In figure 2.10 processes *P* and *Q*, executed by different transputers, communicate using a virtual channel *C* implemented by a link connecting two transputers. *P* outputs, and *Q* inputs; note that the protocol used by the VCP ensures that it does not matter which of *P* and *Q* becomes ready first.

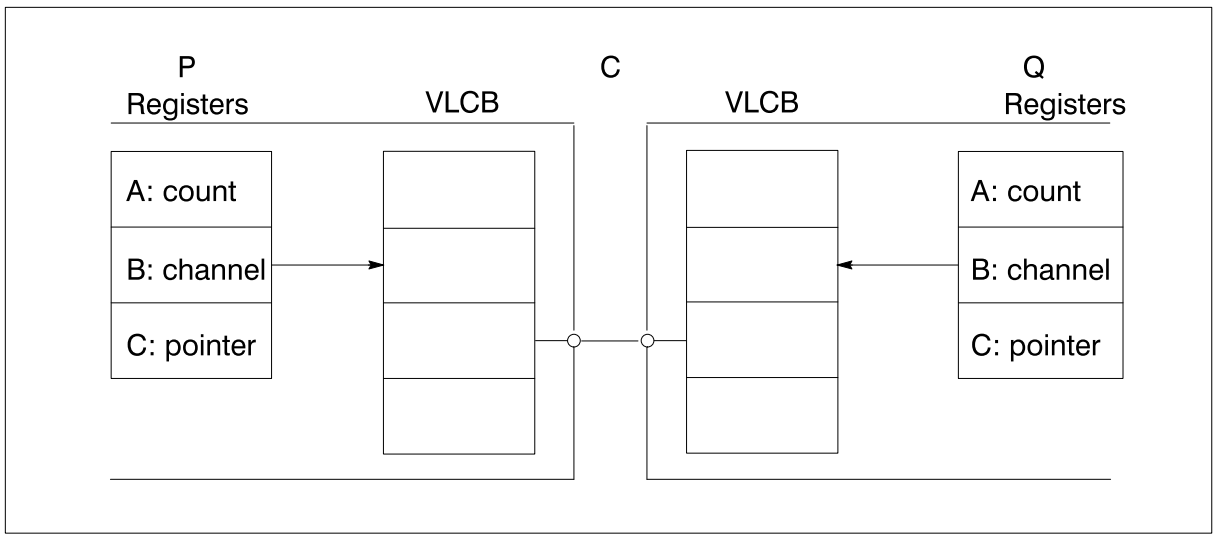

Figure 2.10 Communication between transputers

The VCP, on being told to output a message, stores the pointer, count and process id into the VLCB, and causes the first packet of the message to be sent. The VCP maintains queues of VLCBs for packets to be sent on each link, so the sending of a packet is in two parts; firstly adding the VLCB to the corresponding queue, and then subsequently taking the VLCB from the front of the queue and sending a packet, with the header provided by the VLCB. The queues of VLCBs are illustrated in figure 2.9.

Subsequently, on receipt of an acknowledge packet for this virtual channel, the VCP sends the next packet of the message. This continues until all packets have been sent. When the final acknowledge is received, the VCP reads the process id from the VLCB and causes the waiting process to be scheduled.

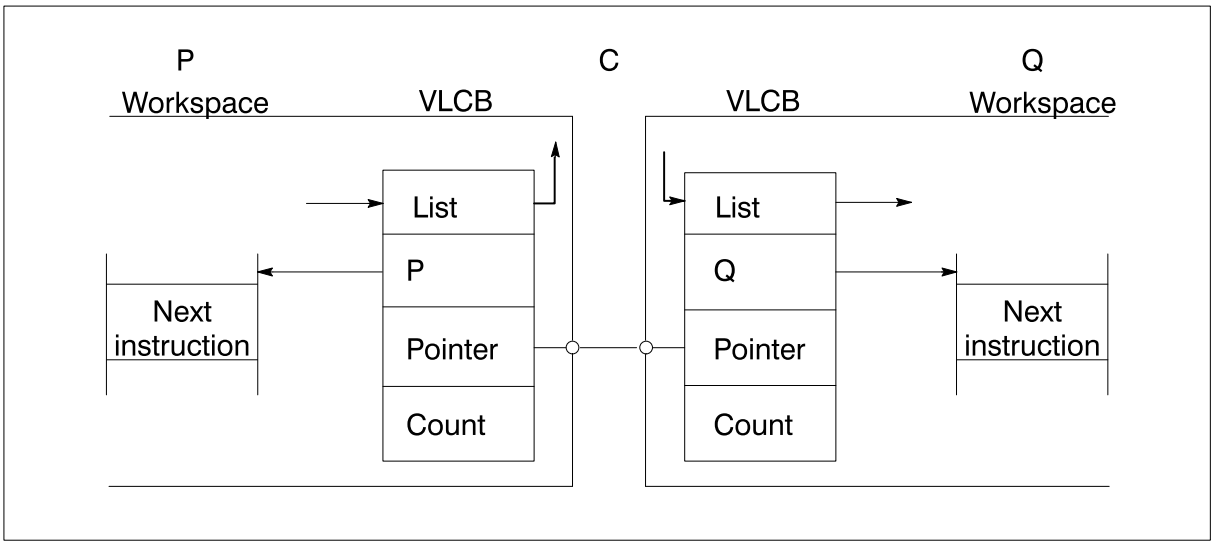

Figure 2.11 Communication in Progress

The receiving transputer's response to the first packet will depend upon whether a corresponding *variable input message* instruction has yet been executed. The VCP can determine this from the state of the VLCB associated with the virtual channel on which the packet has arrived. If an input instruction has not yet been executed, then the VCP stores the packet into the packet buffer provided by the VLCB, and an acknowledgement will subsequently be generated once an input instruction is executed.

When a process executes a *variable length input* instruction, the processor passes the process identifier, the virtual channel address, the pointer, and the maximum length, to the VCP and deschedules the process. The VCP, on being told to input a message, stores the pointer, maximum length and process id into the VLCB and records that an input has been requested. The VCP then examines the VLCB to determine whether a data packet has already arrived. If the data packet has already arrived, it will now be handled; otherwise data packets are handled as they arrive.

When a data packet is handled, the VCP acknowledges the packet by adding the VLCB to a queue for the sending of acknowledge packets. (Acknowledge packets are sent in just the same way as data packets, but use a separate set of queues.) The VCP then stores the data into the memory locations specified by the input instruction, provided that the total amount of data that has been received is not greater than the maximum amount specified. If more data than this is received then all data in excess of the maximum allowed is discarded. When a final data packet is received, the VCP reschedules the receiving process, having first recorded the amount of data received[4] into the process' workspace. This value will be used by a subsequent *load count* instruction.

The message is thus copied through the link, by means of the VLCBs at either end being alternately queued to send data and acknowledge packets respectively, as illustrated in figure 2.11. After all this is done the processes *P* and *Q* are returned to the corresponding scheduling lists as shown in figure 2.12.

4. If too much data is received, a special error value (= $FFFFFFFF_{16}$) is recorded instead.

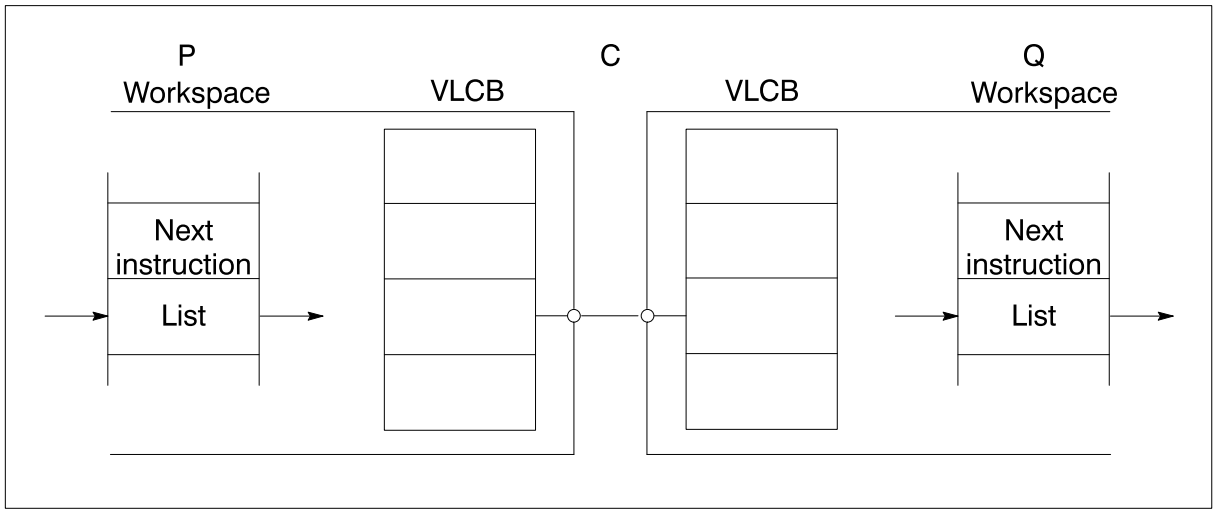

Figure 2.12 Communication completed

2.4.4 Known length communication

In many cases both the sender and receiver of a message know the precise length of the message to be transferred in advance. In this case it is possible to optimize the operation of message passing and the T9000 provides a number of instructions which do this. The most important of these are *input message* and *output message*[5].

These instructions are like *vin* and *vout* except that both the receiver and the sender specify the actual length of message to be passed. There is no need for an instruction which corresponds to *load count* in this case.

The operation of known length internal communication is similar to variable length communication. However, the first process to synchronize does not need to store the length, since the same length will be specified by the second process.

The operation of known length external communication is identical to the variable length case, except for the omission of the *load count* instruction.

2.5 Alternative input

In a system, it is sometimes necessary for a process to be able to input from any one of several other concurrent processes. For example, consider a process which is implementing a bounded buffer between two other processes, one of which (a peripheral of some kind) outputs data to the buffer along a channel, the other (the ”consumer”) requests data from the buffer along another channel, and receives it via a third, as illustrated in figure 2.13. The behavior of the buffer process is determined not only by its internal state, but also by whether the other processes wish to add or to take data from the buffer.

The alternative construct is a means to select between one of a number of guarded processes, each comprising a guard and an associated process; the guard is typically an input[6]. The alternative selects a guarded process whose guard is ready. If a particular guarded process is selected then both the guard and the associated process are executed. Guards may also have a boolean part which force the guard to be disregarded if the boolean is `FALSE`.

5. Note that this is the only form of communication supported by the first–generation transputers.

6. In principle, outputs could equally well be used as guards; however the implementation becomes considerably more complex if both inputs and outputs are allowed as guards. Thus in the T9000 output guards are not allowed.

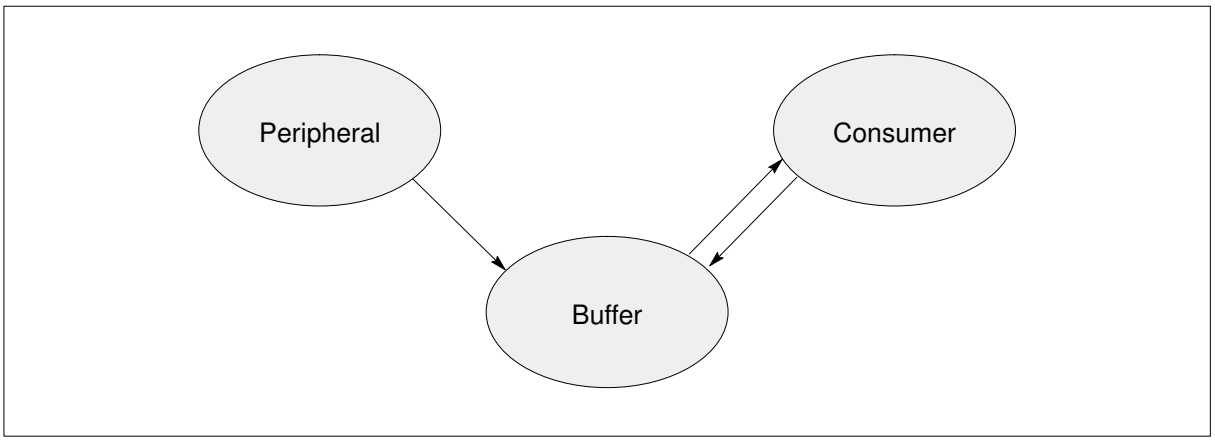

Figure 2.13 Buffer process

The T9000's implementation of alternative separates the selection of a guarded process from its execution. This means that the only new mechanism needed is one to support selection.

The idea behind the selection mechanism is that for each guard, the channel is examined to see if it is ready. If, when all the channels have been examined, no ready channel has been found, the process deschedules until at least one is ready. The process then re–examines the channels and chooses the first one that it finds ready. The key to the mechanism is therefore, the means by which a process can deschedule until one of several channels becomes ready.

The first aspect of this mechanism is that channels can be *enabled* and *disabled*. A channel is enabled (by the process performing the alternative) by executing an *enable channel* instruction. One effect of this instruction is that if the channel subsequently has an output performed on it, the output will signal the process performing alternative that the channel has become ready. An enabled channel is disabled by the process performing alternative executing a *disable channel* instruction, which reverses the effect of an *enable channel* instruction.

The second aspect of the mechanism is the use of a special workspace location by the process performing alternative. This location serves a number of purposes. Firstly, in the case of a straightforward input it is used to hold the pointer to the location to store the message, as discussed previously; consequently it is referred to as the "pointer location". Secondly, whilst an alternative is being performed, it contains one of the special values `Enabling` (= `NotProcess` + 1), `Waiting` (= `NotProcess` + 2), or `Ready` (= `NotProcess` + 3). As no process which is performing a normal input could be descheduled with one of these values in its pointer location (processes being forbidden to input messages to these addresses), the value in the location distinguishes a process performing alternative from an inputting process. Thirdly, it is used to record whether any channel which has been examined is ready. Finally, it is also used to record whether a process performing alternative is active or descheduled.

The implementation of alternative can now be explained.

Alternative start

The first thing that a process does to perform an alternative is to execute an *alternative start* instruction. This sets the pointer location of the workspace to the value `Enabling`, indicating that an alternative is in progress, that no guard has yet been seen to be ready, and that the process performing alternative is active.

Enable channel

The process performing alternative then executes an *enable channel* instruction for every channel guard. This instruction determines whether the channel is ready, and, if it is not ready, the instruc-

tion enables it. If, on the other hand, the channel is ready the instruction sets the value in the pointer location to `Ready`.

For an internal channel, the processor determines whether a channel is ready by examining the channel word. If it contains the identity of another process, then that process has performed an output on the channel, and so the channel is ready. Otherwise, the channel is empty, and so is enabled by writing inti it the process id of the process performing alternative.

For a virtual channel, the processor uses the VCP to enable the channel. The VCP examines the VLCB of the channel; if the packet buffer already contains the first packet of a message then the channel is ready. Otherwise, the VCP records in the VLCB that the channel has been enabled.

Alternative wait

Once a process has enabled all the channels from which it wishes to make a selection, it executes an *alternative wait* instruction. This first writes the value `-1` to location 0 of the workspace, in preparation for the selection process. Then, if the pointer location still contains the value `Enabling`, indicating that no guard is yet ready, the instruction writes the value `Waiting` into the pointer location, indicating that the process performing alternative is not active, and deschedules the process. Otherwise, the pointer location contains `Ready`, indicating that at least one guard is ready, so the process continues to make its selection.

If a process deschedules on execution of an *alternative wait* instruction, it will be scheduled when one of the guards becomes ready. The process will then proceed to make its selection.

Output on an enabled channel

When an output occurs on an internal channel which contains a process id, the sending process distinguishes between a channel which is ready for input and a channel which is ready for alternative input by examining the pointer location of the waiting process. If this word contains one of the special values `Enabling`, `Waiting`, or `Ready` then the channel is in use by a process performing an alternative. In this case the sending process will store information into its own workspace and deschedule as if the inputter were not ready, and may also perform some other actions, depending on the value in the pointer word of the receiving process:

- If the value is `Enabling` then the output instruction changes the value to `Ready`, indicating that an enabled channel is ready.
- If the value is `Waiting`, and hence the process performing alternative is descheduled, then the output instruction changes the value to `Ready`, and schedules the process performing alternative.
- If the value is `Ready`, the output instruction performs no additional action.

When an output occurs on an enabled *virtual* channel, the VCP of the outputting transputer will send the first packet of the message as usual; indeed, the sending transputer has no indication that the channel has been enabled. When the first packet arrives on an enabled virtual channel, the VCP places the packet in the packet buffer, and records that a packet has arrived as is normal for for a channel on which no input has been performed. The VCP also informs the scheduler that an enabled channel has become ready. The scheduler will then examine the pointer word of the process which enabled the channel and performs the same actions as an output instruction executed by a local process, as described above.

Once an output has been performed on an enabled channel two conditions are true; firstly, that the process performing alternative is active (either because it has not descheduled, or because a channel which has become ready has scheduled it); and secondly, the pointer word of the process performing alternative has the value `Ready`. These two, together with the condition for desche-

duling when an *alternative wait* instruction is executed, ensure that a process executes the instruction following an *alternative wait* instruction if, and only if, at least one guard is ready.

Disable channel

The process performing alternative selects a guarded process by executing a *disable channel* instruction for each guard and then executing an *alternative end* instruction. In addition to the channel address, the *disable channel* instruction takes a code offset as a parameter. This is the offset from the *alternative end* instruction to the code for the guard. If the *disable channel* instruction finds that a channel is ready, then workspace 0 is examined; if it contains a value other than `-1` then a selection has already been made, so no further action is taken. If it contains `-1` then this is the first ready channel to be disabled and the code offset associated with this channel is written into workspace 0.

The operation of *disable channel* depends on whether the channel is internal or is a virtual channel.

For an internal channel, the channel word is examined. If it contains the identity of the process performing alternative, an output has not been performed, the channel is not ready, and the instruction resets the channel word to `NotProcess.`. If the channel contains the identity of a sending process, then the channel is ready and may be selected.

For a virtual channel, the processor uses the VCP to disable the channel. The VCP examines the VLCB of the channel; if it contains the first packet of a message then the channel is ready. Otherwise, the VCP removes the information that the channel is enabled from the VLCB.

Alternative end

When all the guards have been disabled, one will have been selected, because guards are not disabled until at least one is ready, and the first ready guard that is disabled will be selected. The process performing alternative jumps to the code corresponding to the selected guard by executing the *alternative end* instruction. This instruction reads the code offset from workspace 0, and adds it to the instruction pointer. In this way the guarded process corresponding to the selected channel is caused to be executed.

A note about boolean guards

In the above, the fact that the guarded processes can have boolean guards has been overlooked. In fact, the *enable channel* and *disable channel* instructions take an additional parameter which is the boolean guard. If the guard is `FALSE` (= 0) they perform no action.

2.5.1 Extensions of alternative

Prioritized and fair alternatives

The T9000's alternative mechanism actually implements a prioritized alternative, the guards being prioritized in the order in which they are disabled. This can be directly useful; for example, consider a bounded buffer where we wish to prioritize receiving data from the peripheral over supplying it to a consumer. This can easily be achieved by always disabling the channel to the consumer process[7] first, so that if both the peripheral and the consumer happen to be ready, the alternative end instruction will always find the offset to the code which interacts with the peripheral.

The prioritized alternative which is actually provided can also be used to implement fair alternatives. For example, if we wish to ensure that the bounded buffer on average favours neither the

7. Since the implementation only provides for input guards, it is necessary to use two channels between the buffer and the consumer process, so that the consumer can perform an *output* to the buffer to indicate its readiness to receive an item.

peripheral nor the consumer, then this can be achieved by always disabling first the channel which was not selected on the previous iteration of the buffer control loop.

Other guards

In addition to inputs from channels, alternatives allows two other types of guard which may be used in addition to, or instead of channel guards.

The first is a `SKIP` guard, which is always ready. This guard is useful in conjunction with boolean guards, and is supported by the *enable skip* and *disable skip* instructions.

The second is a timer guard, which can be used for implementing timeouts, or for arranging for several different time related operations to be scheduled by a single process. The implementation of timer guards is built upon the implementation described above. However, some extra mechanisms are needed, and this necessitates the use of the *timer alternative start* and *timer alternative wait* instructions, rather than *alternative start* and *alternative wait,* for any alternative which contains timer guards. Timer guards are supported by the *enable timer* and *disable timer* instructions.

2.6 Shared channels and Resources

2.6.1 Alternative

The alternative mechanism is very general. It allows a choice to be made between channels, `SKIP`s and timers; each guard of an alternative may contain a boolean part; and the choice between guards is prioritized. Furthermore, there is complete freedom about how the channels are used both within and outside the alternative. It is this generality that necessitates the enabling and disabling of all the guards every time an alternative is executed, a consequence of which is that the cost of an alternative is proportional to the number of guards. This cost is incurred every time a selection is made.

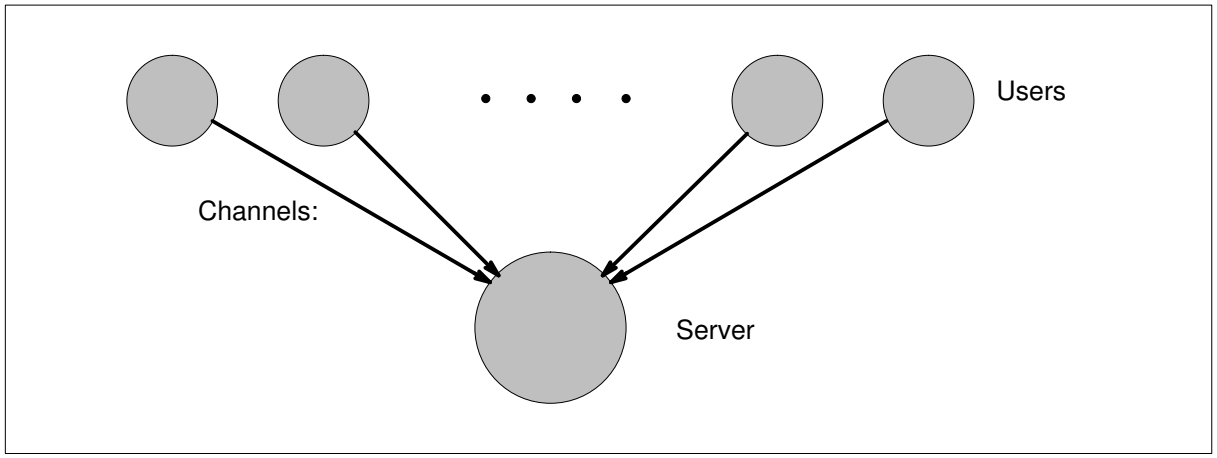

Figure 2.14 Server and users

2.6.2 Servers

One common use of an alternative is to implement a server, or to provide access to a resource. For example figure 2.14 illustrates the notion of a simple server which offers a service to N users, each connected to the server by one of an array of channels.

As the provision of the service may involve further interaction with the user, it is necessary for the code which provides the service to be passed its identity. In this case, the index of the channel in the array identifies the user.

In addition to the potentially large cost of the alternative, there is another potential drawback to this implementation of a server; this is, that the server must know the identity of all the channels connecting to users, since it has to enable and disable them in order to select one. A user cannot use a resource that does not know about the channel along which it communicates. A further difficulty is that fairness between the users is complicated to implement.

The T9000 provides a communication mechanism called a ‘‘resource’’ which overcomes both of these these problems . A resource may be thought of as a shared channel which connects a number of ‘‘user’’ processes to a ‘‘server’’ process.

2.6.3 Sharing a channel by a semaphore

Before describing the T9000's resource mechanism and its use, we will first consider another mechanism that might be used.

Using an efficient semaphore mechanism (which the T9000 does provide), we could implement resources by means of a single communication channel, whose use was shared by means of a semaphore. The resulting system would comprise a channel, used to synchronize with the server, and a queue of processes waiting to use the server, belonging to the semaphore. Whilst this mechanism would work, it has two drawbacks:

- It is not a distributed mechanism – it would work only on a single transputer.
- It no longer allows channels to be used as an abstraction. Rather than merely communicating via a channel, a user would have to first claim the semaphore.

The resource mechanism overcomes both of these problems.

2.6.4 Resources

A resource connects a number of user processes to a single server process. The resource comprises a number of *resource channels*, one for each user, and a *resource data structure* (RDS). A user process communicates with the server by outputting on its resource channel, exactly as if it were an ordinary channel. The server selects a resource channel by executing a a *grant* instruction with the address of the RDS. Once a user process has output on a resource channel, the grant will deliver the *identifier* of the chosen resource channel to the server. The server can then input from the chosen resource channel. Thus the operation of a resource is like that of an alternative, in that the functions of selection and communication are separated.

The identifier associated with a resource channel is a single word value which is delivered to the server on completion of a grant. This is the only information delivered to the server to identify the chosen resource channel, and hence, the user. Although it might seem as though the server should receive the address of the chosen resource channel, this is not always adequate. For example, in the server shown in figure 2.14 above, the service–providing code may need the index of the channel rather than the channel itself, so that it can use this index in an array of reply channels. On the other hand, if the channel address is what is wanted, then the identifier can be set to be the channel address.

Resource data structure

The resource data structure contains one word used to synchronize the server process with a user process, and a pair of words used to implement a queue. Unlike a channel shared by a semaphore, the queue is not a queue of waiting processes, but a queue of resource channels, each of which has been output to by a user process.

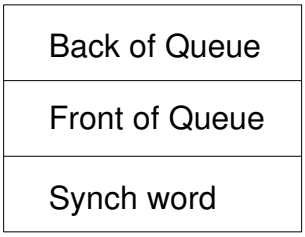

Figure 2.15 Resource Data Structure (RDS)

An RDS is initialized by setting both the synchronization word and the front pointer to `NotProcess`.

Resource channels

A resource channel is a channel together with an a pair of words. In the case of a virtual resource channel, the extra pair of words are associated with the VLCB of the receiving (resource) side. The address of a resource channel does not distinguish it from an ordinary channel, and a resource channel which is not currently part of a resource may be used used just like an ordinary channel, in which case the pair of words is not used.

Resource instructions

In addition to the output instructions mentioned previously, there are three instructions provided to implement the resource mechanism. These are:

- *mark resource channel*
- *grant*
- *unmark resource channel*

The operation of *mark resource channel*

The resource mechanism allows resource channels to be made part of a resource (''marked'') by either the server or by the user[8]. A server may mark a resource channel irrespective of when the user outputs on the channel; a user must mark a resource channel prior to outputting on the channel.

A resource channel is marked as being part of a resource by the execution of a *mark resource channel* instruction. This instruction takes three parameters; a pointer to the resource channel, the identifier, and a pointer to the RDS. There are two possibilities: either the channel is empty, or an output has already occurred.

If the channel is empty, then the identifier and the pointer to the RDS are stored in the extra words associated with the channel. For an internal channel, the special value `ResChan` (= `NotProcess` + 2) is written into the channel word to indicate that it is part of a resource; for an external channel the VCP records this in the VLCB. This is illustrated in figure 2.16.

8. A user located on a different transputer from the server must arrange for a process local to the server to do this. This is discussed in section 2.7.3.

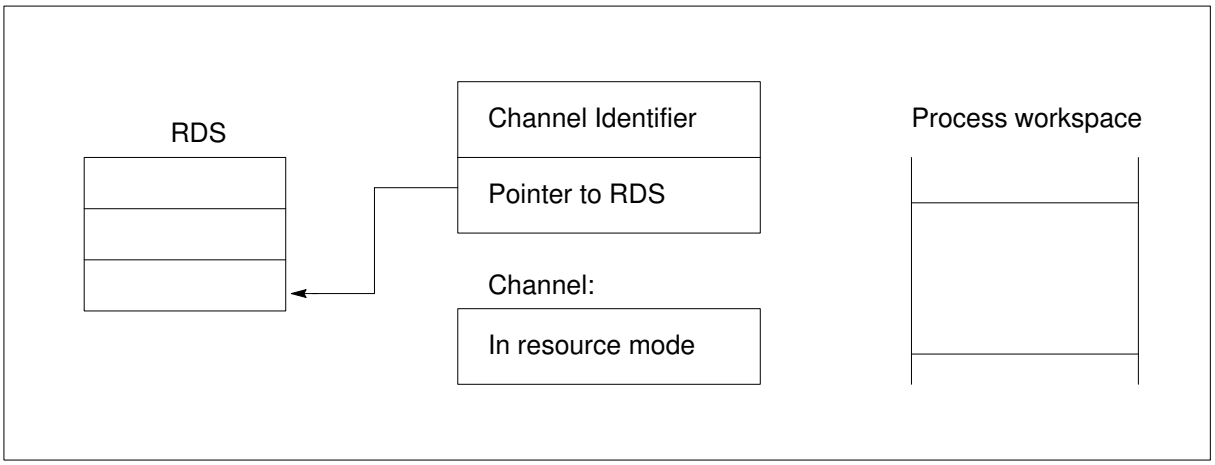

Figure 2.16 Channel after *mark resource channel* but before *output*

If an output has already been executed on the channel, then the *mark resource channel* instruction must be being executed by the server. In this case the channel will be queued on the RDS, using the first of the pair of words to form a linked list, with the second extra word containing the identifier. This is illustrated in figure 2.19.

The operation of *grant*

A server process grants use of a resource by loading the evaluation stack with a pointer to the resource data structure and a pointer to a location which is to receive the identifier of the granted resource channel, and then executing a *grant* instruction.

If there is a resource channel on the queue, it is dequeued and its identifier is written into the location provided for it. The server then continues and can input from the (now unmarked) resource channel.

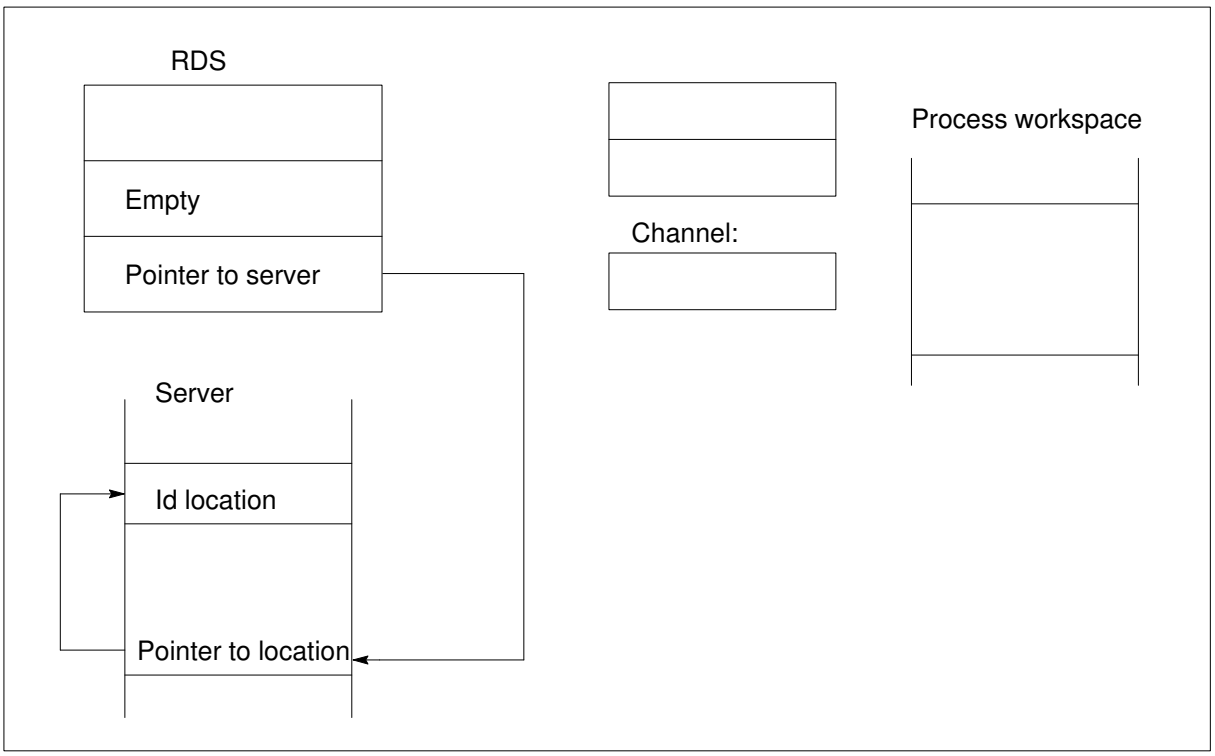

Figure 2.17 RDS and Server after *grant*

If there is no resource channel on the queue, then there is no user process waiting for the resource. In this case the instruction writes the process id of the server into the synchronization word of

the RDS, writes the address to where the identifier will be written into the workspace of the server and deschedules it. This is illustrated in figure 2.17. The server will be rescheduled when a user outputs to the resource. Thus the resource mechanism also provides non–busy waiting, just like alternative.

Note that once a resource channel is granted to a resource it becomes unmarked. It must be re–marked before it can be used as part of the resource again. In the meantime it can be used as a normal channel.

The operation of *output*

An output performed on a unmarked resource channel is indistinguishable from an output on an ordinary channel, as illustrated in figure 2.18.

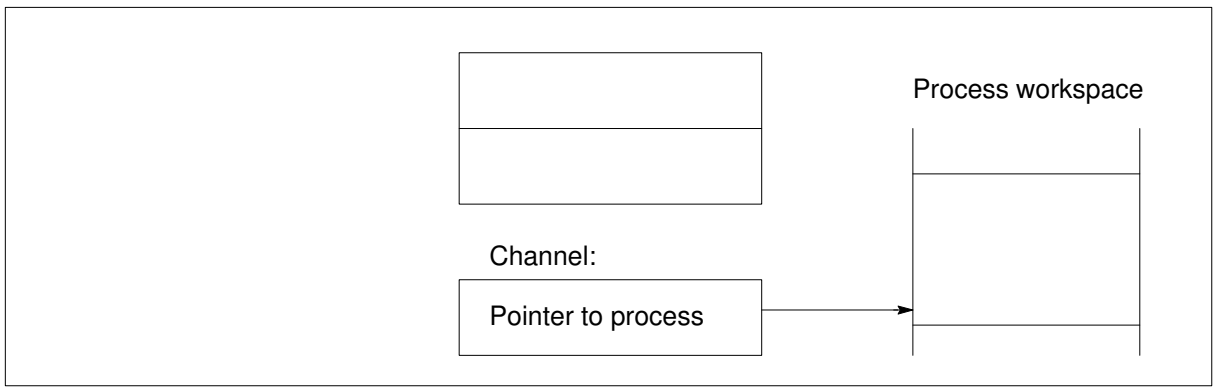

Figure 2.18 Channel after *output* only

When an output is performed on a marked internal channel, the output instruction reads the channel word in the normal way. On discovering that it contains the special value `ResChan`, indicating that it is a marked resource channel, the instruction reads the pointer to the RDS from one of the extra words of the resource channel and examines the RDS.

If there is no server present in the RDS, the output instruction queues the resource channel onto the RDS, as shown in figure 2.19. If there is a server present, then the instruction grants the channel to the server; the channel word is set to the process id of the sending process, the resource channel's identifier is written into the address specified in the pointer location of the server's workspace, and the server is rescheduled, as shown in figure 2.20.

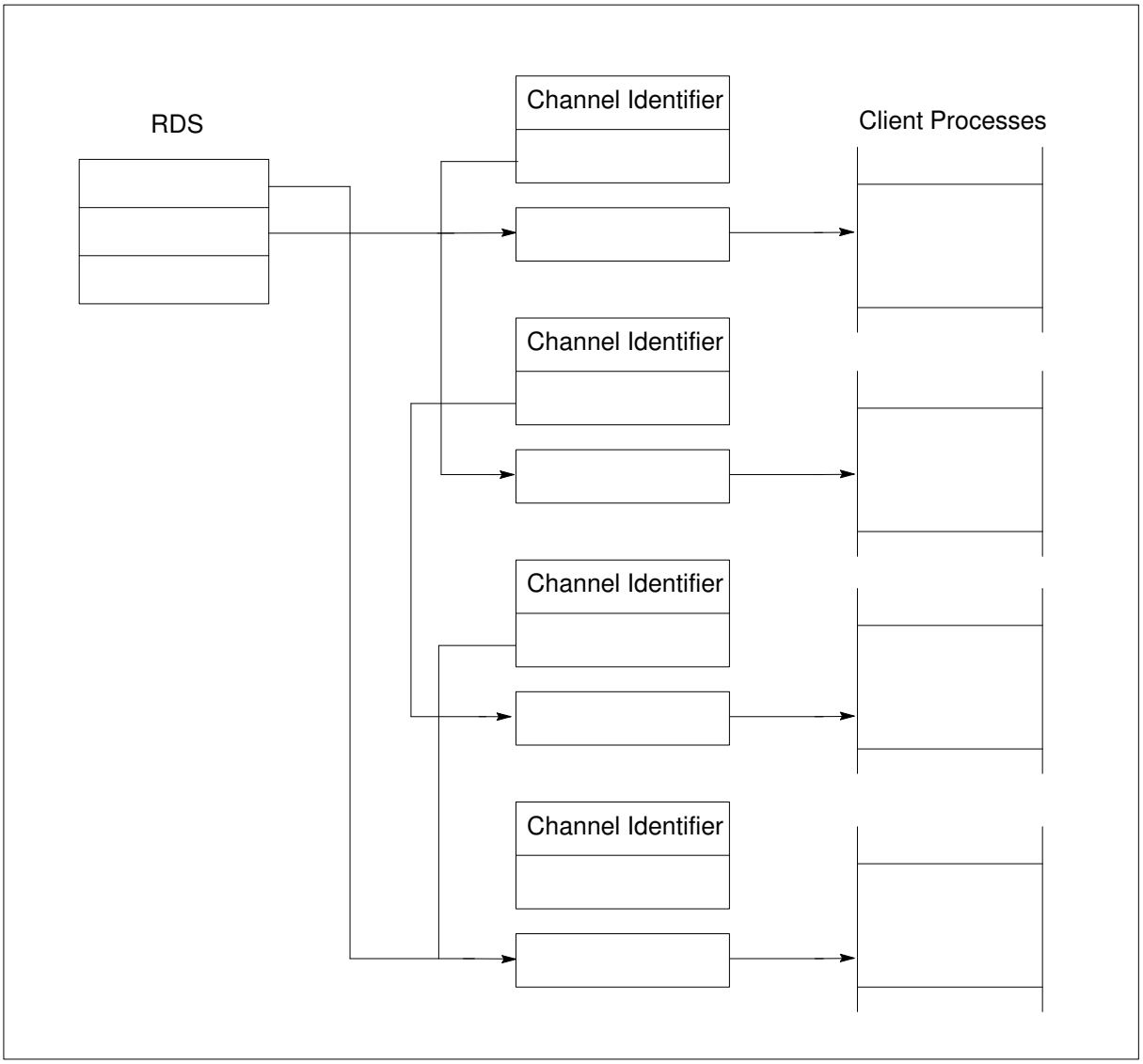

Figure 2.19 Four resource channels after *mark resource channel* and *output*

When an output is performed on a marked virtual resource channel the first packet is transmitted in the normal way. Indeed, there is no indication at the output end of the virtual channel that the channel is a resource channel. When the packet arrives at the receiving transputer, the VCP will notice that the packet has arrived on a marked resource channel, and cause the associated RDS to be examined by the scheduler. If there is no process id of a server present in the RDS, then the scheduler queues the resource channel on the RDS as shown in figure 2.19. If there is a process id in the RDS, then the channel is unmarked and granted to the server. The scheduler reads the pointer to where the server wishes the identifier to be stored from the server's pointer location, stores the identifier there, and reschedules the server as shown in figure 2.20.

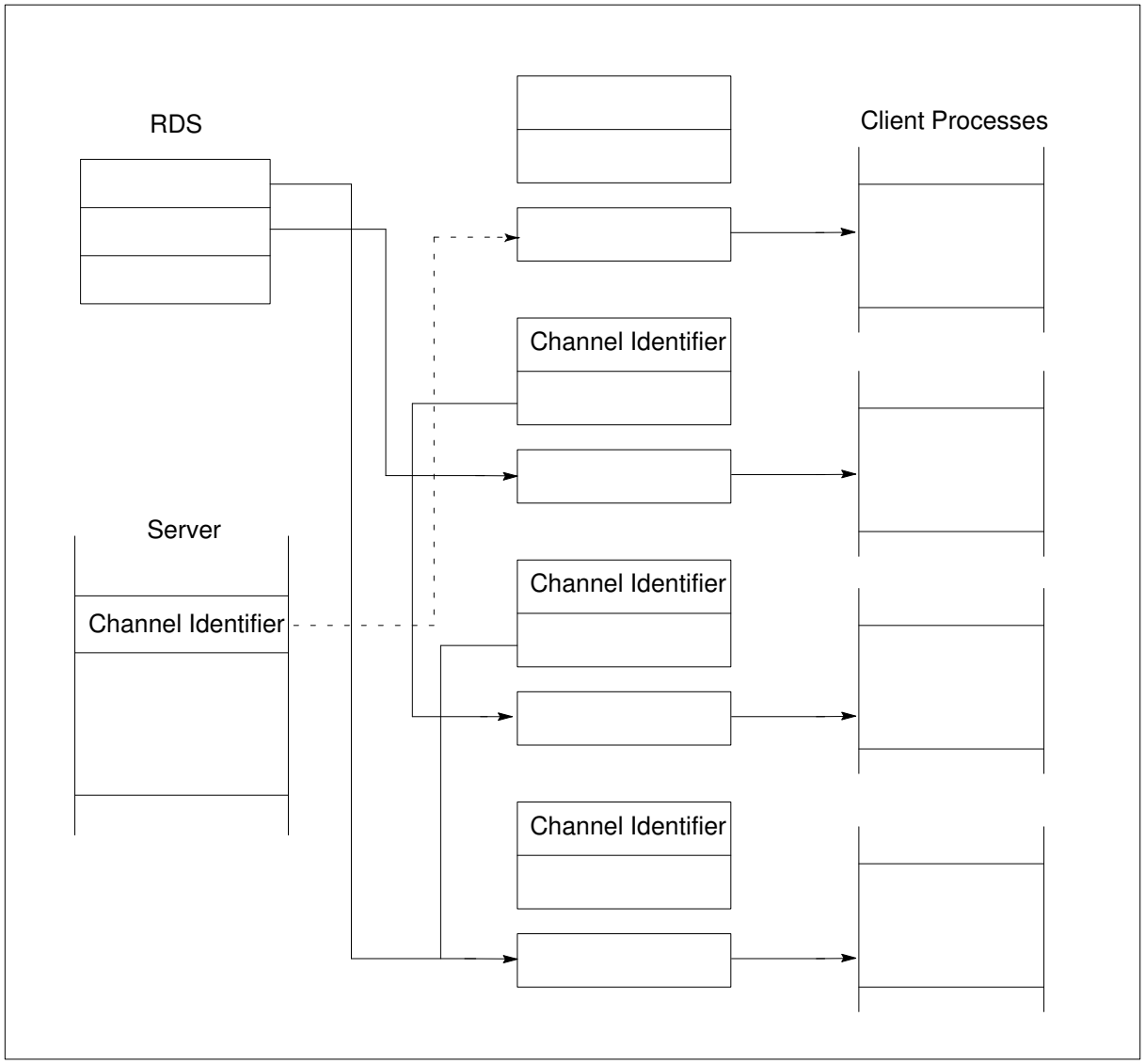

Figure 2.20 RDS with queued resource channels and server after *grant*

Note that in both the internal and external case the resource channel is then in the same state as a channel after an output has been performed and before the corresponding input has been performed, as shown in figure 2.18.

2.7 Use of resources

The T9000's resource channel mechanism can be used in several ways, three of which we now discuss.

2.7.1 Resources as a replacement for alternative; Omniscient servers

Consider the server example shown in figure 2.14, in which a set of users request some service from a server by communicating on an array of channels. We assume that the central server process repeatedly chooses a user which has requested it, provides some service for a time, and then chooses another user. If no user requires the service, the server will wait non–busily.

Although this can be implemented directly using the T9000's alternative mechanism, the cost may be too high if there are a large number of users, and the time taken to perform the service is small. However, if this is so, we can implement the above server using a resource.

The server process first creates and initializes a resource data structure, and then marks all of the resource channels in the array as being part of that resource. The identifier of each resource chan-

nel is set to the index of that channel in the array. The server then repeatedly selects a user by performing *grant*, inputs from the chosen user and provides the service. The granting of the chosen channel enables it to be used as an ordinary channel, and so the server has to re-mark the channel to include it in the resource when the server has completed this iteration. Finally, if and when the server terminates, the channels may have to be placed in a state where they can be used again as ordinary channels. This is done by means of the *unmark resource channel* instruction.

In order that the new code works correctly, the channels must have been allocated as resource channels. This can be achieved either by allocating all channels as resource channels, or by allocating only those channels used in resources as resource channels, in order to optimize memory usage.

This implementation has a one-off set up and take down cost, proportional to the number of users, and a *constant* per-iteration cost which is *independent* of the number of users. The users (sending processes) cannot distinguish between this implementation and one using alternative – or indeed one in which every user is provided with its own server, which simply performs input!

The use of resources instead of alternative is efficient only where a number of constraints are obeyed. Boolean guards and explicitly prioritized selection must be avoided, and the server process must interact with only the selected user, and not with any other users.

2.7.2 Resources in alternatives

Although the above has been suggestive that resources are some sort of a replacement for alternatives, they are in fact complementary. Resources may be used as guards in alternatives by means of the *enable grant* and *disable grant* instructions.

The use of resources in this way is very natural. For example, consider a bounded buffer process, with several providers of data and several users thereof, as illustrated in figure 2.21.

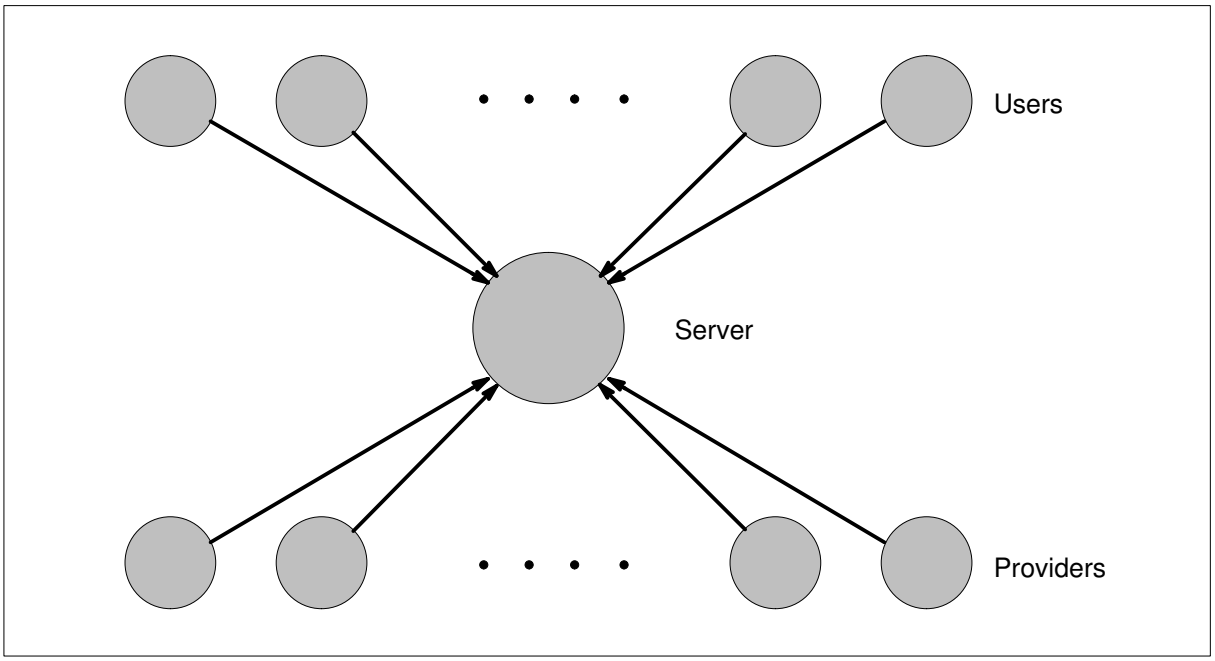

Figure 2.21 Server with users and inputs

This can be implemented using two resources, one for the users and one for the providers. The server can use an alternative to select between the users as a group and the providers as a group, and then within each branch of the alternative it can make a further selection by the resource mechanism as already described. This ensures that the server will wait (non–busily) until either

a user or a provider is ready to communicate. When there are many inputs and users waiting, the server can prioritize either users or providers within the alternative as previously explained.

2.7.3 Ignorant servers

We have seen how to use resources instead of alternatives. In that case, the server knows through which channels its users communicate, and how many users there are, but the users are unable to distinguish the resource from an alternative. We now consider how resources can be used when the server and the users know only the location of the RDS. In this case the resource channels can be generated dynamically as needed.

We start by explaining how to do this where the users are located on the same transputer as the server, and then we explain how to do this where the users and server may be located on different transputers.

Local server and users

In this case the user knows that it is going to use a resource channel and knows the RDS of the resource. The user allocates three words of memory for use as a resource channel, initializes the channel part to `NotProcess`, and executes a *mark resource channel* instruction which specifies the RDS of the resource and gives the address of the channel itself as the identifier of the channel. The user then performs an output on the channel. The server, when it grants this resource channel, will be delivered the address of the channel, and can then input from the user. In practice, it will probably be necessary for the resource to be able to output to the user, as well as the user outputting to the resource. A channel can be established in the reverse direction according to some convention known both to user and server.

Distributed servers and user

The distributed case is more complex because the user cannot initialize and mark a resource channel by itself. Firstly, as the user and server are located on different transputers, a *virtual* resource channel must be used. It must first be allocated, then both ends of the virtual link must be initialized. Once this has been done something must mark the input side of the virtual channel; this something must be executed on the same transputer as the server, not on the same transputer as the user!

However, if we assume the existence of a distributed kernel, capable of allocating, initializing and marking virtual channels, the distributed case becomes straightforward. Firstly, the user asks the kernel to initialize and mark a virtual channel connected to the server[9]. The kernel then cooperates with the kernel on the server's machine to initialize the virtual channel, and then the local kernel waits for the remote kernel to mark the virtual channel. The local kernel then informs the user process of which virtual channel to use, and the user process proceeds to output on that channel.

2.8 Conclusion

The T9000 transputer and C104 router provide the mechanisms necessary for the construction of large concurrent distributed systems. The T9000 provides a process and communication model, based around synchronised message passing over unidirectional point-to-point channels including an efficient and non-busy implementation of message passing, alternative and resources.

The communication system of the T9000 enables channels to be established between processes executing on different transputers, and for the same communication model to be maintained whether processes are located on a single transputer, or on a number of transputers.

9. The kernel can appear as a local server to the user.

When two T9000 transputers are directly connected, many virtual channels are provided in each direction between processes on the two transputers. If C104 routers are used, a network may be built which allows processes distributed over any number of transputers to communicate. The scheduling and communication mechanisms of the T9000 provide efficient support for a wide variety of operating system kernel functions and concurrent programming constructs.