IDN-14

Butterfly Gateway Process Structure

Eric Rosen

February 1984

INTRODUCTION

The processing which the Butterfly gateway will need to perform upon a received datagram can be divided into several steps:

- First, the datagram must be physically received over the device. The gateway will have to control this transfer by means of an input device driver.

- Then the gateway will need to execute some protocol or set of protocols (network access protocols) with the entity that is transmitting the datagram. In some cases (e.g., ethernet) this protocol will be relatively simple and straightforward, but in others (e.g., X.25) there may be two ("level 2" and "level 3") rather complex protocol state machines to maintain.

- Once the network access protocol is completed, and the packet correctly received, the gateway must the determine internet transport protocol being used. It might be a private gateway-gateway protocol (say, used for transmitting routing updates or neighbor up/down packets), or it might be the standard IP. If it is the standard IP, the datagram might or might not be addressed to the gateway, and this needs to be determined.

- If it is addressed to the gateway, further protocol demultiplexing is needed, to be followed by further processing specific to the particular protocol. This might or might not result in the need for the gateway to create and transmit a datagram of its own.

- Otherwise, the gateway will have to transmit the datagram to some other entity. It will have to select an output interface over which to send the datagram.

- It will have to execute the level 3 and level 2 output protocols, and then turn the datagram over to a device driver for that output device.

In order to implement the gateway in the Butterfly, these functions have to be distributed among a set of Chrysalis processes. A Chrysalis process can be thought of as a set of procedures which have been linked together into a single memory image, and which are treated by the Chrysalis scheduler as an indivisible schedulable entity. There are many different ways one might divide the gateway into processes. The purpose of this note is to explain the way we have chosen, and to explore the various trade-offs. The mechanisms we intend to use for inter-process communication are also discussed.

There are a number of mutually conflicting factors to keep in mind when choosing a process structure:

- I/O latency requirements. This factor applies primarily to the device drivers which run synchronous high speed interfaces. When certain sorts of external events occur (e.g., the input device signals "end of message"), there is typically only a small interval of time during which the device driver must run and perform a certain amount of work. If the driver cannot run or cannot complete its task before this interval elapses, inputs may be lost. If this happens

frequently, performance becomes very poor. Similarly, on output, if the next output cannot be started soon after an output completion interrupt, throughput may be low.

- Fair distribution of the CPU resource. The process structure should be such that the "time-slicing" and "declared need" features of the Chrysalis scheduler can be used to good advantage to help ensure that the various gateway functions all get adequate running time.

- Make best use of the built-in services of the operating system, rather than trying to duplicate these services within each process. For example, one would want to avoid having each process implement a lot of internal scheduling if the operating system's scheduler could be made to do the same thing.

- The software should be divided into processes in a way which facilitates the existence of clean, modular interfaces between the processes. That is, one should try to devise a process structure which minimizes the amount of knowledge which each process must have of the internal data structures

or algorithms of any other. (Protocol layering is a special case of this.)

- The number of processes running on a single processor node should be kept relatively small, to avoid running out of SARs, and to generally simplify system overhead.

- The number of memory segments in the virtual address of a process should be kept relatively small, in order to avoid running out of SARs.

- The process structure should facilitate parallel processing wherever that is feasible and would serve to increase the throughput or reduce the delay of the system.

- The process structure should facilitate a variety of different assignments of processes to processors, so that the software configuration in a particular gateway can be matched to the particular hardware configuration for best delay/throughput performance. (However, we do not see a need to have the system automatically "discover" its best configuration; this can be done by means of "config tables" which are created off-line.)

- Both the process structure and the inter-processor communications mechanisms should be optimized for the high throughput path. One should avoid having a lot of overhead devoted to the case of messages which, while they may be high priority, are only a very small fraction of the messages seen by the gateway.

- The process structure should be designed with an eye to minimizing (other things being equal) the amount of interprocess communication that is needed.

Needless to say, these desiderata are not fully objective, nor are they capable of being fully and jointly met. The design process is largely a process of making trade-off among these considerations. The following description of the process structure design will attempt to explain the way the trade-offs were made, and why, in our design.

PROBLEMS WITH DEVICE DRIVING IN THE BUTTERFLY

The Chrysalis scheduler, rather unfortunately, was designed on the assumption that none of the schedulable processes would have to meet strict latency requirements. This assumption shows up in three ways:

1. Only a very small number (4) of preemptive priority levels are available.

2. Scheduling and context-switching overhead may make it hard to meet latency requirements.

3. Within a fixed interval of time known as an "epoch", the scheduler will attempt to ensure that ALL processes (even those of the lowest priority) get some CPU time, even if this means that a higher priority process gets held off to let a lower priority process run. This feature is welcome from the perspective of fairness in the sharing of the CPU resource, but it conflicts directly with the need to meet latency requirements.

The only existing communications applications of the

Butterfly, viz., the Voice Funnel and the BSAT, have off-loaded all high speed device driving functions into a subsidiary processor, the BIO. The BIO contains the complete device driver, and interfaces to the 68000 via linked lists of "channel control blocks", a structure defined in the Chrysalis operating system specifically for communication with the BIO. The BIO also deals specifically in terms of Chrysalis buffers, and will send and receive packets to or from linked lists of Chrysalis buffers, performing the buffer chaining automatically if a single packet does not fit into a single buffer. With all I/O controlled by the BIO, There is no need for the 68000 to field input completion interrupts, thereby removing strict latency requirements from all processes which run in the 68000 under control of the Chrysalis scheduler. The BIO provides similar services on the output side, which eliminate the need for the 68000 to field output completion interrupts also.

However, the gateway will not utilize the BIO or, at least initially, any other I/O interface of comparable sophistication. The Butterfly gateway will have a multibus to which standard network I/O devices can be connected. Plans are

to use the PAMPERS board for 1822 and HDLC interfaces, and the Interlan board for ethernets. These devices require the presence of device drivers in the 68000, since they cannot deal in terms of Chrysalis structures (like buffers), and cannot perform buffer chaining. The Interlan board does use a hardware FIFO, protecting the 68000 to some extent from the strictest of latency requirements, but these devices still leave latency requirements which can be quite strict, or at least too strict to be met by Chrysalis processes running under the usual scheduler discipline. Ultimately, we hope to have more sophisticated ethernet interfaces (a much improved Interlan board is scheduled for the future), and to re-program the PAMPERS board to make it more "Butterfly-friendly", but initially, at least, we need to deal with relatively unsophisticated devices.

DEVICE DRIVERS -- INTERRUPT HANDLERS PLUS ASSOCIATED PROCESSES

We propose to deal with this problem by writing part of the device drivers as 68000 interrupt handlers, rather than as Chrysalis processes. The 68000 has 8 interrupt levels, numbered 0-7 from lowest to highest priority. Chrysalis processes (of all priorities) run at priority 0, and the Chrysalis scheduler itself runs as the priority 1 interrupt handler. Levels 3 and 4 are available for I/O interrupt handlers, i.e., device drivers.

There is no particular problem in writing interrupt handlers for the Butterfly. Since they are not run by the scheduler, however, they must be responsible for saving and restoring context. This seems to require only the saving and restoring of the registers (which can be done by a single instruction), and the saving, setting, and restoring of the ASAR register. (If the interrupt handler does not set the ASAR register itself, it will run in the address space of whatever process was interrupted. This would not seem to be satisfactory, in general, but it might be if it were guaranteed that every process on that processor node had all the buffers in that node's memory mapped in, and if all the device driver's data structures

were in common segments.)

Interrupt handlers "steal" time from the processes they interrupt, in that the scheduler does not take account of them at all. Therefore, if the system spends significant amounts of time in the interrupt handlers, the scheduler's ability to manage the CPU resource is severely compromised, and processes may not get the proportion of time they really need. The device drivers will therefore have to be kept as small (in time) as possible.

Each I/O device will have two (one for input and one for output) associated interrupt handlers. Each interrupt handler will have an associated Chrysalis process, which is the only process with which it can communicate directly. This process will be known as the "driver process". The device driver can thus be thought of as consisting of the two interrupt handlers and the associated process or processes. It is expected that the interrupt handlers will be written in assembler language for maximum efficiency, while all processes will be written in C.

The driver process will need to have the entire buffer pool of the local processor mapped in, so that it can transfer

data to or from any buffer in local memory. (I/O data transfer is not possible with memory on other processors, nor is any interaction with the I/O device possible from a remote processor.) It may be a useful convention to have the interrupt handlers set their virtual address spaces to that of the associated driver processes.

COMMUNICATION BETWEEN THE INTERRUPT HANDLER AND THE DRIVER PROCESS

OR:

I/O CONTROL BLOCKS -- ARE THEY NEEDED?

Is there a need for I/O control blocks, or can all the information that the interrupt handler needs to transmit receive data, as well as to record the status of the transfer, be placed in the buffer itself? There are two issues relevant to answering this question:

- Is there a lot of control and/or status information needed?
- Do we want to exclude the possibility of ever transmitting the same buffer simultaneously over two or more interfaces?

If the amount of necessary control or status information is very large, we wouldn't want to provide space for it in every buffer; that would be wasteful, since not every buffer can be in transmission at any one time. On the other hand, if the amount of information is small, then keeping it in the buffer is very convenient, since that would mean that we wouldn't need to manage yet another dynamically allocated resource (i.e., the control blocks).

However, if we want to be able to transmit a buffer out several interfaces simultaneously (e.g., multi-cast, routing updates), we cannot keep control and status information about each transmission in the same buffer; it would seem that control blocks would need to be used in that case.

Unfortunately, the gateway will probably not be able to make much use of multiple simultaneous transmission. A packet transmitted out several interfaces will generally need a different protocol header for each interface; the very same packet (physically) cannot really be transmitted out each interface. Even if the same protocol is used on two different interfaces, the address to which the message is sent over each interface would generally be different. This differs from the similar situation in the IMP, in which, for the transmission of routing updates, e.g., a single transmisison protocol is used for every "interface" (modem trunk), and no addressing is needed.

It may be unwise to have the design exclude the very possibility of multiple simultaneous transmission, particularly if the basic structure of the system is to be adaptable to other (more IMP-like?) applications. Further, it will not become clear

until the design of the device drivers is taken to a greater level of detail whether the status information which needs to be stored with each packet is large enough to warrant the use of control blocks to save space in the buffer headers.

If control blocks are used, managing them should be the responsibility of the driver process, not of the interrupt handler. That is, the process should get a free block (off a free queue, presumably), and fill it in with the information the interrupt handler needs. Then it should place the block on a queue for the interrupt handler. After transmission is complete, the interrupt handler can place the control block back on a queue for the associated process, which would look at the block and then free it. Given that the associated process needs to be on the same processor as the interrupt handler (see next paragraph for justification), it makes sense for these control block queues to be linked lists, rather than, say, dual queues. To prevent race conditions, the process will have to inhibit interrupts whenever enqueuing to or dequeuing from the list. It will be the responsibility of the driver process to ensure that the queue is properly ordered (by priority, perhaps), if that is a factor.

If control blocks are not used, the driver process can communicate with the interrupt handlers via linked lists of buffers, rather than via linked lists of blocks. Most of the other considerations are unchanged.

To meet latency requirements, or just to keep up throughput, it is necessary to allow the queue between the interrupt handlers and the driver processes to grow to a length greater than one. If control blocks are used, this implies a need for more than one control block per device. It is not clear at the present whether a control block would be needed for each buffer on the queue, or just one for each packet (linked list of buffers) on the queue. Clearly the latter is preferable, if it is feasible.

If a transmission is in progress at the instant that a control block is placed on the queue, the interrupt handler will be responsible for serving the queue when its current transmission completes. If, however, the device is idle, the driver process needs to take some explicit action to initiate the transfer. This is requires the driver process to be on the same processor as the I/O device and its interrupt handler; I/O

registers cannot be manipulated across the Butterfly Switch.

The interrupt handler will assume that any buffers on that queue are in the memory of the local processor. The interrupt handler will need to know the physical address of a buffer in order to use it for I/O. Rather than having to make the mapping from buffer ID to physical address every time a buffer is used for I/O, it seems to make sense for each buffer to carry its physical address in its header; this can be set up at initialization time. (This also simplifies one of the tasks of the DISPATCH process, viz., determining whether a given buffer is on the same processor as a given I/O device.)

Input drivers (at the interrupt handler level) will obtain free buffers directly from the main free queue of the local processor (a dual queue), unless that queue is empty. If it is empty, they will obtain free buffers from a "private" free queue, as explained in the note on buffer management. Since buffers may get placed on the free queue (even the private free queues) by processes running in remote processors, these will be made dual queues. (This allows the PNC microcode to prevent multi-processor race conditions in the enqueuing/dequeuing

operations.) The input driver will also have to obtain a control block from a free queue, which should probably be a linked list of blocks. Since only one process touches this queue, and that must be on the same processor, locking can be done simply by inhibiting interrupts, and the dual queue mechanism has no advantage over linked lists.

A fixed number of I/O control blocks can be assigned to each interface, or they can be made a dynamically allocated resource, like buffers (although there will be many fewer I/O control blocks than buffers). If they are dynamically allocated, then there will have to be "block management", whose algorithms will be much the same as those of the buffer management system described elsewhere. (The block management system could be made a smaller version of the same procedures.)

The interrupt handler will be responsible for linking together all the buffers into which a datagram has been read. When input is complete, either a control block pointing to the first of these buffers, or else the buffer itself, will be queued to the driver process. If this queue is made to be a dual queue, the driver process can be automatically woken up. If it is made

to be a linked list, the interrupt handler will have to explicitly post an event for the driver process to make sure it becomes runnable. This might be preferable, since it might make it easier to devise an algorithm which does not necessarily make the driver process runnable after every input. (The same consideration applies to the queue of returned control blocks which the interrupt handler passes back to the driver process after output completion.)

It is the intention that the interrupt handlers perform as few functions as possible, viz., only those needed to keep the I/O device running at full speed (or as close to that as possible) and to interface with the driver process, and that all other functions, in particular all Level 2 and Level 3 protocol functions, be performed by Chrysalis processes. The interrupt handlers will not be expected to check sequence numbers, validate checksums, perform buffer management computations, allocate control blocks, etc. It is the responsibility of the driver process to provide the interrupt handler with a long enough queue of datagrams, when available, to keep it busy until the driver process can run again.

PROTOCOL PROCESSES

Performing I/O requires more than just driving the device; there is protocol to be executed with the entity on the other side of the device. In some cases, there are two protocols, a level 2 and a level 3 to be executed. These must be executed by Chrysalis processes. There are a number of ways one might try to divide this functionality into processes. At one extreme, one could imagine that a processor might contain only a single protocol-executing process, which executed all the required protocols needed for all the various devices on that processor. This would have the virtue of minimizing the number of processes. At the other extreme, one could imagine that each device has four protocol-executing processes associated with it, one for input and one for output of the level 2 protocol, plus one for input and one for output of the level 3 protocol. This would have the virtue of reducing the amount of code needed by each individual process. There are also an arbitrary number of intermediate choices. We now consider some of the issues involved in choosing how to apportion the required functionality into distinct processes.

MAPPING PROCESSES TO DEVICE: ONE-ONE OR ONE-MANY?

Should a process which is responsible for executing a particular I/O protocol execute that protocol for all the I/O devices (or over all the I/O devices connected to a single processor node) which use that protocol, or should be there one such process per device? A process which is written to handle the protocol over just a single device is certainly going to be simpler than a process which must handle several devices. It will not need to maintain a set of data structures indexed by device, nor will it need to have an algorithm for scheduling the way the various devices are handled relative to each other. The scheduling is left completely to the Chrysalis scheduler.

The major disadvantage of having a separate process for each device is the unwanted multiplication of processes and consequent drain on the SAR resource. Perhaps this will not be too serious if the number of devices on a single processor node is relatively small, as it probably will be.

Leaving the relative schedulings of the various devices solely to the Chrysalis scheduler might be a mixed blessing. It

simplifies the protocol process, but it also removes a level of control from the application software. However, as long as the processes for all the devices run at the same priority level, and have reasonable time-slice and declared-need parameters, it is hard to see how the scheduler will show favoritism to any one device over another. This avoidance of favoritism is about all one can ask, and it is hard to see how the application software could do better.

This considerations suggest having a separate protocol process for each device, UNLESS we run out of SARs. Should this happen, it may be necessary to cut back on the number of processes by having a single process handle several devices. (However, that could only be a factor in gateways which had several devices running the same protocol.)

INPUT AND OUTPUT -- ONE PROCESS OR TWO?

Should the input and output functions needed to run a given protocol over a given I/O device be in the same process, or in two different processes? The major reason for wanting to put input and output in the same process is that the two are

generally intimately related in the state machine of the protocol. If separated into two processes, the output process would still need to know about the input headers in order to know whether to make certain state transitions. There would be a need for continuous inter-process communication which is very much simplified if only a single process is used.

The major reason for wanting to put input and output into two different processes is to avoid having to have the process itself make decisions about the relative scheduling of input vs. output functions. If a process has to serve both an output queue and an input queue, it needs to do some internal scheduling which might otherwise just be left to the Chrysalis scheduler. For example, a single process needs to have an explicit algorithm to ensure that neither input nor output locks out the other (perhaps by alternating its attention between input and output functions); if there are two processes, the Chrysalis time-slicing might prevent this automatically. However, in that case there would be no way for the application to control the input vs. output fairness algorithm. In some situations, the ability to tune this algorithm might be important to performance or robustness.

Fairness in handling input vs. output on a single device differs from fairness among several devices in that the former is much more intimately related than the latter. The ability to do I/O on one device is independent of what is happening on any other device. The ability to do output on a given device, however, may depend on what inputs have arrived recently over that device. One could imagine processing all piggybacked acks in packets on the input queue, then checking to see if this enables more output to be done, then going back and processing the packets on the input queue. This is not necessarily the way one wants to handle it, but it is an example of the close inter-relationship of input and output on a single device, and is something which would be more difficult to do if input and output were handled by separate processes.

Since we are speaking of the implementation of a single protocol, there is no argument from modularity or protocol layering suggesting that there should be two different processes. In fact, one might argue that considerations of layering and modularity favor (though not require) a single process handling both input and output, since that maximizes the simplicity of

replacing one protocol by another without affecting the rest of the system.

LEVEL 2 AND LEVEL 3 -- ONE PROCESS OR TWO?

In some cases, performing I/O over a given device requires only one level of protocol. This is the case for an ethernet interface, or for a phone line directly connecting two gateways (which might run HDLC or even IMP-IMP protocol, for example.) In many cases, however, there are two levels of protocol, Level 2 and Level 3. This is the case for X.25 and HDH interfaces. Should these two levels be implemented in a single process, or in two different processes?

A common misconception is that protocol layering requires that different protocols be implemented in different processes. The goal of layering does require the existence of clean modular interfaces between the procedures that implement the different layers, but these interfaces could well be implemented as subroutine calls rather than as inter-process messages. In fact, from the programmer's point of view, there is not that much difference between the two methods of passing data from one

protocol "module" to the other, since the programmer need not think too much about whether a given routine call is actually doing the work of the next protocol level, or is passing a message to some other process. If one were using subroutine calls to pass information "between" the two protocols, the most flexible and robust way to pass arguments would be to pass pointers to explicitly formed argument lists, i.e., pointers to structures, and there is little difference in programming effort between setting up a structure and setting up an inter-process message.

From a software engineering point of view, each of the two methods has some advantages and some disadvantages. On many systems, one wants to strictly minimize both the amount of inter-process communication, and the amount of process context-switching that is done. This is common when the operating system is a multi-user time sharing system, and one is trying to implement network protocols in user processes. On such systems, one would certainly make one protocol level be a "subroutine" of the other. The Chrysalis/Butterfly environment, however, is optimized for sending messages between processes (even when they

are running on different processors), allows easy access to common memory, and has relatively inexpensive context switching. (The context switching is not so inexpensive that one could countenance a context switch during the processing of each packet, but this is not implied by the scheme of having separate processes implement the separate protocol levels.) However, having more rather than fewer processes involves a cost in SAR usage.

In some situations, one might want to use global variables (i.e., shared memory) for communication between protocol layers, rather than message passing or argument passing. For example, this might be the best way of handling status information about the device itself (rather than individual packets), since that eliminates any delay which might be inherent in message passing. This does not distinguish at all between the two methods, since, in the Butterfly, different processes can easily share any number of memory segments.

In the gateway, we will want to have various mixtures of level 2 and level 3 protocols. We foresee some interfaces which are simply HDLC or possibly IMP-IMP (direct gateway-gateway

connections); some which are simply 1822; some which are HDH (1822 as level 3, HDLC as level 2, with a little special HDH processing thrown in); some which are X.25 (X.25 level 3 plus HDLC); and some which are SATNET/HDH. From a software configuration viewpoint, is it very convenient to ensure that a given process needs to be compiled and linked only once, and is then suitable for loading into ANY gateway, no matter what its hardware configuration. If some gateways will need 1822 combined with HDLC (HDH), and others will need HDLC combined with X.25 level 3, and others will need 1822 without HDLC, and others HDLC by itself, this desideratum is most naturally (at least, most naturally to Chrysalis) handled by having distinct processes for the different protocols. That is not to say it is the only way of handling it however. However, it could also be handled by the use of common library routines and transfer vectors, though the operating system does not really offer much help here at the present time.

(The ability to have library routines which reside in only one place in memory but are callable from several processes would be important here; in a gateway with one HDH interface and

one X.25 interface, one would not want to have separate copies of the HDLC "library" linked into two different processes. This is probably not impossible, but does require a bit of "fighting the system". While we intend to make some use of common libraries with re-entrant code, if possible, this will probably be confined to relatively small and self-contained functions, rather than major protocol modules.)

Since it seems natural to think of different protocols as different processes, and since this way of treating them is more congenial to the Chrysalis environment than the other, we have decided to adopt this procedure, at least unless it results in an unacceptable multiplication of processes. It will be an interesting challenge to try to see to what extent we can make the code independent of this decision. One would hope that switching between the process model and the subroutine model could be done by changing only a small and well-defined number of "interfacing" routines. Certainly, most routines need not know whether the message to which they have a pointer has been removed from an inter-process queue, or whether the pointer has been just been passed down from a higher level routine.

We needn't be dogmatic about the process model, however. We may occasionally need to implement a protocol which is so small, in space and time, that it is much less costly just to make it a subroutine (even if it has to be linked separately into every process that might need it in some configuration.) A good example is the HDH "level 2.5" protocol, which does so little (especially when used in "message mode", which I think is its preferred method of operation), that it would seem to make most sense to make it a subroutine in either the 1822 process or the HDH process.

DIFFERENT DEVICES, SAME PROCESSES -- CONFIGURATION

Thus every I/O interface will be associated with an interrupt handling routine, the driver process that interfaces to the interrupt handler, and one or two protocol processes. Where there are two protocol processes, we will speak of the LEVEL2 process and the LEVEL3 process.

In certain configurations (e.g., an HDH interface), a given process (e.g., HDLC) will, when it has completed its input processing of a packet, need to pass it to another protocol process (e.g., 1822). However, in other configurations (an HDLC line with no level 3 protocol running) the processed input will have to be otherwise disposed of. A similar point can be made with respect to output. For an HDH interface, the 1822 process must pass a processed output packet to the HDLC process. For an 1822 interface (local or distant host), the packet must be passed directly to the appropriate device driver. We would like the protocol processes to be identical in both cases. This implies that they must make the decision as to how to dispose of a processed packet by consulting configuration information in the gateway. We posit the existence, for each I/O device, of a

"device control block", which will be in a global common area. A process should need only to look in this block to determine how to dispose of a processed packet.

The device control blocks should also be used to contain any information about the state of the device (either configured state or dynamic state) which might need to be known by any process. This might include such things as whether it is up or down, its recent error rate, various statistics about its recent utilization, its bandwidth, the delay associated with it, whether it is available for use or blocked by protocol, etc., etc., etc. To the greatest extent possible, information like this should be represented in a device-independent manner. We should be able to represent a considerable amount of information about the device in this block, making the information immediately accessible to all processes without the need for extensive inter-process message passing.

This is especially for such information as interface up/down status which must be readily available to many processes, from the driver to the protocol processes to the routing-update-generation and dispatching processes. It some cases, interface

status changes will need to trigger some action to be taken by "higher level" processes such as routing. Perhaps at initialization time, each process that needs to become runnable when there is an interface status change should write an event handle in the device control block. Then whatever process happens to first discover the change in status can be responsible for posting the events for all the others.

DEVICE AND PROTOCOL INDEPENDENCE

One might ask whether the protocol process which deals with the device driver cannot itself contain a piece of the driver? Why have, e.g., an HDLC process and a simple driver process which does little more than interface with the interrupt handler, when one could simply make the HDLC process itself interface to the interrupt handler, and thereby eliminate the separate driver process?

This makes a lot of sense if (a) the protocol process is on the same processor as the I/O device which it uses, and (b) the interface to the interrupt handler is device-independent. In most cases, condition (a) will probably hold. However, we can imagine a gateway having a few very high throughput devices, in which, to ensure the meeting of latency requirements, a single processor node has to be totally devoted to device driving tasks. In such a case, we would want to put the protocol process on a separate processor, and have a separate driver process.

It is not clear to what extent condition (b) will hold. If it doesn't, then it may be hard to make the protocol processes

device-independent. Of course, if the device dependencies are small, this could be handled by the use of common library subroutines. (While we rejected the idea of implementing entire protocols as library routines, it may be easier to implement this more restricted set of functions as a common library.) Perhaps we should retain the flexibility to have either a separate driver process or not, as demanded by the configuration, and specified in the device control block.

It is generally desirable not only to make the protocol processes device-independent, but also to make the driver protocol-independent. However, in certain circumstances, performance considerations may make this desideratum difficult to achieve. For example, when transmitting HDLC packets, one wants to load in a piggybacked acknowledgment at the last possible instant, so that the maximum number of received packets can be acknowledged as soon as possible. But one also wants to provide the driver with as long a queue as possible, so that the protocol process need not run between every pair of sent packets (which would hold down throughput). These considerations are in conflict. The best way to minimize the acknowledgment delay

while also maximizing throughput might be to have HDLC process store the acknowledgment sequence number in a known place, and to have the driver (presumably the interrupt handler) load that number into each packet upon transmission. However, this removes the protocol-independence from the driver. Perhaps the right thing to do would be to conditionalize this, so that the driver determines from the device control block whether HDLC is running over this device, and does the right thing in either case. There is still a layering violation, but a relatively minor one.

INTER-PROCESS COMMUNICATION BETWEEN LEVEL2 AND LEVEL3

How shall the LEVEL2 and LEVEL3 processes pass messages to each other? The Chrysalis dual queue mechanism is the most natural means of inter-process communication. It is supported by the microcode, and operates in a way which avoids enqueue/dequeue race conditions which might otherwise arise in the multi-process and multi-processor environment. Placing something on a dual queue has the further advantage of automatically making runnable any process which is waiting on the queue. Since dual queues of packets are really just ring buffers of pointers to the packets, there is the further advantage that a given packet can be simultaneously on an arbitrary number of dual queues (assuming an appropriate use count mechanism in the buffer headers, which Chrysalis does to provide). Further, the number of queues on which a packet can reside can be increased at any time without need to redesign the packet headers, or to recompile the entire system.

Dual queues have two major disadvantages, however. The major one is that the ring buffer must be allocated to a fixed size at initialization time, which restricts the number of

packets the queue can contain. The ring buffer requires four bytes per potential queue item. If the maximum queue size greatly exceeds the average size, much memory can be wasted.

The other major disadvantage is that a dual queue cannot easily be sorted so as to implement a priority queue. (Queue items can only be placed at the head or at the tail of the queue). Priority queues can, of course, always be implemented as multiple queues, but this does introduce other problems. If one supposes that the throughput due to high priority packets is low, and the throughput due to low priority packets is high, one would like to have a scheme in which the extra work needed to handle priority packets is done by the process that puts stuff on the queue (the "producer"), rather than by the process that takes stuff off the queue (the "consumer"). It is more efficient to have the producer, which already knows it is handling a high priority packet, insert the packet at the proper place, than to require the consumer to always scan a whole bunch of queues, most of which will be empty most of the time. The former scheme has no penalty if high priority packets are not present; the latter scheme does.

There may be ways around this, however. For example, perhaps the enqueuer can place a special marker at the head of the low priority queue to indicate the presence of a packet on a higher priority queue. The details of such a scheme are problematical; this would have to be considered in more detail whenever it is decided that a given pair of processes need multiple priority queues for their communication with each other.

Another potential problem with the use of multiple dual queues for priority might arise if a higher priority queue filled up while a lower priority queue did not; one wouldn't want to keep the lower priority stuff while discarding the higher. (But on the other hand, one wouldn't want to discard the lower just because one couldn't keep the higher).

In the particular case of LEVEL2 talking to LEVEL3, the priority problem is certainly not very serious. In most cases in which there are both levels of protocol, it is necessary to maintain sequencing among the packets handled by the processes. So there is no application of priority queues when communicating between the two processes.

The size issue may also not be that important in this particular case in the gateway application, since there will generally be only one process producing items for this queue, and one process consuming items from it (though we will shortly consider the case of multiple producers and/or consumers). However, it is also true that the interface between LEVEL2 and LEVEL3 is NOT a good place to start dropping packets because they cannot fit on the inter-process queue. So some mechanism is needed to handle this situation when it arises, even if it arises only rarely.

If LEVEL2 and LEVEL3 are on the same processor, they should run at the same priority level, in order to prevent a context switch after each packet is processed by the first protocol process. The time slicing parameter should be set so that, on average, a certain number of packets can get processed by the first protocol process before it gets sliced out, giving the other protocol process a chance to run.

However, if the two protocol processes are on different processors, the time-slicing mechanism does not function. And at any rate, it will always be possible that the feeding process

will at some time fill the dual queue. This possibility can be dealt with in the following manner. The producer process can, at initialization time, allocate an event block whose handle will be known to the consumer process. The consumer process will post this event whenever it empties the dual queue (or reduces the size of the dual queue below a threshold). When the producer process fills the dual queue, it will cease any processing that would result in more things going on that queue, until that event is posted. There is not too much cost to posting this event unnecessarily (i.e., when the queue is emptied, but was not previously full).

This scheme extends naturally to the case in which there are several processes consuming a single dual queue; whichever one finally empties the queue (or reduces its size below the threshold) is the one which should post the event. If there are several producing processes, the scheme breaks down, since one of them might stop producing once it fills the queue, while the others manage to keep the queue non-empty (or above threshold), while still never quite filling it. In this case, the former process would get locked out.

One could deal with this case by having the process which fills the queue set a lock (a dual queue lock) which prevents the other producing processes from putting anything more on the queue until the consuming process empties it and then signals them all. The need to check the lock before putting any item on the queue is an extra expense. However, this check could be avoided if, according to the config information, there is only one producing process. Then the cost of checking the lock just trades off against the benefit of having several producing processes which could run simultaneously.

Note that this might cause a protocol process to cease its processing of input, while continuing to process output, or vice versa. There is NO requirement that the process sleep when the dual queue it is running fills up. The producing process is not even required to cease processing input, only to cease forwarding the processed inputs to the next protocol level. It will be an issue in the design of each particular process whether it should (a) discard packets it cannot forward, or (b) whether it should let such packets pile up on the queue that it consumes, or (c) whether it should keep packets which don't fit on the

inter-process queue queued up internally (in a linked list).

The same mechanism would be used to pass packets between the LEVEL2 protocol process and the driver process (if these are separate processes), which case is very similar.

If I/O control blocks are used by the device driver, and this kind of process-to-process blockage were to persist in the input path, eventually input would stop, since all the I/O control blocks would be tied up on the linked list which the input interrupt handler prepares for the driver process. Rather than block the input interface like this, it may be desirable to keep a small number of I/O control blocks free, so that input can continue. The driver process can implement this stategy by discarding packets from the linked list as needed to keep its size at or below a certain threshold. This provides the option of receiving certain special packets (routing updates, up/down probes) and processing them "ahead" of packets which are languishing on queues.

If I/O control blocks are not used, input won't get blocked quite so soon, but the same situation applies; one wants

to keep some buffers freed up so that input can always continue.

How can "special" packets get "pushed ahead" if the protocol requires sequencing? There is no reason why the device control block for each interface cannot indicate the offset into the buffer at which the IP header or gateway-gateway protocol header would start. This might vary with different protocols, but could be set up at initialization time. The driver process can thus look ahead at each packet, and if it is a special (gateway-gateway internal protocol) packet, the packet can be passed ahead to whatever special process needs to look at it, even while it is still being queued for LEVEL2 input processing, or even if the queue of the LEVEL2 input is filled. (Remember that with dual queues, a single packet can reside simultaneously on an arbitrary number of queues.)

This architecture does not require that there be a single LEVEL2 and a single LEVEL3 process for each interface. Multiple processes of either level are possible if they become bottlenecks, so that one wants to use the multi-processor parallelism to increase throughput. Unfortunately, however, protocols that tend to be CPU-intensive also tend to require

sequentiality, thereby removing parallel processing as an option.

There is no requirement that the LEVEL2 and LEVEL3 processes be on the same processor as each other, or that either be on the same processor as the I/O device. The Butterfly/Chrysalis environment provides the flexibility to assign the processes to processors differently in different gateways, as the configuration warrants. The gateway initialization routine will have to be able to tell, from configuration data, which processes must be created on which processors.

Regardless of which processors contain the various protocol processes, it is still the case that all the packets being handled by those processes will reside in buffers which are in the memory of the processor to which the I/O device is attached. Therefore, all the protocol processes which serve a particular I/O interface, as well as its driver process, must have all the buffer segments of that processor in their virtual address space. No buffer segments on any other processors need be included in their address space.

If a protocol process is not on the same processor as the I/O device it is serving, it is probably more efficient for the process to block transfer the header of the packet it is processing into its own processor's memory than for it to (transparently) refer several times to that remote memory. In general, only a small number of packets are being processed simultaneously (perhaps one for input and one for output), and the block transfers can be made into a fixed area; no buffers on the process' processor are needed.

DISPOSITION INFORMATION -- SENDING BACK INFO TO A PRIOR PROCESS

In some cases, a particular protocol process may not be able to completely finish its processing of a particular packet before it passes the packet on to the next process. Rather, it may need to learn about the actions which are later taken with respect to that packet (by other processes) before it can finish its own processing of the packet. For example, one might decide that the acknowledgment for an incoming packet should not be sent unless that packet can be successfully queued for output. The intention would be that if some later process had to discard the packet, say for buffer management reasons, one would then not have acknowledged it, and the network entity which sent it would then retransmit it. It is not clear at the present time whether the gateway will actually have an application for this strategy, but this strategy is used in the IMP. At any rate, our design should not be incompatible with such a strategy.

A clean way of having a "later" process inform an "earlier" one of the ultimate disposition of some packet is simply to have the later one mark the disposition somewhere in the header, and then put the packet (probably, only the first

buffer of the packet) on another dual queue for the earlier process. (Remember that a given packet can reside simultaneously on an arbitrary number of dual queues, given an adequate use count facility.) This avoids the need for one process to touch the data structures of another, which would almost certainly cause a violation of protocol layering. It also avoids the need to establish a separate method of inter-process message passing.

This dual queue would have to be large enough to contain the maximum number of packets which the protocol can leave outstanding (i.e., unacknowledged), or else acknowledgments may be spuriously withheld, and spurious retransmissions may result. However, this number is at least bounded, and is often quite small. Furthermore, if making the dual queue ring buffer smaller than maximum size does cause the occasional spurious withholding of an acknowledgment, no great harm is done.

Packets would not need to sit on these "return" queues very long, since packets on those queues are not really waiting for some resource (such as a transmission medium) to become available. Generally, the protocol process which consumes the return queue will only need to note the disposition of each

packet in its internal tables, which can be done very quickly.

In order to implement this in a clean way, the buffer header would have to contain an indication as to which, if any process needs disposition information about this packet, so that later processes know where to find the return queue.

AFTER INPUT IS COMPLETE -- THE DISPATCH QUEUE

Suppose now that a packet has been fully and correctly received and processed through all necessary layers of protocol. What is done with it next? Now the real "gateway processing" begins. This processing will depend on what kind of packet it is:

- "special" gateway-gateway protocol packet, such as a routing update or a neighbor up/down packet. There will be a Chrysalis process devoted to each of these. Since these packets need immediate processing (high priority), but occur relatively infrequently (low throughput), it may be advantageous to have them processed by processes which run at a higher priority than the protocol and device driver processes. (Perhaps though these packets will have already been passed ahead from as discussed previously, from the driver processes to the special processes that consume them. Also, it is not really clear at present whether there is a significant advantage in having a Chrysalis process run at a higher priority than others.)

- a transit packet which has already been seen by some other gateway and which must travel to some other gateway (where the exit gateway is identified in the gateway-gateway header that was created by the entry gateway). The exit gateway must be looked up in the routing tables in order to determine a "next-hop gateway". If the next-hop gateway is a neighbor of this gateway over more than one network, one of the available networks must be chosen. If this gateway has several interfaces to that network, one must be chosen. Then the packet must be enqueued to the protocol process (or to the LEVEL3 protocol process, if that interface needs two levels of protocol), subject to buffer management considerations.

- a packet which is addressed to this gateway. This may be an EGP message, an HMP message, an ICMP message, an RDP message, etc. These packets might or might not have gateway-gateway headers, depending on whether this gateway was also the entry gateway. There will have to be one or more processes to handle these messages. It may be suitable to have these handled by lower priority processes,

especially since Chrysalis ensures that even lower priority processes get to run.

- a packet which is entering the internet system at this point (i.e., has not been seen by any other gateway). It must be determined whether this packet is addressed to this gateway (if so, see previous paragraph). If not, a gateway-gateway header must be fashioned. The routing tables must be consulted to find which gateways are potential exit gateways for its destination network. One of these must be chosen. Then the packet must be processed as if it were a transit packet.

In some cases, it may not be possible to queue a packet for output, because the output queue is full, or because no buffer can be obtained, or because the packet's destination is known to be unreachable, or because access control restrictions would be violated, etc., etc., etc. In some of these cases, the packet can just be discarded, or placed on a "return" queue so its disposition can be indicated to the protocol process which handled it on input. In other cases, an ICMP message might need to be sent to the source host. In the latter case, the packet

would need to be queued to a process (lower priority, possibly) which can create and send ICMP messages.

Should these decisions, and the related processing, be made by the receive side protocol process which finishes the input processing of the packet, or should there be another process (call it DISPATCH) which receive ALL packets that arrive over the interfaces and which makes these decisions? The more complex the decision making is, the more reason there is to have another process, and as we see, the decision making needed here can be rather non-trivial. Introducing a separate DISPATCH process seems a better alternative (for reasons already advanced) than having a large library of dispatching routines callable by all other processes.

Introducing a separate dispatching process also enables better use of the multi-processor environment, since the can be multiple DISPATCHes working in parallel, all consuming from a common DISPATCH queue (which we will call the "DQ"). Advantage could even be gained just from having a single DISPATCH, on a different processor than the protocol processes.

There are other advantages to having a DQ on which all packets will reside at some time or other. It simplifies the gathering of various sorts of statistics about the utilization of common resources in the gateway, and about the throughput and user data flows seen by the gateway. It provides a common place where other processes (e.g., message generator processes, ICMP processes, EGP processes) can place their messages without having to perform routing or buffer management themselves. Without a common DQ, certain simple functions, such as having a message generator send packets to itself without having those packets ever actually leave the gateway, are very much simplified. As long as the DISPATCH process runs at the same priority as the protocol processes, or if there are multiple DISPATCHes running in parallel, this should not have an excessive cost nor should it become a bottleneck.

THE DISPATCH PROCESS

In general, the particular DISPATCH process which processes a given packet need not be on the same process as either that packet's input device or its output device, and these two devices need not be on the same processor as each other. Therefore, the DISPATCH process, after choosing an output device for the packet, must determine whether that device is on the same processor as the input device. If so, it will have to try to obtain enough free buffers on the output processor (subject to buffer management limitations) to hold the packet, and will have to copy it from the input to the output processors. At that point, the input buffers can be freed. If input and output are on the same processor, no copy is necessary, but the buffer management computations must still be applied, and may cause discarding of the packet.

It follows from this that all DISPATCH processes must have all buffer segments from all processors mapped into their virtual address spaces, since they might have to copy from any buffer on any processor to any buffer on any processor. This is another very important reason (perhaps ultimately the most

important) for having a separate DISPATCH process, instead of having all the protocol processes to their own dispatching. Otherwise, all protocol processes on all processors would have to have all the buffer segments on all processors mapped into their virtual address spaces. This would be much more expensive in terms of SAR utilization than it requiring just one (at most) process per processor to have all the buffer segments mapped in.

In most cases, DISPATCH will dispose of a packet by enqueuing it for a protocol process (LEVEL3 if there are two levels of protocol needed) which serves one of the output interfaces. This transfer will be made by enqueuing onto a dual queue, subject to buffer management considerations. If buffer management considerations, or a full dual queue, prevent the assignment of the packet to that output interface, there are some cases in which DISPATCH might be able to choose another output interface (e.g., if there are two interfaces to the same network, or if a given neighboring gateway is a neighbor over two networks). If the packet cannot be queued to any appropriate output interface, then DISPATCH must discard it.

This is the case in which the fixed maximum size of the

dual queue is most likely to cause difficulty. However, this can be alleviated to some extent if we require that the consuming process remove packets from the dual queue as soon as possible, even if they cannot be transmitted. The consuming process can keep such packets on an internal queue which is implemented as a linked list. With this strategy, the dual queue need be sized only according to the scheduling latency of the consuming process, not according to the possible backup from the output interface. This also allows the buffer management scheme to function properly to prevent too many buffers from being tied up waiting for a single interface. Also, this is a point where it does not hurt too much to discard packets if they can't fit on the dual queue.

If we have several priorities of packets, the dual queue between DISPATCH and the protocol processes needs to be prioritized. If several distinct dual queues are used for the different priorities, there is a difficult issue of how to handle the case in which a high priority queue is full while a low priority queue is empty. We don't want to start discarding high priority stuff while keeping low priority stuff. Even with a

single prioritized queue, there is the issue of how to avoid discarding high priority packets while low priority packets remain enqueued. The "proper" strategy, from the perspective of meeting the priority requirements, would be to somehow remove low priority packets from the queue so as to make room for high priority packets.

Discarding a packet will require DISPATCH to maintain statistics as to the number of packets discarded for the various reasons. If it is required to send an ICMP source quench packet after every N discards, or after every N discards for a particular reason, the Nth "discarded" packet should be enqueued to the (lower priority?) process which is responsible for creating ICMP messages. Actually, only enough buffers to contain the IP header need to be so enqueued, and the header of the first buffer needs to be marked so as to indicate that a source quench should be sent to the source of the packet.

If a packet elicits an ICMP redirect, then that packet should be enqueued to the ICMP process, as well as to the appropriate output interface. Of course, its header will need to be marked to indicate that a redirect should be sent.

The same strategy should be followed for any packet which elicits any type of ICMP response. If the queue to the ICMP process is full, then this step should just be skipped.

If a packet gets queued to both an output protocol process and the ICMP process, buffer management should be based on the output interface. If it gets queued to only the ICMP process (perhaps because buffer management prevents its being queued for output), the ICMP process should be treated as an output device from the point of view of buffer management.

When the ICMP process creates a message in response to a particular packet, it may be able to use the same buffer, IF the use count of the buffer indicates that no one else is using it. Otherwise, a new buffer must be obtained from the free queue. It doesn't really matter which processor's free queue is used. In either case, additional buffer management must be done, treating the ICMP process as an input device. When the ICMP message is created, it is just put on the end of the DQ.

When DISPATCH finishes with a packet, it may also need to enqueue it on a "return queue", after marking the packet with its

disposition information. If the return queue is full, this step should just be skipped. No buffer management is involved.

Initially, packets entering the internet system at this gateway can have their gateway-gateway headers set up by the DISPATCH process. Ultimately, as we evolve to a more "circuit-oriented" interface, with control information between the entry gateway and the source host (e.g, to enforce flow control), we may require a separate process.

Packets addressed to the gateway will be enqueued to a special "FORUS" process (ultimately, perhaps wthere will be several such processes), which will be treated by DISPATCH as just another output interface.

DISPATCH should not need to look at the source route option. A packet whose IP destination address is this gateway will be sent to FORUS, which will, if necessary, swap our address with the next address in the list of addresses to be visited, and then return the packet to the DISPATCH queue. (N.B.: If we want the return route field to have the output interface address, rather than the input interface address, DISPATCH will have to

place the output interface address in the return route field the second time it sees the packet.)

Communication with the routing and up/down processes will be considered in a separate memo.

All of the inter-process messages we have discussed so far have been messages which can be transmitted outside the gateway. We have not discussed the use of a special class of inter-process control message whose use would be solely to carry status or commands from one process to another. It is not clear at the present time that we will need any such messages. We have discussed several cases in which status and/or command information from one process to another is piggybacked on the buffer header of a buffer containing a packet. In these cases, the information is about that particular packet.

We have assumed that a buffer might have to be on as many as three queues at once: an output queue, a return queue, and an ICMP queue. The buffer header will have to have enough space to simultaneously contain the information that needs to be passed to all three processes.

In other cases, the status information or commands might be about some particular device, rather than about a particular packet. One example is an interface's going down, which information, once obtained by some process, needs to be made known to the interrupt handler, the driver process, one or two protocol processes, and the routing process. Another example would be the reception by the gateway of, say, an HMP command to flap the ready line on one of its 1822 interfaces. This command would have to be conveyed from the FORUS process which interprets the HMP message to the device driver. In both of these sorts of cases, it might be possible to mark the information (or command) in a control block for the device, and then to signal the processes that need to know about it by posting an events. It is not clear at present whether this technique will work in all cases, but if it does it might be much less troublesome than sending messages from process to process, because it does not require any dynamic message block management, and does not give rise to any flow control problems.