IDN-6

Chrysalis and Processes

Eric Rosen

September 1983

In this message I want to present my understanding of the way the Chrysalis operating system handles processes. I have gleaned this information by reading several times through the QTRs and the Chrysalis Operating System Manual. I will discuss process creation and invocation, inter-process communication, and process scheduling. I hope my misunderstandings will be corrected by those who know better. My main goal in writing this message is to crystallize my own thoughts, but I also hope to encourage some discussion Chrysalis' facilities among newcomers to Butterfly projects, and to provide an introduction to these facilities which may be more accessible than the existing documentation.

Chrysalis' notion of a process is quite conventional, and not much different than that of, say, Unix. A process consists of a set of modules which are linked together by running a link program on the development machine. Every process runs in a virtual address space, but the virtual address spaces of several processes might correspond to overlapping areas of physical memory. Right now all we really have to know about a process is that it is the smallest independently schedulable program entity.

Chrysalis provides a system call ("Make_Template") which will load in a copy of the process's text (instructions) and initialized data from an external load device. Currently, the only supported load device is the VAX, and the Butterfly is connected to the VAX via an ordinary tty line. One of the arguments to Make_Template is the VAX file name of the process image. Text and data are each limited to 64K bytes. The actual loading is done by a separate "loader process" that is part of Chrysalis; this process cooperates with a process running under Unix on the VAX.

At the time of the Make_Template call, one can specify the size of the process' stack, as well (I think) as whether the

stack should be in a memory segment (SAR) of its own, or should be in the low order part of same segment which holds the initialized data. (Stacks grow downwards.) Text and data may not be in the same segment, it appears. (Ignore this if you don't understand it -- it raises some important issues with respect to memory segmentation problems in the Butterfly, but I'll discuss those in another message.)

Make_Template allows you to specify a process starting address, or to take the starting address from data provided by the linking program that ran on the development machine (I guess this is stored as a magic number somewhere in the image file.)

The call to Make_Template creates a copy of the process' text and data, but it does not actually result in the creation of a runnable process. Rather, it just creates a "template" which can be used to make a runnable process. To actually create a runnable process, one must call "Make_Process", which takes a pointer to the template as one of its arguments. By calling Make_Process several times with the same process template, one can create any number of processes which start out with identical text and data. Each call to Make_Process results in a new copy

of the initial data segment, and a new (empty) stack, but the text is shared, not copied.

(There is an option for creating a "clone" process, which copies the present state of the data, rather than the initial data. A clone can be started with an empty stack, or with a copy of the current stack. It is also possible to create a "child" process, which shares the same virtual address space as its "parent", so that no copying of data is necessary. In this case, the parent is responsible for setting up stacks for the children.)

When issuing a call to Make_Process, one specifies various information needed by the scheduler, such as priority, time-slice, and "declared need". I'll discuss these concepts shortly; for now, the point is that all scheduler-related parameters are given their values at this time.

Make_Process seems to allow one to override some of the information that was specified in the call to Make_Template, such as the size and location of the stack, and the process' starting address.

Creating and destroying processes appears to be pretty costly, and is really intended to be done only at initialization time.

One very important thing to understand about the Butterfly is that all its memory is local memory. That is, every memory location is part of the memory of some particular processor; there is no common or global memory. A given processor can read and write memory which belongs to another processor, but a processor can only execute out of its own memory. Therefore, when creating processes or process templates, one must specify a particular processor. A given process never moves from one processor to another, but different processes created from the same template may run on different processors. It is possible for a process on one processor to create a process on a different processor; in this case only, the process text must be copied into the memory of the other processor by being sent across the Butterfly switch.

I have only a vague picture of how a Butterfly application actually gets loaded up, initialized, and started. I imagine that when a Butterfly first comes up, it runs a PROM-

resident boot loader (in some processor or other) which can be force-fed some version of the Chrysalis operating system. Then there must be some way in which it gets a list of VAX file names corresponding to the processes which need to be loaded. (How does it get this list?) Then I suppose it issues calls to Make_Template and Make_Process until it has loaded all the necessary processes in the proper processors. (It's not clear to me whether each processor loads itself, or whether one processor loads all the others.) When a process is created via a call to Make_Process, the process is made "runnable", which means that it will be scheduled.

When a process runs for the first time, it starts at an address specified in the Make_Process or Make_Template call. This address is supposed to be the entry point of some function. The Make_Process call also provides an argument list to be passed to that function. Presumably, this function will be the initialization function for the process.

After this initial invocation of the process, there is only one way it can ever run again. It can only be made runnable if some process (possibly itself) "posts an event" for it. So

let's look at the "event" mechanism in Chrysalis.

When a process is first invoked, and runs its initialization function, it allocates some number of "event blocks". Each of these event blocks is OWNED by the process which allocates it. Each event block is associated with an "event handle", which is a sort of pointer to it. Typically, each event block will have some special meaning to the owning process, in the sense that when a specific event is posted, the process which owns that event will infer that there has been some specific occurrence which demands its attention.

Each process has associated with it a queue of posted events. A process can poll this queue at any time (by issuing a "Receive_Event" call), to see if any events have been posted since it last polled the queue. If anything is on the event queue, this call removes the first event and returns its event handle. Otherwise, the call returns NULL. The related call "MReceive_Event" allows one to poll for a specific list of events, rather than just any event whatsoever. The system calls "Wait" and "MWait" allow a process to dismiss until an event (any event whatsoever or one of a specified list of events) is posted.

(Once a process dismisses in this fashion, it is no longer runnable, and will not become runnable until an event is posted for it. Note, however, that these calls will return without dismissing if there are already posted events on the queue.)

Any process can post any event, as long as the posting process knows the event handle. However, the posted event will always be queued for the process which owns it. (The posting process need not be on the same processor as the process which owns the event; event handles are unique across the entire multiprocessor system. The operating system knows how to post events for processes on other processors, and handles all the inter-processor communication transparently.)

The event mechanism is the basic mechanism for waking up a process (when an event is posted for a sleeping process, that process is marked runnable), and for signalling a process of external events which need its attention. Events are also the basic inter-process communication mechanism. (Though remember that different processes can have some of the same physical memory in their virtual address spaces, providing a less clean mechanism for inter-process communication.) When posting an

event, the posting process can supply a couple of words of data which the receiving process can read from the event block. This data is allowed to be a pointer.

When an event block is allocated, the process which owns it specifies whether it should be legal to have the event multiply posted. (An event is "multiply posted" if it gets posted, then posted again, before the process that owns it has a chance to remove it from the event queue by executing a Wait or Receive_Event call.) If multiple posting is illegal, the posting process gets an error. Note that if multiple posting is legal, the data field of the event block is not useful.

A process can cause one of its own events to be posted at a specified time in the future by supplying an event handle and a time offset to the "Set_Timer" call. Processes can reschedule themselves for a later time by using this call and then waiting on the specified event. Apparently, a process cannot use this mechanism to schedule another process at a specific time in the future, but only to schedule itself. (I think this timing mechanism is less general than that of NMFS, but I'm not sure if that is a real restriction.) Time offsets are specified in ticks

of 62.5 microseconds, and the offset value is a 32-bit long. Timer requests can be cancelled by using the "Clear_Timer" call.

The event posting mechanism in Chrysalis seems to subsume the "goading" and "timing" mechanisms of NMFS. It provides a much more general mechanism for enabling a process which wakes up to determine just why it was awoken, since each event that might wake a process has a special inherent meaning to the process, and the event block can point to any amount of data. (NMFS provides only a skip, and the use of common memory, to allow the process to determine whether it woke up because of a time-out or a goad.) In addition, the confusing "re-poked" and "re-timed" states of NMFS processes are eliminated.

One thing I really like about the event mechanism (and one of its most original features) is the fact that a process must explicitly allocate (preferably at initialization time) all the events that it is capable of receiving. This doesn't appear to restrict the generality of the inter-process communications in any significant way, and it eliminates lots of resource shortage and congestion control problems that would otherwise arise if one needed to do real-time allocation of event blocks while the

application is running.

The alert reader will be wondering about the following problem. In order for process A to signal process B, A must allocate an event block, and B must somehow find out the event handle, in order to be able to post the event. Since the situation is symmetric, one might wonder just how one process ever gets hold of the event handle of another process in the first place; this data can't just be sent from A to B in an event block, for that would just raise a chicken and egg question. Clearly, some kind of communication through commonly accessible memory is needed. Chrysalis provides a global name table for this sort of purpose. A process can assign an ascii string to an event handle, and then execute a system call ("Name_Bind") to have this binding placed in a special table. Any process (on any processor) can then get the value of the event handle by passing the ascii string to a system call.

Much of the event posting stuff is done by microcode. A Butterfly processor node consists of several components. One component is a 68000 microcomputer, which is where one's macrocode programs actually run. Another component, known as the

"PNC" ("Processor Node Controller") or "co-processor" is a microprogrammable processor that extends the functionality of the 68000. Generally, queueing and event-handling operations are done in the PNC's microcode to make them indivisible and to make them fast enough to be useful. The scheduler (to be discussed below) is also partially implemented in the PNC's microcode, apparently because otherwise it would be intolerably slow.

While this use of microprogamming certainly has its advantages, it is my impression that the microcode is not really maintainable, and we wouldn't expect to be able to make changes in it without lots and lots and lots of work. (Also, it is in hard-to-change ROM, rather than on an easily changeable medium). One sometimes hears the claim that if we don't like some detail of the scheduler's operation, we can easily change it just by running the scheduler in macrocode instead of microcode. However, it is not too reassuring to know that if we discover that something doesn't work (because of bugs or inadequate functionality), we can easily replace it with something else that won't work either (because it has too much overhead.)

The Chrysalis scheduler is responsible for multiplexing

the processes which are running in a given processor. Every processor node has its own incarnation of the scheduler. No global or system-wide scheduling is done, and there is no migration of processes from one processor to another. This presents a number of important design issues to the designer of the application software. I want to discuss here the way the scheduler works on a single processor.

The scheduler has a number of features which are not usually found in real-time multi-process systems. When each process is created, it is assigned three parameters:

Priority -- a value from 0 to 3, with 0 being the lowest priority and 3 the highest. The intention is for processes of higher priority to be able to pre-empt processes of lower priority, but as we shall see, this is only true under certain conditions.

Time Slice -- a number of milliseconds. If a process runs for this number of milliseconds without voluntarily dismissing, the scheduler will be invoked, and control may be turned over to another process of the same (or in some cases, even a lower, priority).

Declared Need -- a number of milliseconds. One parameter of the operating system (it is not clear to me when this parameter is assigned a value) is the number of milliseconds in an "epoch". An epoch is an interval of time in which the scheduler attempts to see that every process

gets at least enough time to satisfy its declared need. The actual algorithm used to bring this about is complex, and will be discussed below.

It is the intention that the time slice of a process be smaller than its declared need.

The scheduler can be invoked in three ways:

1 - An event is posted for a process whose priority is higher than that of the process which is currently running. (Remember that the process which is currently running need not be the process which is posting the event; the event may have been posted by a process which is running on another processor.)

2 - A process "dismisses voluntarily". As far as I can determine, the only ways a process can dismiss voluntarily are (a) to execute a Wait or MWait call in order to sleep until a certain event is posted, and (b) to destroy itself (either intentionally, with the "exit" or "abort" calls, or in some unintentional manner which causes an unrecoveable error.)

3 - A timer goes off, either because it is time to post an event (as a result of a prior "Set_Timer" call, or because the currently running process has reached the end of its time slice.)

It is important to understand that the notion of process priority is not wedded to the hardware interrupt levels of the 68000 processor. The 68000 has seven hardware interrupt levels, numbered from 1 (lowest priority interrupt) to 7 (highest priority interrupt). All interrupt levels are maskable, except

for level 7. Chrysalis seems to use these interrupts for only two purposes:

a) It uses interrupt level 2 for the real-time clock. That is, when a timer "goes off", it causes a level 2 interrupt in the 68000. I think that this interrupt is intercepted by the PNC, which actually maintains the real-time clock in microcode.

b) The other major use of interrupts has to do with the invocation of the scheduler itself. A level 1 interrupt invokes the scheduler. These interrupts are intercepted by the Processor Node Controller, which makes the necessary scheduling decisions in microcode. (This use of the 68000 interrupt facility seems is what enables one to fairly easily replace the microcode scheduler with a macrocode scheduler. There is apparently some not too complicated way of preventing the PNC from intercepting the level 1 interrupt, in which case a macrocode scheduler can run in the 68000 as an interrupt routine.)

(Note that although scheduling decisions are made by the microcode, actual context switching is performed by macrocode which runs in the 68000 as an interrupt level 1 routine. The microcode figures out what amount of context switching (if any) is required, and then fools with the 68000's interrupt vectors to make it run the appropriate context switching routine as its level 1 interrupt routine. I was a little surprised to find out that what I thought was the more expensive part of the scheduler, viz., the context switching, was done in macrocode. Also, there

doesn't seem to be any hardware or firmware to help with this very much. Context switching seems to involve saving registers in memory, rather than, say, switching among blocks of registers.)

At any rate, Chrysalis processes running in the 68000 do not run at any interrupt level (or one might say, they all run at level 0), irrespective of their priority. I think this separation of process priority from hardware interupt level is a difference from NMFS (maybe I'm wrong though), but I'm not sure what implications this might have, if any.

The scheduler keeps track of processes via their "Process Control Blocks" (PCBs). A runnable process (as opposed to one which is sleeping while waiting for an event of some sort) always has its PCB linked onto one of eight priority queues. (Although there are only four possible process priorities, there are eight scheduler priority queues; we shall see why shortly.) A process which becomes runnable because an event becomes posted for it, or because it has just been created with a Make_Process call, gets placed at the end of the priority queue associated with the priority that it was assigned in the Make_Process call. Because

of the event mechanism, there is no need for the separate "pending" and "timing" queues that NMFS uses for PCBs.

The PCB contains the initial values of the priority and time-slice parameters that are assigned by the Make_Process call. However, the values of these parameters that are actually used at any given time may differ from the initial values, in a way determined by the scheduler. Thus we can speak of a process' "initial priority" and its "current priority". When a process becomes runnable (because an event is posted for it), the event-posting microcode makes the process runnable by placing it on the end of the priority queue which corresponds to its "current priority".

When the scheduler is invoked, it first looks at the currently running process, and asks whether that process needs to be moved off the head of its priority queue. If so, it has to decide how to reschedule the process. That is, it decides what priority and time-slice the process will have next time it runs. (This will be clarified below.) Then it just runs the process which is at the head of the highest non-empty priority queue. (This might still be the currently running process, if the

scheduler has decided that that process can continue, and there are no higher priority processes waiting to run.) The real "smarts" of the scheduler are in the way it decides how to reschedule a process that it is dequeueing from the head of its current priority queue. I want to look at the operation of the scheduler step by step.

How does it determine whether the currently running process needs to be moved off the head of its priority queue? There are two cases in which this is necessary:

1 - The process has just gone to sleep by executing a Wait or MWait call, and is no longer runnable.

2 - The process has used up most of its allotted time slice (there are no more than "a few hundred microseconds" to go).

If neither of these two conditions obtains, the assumption is that the scheduler has been invoked because an event has been posted for a higher priority process. So the currently runnable process is left on the head of its priority queue to complete the amount of time left in its current time slice. However, when the scheduler exits, it will turn control over to the process at the head of the highest priority queue, rather than the process which has been running. The pre-empted process will be the first

process of its priority to run when there are no higher priority processes to run.

Now suppose, on the other hand, that one of these two conditions is the case. Here is where the scheduler's "smarts" come into play. The scheduler asks itself the following question: "is there enough time left in this epoch so that all processes can have their declared need satisfied before the epoch ends?" Of course, the answer to this question depends not only on the amount of time left in the epoch, but also on the amounts of time that the various processes have already used in this epoch; some processes may already have received their declared need.

If the answer is yes, then the currently running process retains its current priority. That is, the next time it runs, it will run at the same priority level it is now using. If it is still runnable, it is just placed on the end of its priority queue. (Thus the time-slicing enforces a round-robin discipline among processes of the same priority.) However, if the process has already had its declared need satisfied in this epoch, it will not necessarily get allocated its full time-slice. It will

only get allocated enough time so that if it were to begin running again immediately, it would still get sliced out in time to allow all other processes to get their declared needs satisfied within this epoch; this quantity might be smaller than the duration of the time slice specified when the process was created.

Suppose we are running only two processes, A and B. Each process has a declared need of 20 ms. and a time-slice of 5 ms. Both processes are of priority 1. The epoch length is 100 ms. Now consider the following situations:

a - As the epoch begins, process A begins to run. After 5 ms., the scheduler is invoked by timer interrupt. If process B is runnable, A will be assigned a new time slice of 5 ms. and placed on the end of the priority 1 scheduler queue. Process B will then be run. If process B is not runnable at this time, however, A will get to run immediately for at least another 5 ms.

b - After 78 ms. of the epoch, process B has not run yet, and process A has used more than 20 ms. If process B is not runnable, A will get to run again immediately,

but it will be assigned a time-slice of only 2 ms. This gives B a chance to get its full 20 ms. in this epoch if it become runnable in time. (Of course, if B does not become runnable at all in this epoch, it just won't run. And if it never becomes runnable until there are only 10 ms. left in the epoch, it obviously will not be able to run for 20 ms. in the epoch. In general, a process is only guaranteed its declared need within an epoch if it is ready to run at the beginning of the epoch.)

Suppose, however, that there is not enough time left in this epoch to satisfy the declared needs of all processes within this epoch? Then the scheduler asks whether the currently running process has had its declared need met. If it has not, then it retains its priority, and if runnable is moved to the end of its priority queue. But if its declared need has been met, and if the current priorit of the process is the same as its initial priority, then the currently running process has its priority REDUCED, and is assigned its full time slice. If it is still runnable, it is place on the end of the scheduler queue corresponding to its new, lower, priority.

Recall that while there are four process priorities, there are EIGHT scheduler priority queues. When the scheduler moves a process to a lower priority queue, it actually reduces its priority by four levels. THIS MEANS THAT A HIGH PRIORITY PROCESS WHOSE DECLARED NEED HAS BEEN MET MAY NOT BE ABLE TO PRE-EMPT A LOW PRIORITY PROCESS WHOSE DECLARED NEED HAS NOT BEEN MET. For example, if, near the end of an epoch, all processes but one have had their declared needs met, that one process will be the only one that hasn't been moved to a low priority queue. If it does become runnable during this epoch, it will pre-empt any other process which may be running, and will not itself be pre-empted by any other process, until the epoch ends.

To continue the above examples:

c - Suppose that the scheduler is invoked when there are only 20 ms. left in the epoch, and it finds that process A is running. Suppose also that process A has already used more than 20 ms., but process B has used less than 20 ms. Then process A is moved to low queue and assigned a full 5 ms. time slice. If process B is not runnable at this time, process A will just continue

to run. But if B should become runnable at any time in the next 20 ms., it will pre-empt process A and begin to run. NOTE THAT THE SAME THING WILL HAPPEN EVEN IF PROCESS A HAS A HIGHER PRIORITY THAN PROCESS B; IF A'S NEED HAS BEEN MET AND B'S HAS NOT, B WILL BE ABLE TO PRE-EMPT A.

Of course, the priority of a process can only be reduced once. Thereafter, it will retain the same priority until the end of the epoch. Processes which have had their priorities lowered are always assigned their full time slices. This makes sense, since no process can have its priority lowered until its declared need has been met, and any such process is pre-emptable by any process whose declared need has not been met. If all processes have had their declared need met, then the system acts the same as if they are all at their initial priority with full time slices.

At the end of an epoch, all processes are re-assigned their initial priorities and time slices.

One thing to notice: the scheduler does not appear to know why it was invoked. It will slice out a process which is

close to the end of its time slice, even if it was invoked NOT by the timer interrupt, but by the posting of an event for some higher priority process. This can lead to processes being sliced out a little bit ("a few hundred microseconds") before the actual end of their time slices. This seems warranted, in order to prevent the scheduler from getting invoked again in just a few hundred microseconds.

A more significant thing to notice (this pheonmenon is pointed out in the documentation): the particular assignment of priority and time-slice to a process that the scheduler makes at a given time may be valid at that time, but may be invalid at the time that the process becomes runnable again. Another variation on the above set of examples:

d - Process A dismisses with 50 ms. to go in the epoch, having had its declared need satisfied. Process B has not run yet. The scheduler lets A retain its initial priority, and assigns it a full 5 ms. time slice, since there is plenty of time left for B to get its need satisfied. Now process A waits on some event, which doesn't get posted until there is only 21 ms.

left in the epoch. Then process A gets to run again at its initial priority for at least 5 ms. Even if process B becomes runnable one nanosecond later, process A will still get to run for 5 ms., which means that process B will only get 16 ms. in this epoch. That is, process B will not have its 20 ms. declared need satisfied in this epoch, even though it becomes ready to run while there are still almost 21 ms. left in the epoch.

The problem is that the scheduler's decisions are made on the assumption that the currently running process is going to be ready to run again immediately, and this assumption will not generally be true, ESPECIALLY in the case of a process which is not even runnable at the time when the scheduler is making its decisions.

It would seem to make more sense for the scheduler to assign a process its priority and time slice at the moment it becomes runnable (i.e., when an event is posted for it.) This couldn't solve the whole problem, however, since if a process gets sliced out while it is still runnable, then the scheduler

does have to decide right then which priority queue it should be placed on. Of course, the scheduler might have been designed so that it could change its decision when that process reaches the head of the queue. I'm not sure whether these apparent design defects are there for philosophical reasons, for reasons of expediency (e.g., no one felt like writing additional microcode), or for reasons of efficiency (viz., fixing the defects causes so much additional scheduler overhead that it becomes counter-productive.)

Another possible approach would be for the scheduler to lower the priority of a process as soon as the declared need of that process is satisfied. This would ensure that no process whose declared need had been met was running while another runnable process' declared need had not been met. However, this approach would also have a negative side-effect, since a low priority process whose declared need had not been met would pre-empt a high priority process whose declared need had been met, even if it may still be possible to satisfy the declared need of the low priority process in this epoch without pre-empting the higher priority process. I'm not sure it is possible to say apriori whether this is better or worse; the actual effects might

differ in each application. Perhaps this is just indicative of the general problem of scheduling in a real-time system: whatever decision you make, something will probably happen right after you make the decision that invalidates it.

The number of process priority levels (4) seems rather small. In fact, the QTR that describes the scheduler suggests that only 2 of the levels are really intended for use by the application. Priority 3 (the highest priority) is intended for use by the operating system and the debugger. The only example given of a process that might run at priority 2 is a disk controller process. Priority 1 is suggested for the majority of "ordinary" processes, INCLUDING MOST PROCESSES THAT SERVICE I/O DEVICES (ALL ORDINARY TELECOMMUNICATIONS I/O PROCESES). Priority 0 seems to be intended for functions that have no latency requirements at all.

This rather small number of priority levels is a major difference from NMFS (and, I think, from most other real-time systems), which offers 32 priority levels. Why the difference?

The major reason for designing a system with many priority levels is to enable it to hook up to a large number of

I/O devices of various speeds and protocols, where each device may impose a different latency requirement on the processor. If some devices need servicing within a millisecond, others within 5 milliseconds, others within 20 milliseconds, etc., then a system with lots of pre-emptive priority levels provides a very natural way to service all the devices without having them interfere with each other very much. One simply assigns higher priorities to the processes servicing the devices with the stricter latency requirements. Of course, this only works if a process with a strict latency requirement runs to completion in a very short amount of time. A device which needs servicing within 5 ms. can hardly be serviced properly by a process which runs for 10 ms.

Time slicing cannot replace this sort of multi-level priority system, even if all processes are given very short time slices. Since processes of equal priority are run round robin, a process which has been sliced out might have to wait for all other processes of the same priority to run for at least one time slice apiece, which could easily cause the latency requirement to fail to be met. In addition, overly short time slices are likely to result in a large number of unnecessary invocations of the scheduler, with the consequent overhead.

Why doesn't Chrysalis provide more priority levels? There is only one cryptic statement in the documentation which is relevant to this question. QTR 18 states, "the intent is that by providing enough buffering, all ordinary telecommunications I/O processes will be able to run at level one." At first, I was puzzled by this statement, since the amount of buffering that is provided has nothing to do with the ability of the system to meet a variety of latency requirements. To understand the intent of this statement, I think we need to realize that Chrysalis was designed to be used in a processor node which includes the Butterfly I/O module (BIO).

The BIO is an intelligent microprocessor which is designed to handle the high speed I/O for the Butterfly. A BIO is part of a particular processor node, and a single BIO handles I/O to and from the memory of that processor node only. (It is interesting to observe that a SINGLE processor node can have three different processors - the 68000, the PNC, and the BIO.) The BIO does DMA I/O, and executes the link level protocols. It cannot access the memory on any other processor node, however. It is intelligent enough to get free buffers, chain them together if necessary to hold large packets, and to properly dispose of

the buffers after I/O is complete. The BIO is under the control of the PNC. I think this means that the application program running in the 68000 can do things (either execute certain instructions or else write certain locations) which get intercepted by the microcode in the PNC and turned into commands for the BIO.

When I/O processing by the 68000 is necessary, the BIO causes events to be posted, to wake up the associated processes. However, the BIO has considerable leeway as to when to do this. It can decide to wake up the 68000 only after every nth packet is received, or if a packet is received and then no further packet is received after a certain amount of time. This enables it to protect the 68000 from having to deal with very strict latency requirements, as long as there is enough buffering to enable the BIO to keep doing I/O.

I believe that it is the presence of the BIO that has made it seem unnecessary to add more priority levels to Chrysalis. Since the BIO handles all of the processing that imposes strict latency requirements, the need for a large number of priority levels in the 68000 is severely reduced. The NMFS

design, on the other hand, does not assume that most of the processing that requires quick service can be offloaded to another processor, so it provides a richer set of pre-emptive priorities.

It seems to me that the Chrysalis system is somewhat weak in its ability to handle lots of different I/O devices of different speeds and requirements, and if we have a number of such devices, we need to think about offloading a lot of the I/O processing. One option is an intelligent microprocessor, like the BIO, to do a lot of the work. A different option, worth considering, is just to add additional processor nodes to the Butterfly as needed. On the gateway, for example, we might be able to do without a BIO-like entity if we always have enough processor nodes so that no one processor is stuck with too many processes that have strict latency requirements. Or maybe it would be better to put processes with like latency requirements all on one processor, and processes with different requirements on another. But at any rate, unless we have an intelligent microprocessor to do the I/O, I would doubt whether we can build an adequately responsive system with just a single processor node.

Of course, a different approach would be to just modify Chrysalis to provide more priorities. I don't know whether this would end up producing too much overhead, or whether it is even possible (maybe the number of priorities is compiled into the PNC microcode.)

I've been discussing certain advantages which a priority system has over a system which uses time-slicing and a declared need. But of course, there are also disadvantages. In a system like NMFS, there is nothing to stop high priority processes from locking out the lower priority processes entirely. (This has happened often enough in the IMP.) Since the system will generally not function correctly unless the low priority processes run also, one would like to avoid this situation. If there just isn't time for all the processes to run, one cannot expect to get good performance no matter what the scheduler discipline, but if every process can at least get some time, there is a better chance that the system can be made to degrade gracefully.

Making the system degrade gracefully does require careful design on the part of the application, though -- this is not

something that the operating system can be expected to do for you automatically. It is all too easy to get into a situation where the high priority processes lock out the low during the initial part of an epoch, while the low lock out the high during the final part. If the application cannot detect and react to this after a little while, there will be very strange and bursty behavior patterns that do not constitute "graceful degradation". Perhaps it is necesary to keep the epoch relatively short in order to avoid this kind of burstiness.

The fact that an epoch is a fixed interval may contribute to burstiness phenomena even during normal operation. If any external events happen to get into sync with the epoch, the phasing may produce strange behavior.

The ways in which the settings of the time slice interval and declared need for different processes may affect the behavior of the system need to be carefully considered for an application. I think this may prove very hard to understand, much harder than understanding the effects of priority assignments.

In NMFS, a process can prevent itself from being pre-empted by "inhibiting interrupts". In the IMP, this is commonly

used instead of a software lock when accessing a common data structure. Chrysalis provides a good locking mechanism. But it looks like problems can arise if a process is pre-empted or sliced out when it holds a lock for an important common structure. It ought to be easy for a process to prevent itself from being pre-empted or sliced out, since all it would have to do is to mask out the 68000's level 1 interrupt. I haven't seen anything to suggest that Chrysalis provides a system call to do this, but if not, it should be easy to add.

Here is another feature which would seem natural to Chrysalis, but which I haven't been able to find. A process might have certain points at which it is willing to give up control and be placed on the end of its priority queue, even though it has more work to do, and would like to continue running if no other processes of its priority are waiting. However, I can't figure out any way to do this in Chrysalis. That is, I don't see any way to dismiss if and only if there are other runnable processes of the same priority.

In this message I have concentrated on Chrysalis's treatment of processes. A forthcoming message will deal with

Chrysalis' memory management system.