IDN-7

More on Butterfly/Chrysalis

Eric Rosen

September 1983

This message may be considered a continuation of the previous message which discussed processes and scheduling. Once again, corrections and discussion are solicited. My general goal has been to come to an understanding of the environment, rather than to get every last detail precisely correct.

I want to talk primarily about the memory architecture of the Butterfly, as seen by the software, and about the Chrysalis functions for managing the Butterfly's memory. But first, I think it would be interesting to discuss the "dual queue" feature of the Butterfly/Chrysalis system. Lots of people have heard something about it, but it doesn't seem to be described anywhere in the QTRs or the Chrysalis manual. The information below is pieced together from dribs and drabs found in the documentation along with what I remember from a talk given several months ago by John Goodhue.

In real-time communications applications, the following is a very common scenario. A process wakes up and looks at some queue. If the queue is non-empty, it removes the first item from the queue and processes it. The process repeats this (look at the queue, remove the first item, process) until the queue is empty. Then the process goes back to sleep until some other process puts something on the queue again. The "dual queue" mechanism provides a rather elegant mechanism for implementing this sort of procedure. (Dual queue functions are all implemented in the microcode of the PNC, making them fast and indivisible.)

Basically, when a process wants to get an item from the queue, it issues a "Deq_DualQ" call. One of the arguments passed to Deq_DualQ is, of course, a "queue handle" (basically, a pointer to the queue header). Another argument is an event handle. If there is any data on the queue, Deq_DualQ just removes the first datum from the queue and returns a pointer to it. HOWEVER, if there is no data on the queue, the specified event handle gets placed on the queue. (That's right, if the queue has no data, the dequeue operation actually puts something on the queue.)

To place a datum on a dual queue, a process issues a call to "Enq_DualQ". If the dual queue does not have any event handle on it, the datum is just placed on the queue. However, if the dual queue does have an event handle on it (as a result of a previous call to Deq_DualQ at a time when the queue had no data on it), then that event handle is removed from the queue, and the event gets posted. The datum that was specified in the call to Enq_DualQ is put into the event block. This wakes up the process which owns the event, and makes the datum immediately available to it.

This mechanism makes it very simple for a process to cause itself to sleep until some other process places something on a specified queue. And when some other process does place something on the queue, the first process automatically receives an event; if that process is sleeping, this will automatically awaken it. Note that the datum that is NOT left on the queue until the process runs; it is put in the posted event block where it will be seen only by the process which owns the event. Any other process which does a Deq_DualQ will find that the queue is still empty.

The dual queue mechanism is actually more general than this may make it seem. One can think of a dual queue as a data structure which, at any given time, is in one of three states. (a) It may be empty (call this EMPTY state). (b) It may be a queue of 32-bit data items (call this DATA state). (c) It may be a queue of event handles (call this EVENT state). The Enq_DualQ system call takes as argument a 32-bit data item. The function performed by this call depends on the state of the queue. If the queue is in EMPTY or DATA state, the datum is just placed on the queue. (A further argument to Enq_DualQ specifies whether the datum is to be placed at the head of the queue or the tail of the queue.) However, if the queue is in EVENT state, the first event handle is removed from the queue, the datum is placed in the corresponding event block, and the event is posted. (Note that the datum never actually gets placed on the queue if the queue is in EVENT state.)

The Deq_DualQ system call takes as argument an event handle. If the queue is in DATA state, the first datum on the queue is removed, and a pointer to it is returned. If the queue is in EMPTY or EVENT state, the event handle is placed on the queue. (A further argument to Deq_DualQ specifies whether the event handle is to be placed at the head of the queue or the tail of the queue.)

The terminology "dual queue" probably stems from the fact that the queues can have either data or event handles on them, but not both at the same time, and hence can be thought of logically as consisting of a pair of queues, one for data and one for events.

The Deq_DualQ call actually doesn't cause the process to go to sleep if the queue is not in DATA state. Rather, it allows the process to continue doing other things, knowing that it will be signalled (i.e., an event will be posted) if anybody tries to put anything on the queue. (Deq_DualQ provides a boolean return to let the process know whether a datum was removed from the queue or whether an event handle was added to the queue.) If the process does want to sleep immediately if the queue is not in DATA state, it can call "Wait_DualQ". There is also a "Poll_DualQ" system call, which removes a datum from the queue if it is in DATA state, but does not add an event handle to the queue if it is not in DATA state. The documentation suggests that Wait_DualQ will cause the process to wait for that particular event only, and hence is more like "MWait" than "Wait".

The documentation does not specify that the event handle which is passed as the argument to Deq_DualQ or Wait_DualQ must be the handle of an event which is owned by the process making the call. Presumably, this would have to be the case for Wait_DualQ to work, but perhaps Deq_DualQ allows one process to cause another process to be signalled when an Enq_DualQ is later executed by a third process.

Dual queues also provide a convenient locking mechanism. A dual queue in EMPTY state can be used as a lock. To wait on the lock, a process just needs to execute Wait_DualQ. This will put the process to sleep until someone executes an "Enq_DualQ", thereby opening the lock and causing an event to be posted for the waiting process. By convention, a process which opens a lock does so by enqueuing its Process Handle (basically, a pointer to its PCB, but can be thought of as similar to a Unix pid). That way, there is always a record of who last opened the lock.

Of course, a process that needs to use a locked data structure doesn't have to go to sleep until the lock is opened. It can use "Poll_DualQ" to perform a simple test as to whether the structure is locked, or it can use "Deq_DualQ" to cause itself to be signalled when the lock is opened, without putting itself to sleep in the meantime.

It is possible to specially declare a particular dual queue to be a "lock"; in this case, any process that does an Enq_DualQ while the lock is already in the DATA state will receive an error. This prevents a process from opening a lock which is already open.

Dual queues are actually implemented via a header structure which points to a ring buffer of 32-bit items. Since the implementation uses a ring buffer rather than linked lists, dual queues have a fixed maximum size. When one creates a dual queue (by calling "Make_DualQ"), one must allocate enough memory to hold the maximum number of entries. One can use dual queues as general packet queues, where the items on the queue are actually pointers to buffers rather than buffers themselves, as long as one is willing to initially allocate a large enough ring buffer to hold the maximum number of buffers that can ever be on the queue. If this maximum number greatly exceeds the average number, this may not be feasible. This may mean that dual queues are really only of limited use in packet-switching applications.

Another limitation of dual queues having to do with the use of the ring buffer (instead of a doubly linked list) is that it becomes difficult to add things in the middle, or to splice things out. Furthermore, there also does not appear to be any simple way at present of determining how many items are on a dual queue at any given time; this is certainly a problem if one wants to use dual queues for holding packets.

Now to get to the main part of this message, viz., the memory architecture of the Butterfly and the memory management system of the operating system.

Notation: in general, all numbers appearing below are decimal, unless they begin with "x", in which case they are hexadecimal (e.g., "10" means "ten", but "x10" means "sixteen"). However, a number which contains the hexadecimal digits A-F is hexadecimal, and may have the preceding "x" omitted.

Addresses on the MC68000 processor are 24-bit quantities. Thus every process running under Chrysalis lives in a 24-bit virtual address space. However, the maximum amount of physical memory on a Butterfly processor node is around 4 megabytes. Whenever a process makes a memory reference, the Butterfly's memory management hardware must transform the 24-bit virtual address into a 32-bit physical address.

A 32-bit physical address on the Butterfly consists of 8 bits specifying the processor node number, and 22 bits specifying a byte address. (The other two bits specify "subspace", which basically indicates whether the memory reference has should use the Butterfly switch, and whether it is to real memory or to I/O registers or EPROM. But this notion of subspace is relevant only to physical, not virtual, addressing.) Thus each physical address uniquely identifies some byte location in the memory of some particular processor node.

Note that the processor node controller, the BIO module, all I/O boards, the Butterfly's EPROM, and the Butterfly switch, all deal directly with physical addresses. Virtual addresses are meaningful only to the (macrocode) software running in the 68000 processor.

Virtual memory can be considered to be divided into a number of "segments", each of which has a maximum size of 64K bytes. A single process can have as few as eight segments, and as many as 256. The memory management hardware interprets each 24-bit virtual address as an 8-bit segment identifier and a 16-bit byte offset into that segment. So in order to transform a virtual address from a particular process into a physical address, the memory management unit only needs to know the starting physical address of each of that process' segments.

Each processor node in the Butterfly contains 512 of a special sort of register known as a SAR ("segment attribute register"). The SARs are conventionally numbered from 0 to x1FF. Each SAR contains the physical address of the beginning of some segment of physical memory. Each process is associated with a block of up to 256 contiguous SARs. More precisely, associated with each process is

a) the number of some SAR which contains the physical address corresponding to segment zero, offset zero, of that process' virtual address space.

b) a number indicating how many segments are in that process' virtual address space.

For example, a process whose segment 0 corresponds to SAR x10, and which has 16 segments, will be associated with SARs x10-x1F. (The starting address of segment 4, for example, will then be found in SAR x14.)

The two quantities a) and b) above are part of a process' context, and are saved and restored by the scheduler. While a process is running, these values are held in a special hardware register called the ASAR (Address Space Attribute Register.)

Chrysalis has the notion of a "family of processes". Two processes are in the same family if they have the same ASAR value.

Any attempt by a process to refer to an address which is "out of range" of that process' virtual address space will result in a 68000 bus error.

In order for a process' virtual address space to include some memory on a remote processor node, it is only necessary that that process be associated with a SAR containing a physical address which refers to memory on the remote node. References through that SAR will automatically use the Butterfly switch to access the memory. This is transparent to the software. However, if it is desired to transfer more than one or two words of memory at a time through the switch, it is much more efficient for the programmer to explicitly initiate something called a "block transfer". It would appear that good programming practice requires the programmer to be aware of whether he is causing a switch transfer.

The Chrysalis operating system provides a number of facilities by which a process can change the values of the SARs with which it is associated, so that a given process can refer to all the Butterfly's physical memory, even though it may not be able to have all the physical memory in its virtual address space at the same time. We will discuss these facilities shortly.

In addition to mapping virtual addresses into physical addresses, the memory management unit enforces protection on each segment, according to certain attribute bits contained in the SARs. A given memory segment may be made read-only, read and execute only, read and write but not execute, etc., and the memory management hardware will enforce this protection. Any attempt by the application program to violate this protection will result in a 68000 "bus error". The ability to protect areas of memory against certain kinds of accesses seems to be one of the main reasons for having designed a segmented addressing space.

The call to "Make_Process" will generally result (according to a message from John Goodhue) in a process whose text is in its segment 4 (read/execute only, I guess), whose data (that is, global variables) is in its segment 2 (read/write, but not execute), and whose stack is in its segment 3. (When I say "its segment 4", I of course mean SAR number n+4, where SAR n is the zeroeth segment of that process.) In order to use additional segments of memory, the process must execute various operating system calls to allocate and/or map in other areas of memory.

A segment cannot have more than 64K bytes, but it need not have that much. If one wants a relatively small area of memory to have certain protection attributes, then one will want to have a segment of just that size (i.e., one will want to devote a SAR to just that small amount of memory), since all the memory associated with a particular SAR has the same protection attributes. Unfortunately, segments do not come in just any size you might happen to want. There are only 16 possible segment sizes; in numbers of kilobytes, these sizes are 0, 0.25, 0.5, 0.75, 1, 1.5, 2, 3, 4, 6, 8, 12, 16, 24, 32, and 64.

In addition, not only is a single segment limited to 64K bytes, but it cannot cross a 64K byte boundary of physical memory. It would thus appear that the extensive use of small-sized segments could easily result in significant memory "breakage", with a lot of physical memory unusable by the software.

A process' virtual address space can have as many as 256 segments, but actually there are only 6 possible numbers of segments that a process can have: 8, 16, 32, 64, 128, and 256. That is, if a process needs to have, say, 33 segments in its virtual address space, then 64 SARs must be reserved for that process. The 31 unused SARs will be marked to indicate that they correspond to zero-length areas of memory, and they are unavailable for use by other processes. We begin to see another problem: not only is significant memory breakage a potential problem, but we have to worry about "SAR breakage" -- a large number of processes with address spaces containing numbers of segments which are not powers of two can result in a significant number of unusable SARs.

In fact, the situation is a little more complicated. From my description so far, one might think that a process with 16 segments might have its segment zero correspond to, say, SAR 5. To find memory in segment 4, then, the memory management hardware just adds 4 to 5 and looks in SAR 9. Unfortunately, things are not so simple. The memory management hardware does not ADD the segment number to the address of the process' first SAR. Rather, it ORs the segment number with the address of the process' first SAR. It does this to make the memory mapping faster. (These hardware guys seem to be able to get away with all kinds of hacks and kludges.) Unfortunately, OR'ing is equivalent to ADD'ing only if one of the operands is guaranteed to have zeros in all the right places. So for this to work properly, the first SAR of a process with 64 segments must have a number which is divisible by 64 (i.e., must be 0, x40, x80, xC0, x100, x140, x180, or x1C0). More generally, if the number of segments in a process' virtual address space is 2**n, then the number of the initial SAR of the process must be such that its rightmost n bits are zero.

The use of OR instead of ADD results in another hack. Since no process may have fewer than 8 SARs, the number of the initial SAR of a process will always have its three rightmost bits zero. Since there are 512 SARs, each SAR number is nine bits long. Therefore, if a process refers to the segments numbered from xF8 to xFF, the memory management hardware will always map those segments with SARs xF8-xFF or x1F8-x1FF, no matter what the initial SAR number of the process is. This quirk is actually used extensively by Chrysalis, which guarantees that SARs F8-FF have contents which are identical to those of SARs 1F8-1FF. All processes may refer to memory in segments F8-FF, no matter how many segments the processes are otherwise associated with. (An eight segment process whose initial SAR is, say, x10, can actually reference 16 segments, those corresponding to SARs x10-x17 and F8-FF. It references these as segments 0-7 and F8-FF.) This automatically provides 8 segments of memory which are accessible by all processes.

Segments 0, F8, and FF of every process have standard uses. Segments 0 and FF are used to reference the operating system kernel and the standard protected library routines (all of which must appear, it seems, between locations 64K and 96K of physical memory), as well as the EPROM, the PNC registers, the real-time clock, the 512 SARs, the ASAR, I/O registers, etc. Believe it or not, 68000 instructions which use virtual addresses that correspond to segments 0 and FF are one word shorter than 68000 instructions using other virtual addresses; hence the use of these two segments by the operating system. (Of course, this has to do with sign extension.)

Segment F8 is a very important segment. It always corresponds to the low 64K of memory (on this processor node). This means that the memory at offset n of segment F8 is actually found at physical memory location n. The operating system apparently makes much use of this fact. ALL IMPORTANT OPERATING SYSTEM DATA STRUCTURES MUST BE LOCATED IN SEGMENT F8 (i.e., must be in the low 64K of physical memory.) This includes event blocks, dual queue headers, process control blocks, I/O device control blocks, channel control blocks, and the "object attribute blocks" used by the object management facility of the operating system (to be discussed shortly.) User processes get read-only access to segment F8.

The existing documentation is unclear as to whether segments F9 through FE are used by the operating system or not. If not, they could be useful for storing global variables that needed to be referenced by several processes. Walter Milliken looked at the Chrysalis listing and concluded that some of these segments, at least, are used by the operating system. Probably we should just assume that if it doesn't use them now, it will use them later, and they should be off-limits to the application.

It is not hard to see that an application, if not properly designed, could easily run out of SARs. This is most likely to happen if the application contains a large number of processes, each of which has a large number of small segments. Probably this means that the application really has to minimize its use of SARs by having large segments (a strategy which would also seem to minimize memory breakage.) The disadvantage of this is that it gives away much of the hardware memory protection, since every location in a given segment has the same protection attributes.

There are also other situations which might cause the application to run out of SARs. Suppose, for example, the application needs more than 64K of buffer space. If there are several processes which need to be able to access all the buffers, then each of these processes will need to devote several SARs to mapping in the buffer space. Or the application might need less total buffer space, but might need it spread across several processors. (Segments cannot cross processor boundaries.) Since every I/O device must be part of some processor node, and can only do I/O with the memory of that processor node, a multi-processor Butterfly with its I/O devices spread around would need buffering on each processor. Yet some processes might need the ability to refer to any buffer, and so need to have a large number of SARs devoted to mapping in the buffer space.

Well, strictly speaking, one could get by using just one SAR, which one continually changes, mapping in the appropriate buffers as one needs them. However, that would appear to be too inefficient to countenance in a real-time application, and at any rate, one does not want to be stuck with a programming environment in which one is always fussing with the mapping registers. So any potential SAR shortage is a serious matter.

In general, when a process is started, it can reference at least 12 segments: 0, 2, 3, 4, and F8-FF. If it needs to have additional segments, it must take explicit action to map in additional memory. Chrysalis provides a number of facilities for doing this. These facilities are known as the "object management system", and additional bits of memory are known as "objects". Actually, the notion of "object" is very general. Processes themselves are a sort of object, as are process templates, event blocks, dual queues, channel control blocks, and just about everything else which uses up memory.

The most basic such facility can be thought of as similar to the Unix "malloc" or "calloc" functions, which just extend the virtual memory space by making it appear as if new memory has been added. To get the effect of a Unix "malloc", one must execute two Chrysalis calls: "Make_Obj" and "Map_Obj".

The call to Make_Obj takes as its arguments (a) the number of bytes needed, (b) the processor node on which to allocate the memory, and (c) the protection attributes desired for this memory. Make_Obj allocates an "Object Attribute Block (OAB)" in the low 64K of memory (segment F8), and allocates the specified amount of memory from other unused physical memory. The OAB is just a header structure containing some information about the object which is needed by the operating system. For certain special kinds of objects, the OAB is followed by additional structures which are specific to that type of object. A process control block is an OAB which contains information about the state of the process. An Event Block is also an OAB of a special sort. For the application-specific objects (known as "general memory objects" or "general user objects"), the OAB is just followed by a pointer to the physical memory allocated by the object.

Make_Obj always returns a block of memory which has been zeroed.

If the call to Make_Obj is successful, it returns something known as an "Object Identifier (OID)" or "object handle". An OID can be thought of as a pointer to (i.e., the physical address of) the OAB. (Actually, the OID is not really a physical address, but a (32-bit) quantity containing a processor node number and an index into a table of OAB addresses kept in segment F8 of that processor node.) Given an OID, a process can then call "Map_Obj" to map the object into its virtual address space. All that Map_Obj need do is use the OID to find the OAB, use the OAB to find the physical address and protection attributes of the object itself, and load up one of the process' SARs with the address of that object. The object will then be in one of that process' virtual address segments. In fact, it will be the ONLY thing in that particular segment. Map_Obj will return the object's VITRUAL address.

Of course, the call to Map_Obj will fail if the process making the call has no free SARs. It allows the process to specify the SAR to be used, but will make its own choice if no SAR is specified. In the former case, it is an error to specify a SAR which is not free.

In a C program, the virtual address returned by Map_Obj can be assigned to a pointer, which can then be used to reference the object in the standard C manner. Thus the C programmer need not explcitly concern himself with SARs.

There is a system call "Unmap_Obj", which removes an object from your virtual address space (and frees up the SAR).

The process which "makes" an object (through the call to Make_Obj) becomes the "owner" of that object. You don't have to own an object to map it in, though. The owner of an object need not be a process. It would appear that any object can be the owner of any object. There is a system call, "Disown_Obj", by which the owner of an object can transfer ownership to another object (or to the operating system). (At least I think that the Disown_Obj call is restricted to the owner of the object, since the documentation specifies a "Not Your Object" error message. Does this mean it is impossible to transfer ownership of an object which is not owned by a process?)

Events, by the way, cannot be disowned.

Each OAB has both a "use count" and a "deleted" flag. The use count is incremented each time the object is mapped in, and is decremented each time it is unmapped. The "deleted" flag is set by the call to "Del_Obj". (There is nothing in the documentation to suggest that this call is restricted to the owner of the object.) If the use count is zero at the time when Del_Obj is called, the object will actually get deleted. That is, its memory gets de-allocated, and may be reused by future calls to Make_Obj. If the use count is not zero when Del_Obj is called, its only effect is to set the flag. If at any time, a call to Unmap_Obj causes the use count to go to zero, and if the deleted flag is set, Del_Obj will be called, and the memory will be de-allocated.

If an object is actually deleted (i.e., its memory de-allocated), then all objects it owns will also get deleted by the "garbage collector". I'm not sure what the garbage collector really is; there are a bunch of cryptic references to it, but no description. It appears to be a process which runs periodically, scanning the OABs in segment F8, and taking note of which objects have owners which don't exist any more. This seems rather time-consuming and unnecessary. I think the typical application (this will certainly be true of the gateway) will do all its memory allocation at initialization time. Objects, once created, will rarely if ever be deleted, unless due to a bug. Does the garbage collector use up time in this case? If a process aborts, due to some unanticipated error, presumably all objects it owns will get deleted. If there is such a bug, does the garbage collector destroy valuable debugging traces?

When an object is actually deleted, its memory gets zeroed out. This has the advantage of ensuring that any newly allocated memory contains only zeroes, but the potential disadvantage of destroying debugging traces.

It is probably a good idea, as a matter of policy, to allocate and map in all needed objects at initialization time, insofar as this is possible, rather than to be allocating memory and mapping stuff in and out while the application is running. This eliminates the possibility of unexpectedly running out of memory or SARs. It also means that the application will need to do a lot of its own memory management; this cannot be totally left to the operating system. Certainly the programmer will need to maintain some level of awareness about how many SARs his processes use, if just to make sure that he can really fit all the processes he needs into the 512 SARS.

Another potential limitation is due to the fact that all OABs go into segment F8 (the low 64K of physical memory). Presumably, if one is not careful in the allocation of objects, one could have lots of memory available, but still run out of space in F8, thereby making the memory inacessible to the program.

There is the following known problem in Chrysalis' memory management. Suppose there are a large number of relatively small blocks of memory (e.g., buffers) which need to be accessible to several processes. Suppose further that it is necessary for processes to pass each other pointers to these blocks of memory. We have the question: should these pointers be virtual addresses, or should they be physical addresses (i.e., object handles)?

Making them be object handles is perhaps the conceptually neatest way. But that means that each of these small blocks of memory has to be an object. That would have two major disadvantages. For one thing, it would require an OAB in F8 for each block, which could put a strain on F8's capacity. For another, it would be impossible for a process to keep all the blocks mapped in at once, since each block would require a SAR of its own. Therefore, the process would have to map in a block whenever it needed it. Unfortunately, the time it takes to map in an object appears to be too large a cost to incur every time one wants to access a buffer.

Using virtual addresses as pointers to be passed between processes can work, and seems very simple and straightforward, but it requires that the blocks of memory have the same virtual addresses in all processes which might reference them. This can be useful for certain applications. However, if there are a large number of segments of buffers, and each process needs to have only a subset of the total buffer space mapped in, it is going to be almost impossible to ensure that each buffer has the same virtual address in each process that might have to refer to it.

Having ruled out the use of both physical and virtual addressing, what is left? It might seem that the only alternative is to use the object management system to map in large chunks of memory, and have the application explicity carve up these chunks into smaller blocks or buffers, and do its own management of the smaller blocks. In fact, however, the application is saved from having to do a lot of the bookkeeping that would go along with this; the operating system does provide a mechanism for managing "buffers" in a way that avoids the problems discussed in the last two paragraphs. This mechanism seems quite useful, but is somewhat at odds with the basic memory management principles of the Chrysalis operating system.

Chrysalis provides a special kind of memory object known as a "buffer pool". When one allocates the memory for a buffer pool, Chrysalis assigns it a special "buffer pool identifier", which is unique across all processors. When a process maps in a buffer pool, Chrysalis sets up a table (in the process' address space) from which it can quickly translate the buffer pool identifier into a segment number. When it is necessary to pass around a pointer to a buffer, one actually passes a special kind of 32-bit quantity known as a buffer ID. A buffer ID consists of 8 bits of "buffer pool identifier" and 16 bits of offset. To convert a buffer ID into a virtual address, Chrysalis looks up the segment number of the appropriate buffer pool in the process' table of buffer pool identifiers, and just adds in the offset. Apparently, this can be done very quickly. At least, it is much quicker than converting a physical address or OID to a virtual address, and does not require that a given buffer have the same virtual address in every process.

Of course, this procedure for converting a buffer ID to a virtual address will only work if the buffer pool containing the buffer has already been mapped in. The intention is that processes will map in all necessary buffer pools at initialization time.

How does Chrysalis ensure that the buffer pool identifier it assigns is really unique across all processors? A compile-time constant sets the maximum number of possible buffer pools that may be needed, and at system initialization time all possible buffer pool identifiers are created and placed on a dual queue. When buffer pools are created, identifiers are taken from this queue.

It is interesting to consider whether this sort of facility might be useful for managing small blocks of memory which are not generally considered to be "buffers" (e.g., connection blocks in the IMP.)

One allocates space for a buffer pool by calling "Make_BFpool". This routine takes as arguments the number of buffers in the pool, the size of each buffer, and processor node on which to create the pool, and the protection attributes which are to be given to this object. (Since the buffer pool is an object, but the individual buffers are not, the individual buffers are not separately protected. That is, all buffers in the pool are in the same memory segment.) One must also specify the object handle of a dual queue, known as the "free queue". Make_BFpool will initialize all the buffers in the pool, and place the buffer ID of each buffer on the free queue.

One can, of course, specify the same free queue when allocating several different buffer pool segments, thus getting the effect of a buffer pool which spans several segments (and possibly several processors). However, any process which needs to access those buffers will have to have all the buffer pool segments mapped in.

The system calls "BFmap_pool" and "BFunmap_pool" are used for mapping and unmapping buffer pool segments. "BFmap_buf" converts a buffer ID into a virtual address. There is a system call for locking a buffer; what it actually does is use a dual queue lock which locks all buffers in that segment. There are a number of other routines and conventions for handling buffers, which make it simple to use a chain of buffers to hold a single packet. There is also a "use count" mechanism, similar to that in the IMP; a process which wants to hand a buffer off to another process while simultaneously keeping a pointer to the buffer itself must increment the use count. A system call is provided for this purpose; it will lock automatically if necessary. There is also a system call for freeing a buffer, which will decrement the use count and, if it goes to zero, place the buffer ID on the free queue. I'm not sure whether one can specify which of several free queues the buffer should go on, or whether it has to go on the original free queue that was specified in the call to Make_BFpool.