AN INTRODUCTION TO THE HUB OPERATING SYSTEM

Mike O'Dell, KB4RGM
13110 Memory Lane
Fairfax, VA 22033

Introduction

The AMRAD Packet Switch marks a significant departure in the evolution of Amateur packet radio. Historically the computers used for packet service (TNC's and 'digipeaters) have been small, memory-poor, and worked at their limits to do the job. But they were inexpensive, and "cheapness" was an absolute requirement if packet radio were to ever be successful. The demands of creating a transit backbone for connecting local area networks far exceed the capabilities of the first-generation 8-bit machines, so while cheapness is still very important, the AMRAD switch is being designed around an IBM PClone. Further., as the cost of higher bandwidth machines continues to fall, easily rehosting the switch as hardware advances is an important consideration. With the hardware base for packet communications moving forward. AMRAD started looking for a similar vehicle on which to base the software necessary for a serious, high-performance packet switch. The Hub system is the current choice for the underlying operating system. It is very fast, simple, and is written in C. It has the capabilities needed to do the job, and will easily outlive any hardware hosting it.

Not too long ago, software was relatively foreign to many amateurs. While this has changed dramatically, operating systems beyond the scale of a few utility routines burned into a ROM, are probably not widely understood. The purpose of this document is to discuss the architecture, motivation, and implementation of the Hub system chosen for the Switch. To accomplish this without addressing only those amateurs already versed in operating systems design (wizards), a brief tutorial overview of "traditional" process-based operating systems design theory is provided. It cannot be complete, but a bibliography is provided for those interested in entering this wonderful, subtle world. After this introduction to operating systems, we will discuss the Hub in more detail. While there may be sections whose detail is necessary but inscrutable to operating systems newcomers, such readers are encouraged to skim for the highlights and press on, reading for the general philosophy. But before jumping into the technology, we provide a short historical recitation, and roll the credits.

History and Credits

The basic architecture of the Hub was created circa 1975 by Gary Grossman at the Center for Advanced Computation (CAC) at the University of Illinois in Champagne-Urbana. The people at CAC were involved with ARPAnet since "before the beginning," and were quite experienced with communications programming, protocol implementation, and "message-based" designs. (The CAC group put the first minicomputer on the ARPAnet.) While message-based designs were elegant and simple on the blackboard, they were quite inefficient in practice. On the other hand, the industrial "real-time" community built systems which could go very fast, but were nearly unmaintainable because of their intricate, monolithic design. Grossman was looking for a way to combine the efficiency of the industrial designs with the elegance of message-based schemes. The Hub was his solution. The general Hub architecture has been used over the years in several large, successful communications projects by several companies, all of which have their roots in the Center and Grossman's luminary visions. The system described here is loosely based on the system described in a thesis by [Masamoto] (Grossman's Masters student), mixed thoroughly with good ideas borrowed from other sources, and tempered with hard-won experience gained in other network projects.

An Overview of Traditional Process-based Systems

This section is intended to be a very brief excerpt from basic operating systems theory. The experienced reader can probably proceed to the next section, while the newcomer is commended to the Bibliography, particularly [Comer], [Hansen], [Organick], [Jansen], and the-like.

There are many facets of an operating system, but for the purposes at hand (communications programming), the most important functions provided are:

1 - Concurrency

This is how the operating system creates the illusion the computer is doing several things at the same time. It is not; the processor is switching between several different jobs fast enough that for most purposes, the concurrency is real. Concurrency is also known as "multiprogramming", and sometimes as "multi-tasking", although some people erroneously think these are different.

2 - Resource Management

If several activities are proceeding concurrently, it is inevitable that left to themselves, two activities will fight over some resource, be it memory, an I/O device, or access to a third activity. The resource management function coordinates the use of resources which are either scarce, or which must be shared in an orderly fashion to accomplish the required functions.

The texts cited above will spend a great deal of time discussing these issues, as there are a great many details to get straight. We intend to gloss over most of these details and go for the general philosophy.

Before explaining the traditional model of concurrency, it is valuable to briefly discuss why it is needed. At the top level of detail, any system only does one thing: that which it does! But accomplishing that "one" jobs often involves many other small jobs largely unrelated to one another, or related only at the "edges." Instead of creating one giant program which does everything, it is easier to organize a system as a collection of smaller programs which operate independently but cooperatively to accomplish the job. The execution environment of these programs must provide some mechanism for independent execution, as well as mechanisms for cooperation. This "execution environment" is called an Operating System. They come in many shapes, sizes, and flavors, but all exhibit a basic set of characteristics. The most fundamental of these characteristics is concurrency; i.e., the notion of "independent execution."

In "traditional" process-based designs, the unit of concurrency is the "process". Essentially, a process is the independent executing state of some program, or piece of program, which is conceptually executing in parallel with other processes. In process-based designs, processes interact with other processes,

with responsibilities for the job being accomplished somehow divided between the processes. Generally speaking, a process executes internally, performing some computation, until it must interact with another process before it can proceed. Maybe it needs a piece of information before it can go on, or maybe it has produced a result needed by the other process; the reason doesn't matter. What is important is that the process, for whatever reason, cannot continue its computation until the interaction completes. A process in this condition is said to be "blocked" (from proceeding further), and suspends its execution by calling a primitive system function "block()" [this is the function-call syntax for some arbitrary language]. In theory, the process somehow simply "waits". In reality, by some slight-of-hand, the state of the executing process (its registers, program counter, stack pointer, condition codes, and whatever else dictated by the hardware) is preserved in suspended animation until such time as the necessary interaction is complete. When said interaction completes, a "resume()" function thaws the frozen process which can then continue with its computation.

Nowhere have we explained how such interactions are orchestrated, or who calls "resume()" to bring the sleeping process back to life. This isn't particularly important at this juncture (see the texts for the myriad ways such matters can be conducted). What is important is the model - processes compute and communicate, compute and communicate, in an endless cycle. Moreover, to the frozen process, the block()/resume() sequence appears like a "no-op" instruction (unless a process has access to some external timing information, which makes block()/resume() appear like a very slow no-op!). This style of concurrency promotes programs with multiple processes which either compute or wait on one another to communicate, with the sequencing of those events controlled almost entirely within the individual processes.

One important implication of this structure has to do with the state-saving slight-of-hand described above, and its implementation on single processors. Since a real (single) CPU can only execute one instruction stream at a time, the CPU must be multiplexed between the processes which are logically executing concurrently. The block()/resume() pair form the basis of this multiplexing. If only one of the processes is runnable ("not waiting") at any given time, the choice of which process to run is easy. When several are runnable (the overwhelming case), policies establish how the CPU cycles are to be divided. This is called the "scheduling policy," and it too is discussed at great length in the cited texts. For our purposes, the important issue is whether the "which process to execute" decision is made only when the running process block()'s (thereby voluntarily giving up the CPU), or whether by some magic (usually related to some kind of real-time clock) "the scheduler giveth and taketh away" the processor arbitrarily. This consideration is important because this policy controls the apparent relative rates of execution of the processes. For communications purposes, this controls the "granularity" of the concurrency. [For instance, if programs are allowed to run to completion before the CPU is switched to another process, then the system would essentially not multiprogram.] The actual process of transferring control of the CPU between processes is called a "context switch." Context switches are surprisingly expensive operations, usually 5 to 20 times more expensive than a full-blown subroutine call (a subroutine call is not merely a "jump to subroutine" instruction, but includes passing arguments, etc.).

By now the reader should have a general idea of the basic mechanisms inside a multiprogrammed operating system. The reason for explaining all this is that the Hub is different.

An Architectural Overview of the Hub

This section will describe the basic structure and mechanisms of the Hub using analogy with modern microprocessor architecture. The next section will describe the implications of these mechanisms and contrast them with the traditional process model.

The classic fetch-execute cycle found at the heart of almost all processors (micro or otherwise) is also the basic mechanism of the Hub. The Hub system (Figure 1) takes its name from the Hub Queue which stores unexecuted instructions. The Hub queue is essentially the "instruction prefetch queue" for the "execution unit" of the Hub system. An important feature of the Hub, however, is the source of instructions stored in the Hub queue. In most familiar processors, the instructions to be executed ("the program") are stored in a memory, and they are sequenced into the instruction prefetch queue using the value of "the program counter." Conventionally, the program counter simply increments, causing the instruction at the "next" memory location to enter the prefetch queue. The only exception to this linear sequencing is when a "branch" instruction loads the program counter with some arbitrary new value.

In the case of the Hub queue, there is no static "program" of instructions waiting to be sequenced from memory into the Hub queue by some "program counter." Rather, instructions are created and sequenced into the Hub queue by the execution of other instructions! This means that most (but not necessarily all) instructions generate at least one new instruction when they execute. From this viewpoint, the "program" is created as it executes! Like branches in microprocessors, there is one special case of loading the queue: initializing the system. Some agent must initialize the Hub queue with instructions before the Hub fetch-execute cycle can be engaged, lest there be nothing to do!

With this basic understanding, Figure 1 should be more comprehensible, but there are some additional twists. Modern processors often have partitioned execution units - floating-point add-on chips which interpret special arithmetic instructions are a good example. When an instruction comes to the front of the prefetch queue, the instruction decoder inspects the candidate instruction and dispatches it to the appropriate functional execution unit for interpretation. A similar operation takes place when the Hub dispatches an instruction. In the Hub, an execution unit which can process instructions is called a TASK. (My apologies to readers who already have a favorite interpretation for this and several other words to follow - there are only so many such words which are short and euphonious.) In the Hub system, the Task is the fundamental mechanism for organizing an activity and providing concurrency. A Task is represented by a data area called a TASK STATE VECTOR (TSV) and represents the "register set" of an execution unit. Further, a Task must have some specification of the work it is performing, so a Task is said to be obeying a particular TASK PROGRAM (TP). The Task Program is essentially the "microcode" for the execution unit. Formally, a Task is a binding of a TSV to a TP. Such bindings are not arbitrary as a TP expects a particular data layout in the work area of an associated TSV, but many TSV's can be bound to one TP (even if the TP code isn't truly reentrant). This is like having more than one floating-point register set in a floating-point unit. Both a many-to-one and one-to-one TSV-TP binding is indicated in Figure 1. Two questions are raised by Figure 1: If a TSV is a data area, why do arrows labeled "procedure calls" point to TSV's, and how is the binding between a TSV and its associated TP represented? To address this question, it is valuable to look at Hub instructions in a bit more detail.

Figure 2 is a simplified Hub instruction. At the outset it appears quite familiar - an opcode and a few operands of some fashion. But note the added fields: the source and destination TSVid's. A TSVid is a handle, represented as a small integer, which can be used as a shorthand to indicate a particular TSV, and therefore, a particular Task. (The TSVid is returned to the proud parent when a new Task (TSV) is created.)

When the fetch-execute dispatcher examines the instruction at the front of the Hub queue, it uses the Destination TSVid in the instruction to determine which execution unit (TSV) should process the instruction. The opcode field then specifies which operation is to be performed by that particular execution unit. From the

processor analogy viewpoint, we have simply split the opcode into two pieces for easy decoding: one part specifies which CPU chip should be used for the instruction (floating-point, decimal, normal integer), and the opcode is then private to the particular chip. In the Hub system, however, the Opcode field is at least partially specified across all Tasks: a specific, small set of instructions is implemented (or at least treated rationally) by ALL Tasks. Tasks can also have private instructions in addition to this basic core set, but usually the core set provides all that are needed.

We now return to the question of binding-the TSV with a particular TP and how instructions actually get dispatched. Like the common Opcode set, the first chunk of all TSV's has a fixed format. This area with a common format is called the "TSV Prefix". One of the key items in the TSV prefix is the TASK PROGRAM ENTRYPOINT (TPE) dispatch table. The TPE dispatch table is indexed by the Opcode value and contains a pointer to the function within the TP code body which performs the interpretation of the corresponding Opcode. These function pointers form the binding between the TSV and the TP. This indirection through the TPE dispatch table in the TSV Prefix is why the "procedure call" arrows in Figure 1 go through the TSV's. To tie all the pieces together, the main fetch-execute loop of the Hub Operating System is essentially:

```
Forever {
        Fetch the next instruction from
            the Hub Queue

        Use the Instruction's Destination
            TSVid to locate the TSV which
            is to execute the instruction

        Use the Instruction's Opcode to
            retrieve the appropriate Task
            Program Entrypoint function
            address

        Call the TP function at the
            specified address to actually
            execute the instruction
}
```

Architectural Implications

The Dispatcher loop described above is really all there is to the Hub. There is no notion of "blocking". Task Program Entrypoint functions run to completion and then return to the Dispatcher loop. In the process model, the state of an activity's computation is implicitly contained in the variables (registers, storage, stack segment, etc.) of the process encapsulating the activity, and this thinly-diffused state information must be maintained inviolate by the context switch mechanism. In the Hub, the TSV represents all the storage needed by an activity. Any state information which must be retained across TPE invocations must be recorded in the TSV.

There is only one stack segment in the Hub; the Dispatcher invokes the TPE functions with subroutine calls, so no context switch is needed. (This also means no tricky assembler code is required, since context switch functions are almost always written in assembler.) Further, in process systems, processes often "idle" internally when they have nothing to do. In the Hub, a TSV with nothing to do doesn't get dispatched. If the Hub queue runs out of something to do, the Dispatcher idles waiting for an instruction. Where do these instructions come from? Interrupts.

In most systems, interrupts are viewed as nasty creatures which are to be sterilized as quickly as possible. In process-based designs, the usual litany is to make the interrupt simply awaken a sleeping process which then does whatever is necessary to placate the device. In systems with high interrupt rates, this means high context-switching rates, and poor performance.

In the Hub environment, interrupt code usually does some simple work to clean up after the device, and start a new operation if there is an outstanding queue. Notification of the client activity that the operation has completed is accomplished by simply queuing an instruction. Therefore, if interrupts are enabled and devices are running, it is allowable for the Hub queue to empty. The system is simply waiting on I/O. One other source of interrupts is the real-time clock. This is implemented by wiring a periodic source to an interrupt input so as to produce a periodic interrupt heartbeat at a known rate (usually 60-100 Hz). In communications systems, the situation often arises where there is no other I/O pending, but the system is idling, waiting for some time to pass before some Task starts dumping a retransmit queue, etc. The clock interrupt provides the timebase which drives the Timer Management functions, which in turn provide timer services to TSVs.

Hub Support Services

In addition to providing concurrency, an operating system also provides support services for activities. Memory management is an important service, as are time management, buffer, and queue management. These services provide useful primitives which need not be reengineered by each Task, and are orchestrated in such a manner as to promote cooperation between Tasks.

Memory and Buffer Management

In this Hub implementation, memory management and buffer management have been lumped together. They can easily be split back apart (even in this implementation) but using the same allocator for buffers and what would normally be "general memory requests" is very attractive. The overwhelming experience with communications programming is that internal fragmentation induced by fixed size memory allocation is much easier to tolerate than the external fragmentation which results from variable size memory allocation. Further, fixed-size allocators are always faster, usually by a large margin, since they don't have to look around to find a piece of memory big enough to satisfy the request, nor try to coalesce chunks when they are freed. Two problems arise with fixed-size allocators: (1) the small-object/large-object problem, and as a result of addressing(1), we find (2) a maximum allocation limitation.

Network traffic comes in two sizes - small packets containing terminal traffic, and large packets containing file transfer data. If storage is allocated in chunks big enough for the large packets, much storage will be wasted. If we chose to allocate small chunks, we must insure the overhead of keeping track of them doesn't consume too much storage or processor bandwidth. The compromise is to allocate storage in medium-sized units.

The storage allocation units are sized to accommodate some large percentage of all requests (ranked by size and frequency). When large packets are required, the buffer header contains pointers which can be used to chain buffers together to form multi-buffer "messages", in addition to the usual pointers for chaining buffers into a queue of distinct messages.

A direct implication of a small/large compromise allocation size, is a rather severe upper bound on the size of an object which can be created using storage acquired from the allocator. In a general-purpose programming environment, this would be unacceptable. For communications programming, however? this doesn't usually pose a hardship, although it occasionally complicates some algorithms a bit more than they might be in other circumstances. If future applications of this Hub implementation find themselves in hardship situations, as was stated above, the memory and buffer allocator can be split apart to provide the necessary level of service.

Clock and Timer Management

In this implementation of the Hub architecture, we assume a single, fixed-period real-time interrupt which is vectored (somehow!) to the Clock Interrupt Handler. One interrupt is called a "tick", and represents the smallest real-time increment which can be directly measured or created in a given implementation. The Clock and Timer support routines provide Task Programs with

functions for scheduling timer notification events after a specified interval, canceling scheduled events, measuring real-time intervals, and maintaining the current time-of-year.

The Timer functions maintain a queue of upcoming timer events with the queue elements recording the difference between the time of their event and the previous event, measured in ticks. This way, the clock interrupt code need only decrement the time remaining in the first element of the timer event queue until it is zero. When the difference has been reduced to zero, the appropriate TIMER instruction destined for the recipient is placed in the Hub queue, and the timer queue element recycled. This process of "decrement once per tick until a zero difference is detected" repeats as long as there are timer queue entries.

This does place a burden on the queuing functions. The timer queue element must be sorted into the queue in the correct location, while adjusting the requested absolute time to the appropriate difference while scanning the current queue. When a scheduled event is deleted from the queue, only the following element must be adjusted. The algorithms are not complex, but do require careful attention to the boundary conditions.

Queue Management

In this implementation, queues are doubly-linked and are headed with a dummy queue cell. This affords easy insertion, deletion, and sanity-checking. The usual assortment of insert, delete, and discard functions are provided. They are optimized for queues of buffers.

Unexplored Issues

This paper, while providing an overview of the Hub (and some background for the operating systems newcomer), has left some important issues unaddressed. While the "CPU" model of instruction execution spawning new instructions which are added to the Hub Queue is fairly straightforward to grasp, it offers only modest direct insight as to how one structures concurrent systems using the Hub primitives. The actual subroutine calls necessary to do the work in a Hub System, and how one goes about rehosting a Hub implementation are also important issues which weren't addressed either. But this paper only claimed to be an Introduction to the Hub Operating System, so some these issues remain ideal topics for other documents, and others are best addressed by reading the code. We intend to report our experiences in all these areas.

Bibliography

Comer, D., "Operating System Design: the XINU Approach," Prentice-Hall, 1984

Jansen, P.A., "Operating Systems Structures and Mechanisms," Academic Press, 1985

Hansen, B., "Operating Systems Principles," Prentice-Hall, 1973

Masamoto, K., "Implementation of HUB System," University of Illinois Master's Thesis, 1976

Organick, E.I., "Computer Systems Organization: The B5700/B6700," Academic Press, 1973

Organick, E.I., "The Multics System,", MIT Press, 1972

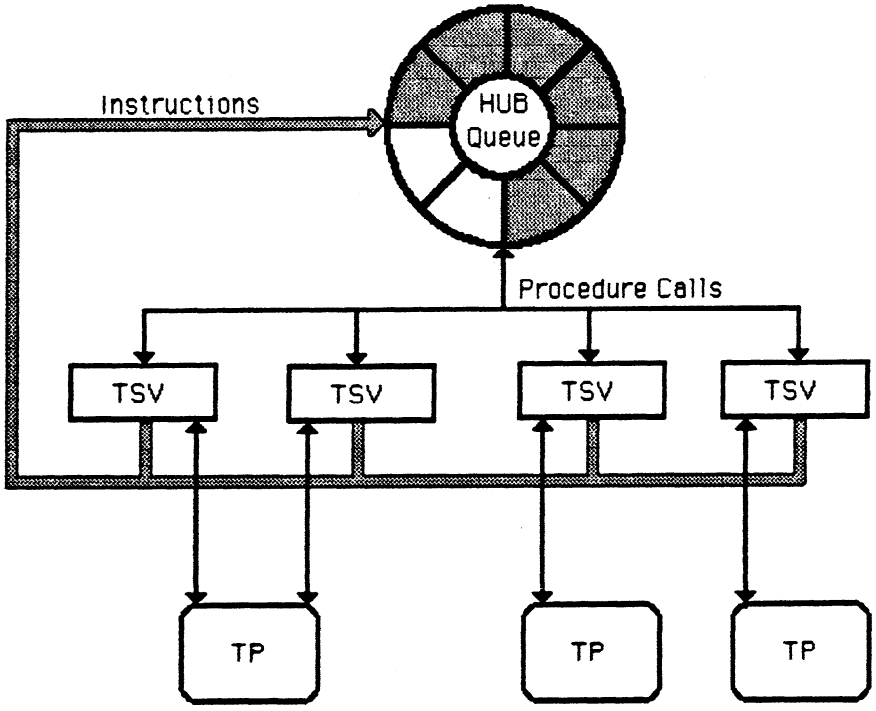

Figure 1
The HUB System

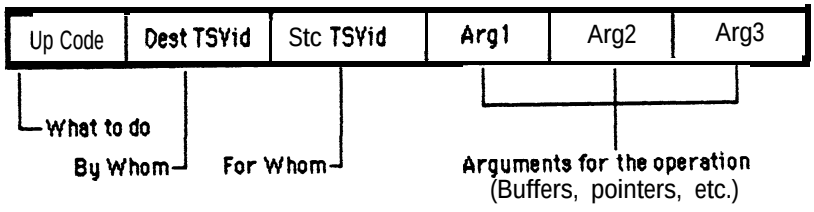

Figure 2
A Hub Instruction