The Kernel within the APM

Conclusions first!

Functions required in the software kernel (** means that a conventional single-machine mechanism needs generalisation to support distribution):

Basic hardware management

** Hardware grope and reconfiguration

Hardware fault handling and basic diagnostic support

Maintaining the environment (including virtual memory) for individual tasks

** Resource allocation

** Message passing between tasks

Access to shared resources

Task synchronisation with external events

Error trapping for tasks

Filestore access

Creation and deletion of tasks

Basic monitoring and logging mechanisms

These have to divided into global control functions and local control functions.

John Wexler
26th. September 1985

In my back-of-envelope sketches, I envisage a distributed system which is represented in each APM by a small limited kernel of software. Some of the functions of this kernel might be:

Basic hardware management.
Essential.

Hardware "groping" and reconfiguration.
Very useful, but not absolutely essential.

Hardware fault handling.
Essential.

Multiprogramming several independent tasks on the single processor.
Essential.

Maintaining the separate environment for each task.
Essential. This heading covers "virtual memory support" - see below.

Protection between tasks.
Essential.

Allocation of resources to tasks.
This might be left to a special task (e.g., accepting requests in the form of messages from other tasks and sending response messages when the resource is available), or it could be done at kernel level. The function is essential, so the "special task" would have to be a permanent companion of the kernel in any case.

Message passing between tasks.
Essential (except that some people prefer equivalent mechanisms under different names).

Providing interfaces (and access discipline) to resources which are shared rather than allocated.
This can be very useful, but there seems to be a tendency to present all resources through an allocation scheme even if one is only allocating a momentary permission to access an essentially permanent shared object. If one wants simpler mechanisms for shared resources, the kernel is the right place for them.

Task synchronisation with external events.
Essential, but need not be distinct from the message-passing mechanism.

Error trapping for tasks.
Essential.

Filestore access.
Could be done at library level, but it seems so universally useful that the kernel is the more appropriate place.

Creation and deletion of tasks.
Essential.

Monitoring.
The kernel should provide basic facilities to be switched on and off and controlled by higher-level (library) software.

Logging.
If the kernel maintained an in-store buffer of records of recent activity, it could be a higher-level (library) function to make occasional permanent (filestore) copies of the buffer.

Diagnostics.
Diagnosis of hardware trouble would be desirable (if it is practical), and perhaps some basic diagnosis of software problems within tasks but most diagnosis of program errors should be handled by library software.

Providing simple interfaces for facilities whose low-level operations are unnecessarily complicated.
This should only be provided for mechanisms which are universally required; the library is the proper place for most simple interfaces.

Providing common interfaces for conceptually similar facilities whose low-level operations are unnecessarily different.
This kind of thing should be in the library.

External communications.
A high-level facility, not appropriate at the kernel level.

Spooling.
A high-level facility, not appropriate at the kernel level.

Each task should run in its own independent virtual memory (rather than in its own part of a system-wide virtual memory). The support of virtual memory is thus part of "maintaining the separate environment for each task".

The kernel as described so far is an ordinary operating system kernel for a single machine, with no special provision for "distributed operating". Distributed or not, there has to be some mechanism to look after what goes on in an individual machine, and it seems natural for that mechanism to reside in the machine itself: does it make sense to try to run the kernel for machine A on the processor of remote machine B? The provision for distribution should be made by extending certain kernel functions; it should not be necessary to add totally new functions. The areas which would need generalisation include:

1. Hardware groping (to recognise remote machines, etc.)
2. Allocation of resources
3. Message passing
4. Access to shared resources (but simple sharing is probably not appropriate for remote resources)
5. Synchronisation (but message passing is probably the appropriate mechanism for remote events)

So far, I have written of the "kernel" and "tasks" as if they are totally separate, suggesting a simple scheme with a single privileged kernel looking after a number of non-privileged tasks. This is rather too simple. What we are aiming for is to run independent streams of non-privileged user code within tasks; and we can expect to find a single unit of privileged "global control" code at the heart of the system; but the "protection boundary" between kernel and user code need not be the same as the boundary between global control code and tasks. It is reasonable (and useful) to have some privileged kernel code running within individual tasks, so that the protection boundary divides each task into a "local control" section and a user code section. (On the other hand, I can see no sense in putting the protection boundary "below" the task boundary, but someone might find some interest in the idea.)

In a structure like this, the kernel is divided into global control and local control code. The list of kernel functions must therefore be divided into global control functions and local control functions.

If we could work with multiple levels of protection, more elaborate schemes could be devised. For instance, library facilities might be more privileged than user code but less privileged than the kernel. For the time being, however, I am assuming that library facilities would run as user code.