Asynchronous I/O

One of the important features of the file system interface is the strong emphasis on asynchronous I/O. We've seen that the message system is intrinsically asynchronous in nature, so this is relatively simple to implement.

Processes can choose synchronous or asynchronous ("no wait") I/O at the time they open a file. When a file is opened no-wait, an I/O request will return immediately, and only errors that are immediately apparent will be reported—for example, if the file descriptor isn't open. At a later time the user calls `awaitio` to check the status of the request. This gives rise to a programming style where a process issues a number of no-wait requests, then goes into a central loop to call `awaitio` and handle the completion of the requests, typically issuing a new request.

Interprocess Communication

At a file system level, interprocess communication is a relatively direct interface to the message system. This causes a problem: the message system is asymmetrical. The requestor sends a message and may receive a reply. There's nothing that corresponds to a file system `read` command. On the server side, the server reads a message and replies to it; there's nothing that corresponds to a `write` command.

The file system provides `read` and `write` procedures, but `read` only works with I/O processes, which map them to message system requests. `read` doesn't work for the interprocess communication level, and in practice `write` also is not used much. Instead, the requestor uses a procedure called `writeread` to first write a message to the server and then get a reply from it. Either the message or the reply can be null (zero length).

These messages find their way to the server's message queue. At a file system level, the message queue is a pseudofile called `$RECEIVE`. The server opens `$RECEIVE` and normally uses the procedure `readupdate` to read a message. At a later point it can reply with the procedure `reply`.

System Messages

The system uses `$RECEIVE` to pass messages to processes. One of the most important is the *startup message*, which passes parameters to a newly started process. The following example is written in TAL, Tandem's low-level system programming language (though the name stands for "Tandem Application Language"). TAL is derived from HP's SPL, and it is similar to Pascal and Algol. One of the more unusual characteristics is the use of the caret (^) character in identifiers; the underscore (_) character is not allowed. This example should be close enough to C to be intelligible. It shows a process that starts a child server process and then communicates with it.

The first piece shows the parent process (requestor):

```
    call newprocess (program^file^name,,,,,, process^name); -- start the server process
    call open (process^name, process^fd);             -- open process
```