Sample Solutions for Problem 3 in Problem Set 1

Here are sample solutions for the two parts of Problem 3 in Problem Set 1. These are worth looking at for being well thought out and surprising.

> ***3a.*** *Sketch a comprehensive design for an asynchronous I/O subsystem, including disks, or argue that one of the current designs is best. Think about buffer management, event notification, the cost of system calls, ease of use, and portability to application-level interfaces. (Many servers provide an I/O-like interface; think of an HTTP server, which is basically* `open`/`read` *if you ignore* `POST`*. Could your design apply there too?) Don't be afraid to present something partially-baked, as long as it's interesting and different.*

[From Steve VanDeBogart]

. . .

As we saw in question one, all events aren't currently representable by file descriptors. In order to make a complete asynchronous I/O system, this would have to be remedied. For example a system call that returns a socket that gets events when signals are delivered to the process.

A wacky idea to prevent blocking (which is what we are trying to do in an asynchronous I/O subsystem) is to support a *list* of things to do. For example in a webserver, we want to open a file, read it, write it to a network socket and then close one if not both of the file descriptors. One could add a syscall which takes a micro-code of what to do before signaling an event. This would kind of be like DMA for a program. Instead of getting a device to send something directly into memory, you're telling the kernel what you want to do with some files. It can then do those things without having to return an event and ask what to do next at each step.

If an error occurred with any step, the kernel could return early with a pointer into the microcode saying where something went wrong. In this case the webserver would just be a microcode composition engine and a set of error handlers. This may lead to scalability problems because of transferring the microcode into kernel space repeatedly. In order remedy that you could name a piece of microcode and let it contain parameters. Then a program would only have to transfer a given piece of microcode once, but could use it with low overhead for each of its operation sequences.

Of course, if you take this to it's logical extreme, the microcode would be an entire webserver, handling all possible errors internally and you'd make just one call, to start it. That is mostly what Tux is, an in kernel webserver, except the 'microcode' is already resident in kernel space.

> ***3b.*** *Sketch a piece of programming language technology that could make asynchronous I/O easier to use. The papers by Mazières et al do it one way for C++; pick a language of your choice, or make up syntax to be added to some language. For instance (language nerd alert), what could you do with Haskell's monadic I/O?*

[From John Handy]

One half baked idea for hiding the complexity of an event-driven program is to employ a higher level programming language that allows definition of the sequence or pattern

of events that each independent application session must follow to properly service the application. The language processor will generate from a program in this higher level language the executable program that simulates the finite state automaton that accepts this pattern of events and that can interleave the events of any number of simultaneous instances of the pattern of events. One simple way to define a pattern is with a regular expression where the symbols correspond to notifications of successful I/O operations or of error events. Maybe a more powerful language than regular expressions would be needed in general, but a finite state machine (FSM) augmented with data structures for storage of data from the I/O operations and some method to invoke operations on the data structures as specific symbols of the regular expression could be very expressive and easily implemented. The FSM would accept a pattern of notifications while generating a pattern of actions to initiate operations. For example the very simple web server that we have been discussing has each connection go through these steps: accept connection, read contents of request, open disk file for request, `while () {` read disk, write to network connection } close disk I/O, close connection. As a regular expression:

⟨Connection Accepted⟩ [Get request] ⟨Request obtained⟩ [Open requested file] ⟨Requested File Opened⟩ ([Read from opened file] ⟨Data obtained from opened disk file⟩ [Write to network connection])* [close file] [close connection]

A phrase in ⟨·⟩ are input symbols (notifications) for the FSM. A phrase in [·] are output symbols (actions) of the FSM.

The language processor could output C code in the same way as LEX and YACC, or could create object code directly. The actions would likely correspond to function calls to C functions of the proper form.

In fact, I have designed and partially developed a similar technique for a different application for event-driven programs, graphical user interfaces. I plan to present this as my project for this class.