Building FBP from Scratch

Basics of RTOS's (O/S's)

- RTOS == Real Time Operating System
- supports notion of “processes”
- O/S == operating system, basically an RTOS in disguise (embellished by lots of “useful” libraries)

CPU's

- Most “standard” CPU's are based on CALL / RETURN principles with a single stack pointer.
- Most CPU's are single-threaded, synchronous
- (Yes, there are variations, but the above is mostly true and sufficient for the following discussion).

So, How Do Asynchronous Processes Run on Modern (synchronous) Hardware?

Process

- A “process” is just a subroutine.
- A “process” is described by a small record, called the PID (Process Identifier)
- The PID contains pointer / state (+ maybe a pointer to the process' stack, if any)

Scheduler

- The “scheduler” is simply a subroutine that runs the system – it's the heart of an O/S.
- It “calls” other processes (simple subroutines).
- When a process “suspends”, it executes a “return” to the scheduler.
- The scheduler contains several queues, e.g. 'ready', 'blocked on I/O', 'blocked on queue read', 'blocked on queue write', 'blocked on signal', etc.

Blocking

- A process “blocks” itself when it wants to wait for something.
- A process “blocks” itself by placing it's own PID on the appropriate 'blocked' queue, then it executes a “return” to the scheduler.

Kernel

- The common operations, such as blocking, are often refactored and collected into a library called the Kernel (for RTOS'), providing an “API” for the processes to call.
- An Operating System (Windows, Mac OS X, Linux) are simple kernels embellished and generalized with extra library routines (file I/O, MMU handling, windowing, etc. - all done in 'blocking' style).

Minimal FBP

- FBP is often implemented using existing operating systems (Windows, etc., generalized threading libraries (green threads, boost, etc.))
- If one considers an FBP program to be a 'program' instead of a bunch of mutually harmful processes, one can discard *most* of the operating system.

Minimal FBP (con't 2)

- input and output queues
- a way to block (enqueue + return)
- a scheduler (just a subroutine)
- syntactic sugar

Minimal FBP (con't 3)

- A “stack” is needed only if time slicing is allowed.
- Stack == captured state, with a return-point (stored in the PID or at the top of the stack).

Minimal FBP (con't 4)

- A “stack” is *not* needed to implement nested sends, if a local “output queue” is provided for each component.
- A “send” places the IP on the local output queue. Dispatch occurs only when the component blocks.
- Scheduler moves IP's from the local output queue to the appropriate destinations, when the called process return's to the scheduler (i.e. blocks).

Conclusion

- FBP uses only a minimal subset of Kernel functionality.
- The rest of the O/S can be thrown away, resulting in a very efficient implementation (aka “programming language”).
- FBP (and its scheduler) can be implemented in any language / browser that supports CALL/RETURN.
- Processes, yields, generators, etc. are not needed (and confuse the issue(s) with over-generalization).

Collate

- A demo of the above techniques can be found at:

  https://github.com/guitarvydas/collate-fbp-classic

- (This sample is very “bare bones”. A deliberate decision was made to represent the flow graph as code instead of hiding the graph in a data structure).