Title:	Programming Problems and Approaches	Created:	2022-11-22
Status:	Initial collection	Author:	Lion Kimbro

Intent

This is a collection of problems that I work on, and approaches towards resolving those problems.

I intend to use it as a reference, because I find that I often revisit problems in the future, that I have visited in the past – to try and solve it another way with a new insight, or, alternatively, to implement something that I've already solved in the past, with the benefit of knowledge gained in the past.

Log (L)

2022-11-22	started document: L, P

Problems (P)

Pr#	*Added*	*Summary*	*Flags*
1	2022-11-22	Transactions Recording and Restoring Transaction	
2	2022-11-24	Safe Data Storage Recording Data in a Durable, Safe Way	
3	2022-12-04	Function Inventory Keeping track of what functions are intended to be used in what ways	
4	2022-12-07	Component Connecting How components are connected together into a working system	
5	2022-12-07	Sequencing Bus Broadcasts Controlling sequences amidst bus subscription and publications.	

Problems in Detail

Pr1: Transactions

id	Pr1
title	Transactions
descriptor	recording and restoring transaction

2022-11-22	wrote up
2022-12-07	(added this header block)

Parts of the Problem:

1. Recording Transactions
2. Recovering from Uncompleted Transactions
    - How is the incomplete transaction detected?
    - When is the recovery performed?

Approach: Append transactions to a Journal.

Variant: Append JSON transactions to a JSON List

For working with a JSON List (a text file in which each line is an independent JSON expression; taken together, all of the lines can compose a list of JSON data objects), write JSON entries, and then write "COMMIT" on an independent line.

When reading the file, if there's a trailing end without "COMMIT" at the end, append a "DISCARD" line, and then carry on.

Because "COMMIT" and "DISCARD" (minus the quotes) are not valid JSON lines, and discretely occupy an entire line, they can be detected without problem.

Approach: One File Per Transaction

Pr2: Safe Data Storage

id	Pr2
title	Safe Data Storage
descriptor	recording data in a durable, safe way

2022-11-24	wrote up
2022-12-07	(added this header block)

Approaches:

1. Store transactions. (See Pr1.)
2. A/B File Writes

If you want a time-traversable history of edits, store transactions in appends – though, be aware, it is considerably costlier to write, because you need to encode your method of writing to the data store.

Approach: Store Transactions in Appended Events

How it works:

Read the data from an event source, and apply it to an in-memory representation. Whenever you want to change it, do so through a transaction. The transaction is encoded to disk, and appended to a file. After the transaction is fully encoded, write “COMMIT” into the file, some how. When reading, queue changes until reading “COMMIT”, and then apply them and empty the queue. If there is a remainder, with no COMMIT, either delete the remainder, or write “DISCARD” into the log. That was an interrupted transaction that cannot be applied.

See Pr1 for other approaches to storing transactions.

Things to consider:

- Requires a serializable notation for edits.
- Requires application of changes through an explicit API.
- Permits going backwards and forwards in time. Snapshots can optimize, and reverse edit logs can be constructed to optimize reversing through time.
- “Note” events can be stored into the transaction log, to label transactions. To use these, you need to support an API that allows for retrieving and understanding the transaction history.

Approach: A/B File Writes

How it works (1):

Keep two file targets – A and B. Have a file that says, “File A is here, File B is there.” Then have a pointer file that says “A” or “B”, in a single byte. When you read, read from the file that is named in the pointer file. Write to the *other* file, and then once you know that the write has been safely concluded and flushed to disk, edit the pointer file to point to the file you just wrote.

How it works (2):

Keep two file targets – A and B. Writing: When you write a file, write it into the B position. Then, copy it into the A position, and then, delete the B position. Reading: When reading, read file A. If it loads completely, delete B, if it is there. If A does NOT load completely, then load file B. It should read completely. Then copy B into the A position. Then delete B. This approach does not require keeping a pointer file. You will need to create a convention for naming file B, after file A – perhaps add “.safewrite” to the filename for file A, to create the filename for file B.

Things to consider:

- x2 disk space (though: you can delete the prior file, after you’ve written to the first file – still; you require the headroom to store two side-by-side while writing)
- no historical log, beyond the prior file (if you don’t delete it)

Pr3: Function Inventory

id	Pr3
title	Function Inventory
descriptor	keeping track of what functions are intended to be used in what ways

2022-12-04	wrote up
2022-12-07	(added this header block)

The Problem

The problem is about – knowing what functions can be used in what ways.

There is today, only sporadic tracking of what functions can be used in what contexts. This is deeply surprising to me, because it's deeply important to maintaining the promises that a system must uphold, and it seems rather obvious to me. It's further surprising to me, because there is so much popular emphasis for just about any mechanism that could increase the safety and secure completeness of a system – in a way that I consider even overboard today.

Nonetheless, I can't think of any programming language that supports these concepts. "Public, Protected, and Private" approaches this, and before that, declaring "extern" and such. But this is far more detailed. It's easy to imagine how compile-time and execution-time systems can verify these declared assumptions.

Approach: Label Functions

1 Char	*3 Char*	*Title*	*Meaning*
`O`	`OBS`	Obsolete	This function is not to be used for any purpose, and is essentially obsolete.
`D`	`DPC`	Deprecated	This function is not to be used except by old code.
`1`	`1`	Delegation/Singular	This function is to be called from only one specific other place. It's essentially just code that was "broken out" from that larger block of code. It should not be called from other than that one location.
`2/N`	`2`	Limited Calls In	This function is to be called from only a specific, limited number of locations. Nobody else is supposed to call this code. It makes some behavior uniform between those functions, but is otherwise not intended for general use. Most likely has code "Internal Only" (`I`), but it's conceivable that it would be otherwise – for example, a framework that requires that the user of the framework call a specific function, that will only be called from one location.
`|`	`ONC`	Called Once	This function is to only be called once in the lifetime of a program execution.
`C`	`CLI`	Client Callable	Those modules that make use of the module that this function is located in, are encouraged to call this function. It is intended as an entry into this module's functionality. Implies External Use OK (`X`).
`U`	`UTL`	General Utility	This function is a general utility function. Anything and everything can call it.
`[STATE]`	`[STATE]`	State-Specific	This function should only be called when the module is in a specifically configured (and named here) state. Calling outside of that state will likely result in certain errors. Don't use this for "living," though, described by "Living"
`A`	`RUN`	Setup	This function is called to setup, initialize, or configure the system. After it is configured, it is "living."
`Z`	`END`	Teardown	This function is called to tear down, free, or otherwise close down the system. Afterwards, it is "dead."
`I`	`INT`	Internal Only	This function is only to be called from within the module.
`X`	`EXT`	External Use OK	This function is ok to call from outside the module. (Consider also: `C`, `U`)
`P`	`PLG`	Plug	This function is intended to be called by a specific something from outside of the module. It is part of a "plug." If there are multiple different plugs, they should be grouped together somehow – PLG1, PLG2, PLG3, likely with identifiable names. You might want to have a single function call, representing the entire plug, and then use context or a variable to indicate which sub-functionality is intended.
`G`	`GET`	Getter	This function is a getter.
`?`	`CHK`	State Check	Check the state of the system; Does NOT change the state of the system in doing so.
`S`	`SET`	Setter	This function is a setter.
`N`	`NFY`	Notifies	In the execution of this function, it performs a notification, that can lead to a chain reaction of events. That notification may be acted upon immediately, or at a later time, that is not discriminated here. (I might want to create a code for notifications that can cause chain reactions immediately – perhaps "NOW" / "!" for "acts now" or "LTR" / ">" for "acts later". – "N!" or "N>")
`!`	`NOW`	Right Now!	(Intended for use with "Notify"/NFY) Use to specify that something else will be invoked right now, rather than scheduled for later. Use only to disambiguate, since… …most calls cause something to happen, when the call is issued…
`>`	`LTR`	Later	(Intended for use with "Notify"/NFY) Use to specify that something else will be invoked at a later time, by some kind of scheduling typically, rather than occurring immediately.

R	RCR	Recurses	This function calls itself, or calls another function that calls itself again – regardless, recursion is a major theme in or around this function.
L	LIV	Living	This function requires that the system is "Living," in order to be called. This is a special, but super-common, case of State-Specific ([STATE]) that is called out here because it's just so important and common.
Y	LWY	"Yes" / Always	This function can be called regardless of whether the system is living or not.
*	*	Special Note	Be sure to read and understand the special notes attached to this function, before use.

(Used: "[|*!?>12ACDGILNOPRSUXYZ")

Futher, before I have a programming language that implements these assumptions, I can use naming conventions:

3 Char	*Form of Function Name*	*Imagined Example*	*Note*
OBS	obsolete_xxx(...)	obsolete_adder()	
DPC	dpc_xxx(...)	dpc_adder()	
1	xxx_1(...)	notify_clients_1()	
2/N	xxx_2(...) or xxx_3(...) or ...	notify_clients_3()	
ONC	xxx_callonce(...)	setup_callonce()	
CLI	xxx(...)	foo()	
UTL	util_xxx(...)	util_map(...)	
[STATE]	state_STATE_xxx(...)	state_READING_visit(...)	
RUN	setup_xxx(...)	setup_printer_callonce()	
END	teardown_xxx(...)	teardown_callonce()	
INT	internal_xxx(...)	internal_foo()	Alternatively – Python recommends using "_" prefix, so: _foo().
EXT	xxx(...)	foo()	
GET	get_xxx(...)	get_module()	
CHK	check_xxx(...)	check_running()	
SET	set_xxx(...)	set_sel(...)	
NFY	xxx_notifies(...)	set_sel_notifies(...) or set_sel_notifies_deep(...) or set_sel_notifies_actlater(...)	
NOW	xxx_notifies_now(...)	set_sel_notifies_now(...)	
LTR	xxx_notifies_later(...)	set_sel_notifies_later(...)	
RCR	xxx_recurses(...)	fib_recurses(...)	
LIV	xxx(...)		No special designation.
LWY	always_xxx(...)		Do not apply to setup or teardown functions.
*	xxx_SPECIAL(...)		The "SPECIAL" in the name means to make sure that you read the special notes with regards to the function – there's something highly irregular about it.

Pr4: Component Connection

id	Pr4
title	Component Connection
descriptor	how components are connected together into a working system

2022-12-07	created; wrote up approach: Bus System, focusing on non-determinism as a problem, and ways to potentially ameliorate the problem of non-determinism

Approaches:

- Hard-Coding: Manual Instantiation and Connection
- Bus System

Approach: Bus System

(This article is written 2022-12-07.)

Advantages:

- future adaptable – as long as new things “play nice,” future elements can be dropped in, and the system will just work

Drawbacks:

- separation of cause and effect – you have to track chains of activity across time, through additional layers of communication and buffering – this can mean that it’s harder to debug, to connect something that happened, with the cause that led to it happening
- non-determinism – the sequence of events and communications can depend on arbitrary attachment timings; this can lead to mysterious hard-to-isolate bugs that show up in one environment and not in another
- complexity – it’s just a more complex system now, with additional layers of separation

One way to ameliorate non-determinism, is by deliberately sequencing messages in a consistent manner. See Pr5: Sequencing Bus Broadcasts, for consideration of ways to do this.

Pr5: Sequencing Bus Broadcasts

id	Pr5
title	Sequencing Bus Broadcasts
descriptor	controlling sequences amidst bus subscription and publications

2022-12-07	created

This is about strategies for controlling sequence when you have a system of bus subscription and publication. I am particularly focused on this as a solution to Bus solutions to Pr4, and I am particularly focused on an environment in which notification occurs through nullary function calls, thus requiring an establishment of context before the nullary function is called (since the parameter stack is not usable for providing context.)

Approaches (Nullary):

- Ordering Dispatch – sequence messages
- Ordering Dispatch, Registering Context With Message – attaching context to message
- Hard Coded Categories of Delivery – messages only issue when a context for the message is available

Approach: Ordering Dispatch

The idea here is to look at the messages to be delivered, and then to sort them before dispatch, into some specified standard sequence, known to produce good results.

It might look like this:

```
messages = set()

foo = lambda: print("foo")
bar = lambda: print("bar")
baz = lambda: print("baz")
something_else_1 = lambda: print("something else 1")
something_else_2 = lambda: print("something else 2")

def register(fn):
    messages.add(fn)

def dispatch():
    for fn in [foo, bar, baz]:  # this is the deterministic ordering
        if fn in messages:
            fn()
            messages.remove(fn)  # these calls are deterministically ordered
    for msg in messages:
        msg()  # these calls are non-deterministically ordered
    messages.clear()
```

The "something else" messages are ambiguous, but foo, bar, and baz will be well-ordered.

One problem with this approach, in conjunction with my preference for nullary functions, is that should lambdas be passed in, in order to perform context-setting for a function, the function cannot be identified: the lambda conceals the intended message, and thus cannot be ordered. This leads me to a related approach – publishing not only the message, but also the context, and making sure that they are paired.

Approach: Ordering Dispatch, Registering Context with Message

```
messages = set()
contexts = {}

def register(fn, context):
    messages.add(fn)
    contexts.setdefault(fn, []).append(context)

def order_contexts(L):
    …perform a discrimination of contexts, in order to determine the ordering L2…
    return L2

def set_context(context):
    …perform system operations, to restore the context for the call…

def dispatch():
    for fn in [foo, bar, baz]:
        if fn in messages:
            for context in (order_contexts(contexts[fn])):  # contexts are ordered here
                set_context(context)  # context is applied here
                fn()
            messages.remove(fn)  # these calls are deterministically ordered
            contexts[fn][:] = []
    for msg in messages:
        msg()  # these calls are non-deterministically ordered
    messages.clear()
```

So, it's a little more complicated, but it ensures that you can completely control the sequence of execution.

Still, it feels a bit clunky. Here's another approach, that I think I favor.

Approach: Hard-Coded Categories of Delivery

```
contexts = {foo: [], bar: [], baz: []}

def register(fn, context):
    contexts[fn].append(context)

def dispatch():
    for fn in contexts:  # relies on Python ordering dictionary keys, true since Python 3.5 (formally since 3.7)
        for context in contexts[fn]:
            set_context(context)
            fn()
        contexts[fn][:] = []
```

Essentially, by this approach, the message broadcast categories are hard-coded, always execute sequentially, and the only information attached to them is the context. Another way to write this, that might be clearer, requiring only a small amount more of code, would be:

```
foo_contexts = []
bar_contexts = []
baz_contexts = []

def dispatch():
    while foo_contexts:
        set_context(foo_contexts.pop(0))
        foo()
    while bar_contexts:
        set_context(bar_contexts.pop(0))
        bar()
    while baz_contexts:
        set_context(baz_contexts.pop(0))
        baz()
```

(PS: There's a problem with this code – the contexts themselves are not ordered before iteration. Correcting that will render this pretty code decidedly less pretty.)

There is a quadruple redundancy here – "`foo_contexts = []`", "`while foo_contexts`", "`foo_contexts.pop`", and "`foo()`", but that doesn't seem *so* bad to me, when this unrolling provides such crystal-clear clarity. And I think there is value in being able to adjust the specific context-setting function, without much complication.

Then again, perhaps it should be:

```
bus_functions = {foo: {FN: foo, CONTEXT: set_foo_context, ORDER: order_foo_contexts, ENROLLED: []},
                 bar: {FN: bar, CONTEXT: set_bar_context, ORDER: order_bar_contexts, ENROLLED: []},
                 baz: {FN: baz, CONTEXT: set_baz_context, ORDER: order_baz_contexts, ENROLLED: []}}

def set_foo_context(context):
    …  # configures global state for this specific context

def order_foo_contexts(contexts):
    return …  # returns a well ordered sequencing of the incoming contexts

…  # other context-setting functions, and ordering of contexts

def register(fn, context):
    bus_functions[fn][ENROLLED].append(context)

def dispatch():
    for D in bus_functions.values():
        for context in D[ORDER](D[ENROLLED]):
            D[CONTEXT](context)  # establish the specific context
            D[FN]()  # call the unary function
```

One advantage to this method, is that the `bus_functions` table can be manually adjusted by extensions. That said, *this is also its drawback*, because there is the risk that two extension authors will attempt to manipulate the bus_functions table in incompatible ways. However, there's a fundamental problem in that if you make something extensible from a base, in ways that are unpredictable, then the unpredictable cannot be ordered in a way that takes into account *all* unpredictable additions. What I mean is that you can only account for what you already have cognitive closure around. Once you allow strangers to walk in, and the strangers don't necessarily know one another, then there is no way to account for that. You could have people pass in a "priority" or a "sequence number" with their new messages, but you will have no guarantee (without some further strategizing) that they will coordinate with one another to make a sensible sequencing scheme between them. At some point, somebody has to step in, and take into account all of the sequencing needs, and then perform another enclosing scheme, to make sure that everything works right. But from the standpoint of not knowing how it will all unfold, what people will need, what people will attempt, you cannot exercise control to make a sequencing scheme that will work for everybody.

Regardless of the specific method used, these are all good example of hard-coding the categories of delivery.

I think it's a good strategy.