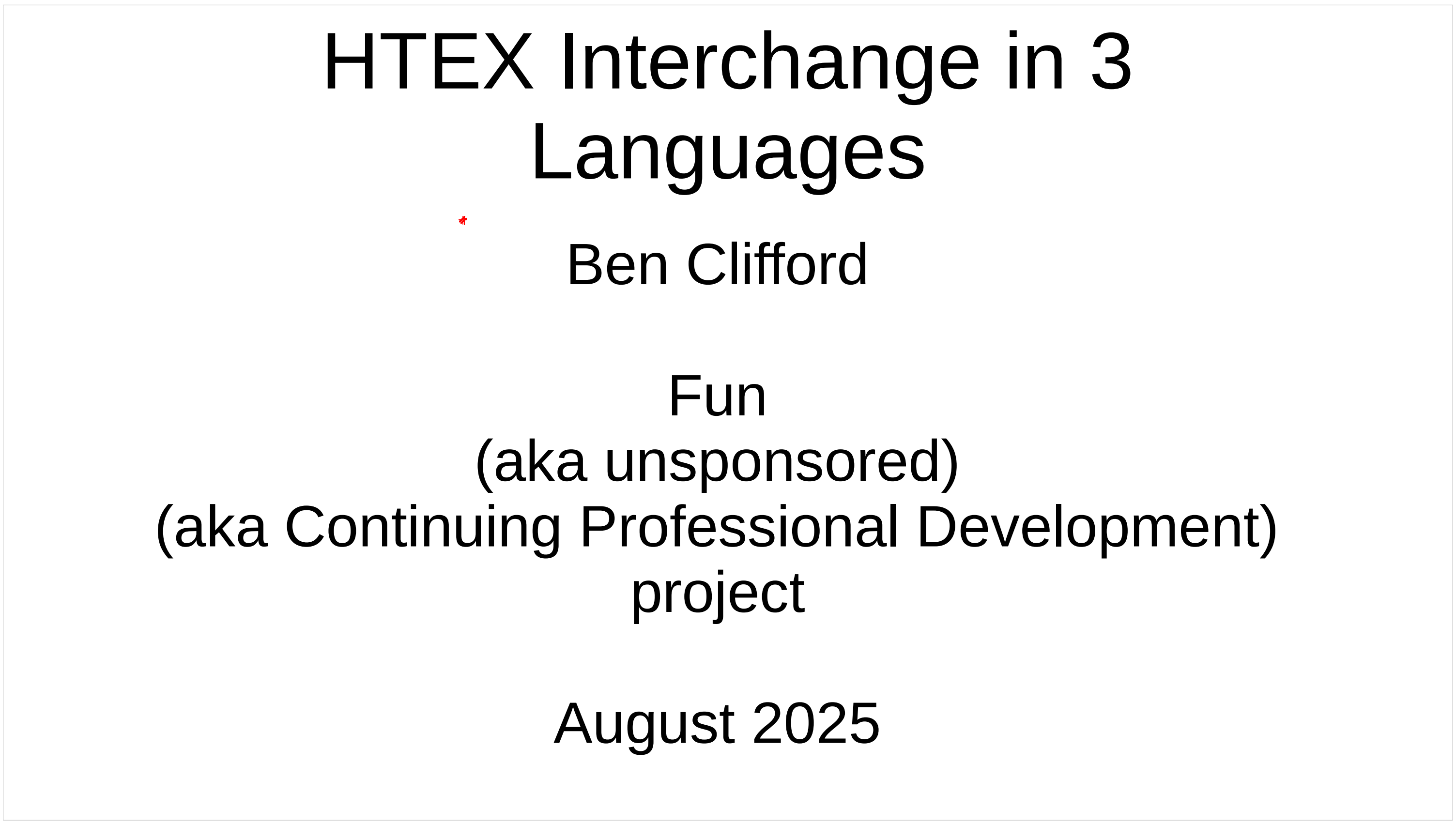


HTEX Interchange in 3 Languages

Ben Clifford

Fun
(aka unsponsored)
(aka Continuing Professional Development)
project

August 2025

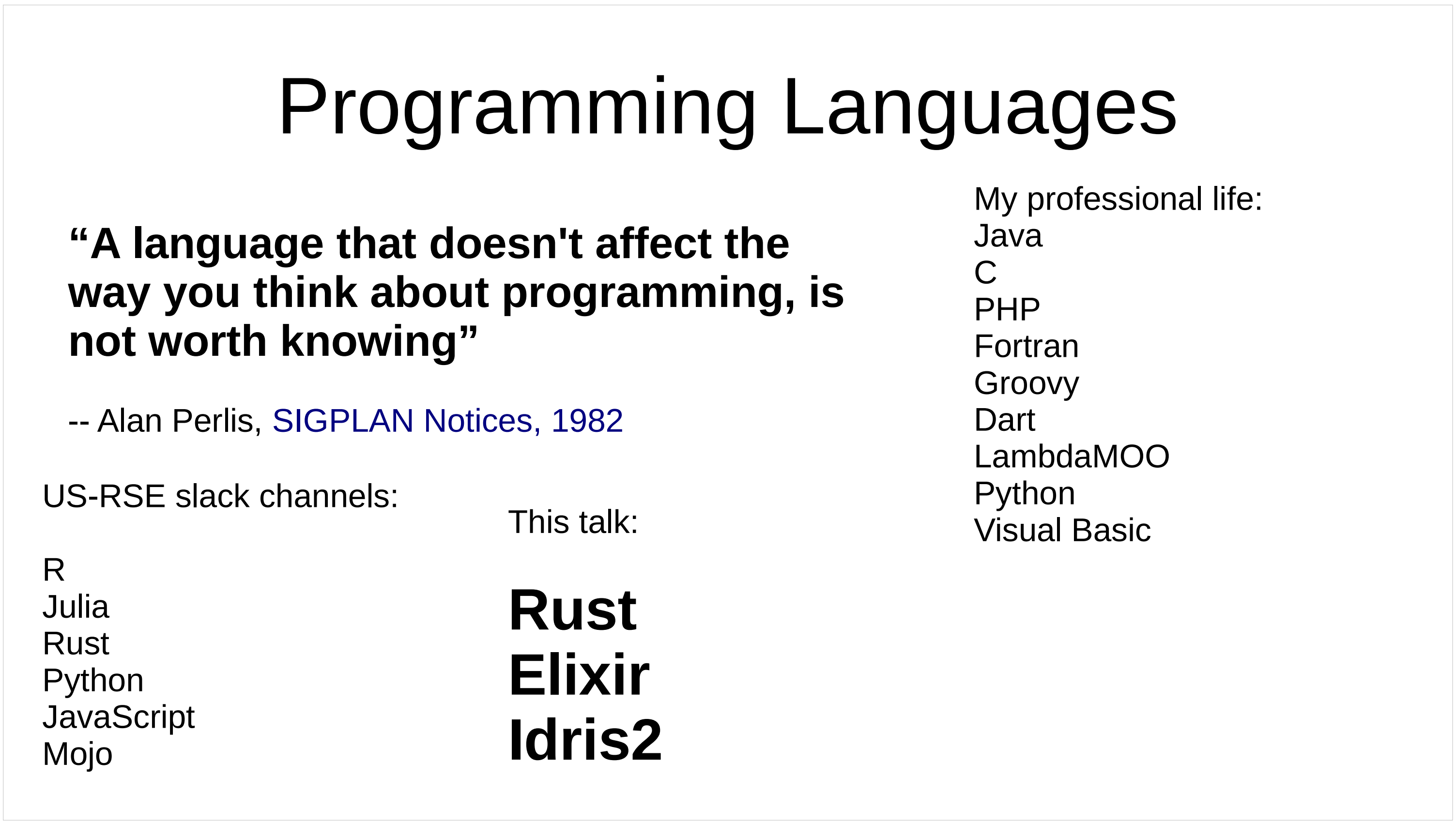
structure:
21 slides

so 3 languages and 1 overview

5 slides per language
and
6 slides for overview

OR

6 slides per language
and
3 slides for overview (introduce more stuff as-we-go-along)



Programming Languages

“A language that doesn't affect the way you think about programming, is not worth knowing”

-- Alan Perlis, SIGPLAN Notices, 1982

My professional life:
Java
C
PHP
Fortran
Groovy
Dart
LambdaMOO
Python
Visual Basic

US-RSE slack channels:

R
Julia
Rust
Python
JavaScript
Mojo

This talk:

Rust
Elixir
Idris2

I’m going to talk about work I’ve done with different languages and some of the fun I’ve had with Parsl and some influences on production Parsl

This is a talk of me screwing around, not me being paid to do something.

interested in other/all languages

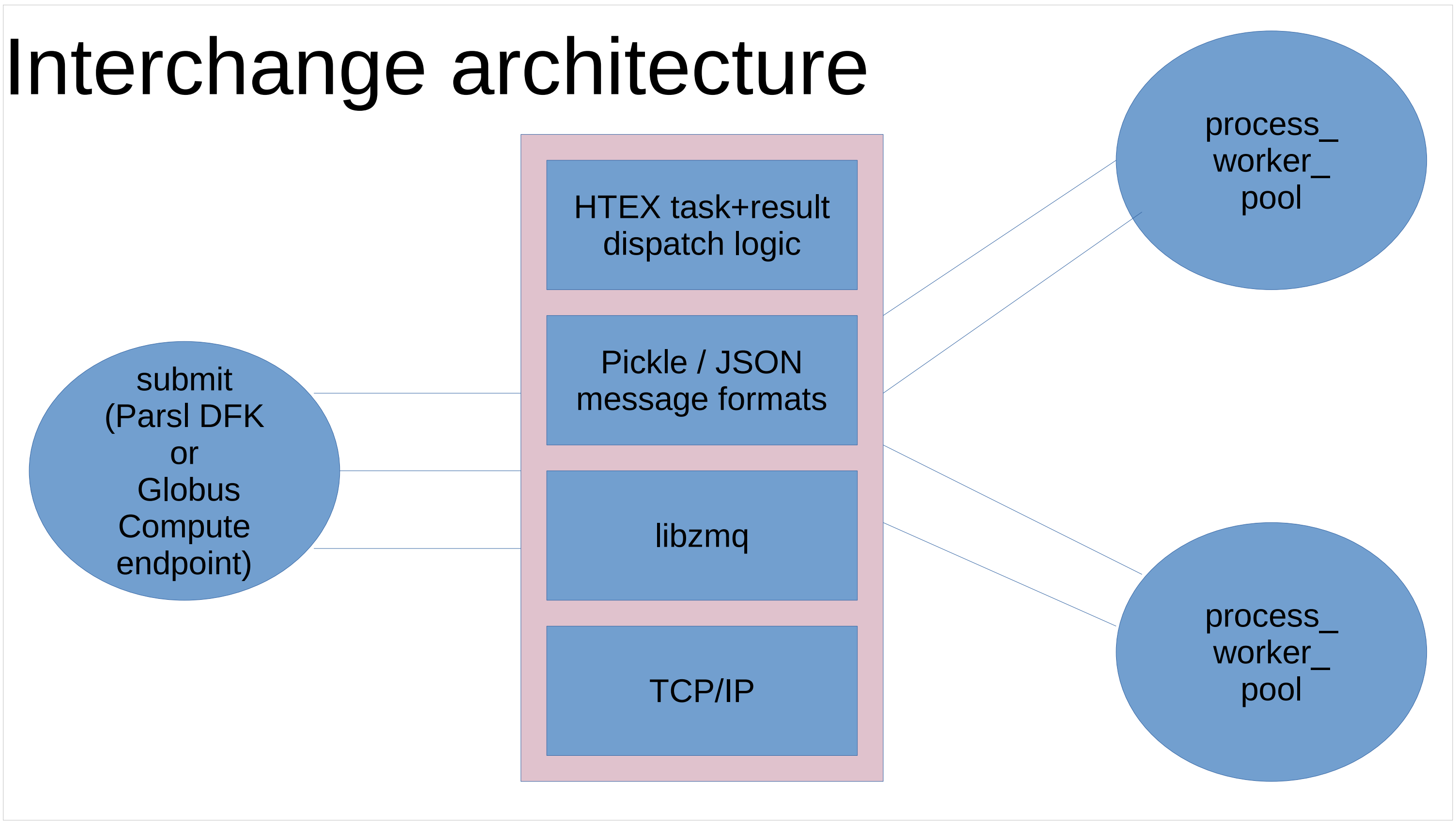
why care about different languages? they’re all turing-complete / turing-equivalent: if you can write a program in one, you can write it in any other - in that turing completeness sense

more interesting is human expressivity: can i write things easily / readably / maintainably?
and not to specifically to reimplement “the same thing” in a different language, but to use it as an excuse for thought.

lots of stuff in Parsl/GC stack is not in Python, despite GC/Parsl being Python-oriented products:
libzmq is C
rabbitmq is Erlang
? numerics in fortran in many apps?

motivated by a throwaway comment I made about thinking about this: regard the component at the other end of a connection or message path as written in an arbitrary other language - which happens to be quite like the impl we are working on, but not quite the same - because its a different Python or a different Parsl or both. So dont’ let that similarity fool you into complacency. not because I think we should be rewriting pieces of production Parsl in other languages.
so “you should be able to rewrite the htex interchange in rust”

... but then? what would this really look like?
to rewrite the interchange in rust?



Interchange architecture

submit (Parsl DFK or Globus Compute endpoint)
HTEX task+result dispatch logic
Pickle / JSON message formats
libzmq
TCP/IP
process_ worker_ pool
process_ worker_ pool

interchange architecture as far as this talk is concerned

[interchange is a separate unix process. i hacked the code to use arbitrary command line, then Reid did it properly in PR #NNNN]

task + worker pool / dispatch - interchange logic
Pickle/JSON message format - built into Python
ZMQ - libZMQ
TCP/IP - OS

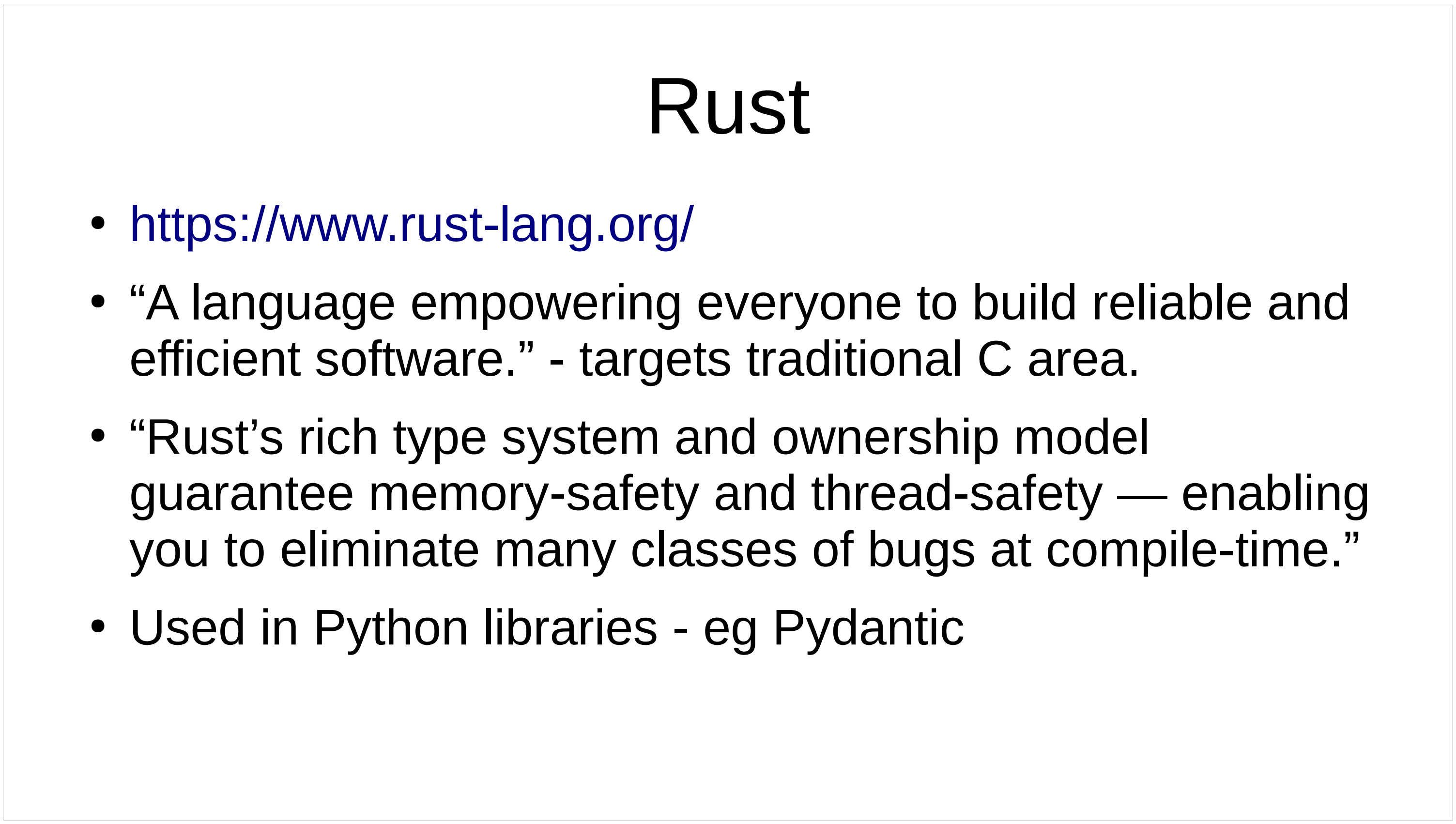
[JSON was removed in PR #3871 by Kevin]
pickle is a whole adventure in itself - see PyCon LT 2024 https://www.youtube.com/watch?v=Qn-1hGLrzR0 The Ghosts of Distant Objects

submit side (dataflow kernel or endpoint)

worker side
[worker side - 2 connections merged into 1 connection by Kevin in PR #NNNN]

history: interchange already well separated as a separate unix process, communicating via ZMQ.
this comes from early work on GC-like behaviour where the interchange might have been running remotely - talk to Yadu about that. that particular line of development was abandoned but the concepts sort of turned into the GC endpoint.
code example: start_local_interchange_process has the word “local” there because of this distinction.

(mention work to remove multiprocessing fork? and make things either use multiprocessing spawn or explicit processes? not sure if thats relevant to this talk?)



Rust

- https://www.rust-lang.org/
- “A language empowering everyone to build reliable and efficient software.” - targets traditional C area.
- “Rust’s rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.”
- Used in Python libraries - eg Pydantic

Single threaded (maybe real interchange is single threaded now? PR NNNN. If so, this drove some of that thinking / feasibility study). Parsl has been a masterwork of race conditions that I've been spending the last 8 years untangling: concurrency is hard to reason about and to a first approximation should be banned. Which is superficially a funny thing to say when parsl is (from one view a concurrency library) but at the same time, that's also the argument of the task based model, for users: concurrency is hard, here's a simpler safer concurrency model.

ZMQ poll based. Explain poll as blocking call, not a "sit in loop hard looping and checking many conditions every 1ms" - Unix select call introduced in... what year and Unix version?

ZMQ monitoring was really helpful - ZMQ is quite async, to the extent that it doesnt tell you when a connection has been opened at the TCP layer. Which can then disguise some errors. It has a monitoring layer which reports progress and I gatewayed that to log messages. I recommend to the remaing core developers to implement this in mainline parsl . Show an example of what messages I got - suggestion to use this in mainline Parsl for debugging. is there issue #NNNN? if not, open one.

polling not tight looping
(here and in idris2?)
Poll based stuff needs all your pollables to share a pollable representation. In ZMQ poll, can us ZMQ sockets and Unix file descriptors. But in python impl, for example, can't poll on both a ZMQ socket and a multiprocessing queue - which has led to some ugly busy loops in the mainstream impl of Parsl in htex and in monitoring. which uses CPU, and when delays are introduced to avoid that CPU burning, introduces latency on message processing. That can give a recommendation/pressure to choose between concurrency structures.
mentioned radios in monitoring talk earlier

Rust: Static Typing

- “eliminate many classes of bugs at compile-time”

```
struct ManagerInfo {
    alive: bool,
    last_heard: Instant
}
```

Python Gradual Typing, BOB Konferenz 2022
https://bobkonf.de/2022/clifford.html

PR #2392, August 2022

```
class TaskRecord(TypedDict, total=False):
    """This stores most information about a Parsl task"""

    dfk: dflow.DataFlowKernel
    func_name: str
    status: States
    depends: List[Future]
    app_fu: AppFuture
    exec_fu: Optional[Future]
    ...
```

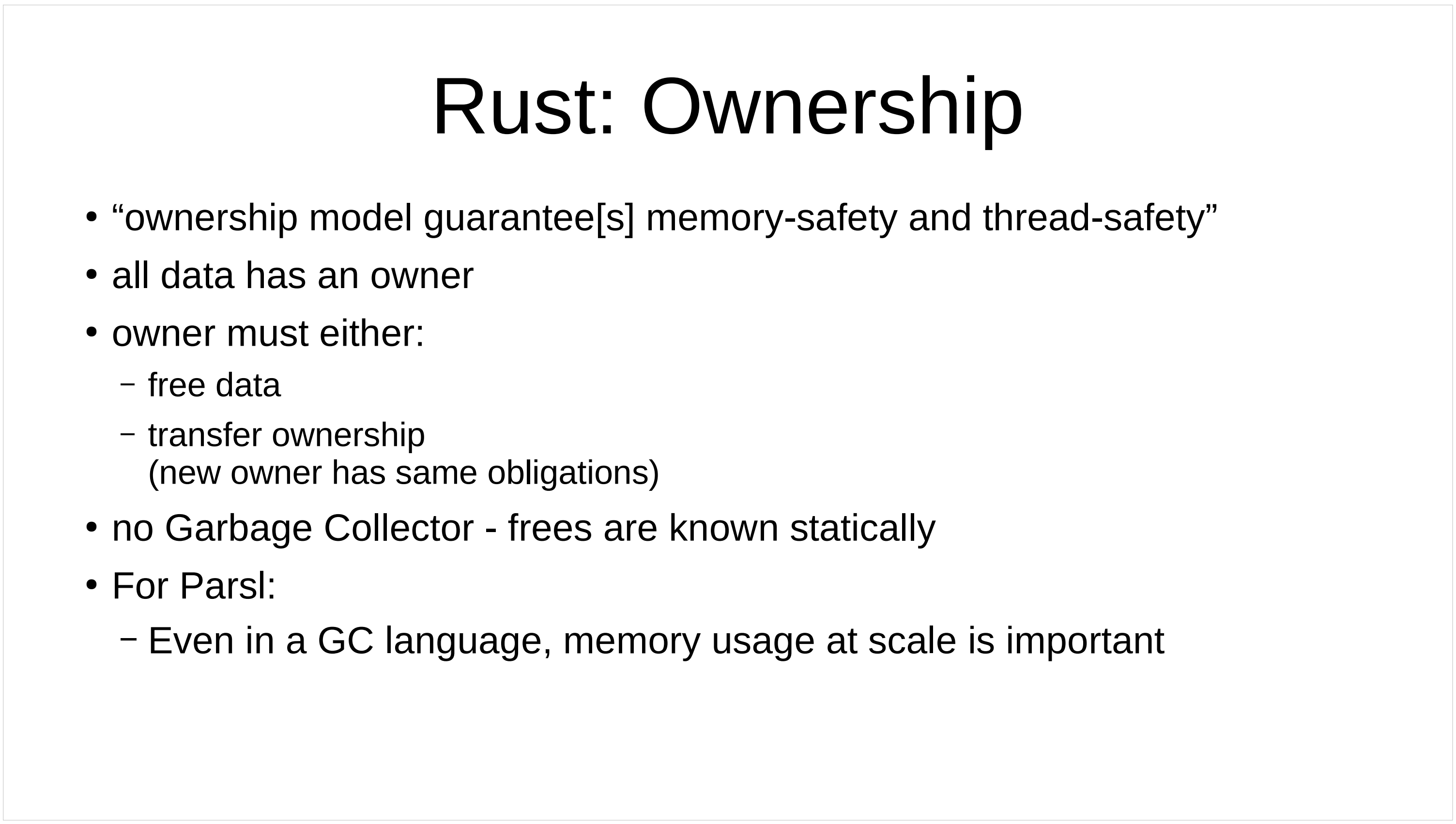
“Eliminate bugs at compile-time” - what does that mean for Parsl?
In Parsl case: we don’t really case about “compiling”, but what is relevant is that it is before the code reaches `master` branch - at “developer time”.

Well this is something we’ve been pushing into Python Parsl, and were even explicitly funded for, for several years!

Where we originally used a regular Python dictionary for task information, we started using a typed dictionary.

Code that asks func_name for it’s area (for example) is “obviously” wrong: declaring it as a string lets tooling also see that obvious mistake.

So this isn’t something new from this current project, but Rust leans into static typing much harder.



Rust: Ownership

- “ownership model guarantee[s] memory-safety and thread-safety”
- all data has an owner
- owner must either:
  - free data
  - transfer ownership
    (new owner has same obligations)
- no Garbage Collector - frees are known statically
- For Parsl:
  - Even in a GC language, memory usage at scale is important

“ownership” is probably the most famous piece of Rust.

Using the static type system to help with memory safety and thread safety. Track statically (at developer time) which code is allowed to modify “objects”, and which threads are allowed to access them.

From a Parsl perspective, we’ve had various bugs from objects passing between threads that should not have - mostly in the context of the ZMQ library. Ownership model is trying to stop that statically.

This is something to think about in every language - even when it isn’t statically enforced. Just because an object is accessible everywhere doesn’t mean it is right to access it anywhere.

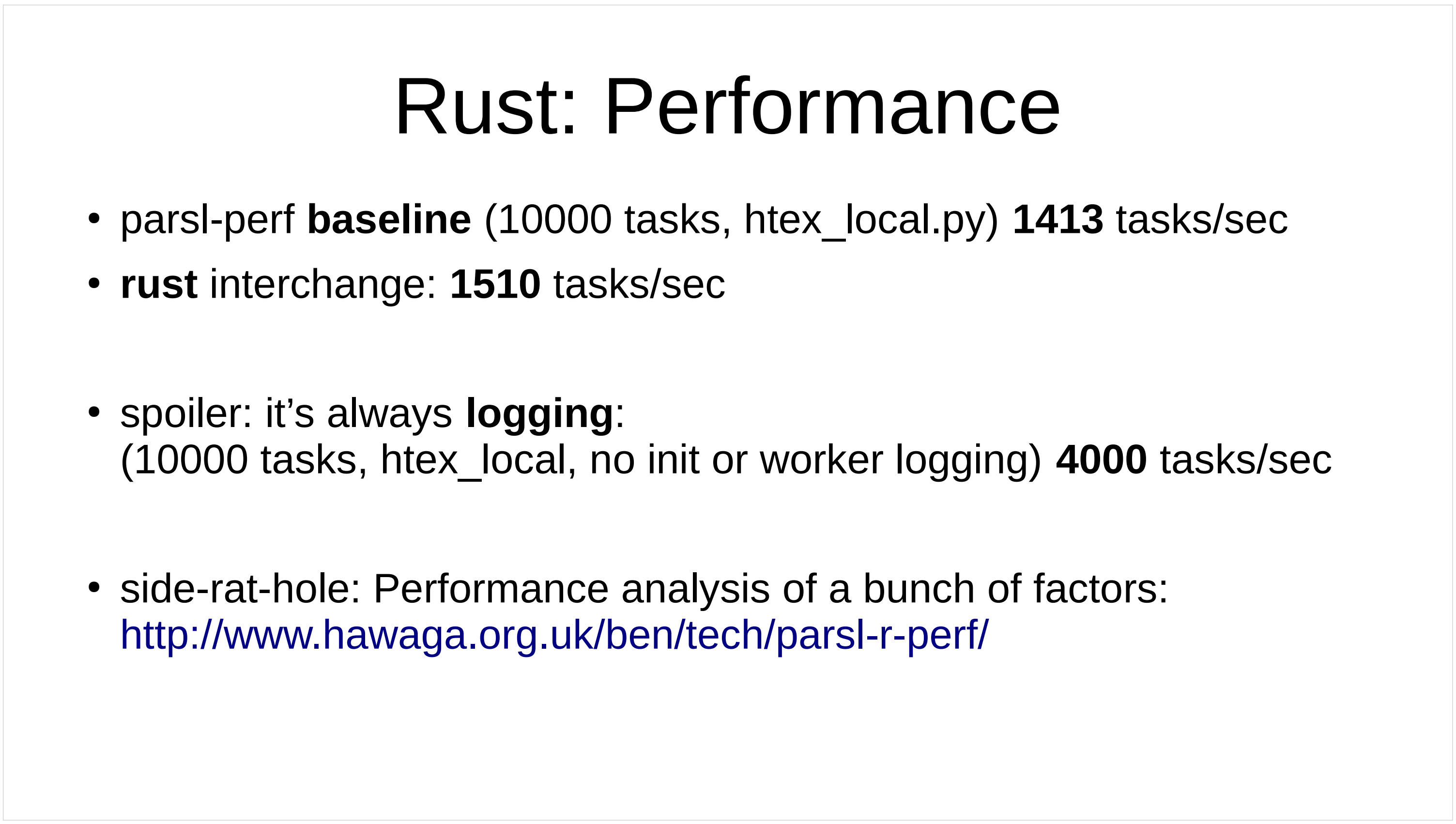
“ownership” model is sometihng you have to think about in other places - even with a garbage collector -- “memory leaks” in a garbage collector model look like keeping. not just about freeing memory, also about what is safe to do at what time.

free memory explicitly-ish (statically?) rather than garbage collector - from the haskell world, see https://pusher.com/blog/latency-working-set-ghc-gc-pick-two/ about GC vs message routers especially. although python reference counting maybe makes that not so bad?

ownership not just about freeing memory - but also about who is allowed to do thigns to objects

Python does reference counting and garbage collection at runtime. Rust tries to do this statically. Which means you have to reason about who owns objects so that you know when an object is finished with. That can be awkward to get strictly right. But in the Parsl codebase in Python, it still makes sense to think sometimes about who owns an object informally. (Example? For example who is allowed to make calls on a particular object? Rather than chaotically through the source code? Or perhaps that having this kind of formality in general is useful - not so much just ownership)

ownership also thinks about sending things between threads - most stuff is ok. but in Parsl, for example, we’ve had various bugs that come from libzmq structures that cannot be (easily) passed between threads.



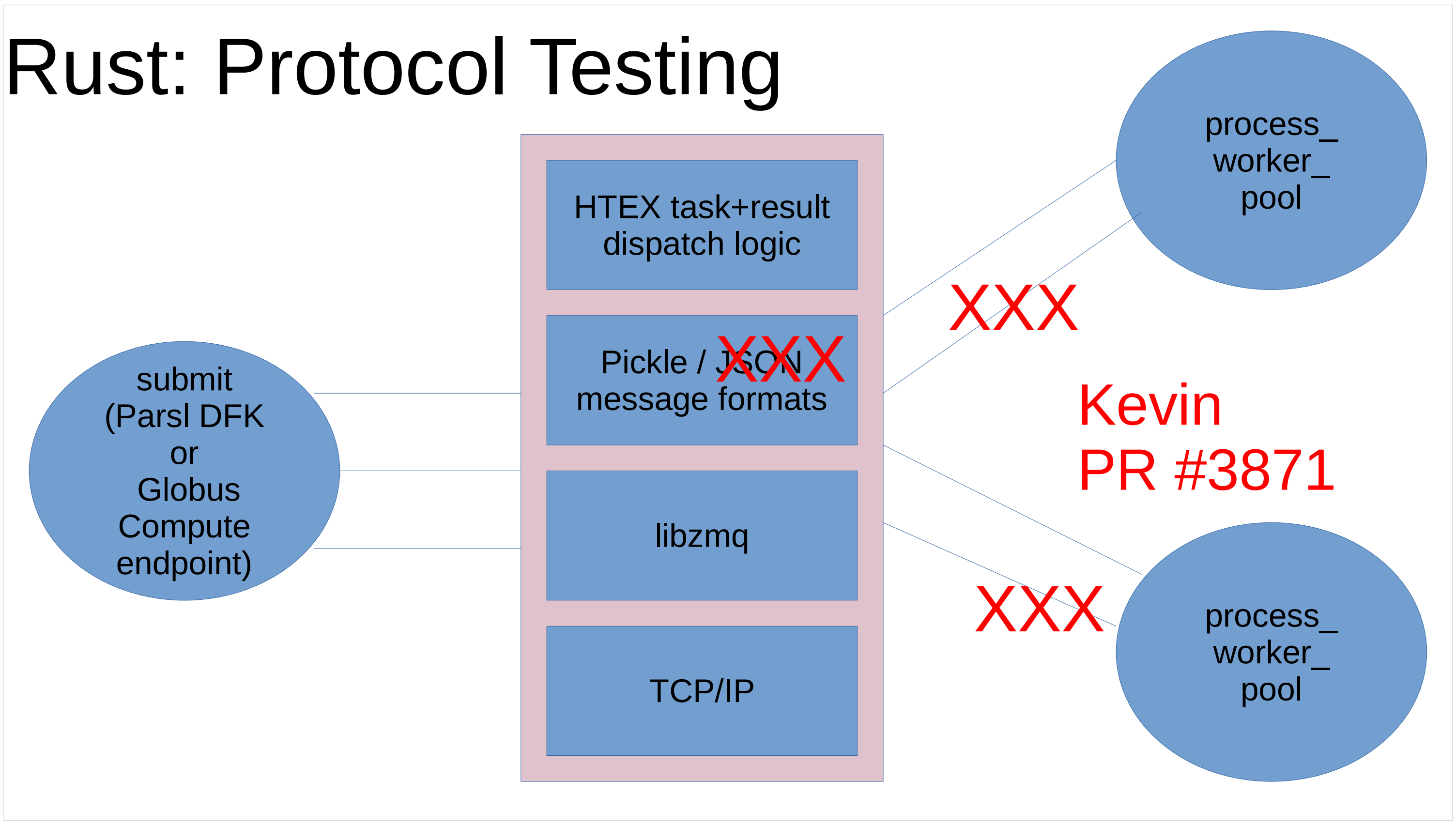
Rust: Performance

- parsl-perf **baseline** (10000 tasks, htex_local.py) **1413** tasks/sec
- **rust** interchange: **1510** tasks/sec
- spoiler: it’s always **logging**:
  (10000 tasks, htex_local, no init or worker logging) **4000** tasks/sec
- side-rat-hole: Performance analysis of a bunch of factors:
  http://www.hawaga.org.uk/ben/tech/parsl-r-perf/

I assumed (without justification) that the interchange was the rate limiting component.
And that a compiled Rust interchange would therefore be faster than Python and make task throughput much faster.

WRONG!

Spoiler: it’s always logging.



Rust: Protocol Testing

process_ worker_ pool
HTEX task+result dispatch logic
Pickle / JSON message formats
XXX
XXX
submit (Parsl DFK or Globus Compute endpoint)
Kevin PR #3871
libzmq
XXX
process_ worker_ pool
TCP/IP

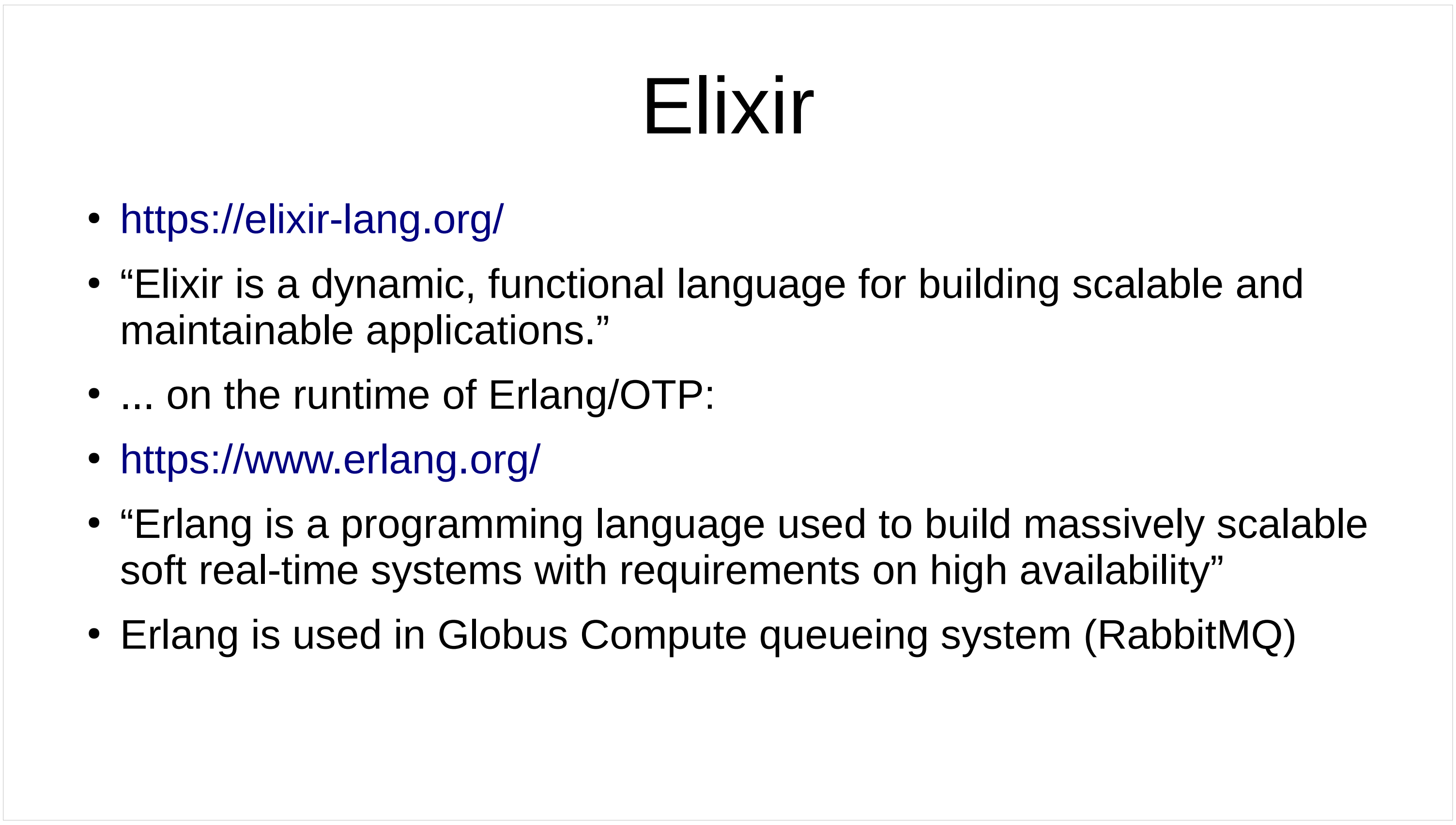
Reimplementing the protocols made me think about them more and open GitHub issues for a bunch of protocol jumble

Fixed by Kevin:
JSON in addition to Pickle - use one framing protocol
Two TCP connections per worker instead of one - historical reasons.

But:

i) I’m only implementing against tests, and we don’t test much protocol, so I got away with not reimplementing a lot of the interchange.

ii) some pressure for worker pool <=> submit side work from different Parsl versions... more protocol focus in testing.



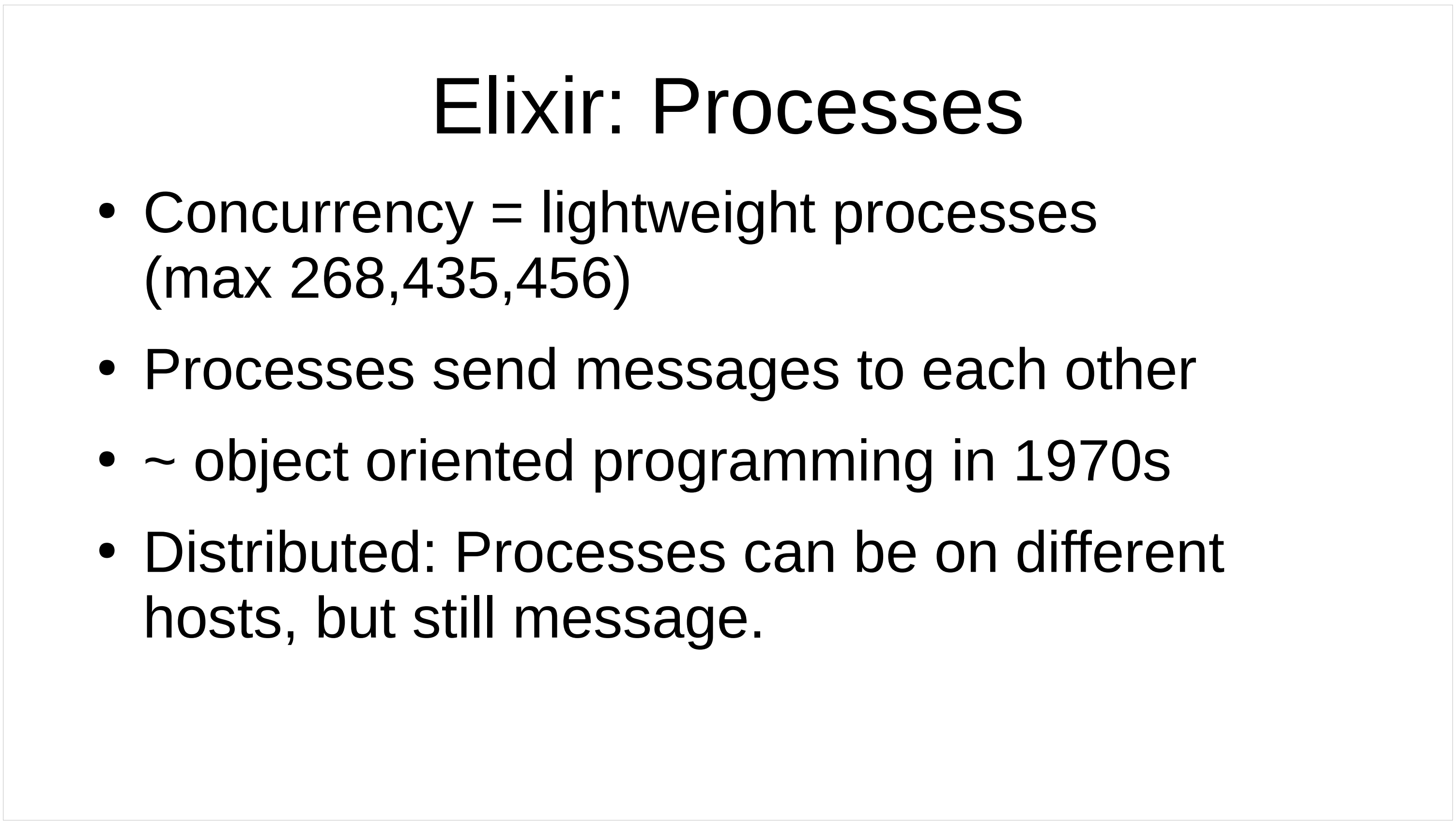
Elixir

- https://elixir-lang.org/
- “Elixir is a dynamic, functional language for building scalable and maintainable applications.”
- ... on the runtime of Erlang/OTP:
- https://www.erlang.org/
- “Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability”
- Erlang is used in Globus Compute queueing system (RabbitMQ)

Elixir
Erlang runtime, various languages on the front: also LFE and Gleam. Ruby inspired syntax.

erlang - telecoms style applications - headline?
lots of my feelings here come from erlang, not from elixir

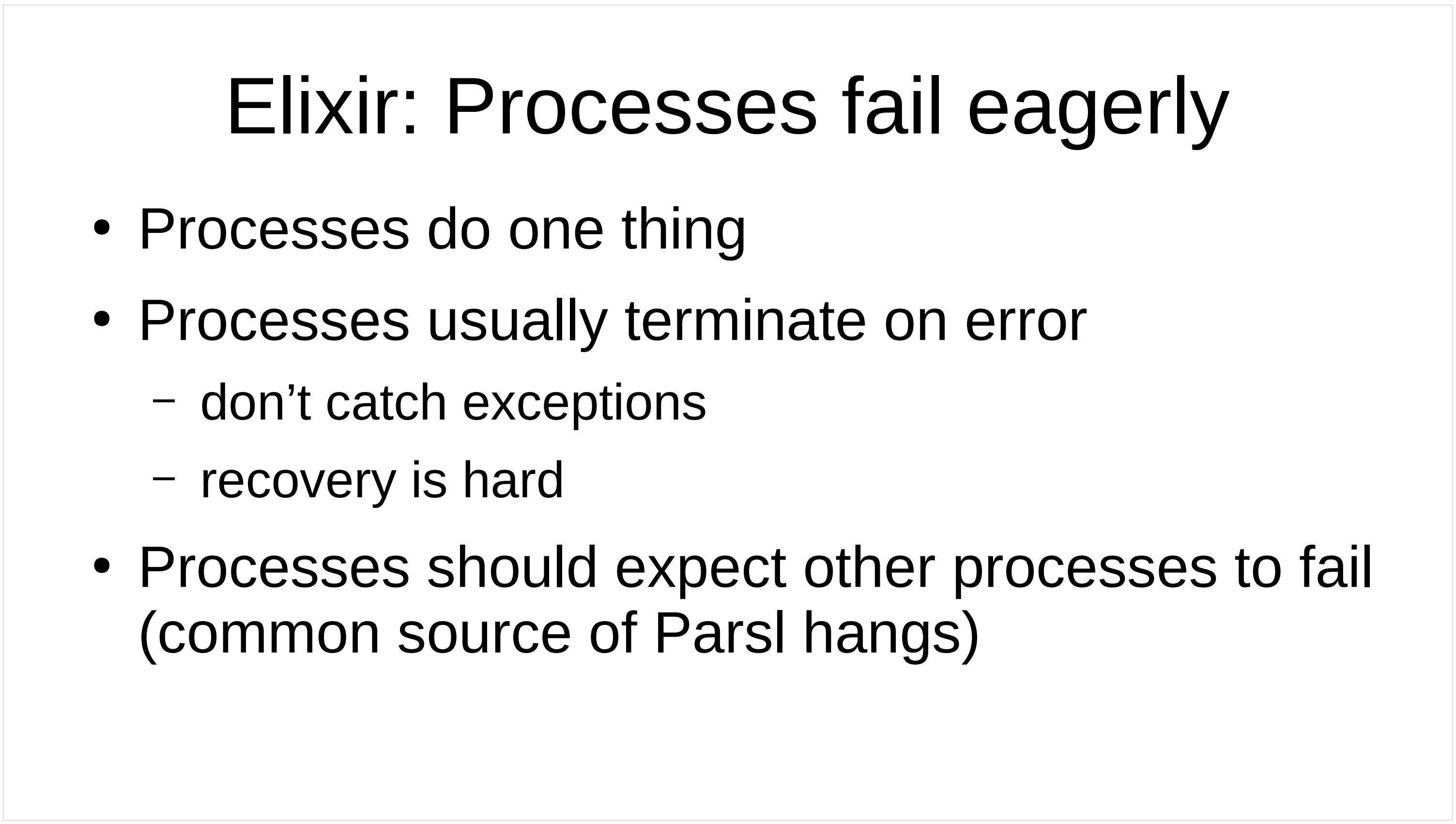
Libraries also Erlang. If you are using htex via GC, then your tasks are already passing through an erlang layer - rabbitmq(?)
Contrary to rust impl, this is *many* threaded / what the runtime calls a process, but it is not a unix-level process
Erlang has principle that process should die easily and other stuff should deal with that. One thing parsl has suffered from architecturally is threads and processes dying but then the rest of the pile not detecting that. I've pushed a bit on detecting that in Parsl. But for example it's hard to poll on a multiprocessing queue and also on a process exiting at the same time.
Python threads look like OS threads (which in Linux is very similar to being a Linux level process) - in elixir/erlang, that isn't the case.
Also a principle I've become fond of: you should be eager to raise exceptions but don't try to catch and handle them because you often can't/need to reason very hard about it
Thats their approach to concurrency. Messaging. Processes die easily. Tied processes - if one dies so does a connected process (I have pushed that a bit into Parsl and recommend push more). other shared structures built on that.
htex command client: shares a non-thread-safe structure between threads. can lead to weird behaviour - rare enough that other devs won’t fix it. frequent enough I encounter it with my users. erlang runtime puts external interactions on its own process/thread and uses internal messaging to talk to that [beam] process from other [beam] processes more safely. [suggestion - use a cross-thread RPC internally to talk to command client? for example, using ZMQ local connections? it’s more bureaucracy but it’s thread safer?]
this doesn’t eliminiate race conditions - but its a model that can be easier to reason about than Python style shared objects.
one informal effect of moving task launches to own thread (although that wasn’t the core intention - was more about stack sizes) but has a more message based model now with queue.
(queues / message queue is the core concurrency primitive in beam) -- PR #NNNN
these BEAM level processes can run across cores and across machines. which means theres a scaling possibility for having a multi-host megainterchange... if the process model was suitably architected. but what is that architecture?
c.f. python asyncio/coroutines: much lighter-weight than Python OS threads (although less capable for multiple CPUs). and yet again a different “poll language”.
issue #3761 - interchange should notice when submit process goes away (and vice-versa)
erlang/beam built for serialization - python object model is not (and I have had a lot of fun there ...)
exit handling -- idea that linked processes: if one process dies, the other process should explicity care about it. a little bit manifested in htex heartbeats, but not everywhere - cause of hangs.



Elixir: Processes

- Concurrency = lightweight processes (max 268,435,456)
- Processes send messages to each other
- ~ object oriented programming in 1970s
- Distributed: Processes can be on different hosts, but still message.

processes do one thing. there can be many processes.
in my impl i made each task be handled by its own process. yes thats deliberate overkill.
yes that’s a lot of processes.
and yes it scales much less well than the python impl.
(which then leads to backpressure point)
run more threads! (run *many* threads)
parsl monitoring for example had one thread that was both a zmq radio receiver and a udp radio receiver. these were not done in a select/poll way, but making a blocking call to wait, for each. and alternating between the two, using cpu rapidly polling one then the other.
two thread model -- what monitoring looks like now -- we can make a blocking call, and sit there for much longer. we don’t have to alternate between checking two things.
but then, because these two threads are separate, it’s much easier to see that actually we often don’t need the behaviours of one or both, and not start the thread at all -- what looks like “adding threads” actually resulted in removing load.
“message mover style”
- lots of bits of parsl move messages around between queue-like structures.
- and have sometimes done other stuff on the way
- which conflates ownership of responsibility
- push towards message mover threads *only* doing moving. not doing other stuff “by the way”.
(db manager shutdown race, example)
(i think some other examples in htex?)
this is part of the “processes do one thing” model
processes communicate by messaging - no shared state. represent shared state by making a state-containing process that can be messaged by other processes. - if you’re used to MPI programming (something Parsl in part tries to help you avoid) - you’ll have some feel for message passing style.



Elixir: Processes fail eagerly

- Processes do one thing
- Processes usually terminate on error
  - don’t catch exceptions
  - recovery is hard
- Processes should expect other processes to fail (common source of Parsl hangs)

code should be expecting processes to fail, and to do “something” when that happens.

this can be very simple: compare: if the interchange process exits: parsl stack does all kinds of stuff: waiting for the interchange to reappear and reconnect. that will never happen.

* should the interchange be restarted?
* should the htex instance become invalid/all tasks on htex fail? (in the same way that manager or worker lost happens ... InterchangeLost?) -- that’s my preferred behaviour. (I made a hacky implementation in my project)
* what happens if submit process goes away without telling the interchange to exit? - i think there’s an open bug now about interchange not noticing? issue #NNNN
- in some sense the interchange and the submit process are paired and one failing needs to lead to the other failing/ending too - or at least clearly seeing the error.

code principle: raise exceptions eagerly. don’t try to be Very Clever when handling them. usually, just fail, unless you can really reason about why you should do something else. (eg. parsl task retries)

common cause of Parsl hangs in the past: a process/thread dying doesn’t cause an exception anywhere else, in default Python model.
some parts have heartbeats which perform a similar role. but only in a few limited places.

Elixir: Back-pressure

- My interchange implementation was very inefficient and unscalable
- One task = one process
- Neither theory nor mechanism on how to slow down a workflow
- Billion Task Parsl: https://www.youtube.com/watch?v=5brIeAvZG1c

backpressure: noticed this especially with my elixir impl because it scales very badly.

explain what is backpressure: too many tasks = stuff crashes, rather than slows down. how does the interchange push back into the submit side “don’t send me any more tasks, I am full”. full means some resource exhausted. eg submit host RAM!

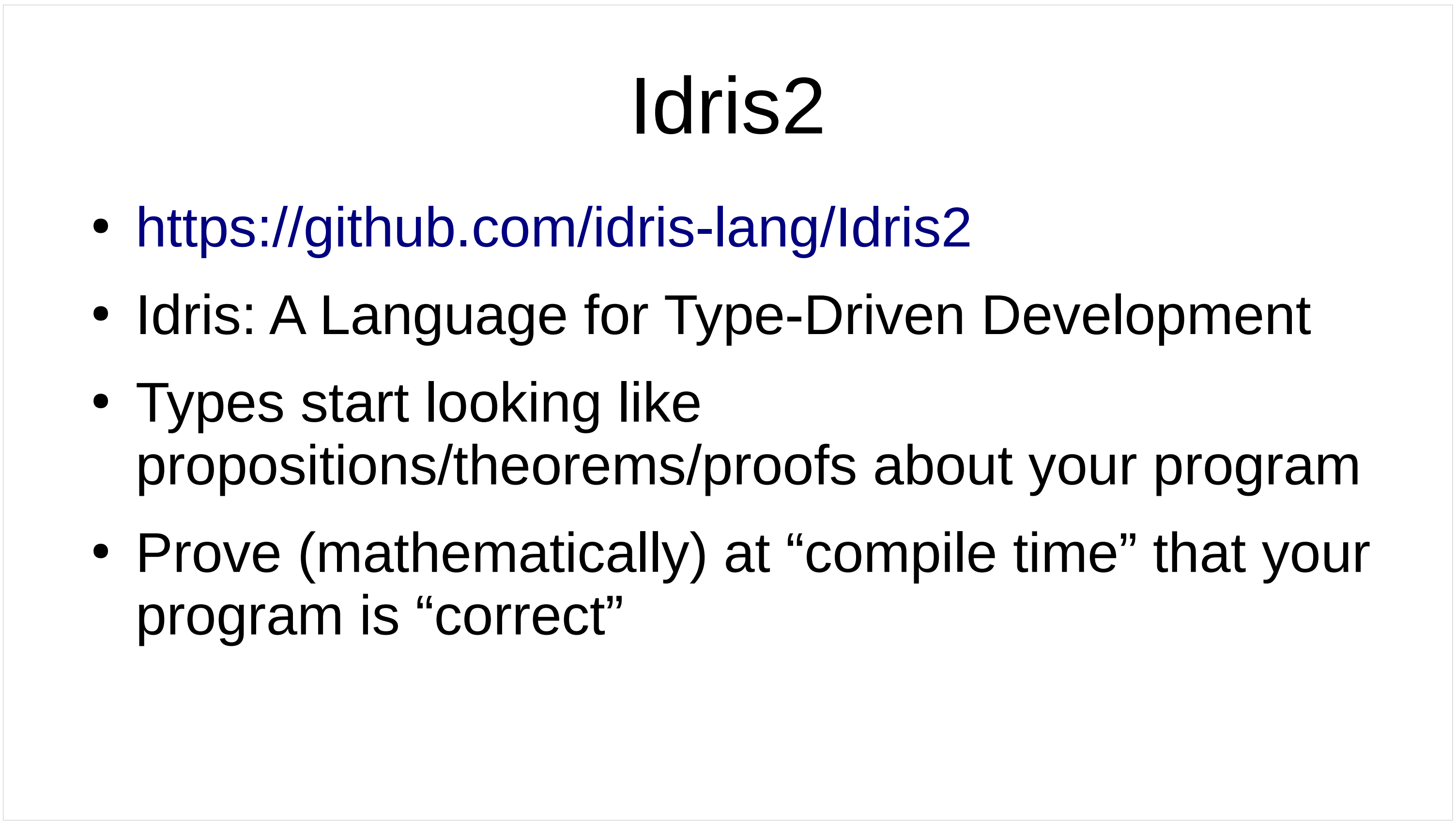
but this is a problem everyone has with “many” task workflows. (for some definition of many)
- “just” don’t submit too many tasks to Parsl - which makes your workflow code possibly less beautiful.

there aren’t any good backpressure mechanisms (in the workflow language) to say “please stop submitting more tasks” - also in the DFK, not just at the execution layer.

thoughts about queue length as an important thing.

Elixir: replace entire worker pool mechanism?

- Rather than replace only interchange...
- This is already a distributed system
- Replace worker pools + worker protocols
- Previous prototypes:
  Extreme Scale Executor - used MPI
  Low Latency Executor - no error handling



Idris2

- https://github.com/idris-lang/Idris2
- Idris: A Language for Type-Driven Development
- Types start looking like propositions/theorems/proofs about your program
- Prove (mathematically) at “compile time” that your program is “correct”

Idris2
headline? purely functional dependantly and linearly typed language [its programming language research] - starting to cross over with proof langauges / theorem provers.
Runtime on top of scheme. Syntax is very much Haskell but different - which standard way to do things (and Haskell was designed to “serve as a basis for future research” - [influence box: that’s part of what leads to my thinking of Parsl being a hackable pluggable core on which people build their own particular chaos] - many language features in mainstream languages developed in the likes of Haskell
Much more experimental language. So a lot rougher to use. (eg. error message quality - something Rust people have put a lot of work into; package management is rough; and roughnesses that maybe compiler bugs or unimplemented compiler functionality - everyone here is used to Research Code) and i have slighly modified the compiler stdlib. (although I’ve also done that for my own Python installs...)

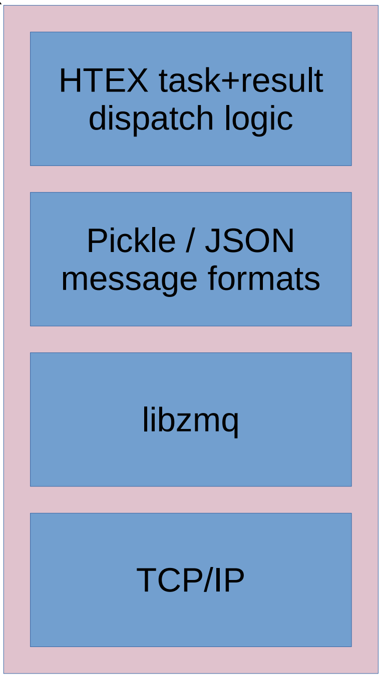
So modelling event loop as fd based.
Single threaded.
Strongly typed like rust...But even more so.
Static checking resource management - not just that we want to ensure that free is called.

I used pidfd (signals for other reasons aren’t delivered to the right process but I would like to catch signals via fd too)
Classic example of dependent types is reasoning about vectors: like a list but type safe around length.
Example of linear types is resources rather than values: eg. memory buffer

talked earlier about developer time type checking to increase code quality before it reaches users - with the kind of type checking that the idris2 research crowd lives in, that can start to look like more serious theorems about a program, rather than “this should have been an int, not a string”

Idris2: library stack

- htex dispatch logic
- but also... (because it’s a research langauge)
- my own pickle encoder/decoder:
  The Ghosts of Distant Objects (PyCon LT 2024)
- my own libzmq bindings
- my own OS interfaces

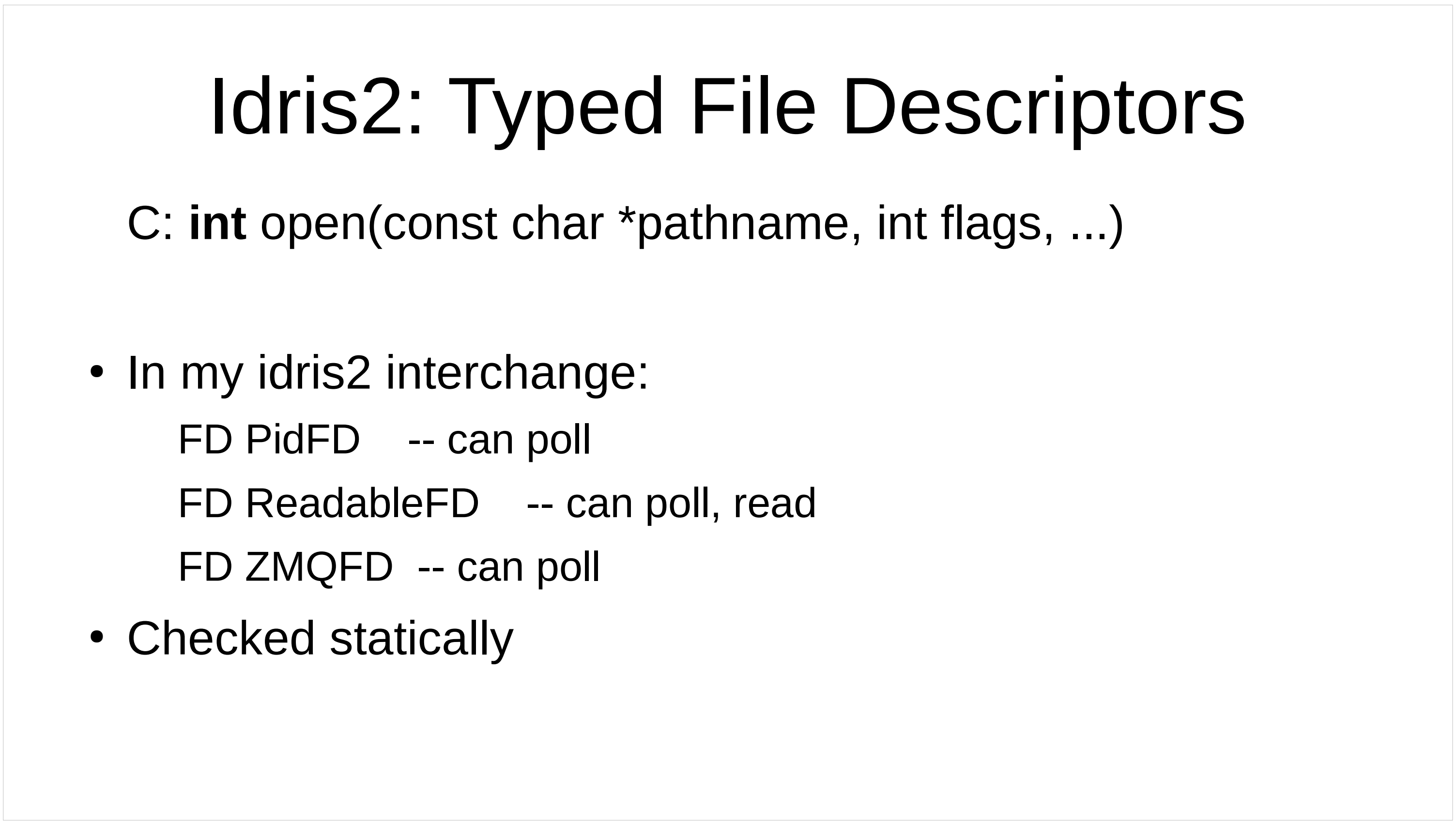

HTEX task+result dispatch logic
Pickle / JSON message formats
libzmq
TCP/IP

own zmq bindings

own pickle encoder/decoder (just enough to work for interchange, not a full impl)
- in Parsl, over the years, I’ve come to learn far too much about Pickle, so it was all in my head anyway

slide: show code sizes of all .idr and .c files
(and compare with python, rust, elixir top-level source file)

built my own lower level libraries - pickle imp, zmq C binding, unix/file descriptor interfacing, bytes, logging
- show sizes



Idris2: Typed File Descriptors

C: **int** open(const char *pathname, int flags, ...)

- In my idris2 interchange:

```
FD PidFD    -- can poll
FD ReadableFD    -- can poll, read
FD ZMQFD  -- can poll
```

- Checked statically

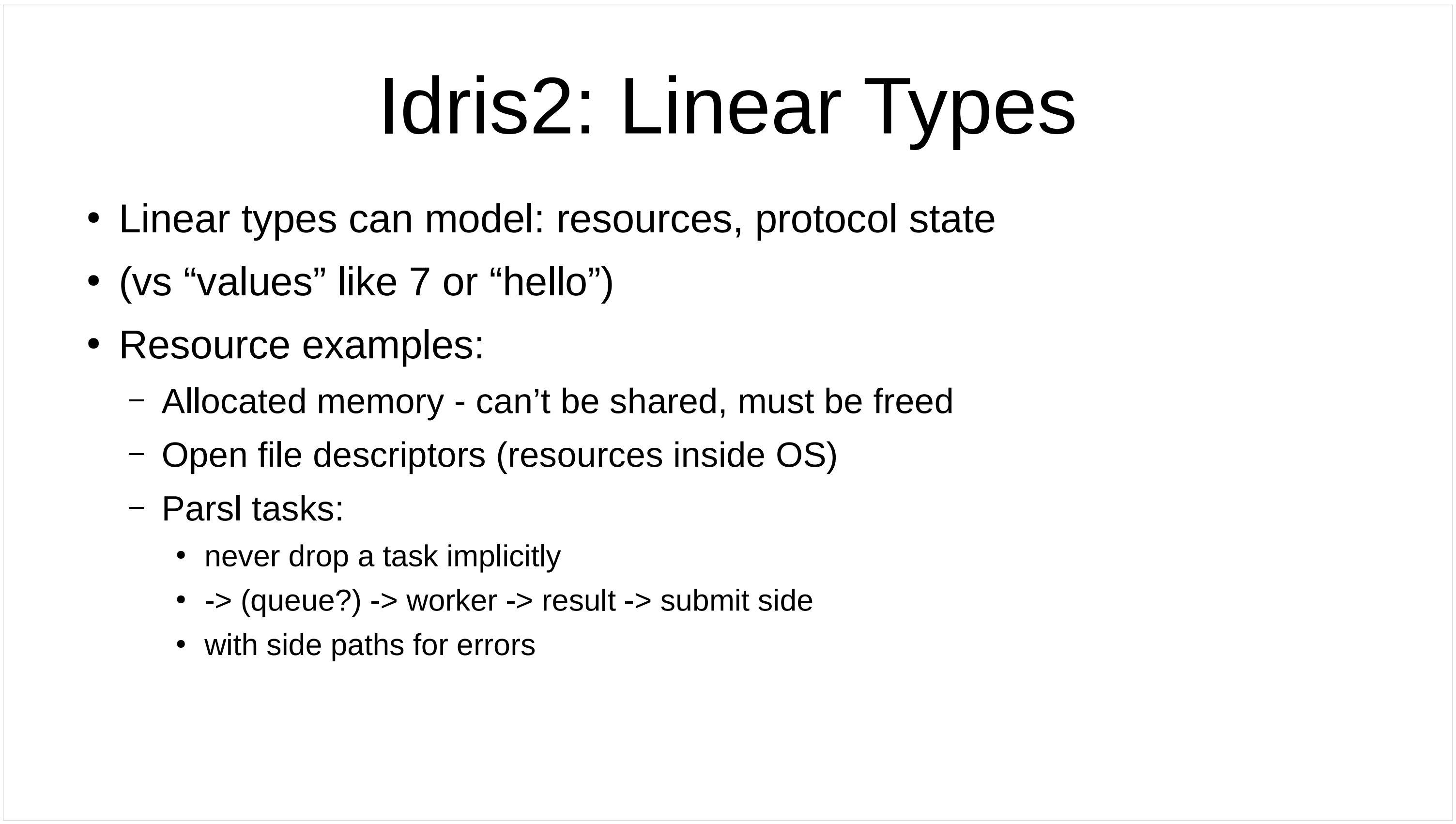
I went on another massive sidetrack learning more about kernel implementation of file descriptors and then brought some of the thoughts back to my Idris2 implementation.

Single threaded polling loop, like the rust impl, unlike the elixir impl. Using unix file descriptor polling as the polling primitive.

file descriptors are *not* files, despite the name
(or rather “everything this a file” unix phrase only works with an extremely loose definition of file

- both as an example of thinking about the unix model in the context of a type system, and how to represent that model. eg what FDs can we call poll on?
what FDs can we write log messages to? [not pidfd, for example]

phantom typing on FDs - code example
UNIX side track - not what I was planning, but got distracted having a nice look at kernel stuff which I hadn't looked at for a long timeUNIX poll based. ZMQ exposes a Unix FD for the purpose of using with unix poll. Also novel poll types (I think pidfd? Or signal FD? Or both?) - pidfd lets us poll on other processes ending. So interchange ends when the submitter ends. (Did I implement this parent-pid in mainline parsl? If so, pr #NNNN - but looping poll not poll based poll) -- pidfd for doing that Erlang style “if that other process goes away, this process should too”. Traditional parsl bugs of processes not going away.
poll, ioctl, special opens, special message formats - these are not “files”
quick snapshot of other interesting things that can be represented as fds. FD abstracts a "kernel object" not a file. (My preferred phrasing to "everything is a file" that Unix people like) eg poll for a raspberry pi IO pin change? should only list the FD types I actually use: zmq poll, stdout (for logging), pidfd
- pidfd to notice when submit side has gone away. could also use signal fd if I made the unix processes be a bit different.
Polling thoughts here really made me think about common model for polling otherwise stuff doesn't work.



Idris2: Linear Types

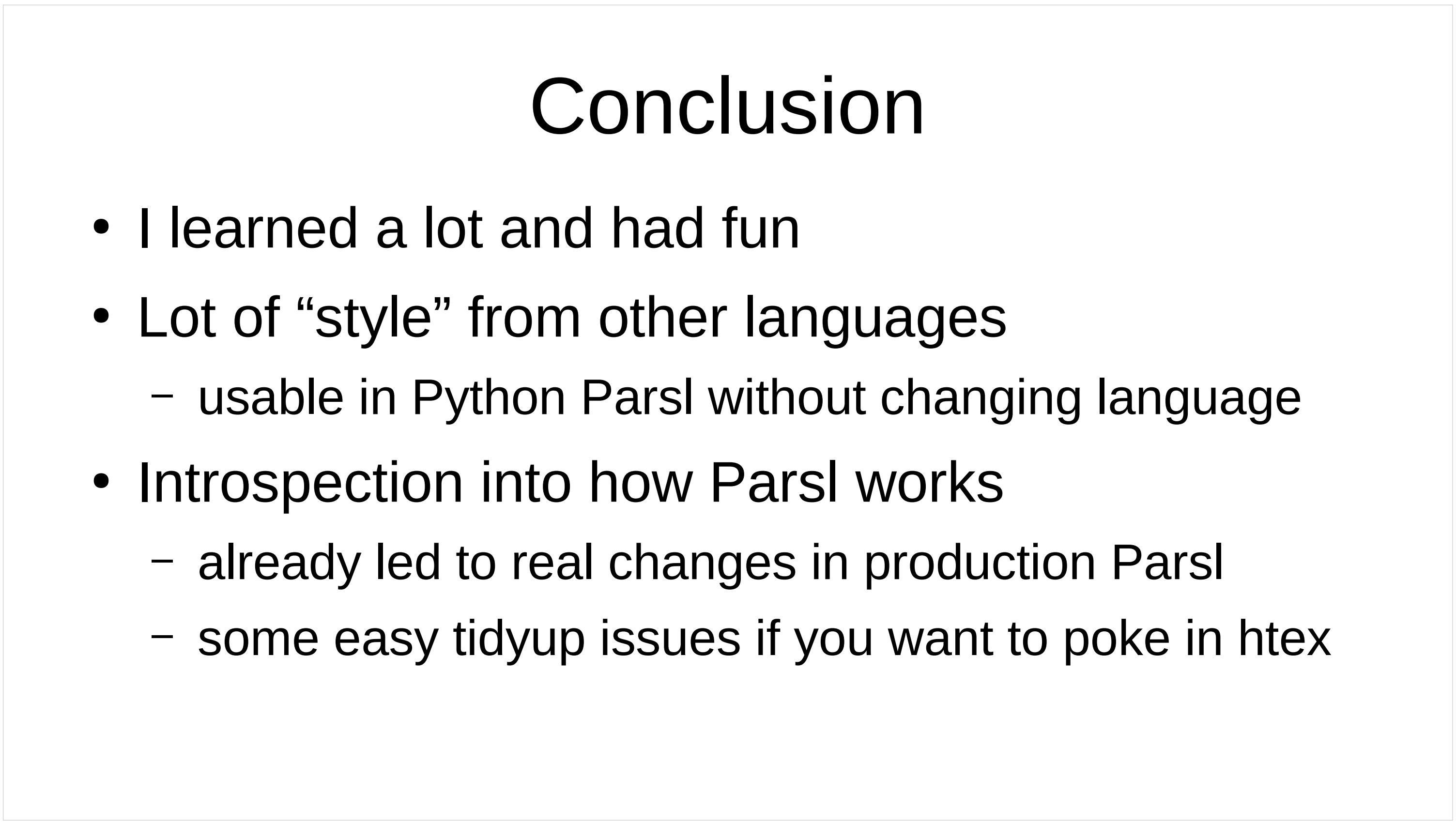
- Linear types can model: resources, protocol state
- (vs “values” like 7 or “hello”)
- Resource examples:
  - Allocated memory - can’t be shared, must be freed
  - Open file descriptors (resources inside OS)
  - Parsl tasks:
    - never drop a task implicitly
    - -> (queue?) -> worker -> result -> submit side
    - with side paths for errors

didn’t really get much done with linear types. which was the point of using idris2. oh well. but I could talk about my usage of Byte buffers a bit?

transform into something managed by a GC is a “successful consumption” of the value - even though it isn’t a rust-style release.

linear types: can check a value is used exactly once. quite similar to rust’s ownership and implicit frees when no longer needed. but might imagine things like: “you have to do *something* with this task - don’t implicitly free it, because implicitly freeing it is memory-safe but it’s not higher level safe: we should *never* silently discard a task. worst case would be to log a critical error when doing so. but other thing we can do is dispatch it to a worker and then insist we deal with a result? (which also involves some wire protocol). Maybe i can do some stuff around there in the next few weeks?

(“implicit” frees can be bad because you don’t necessarily know when they’ll happen: plenty of parsl hangs, especially around exit, have come from the __del__ method of objects; which fires in an arbitrary thread at an arbitrary time. this works in parsl especially badly with ZMQ)



Conclusion

- I learned a lot and had fun
- Lot of “style” from other languages
  - usable in Python Parsl without changing language
- Introspection into how Parsl works
  - already led to real changes in production Parsl
  - some easy tidyup issues if you want to poke in htex

I learned a lot.

It generated a bunch of issues and pull requests that are improvements to mainstream Parsl.

Some broader forward looking topics.

It gave me some good thoughts about what the future of Parsl might be (hint: you should hire me if you want more Parsl)

What can you do?

Think about this stuff in your own work... play with this stuff in your own work...
about programming as a thing in itself, not just a tool to get whatever your application specific research is by the shortest path you already know.

Pick up some of the ideas/issues to work on Parsl.