Planning for Overload

maybe you had overload

error_logger blowing up

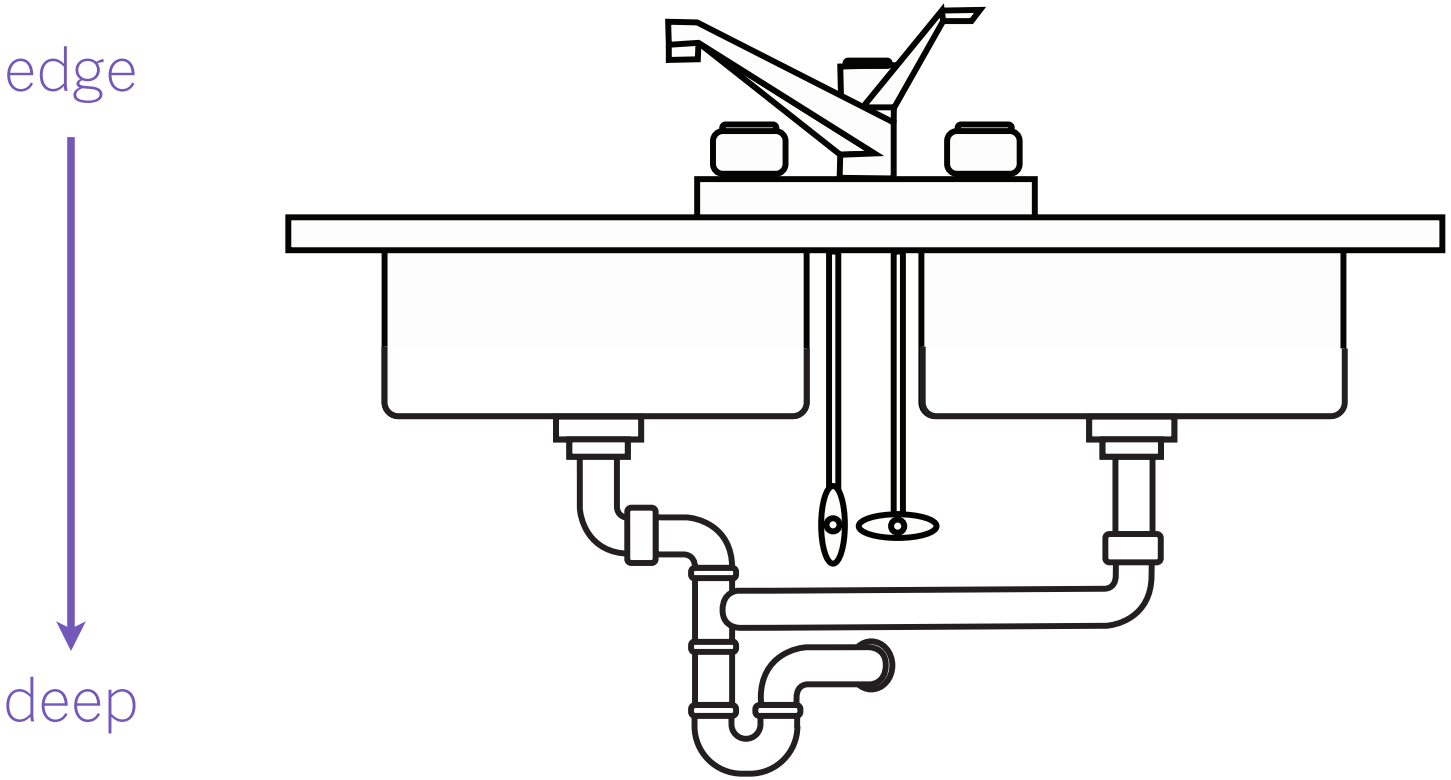
blocking operations backing up

message queue explodes

eheap_alloc: Cannot allocate 298930300 bytes of memory (of type "old_heap")

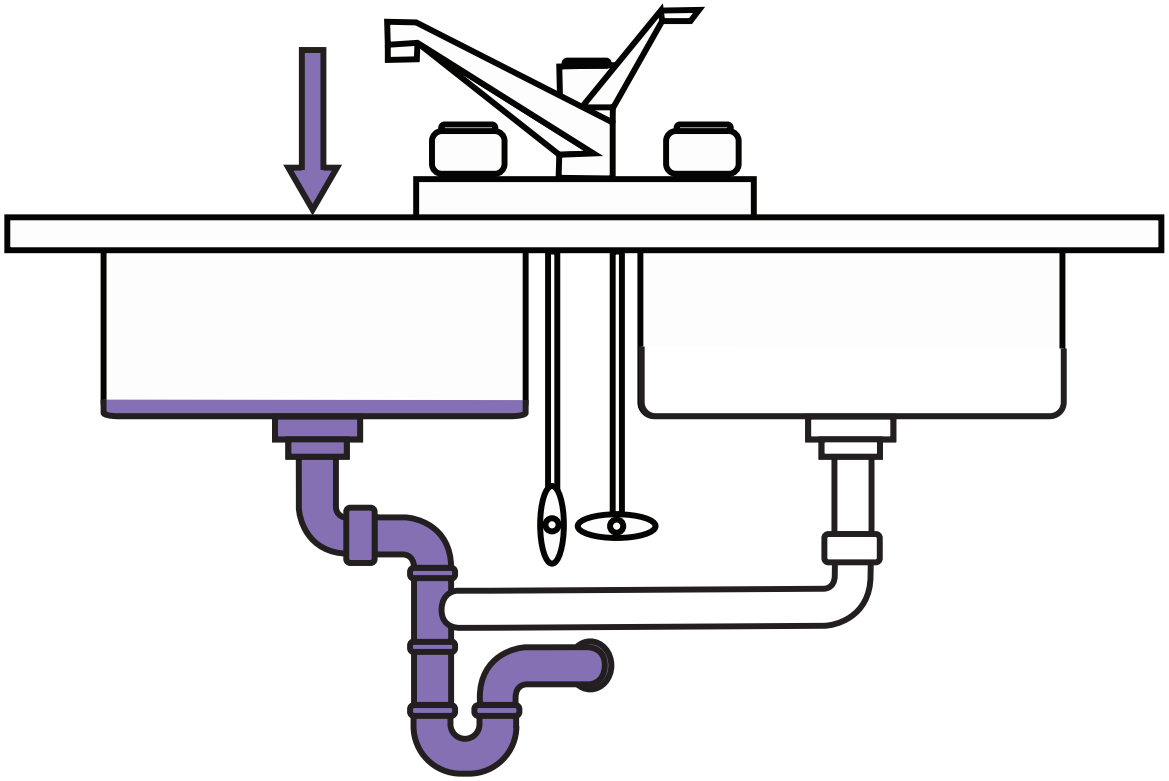
your system is a bathroom sink

edge

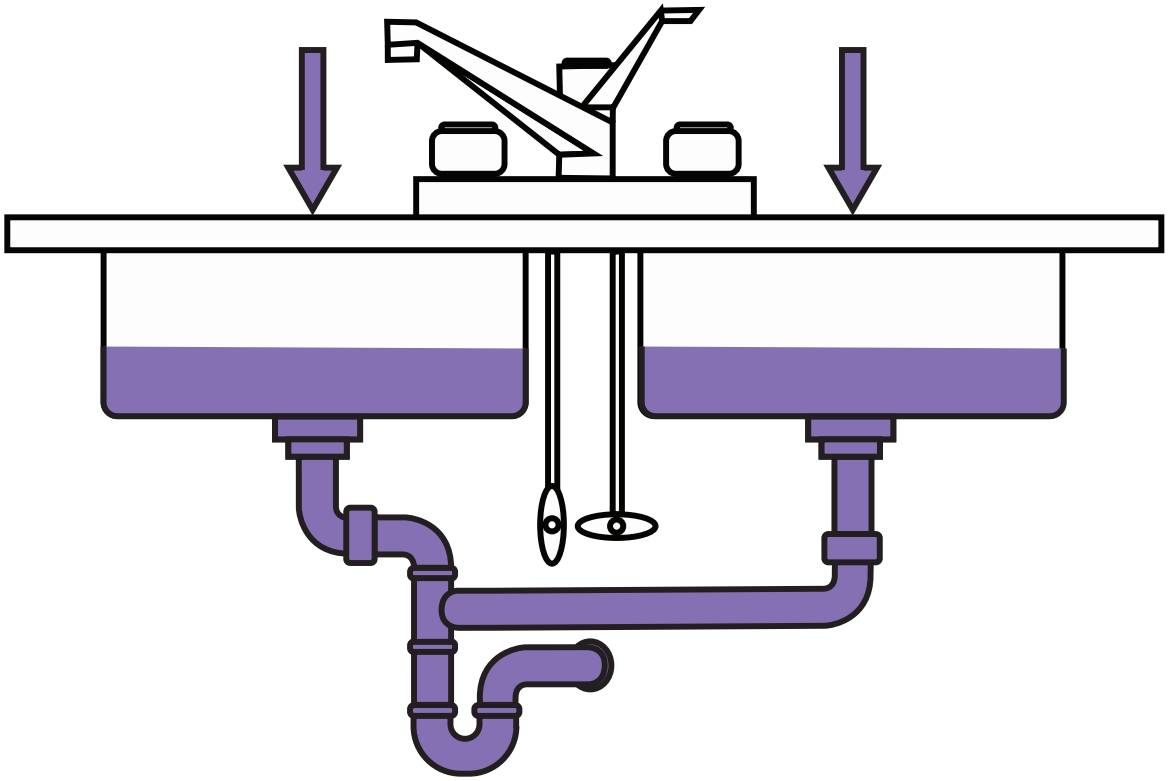


deep

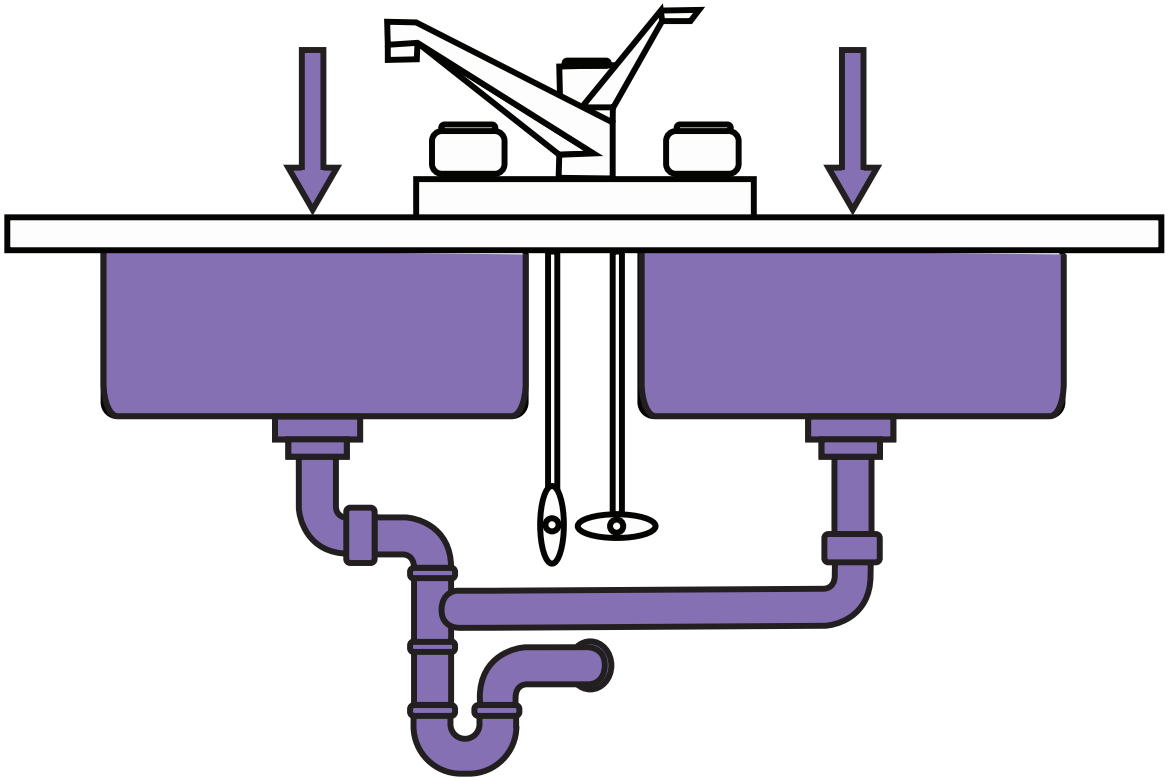
normal operations



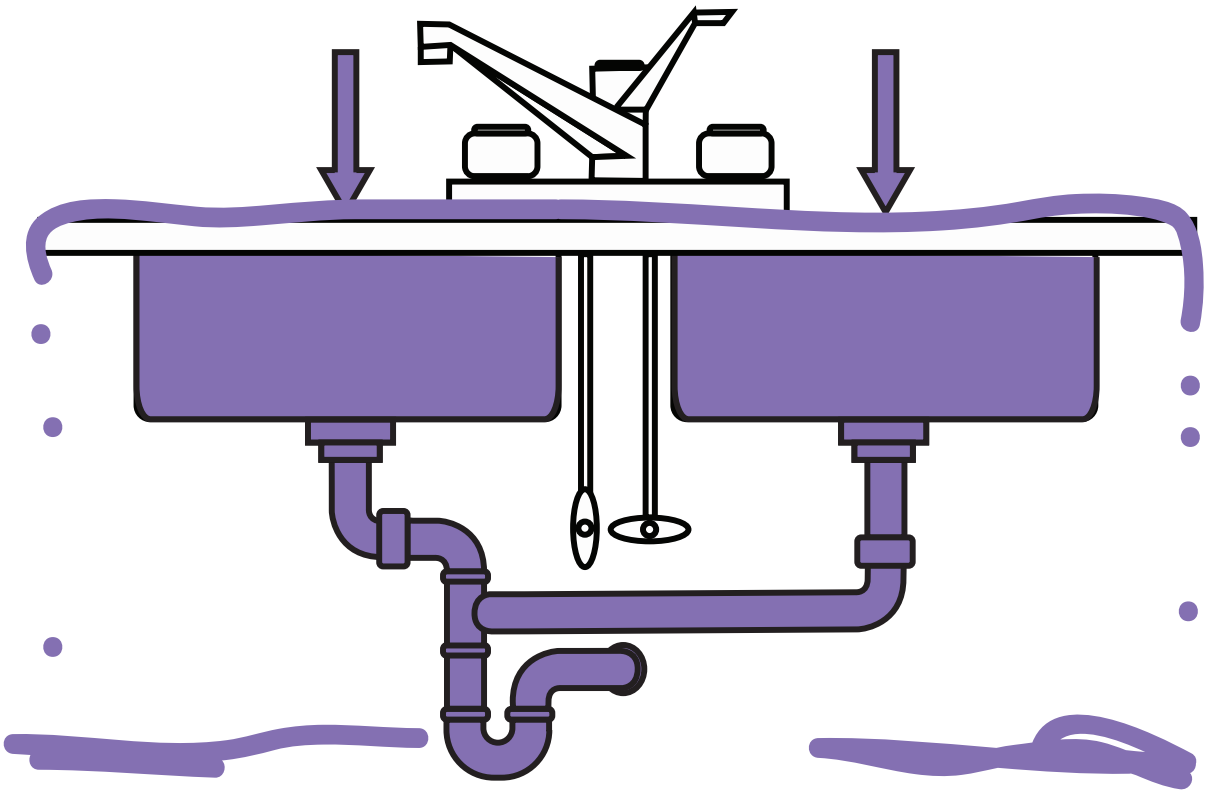
temporary overload



prolonged overload

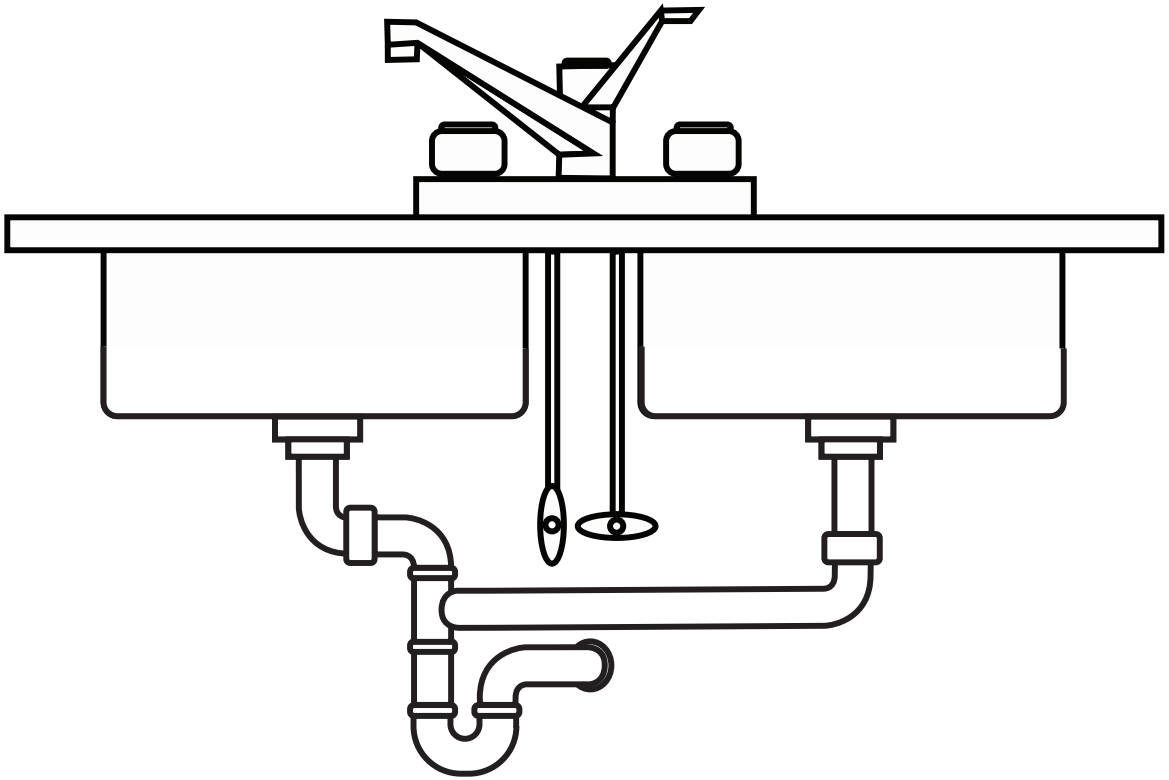


crash dump!

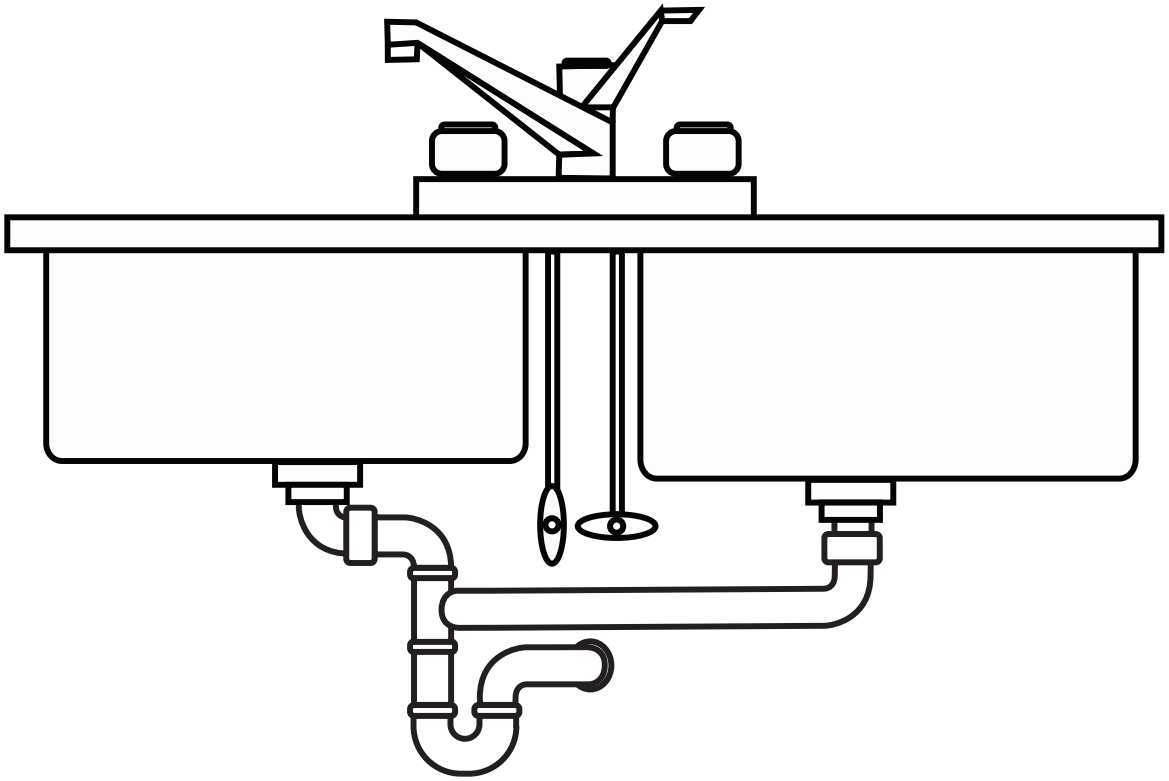


if we make it bigger, it's gonna handle more flow

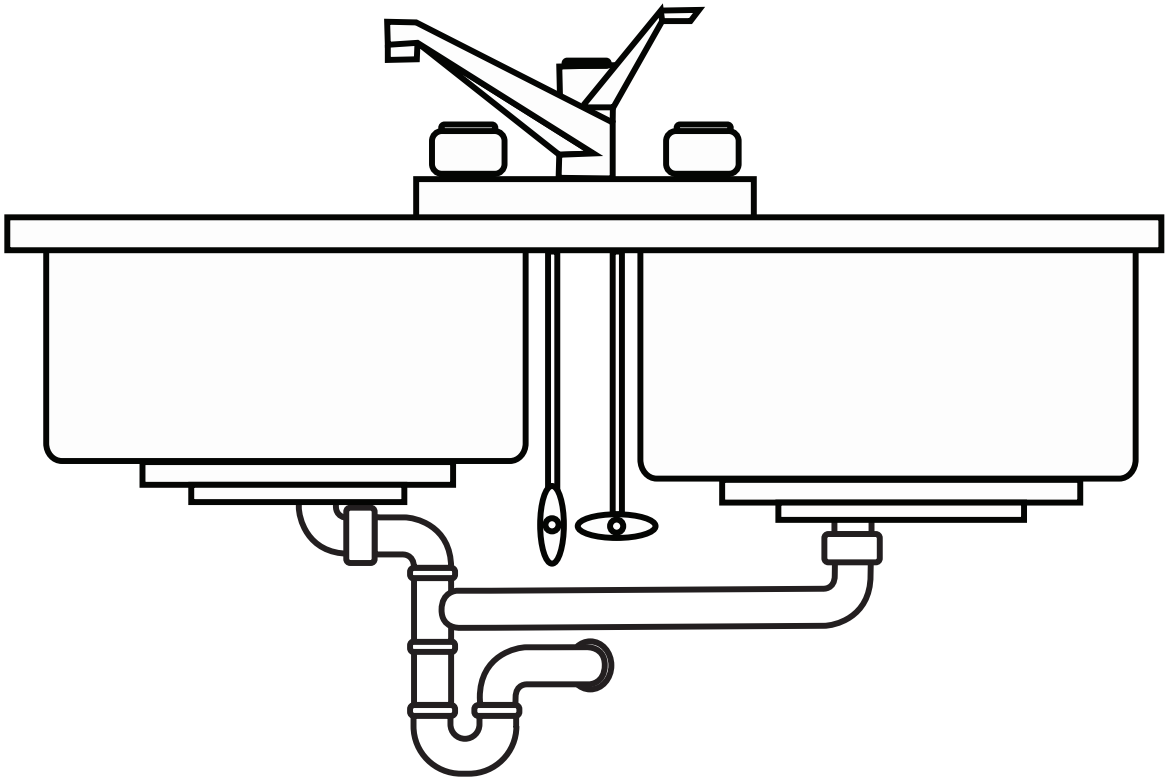
optimize away!



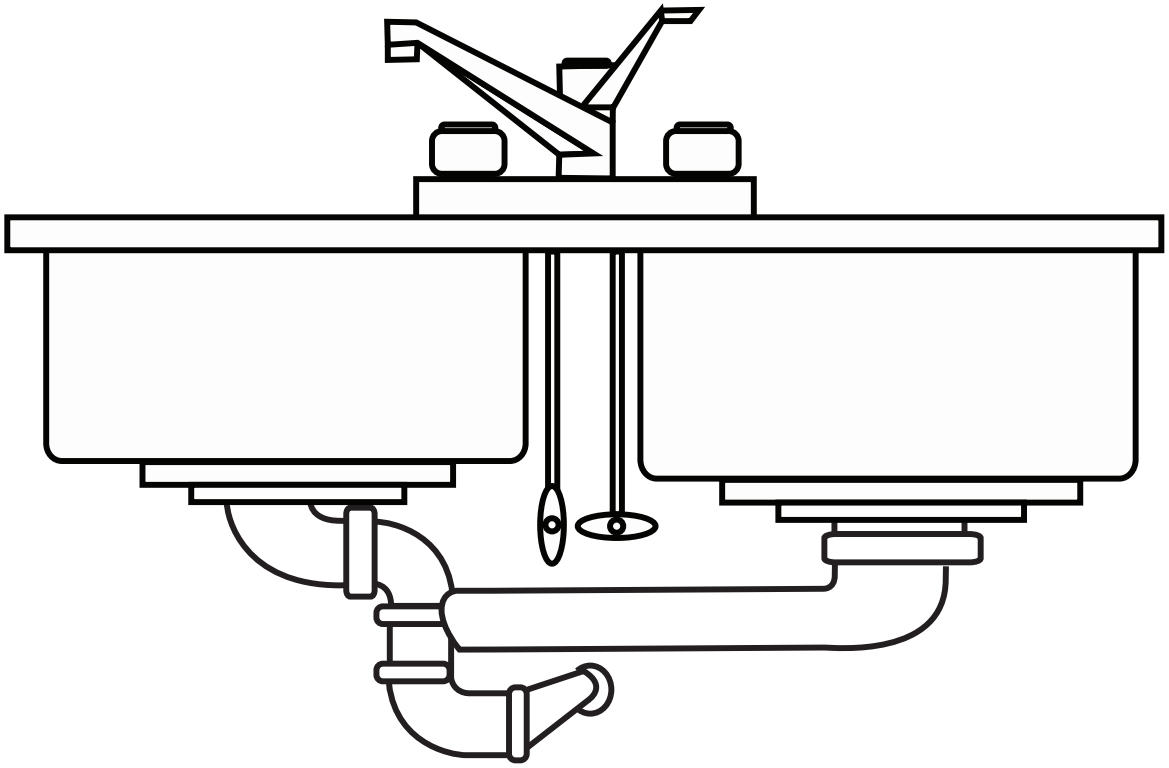
bigger sinks!



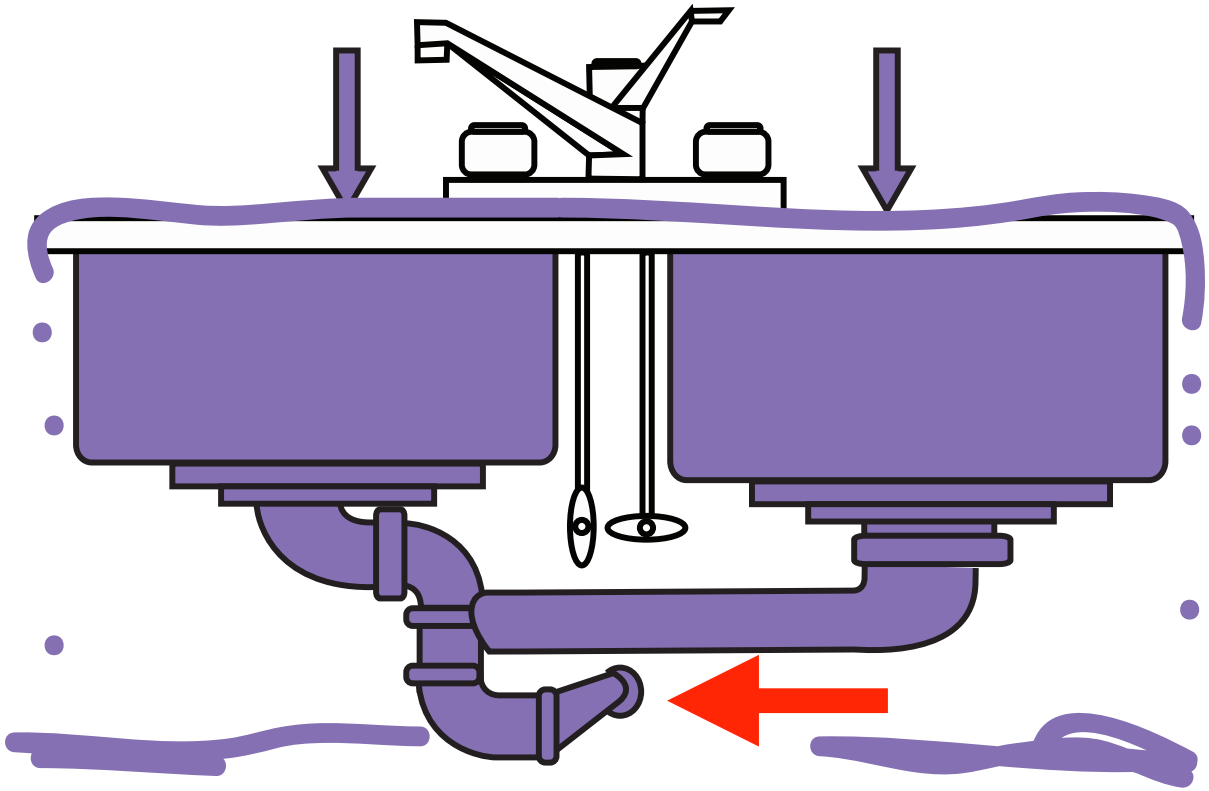
bigger drains!



bigger pipes!



bottlenecks you don’t control



paid to solve the wrong problem

Overload must be planned for

it defines how you die

it defines premature optimization

it defines your margin of error

it defines your API

it defines your engineering

pick what has to give

block on input
(back-pressure)

drop data on the floor
(shed load)

it's a business decision

simple back pressure

Make all calls that matter synchronous

do it all the way down

restrict the pool of callers at the edge

automate back-pressure through blocking

slow the users down (how do you tell them?)

issue: timeout management

what's the typical operation delay? p99?

"Some application developers may push for no timeout and argue it is OK to wait indefinitely. I typically propose they set the timeout to 30 years.[...] Why is 30 years silly but infinity is reasonable?" - Pat Helland

timers at the edge > timers deep

ask for permission

explicit form of back-pressure

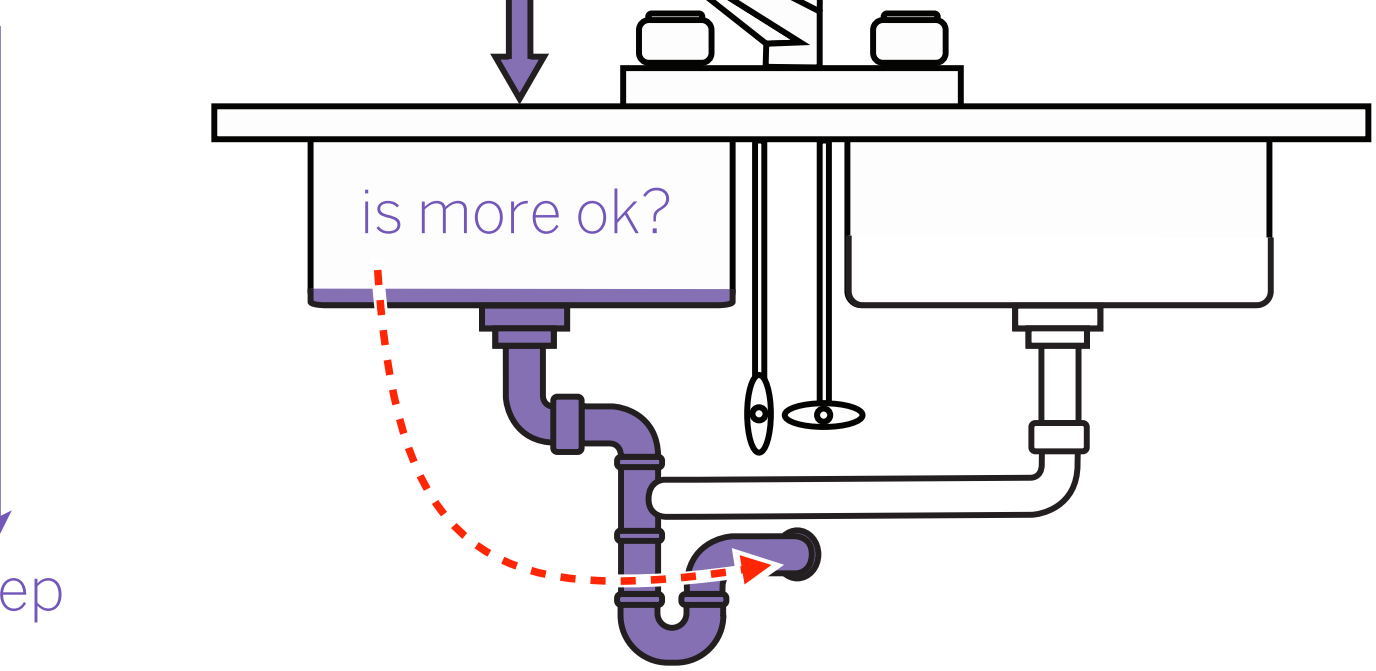
identify the resources you want to block on

make the edge ask for permission

easy to underutilize a system

ask for permission

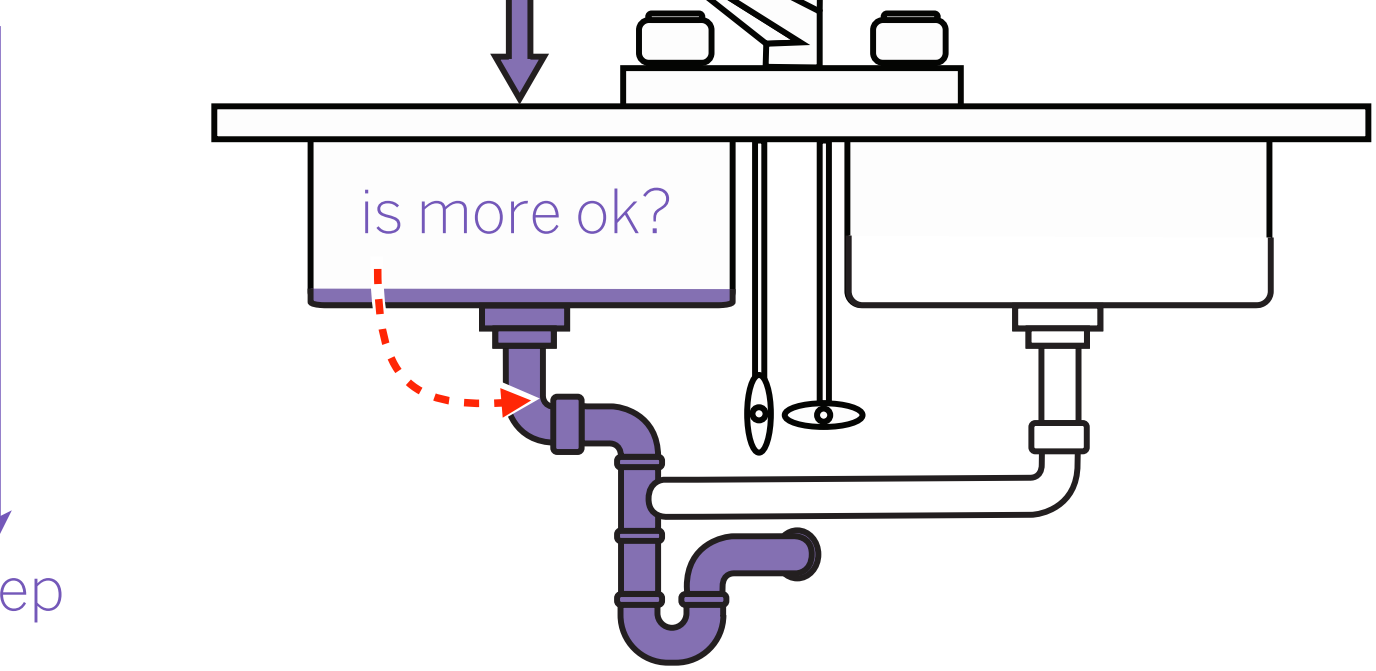
edge



deep

ask for permission

edge



deep

implementations

use processes or ETS tables to ask permission

memsup, cpusup, disksup in os_mon

overload module in SASL

https://github.com/jlouis/safetyvalve

https://github.com/uwiger/jobs

load shedding:
“I don’t even need these requests”

random drop

when some loss is acceptable (sample size!)

can be made adaptive

works even better producer-side

```
case drop:random(0.95) of
    true -> send();
    false -> drop()
end
```

```
random(Rate) ->
    maybe_seed(),
    random:uniform() =< Rate.
```

queue buffers

more control than random drop

can drop from either end of the queue if full

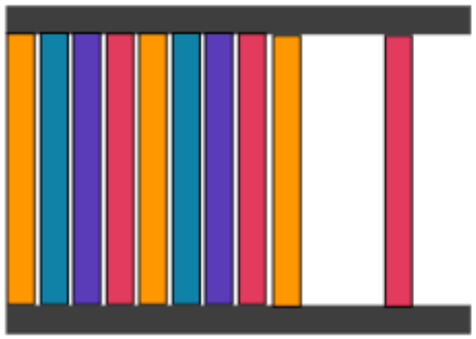
useful if you need messages in order

stack buffers

better for low latency

no requirement for ordering

discard oldest data, or all data too old

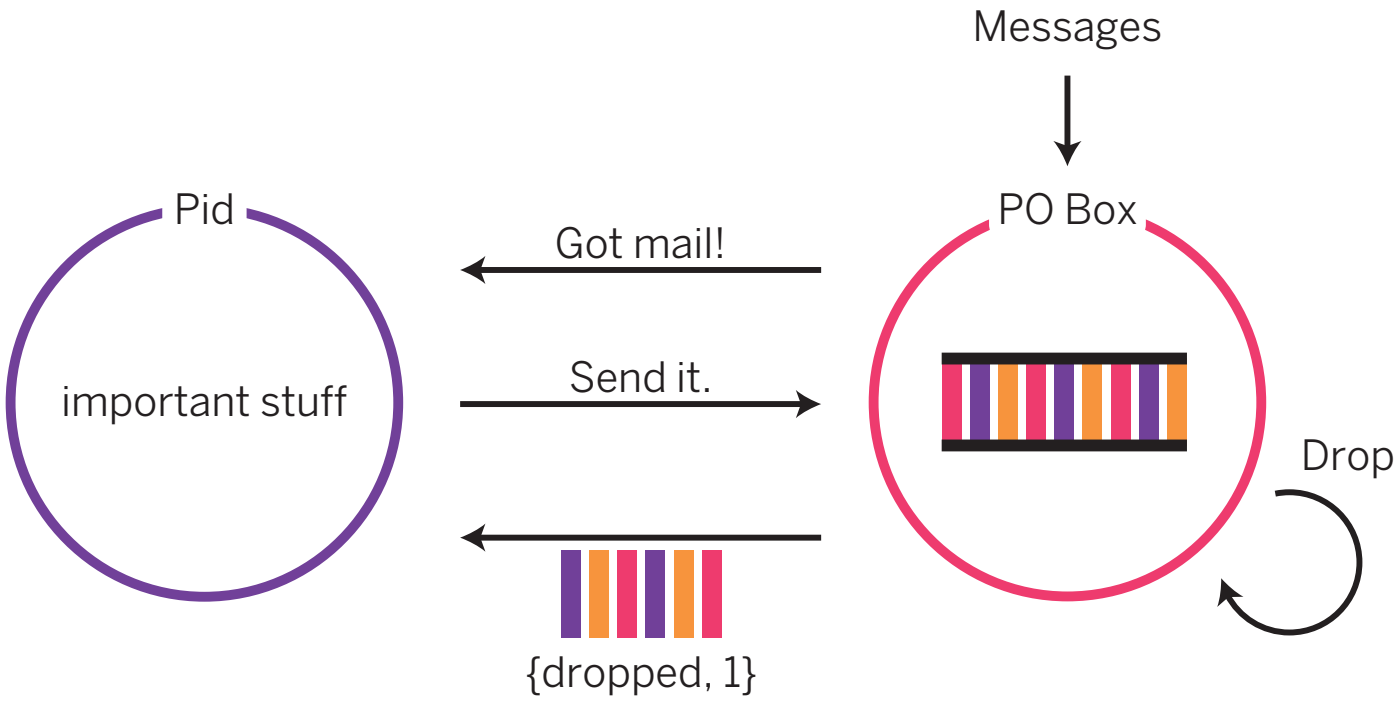


implementations

lager does it on OTP errors for cascading failures

https://github.com/ferd/pobox

https://github.com/ferd/dispcount


Messages
Pid
PO Box
Got mail!
important stuff
Send it.
Drop
{dropped, 1}

PO Box Model

active call

manipulate the PO Box's mailbox

```
filter(_, 0) -> skip;
filter(<<>>, N) -> {drop, N};
filter(Msg, N)  -> {{ok,Msg}, N-1}
...
pobox:active(BoxPid, fun filter/2, 25)
```

how do you tell users?

Block on sessions for back-pressure

Put usage limits, however high

Tell about losses (logplex L10 messages)

Respect End-to-End principles

Make idempotent APIs

Questions?


Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert
Foreword by Joe Armstrong
Learn You Some Erlang for Great Good!
Hébert