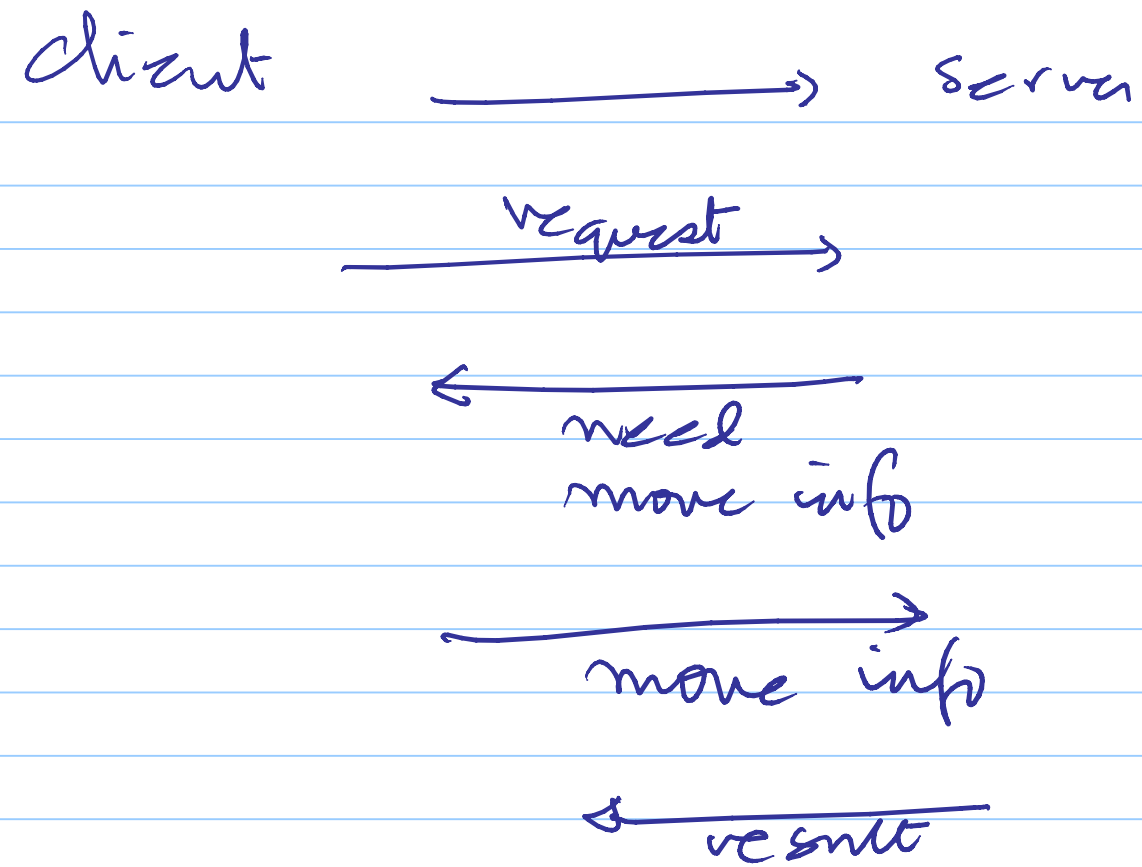
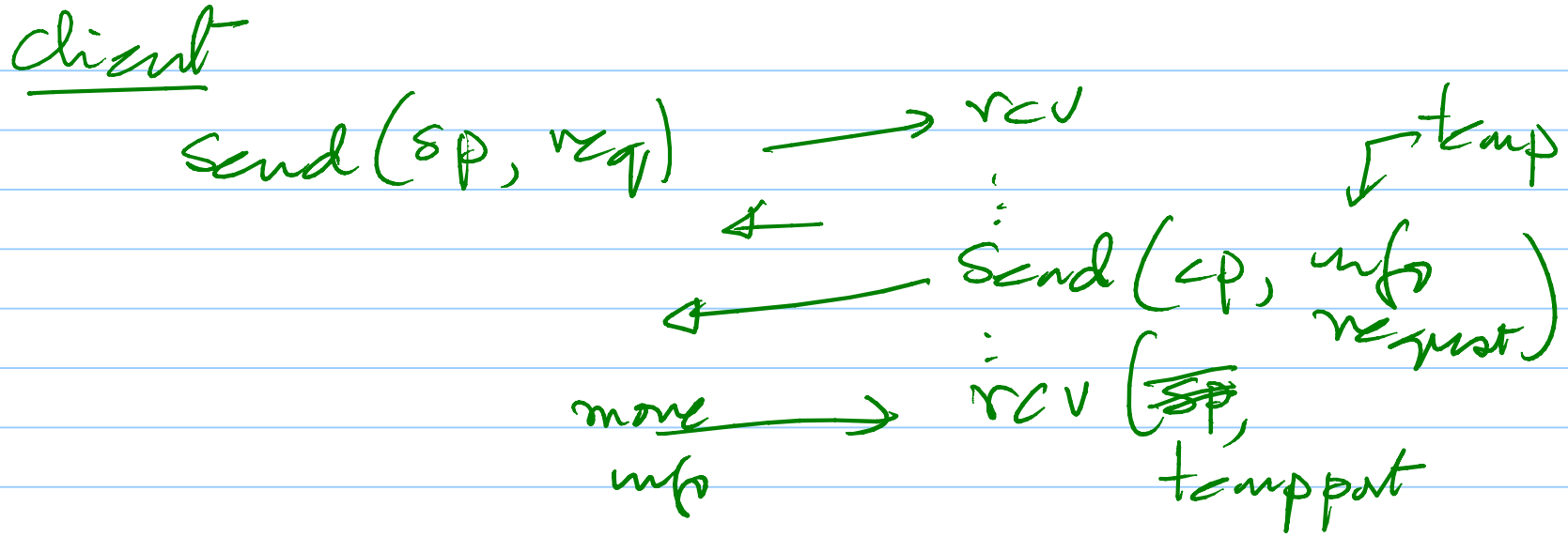
Message passing
- ports
- global naming scheme
- send & receive
- ports can be dynamic
- Name server needed

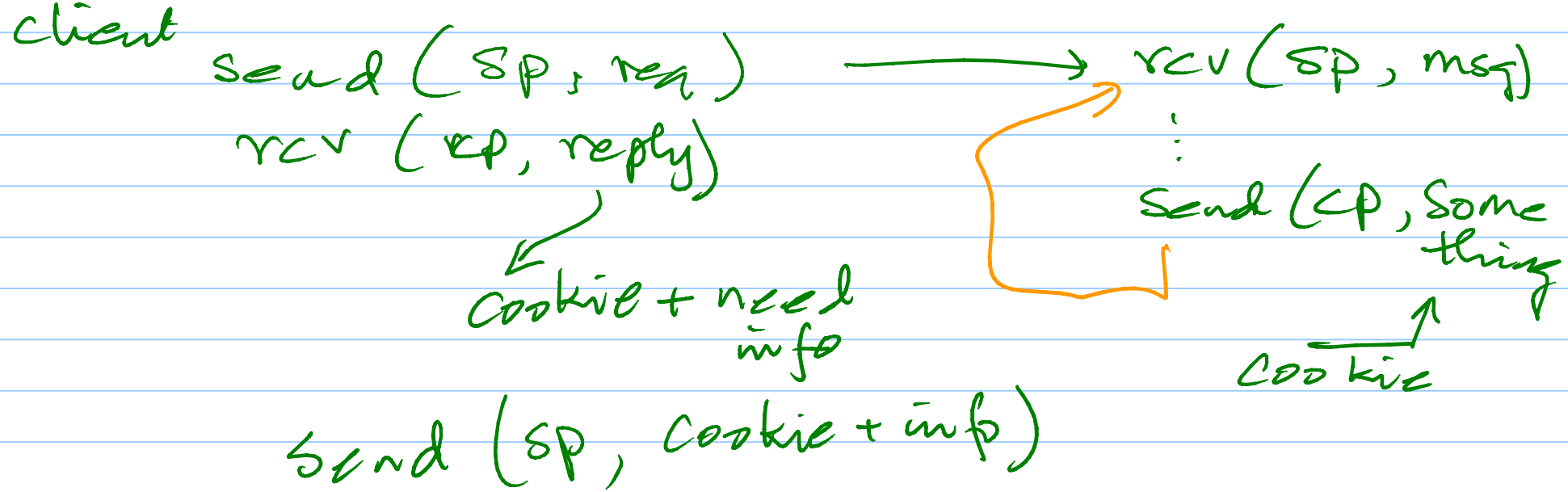
stateless server.

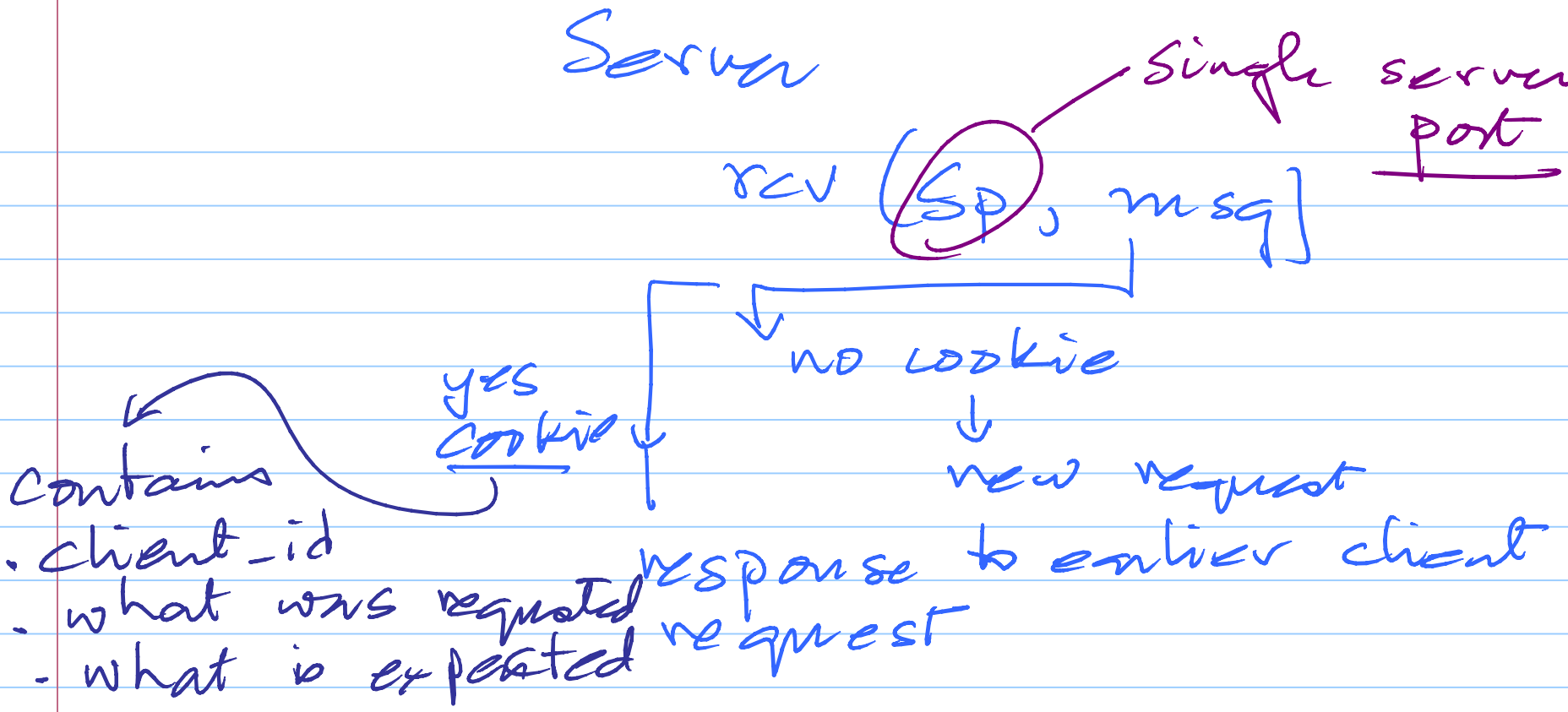
- simpler (?) does not maintain client data
- not everything can be done →


client
server
request
need
more info
more info
result


client
send(sp, rcq)
rcv
temp
send(cp, info request)
more info
rcv(sp, tempport)

Cookies.


client
send (sp, req)
rcv (cp, reply)
cookie + need info
send (sp, cookie + info)
rcv (sp, msg)
send (cp, Something)
cookie


Server
single server port
rcv [sp, msg]
no cookie
new request
to earlier client
yes cookie
response request
contains
. client_id
. what was requested
. what is expected

Servers need to store client data

- stateless
  - package up client data & send to client (cookie)
  - comes back → reconstruct client state serve request

client

send

recv

recv

send

Server data

- in core
- on disk
- eg a counter → counts the # of requests served.

global shared data

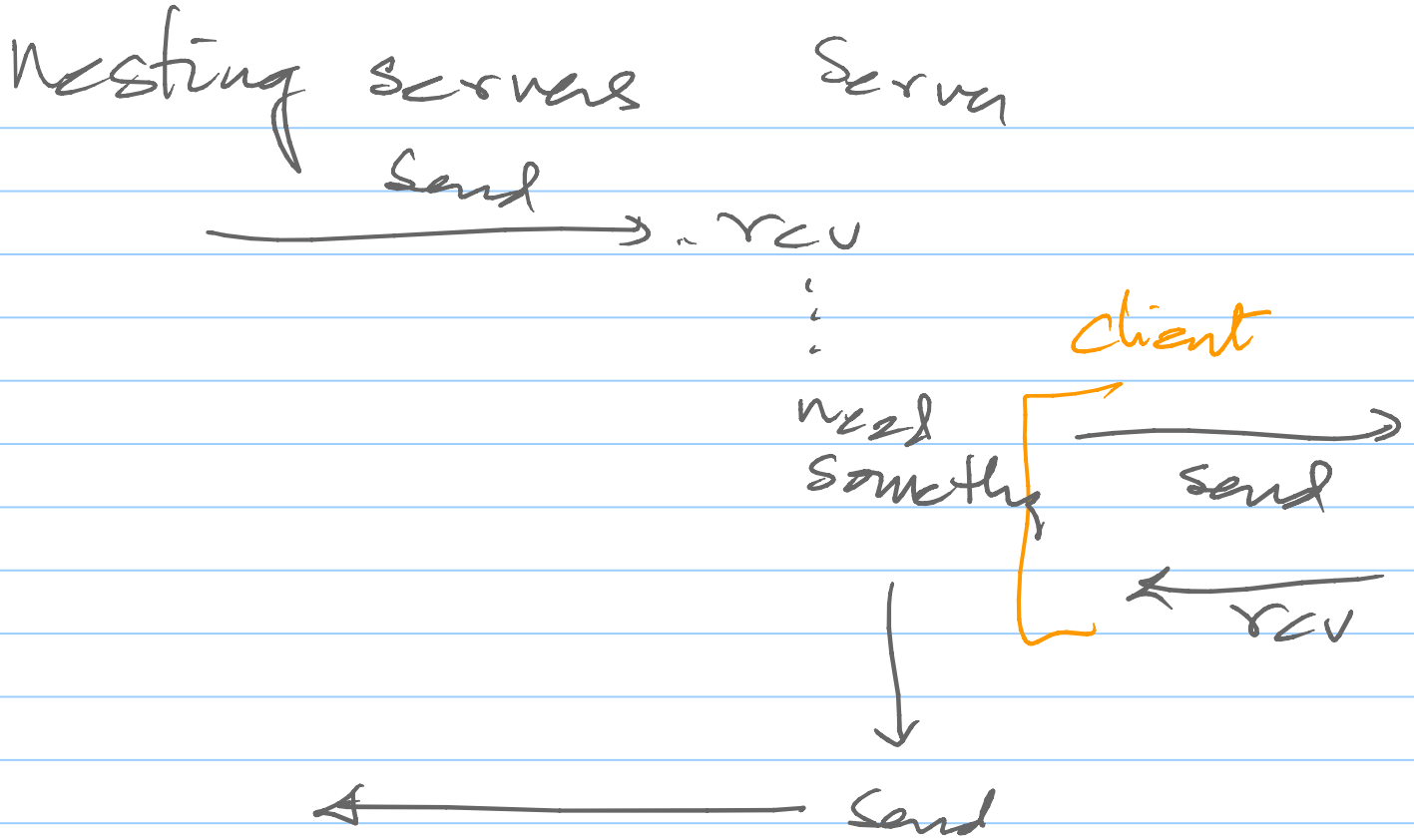
↳ where?

shared data

- on disk → stateless ? yes

- in memory → not stateless

  → but stateless if this data is not client specific


Nesting servers
Server
Send
rcv
client
need something
send
rcv
send

recursion? (call itself)

```
rcv(sp, ...)
  :
send(sp, ...)
rcv(sp, response)
```

multithreaded servers

→ parallelism

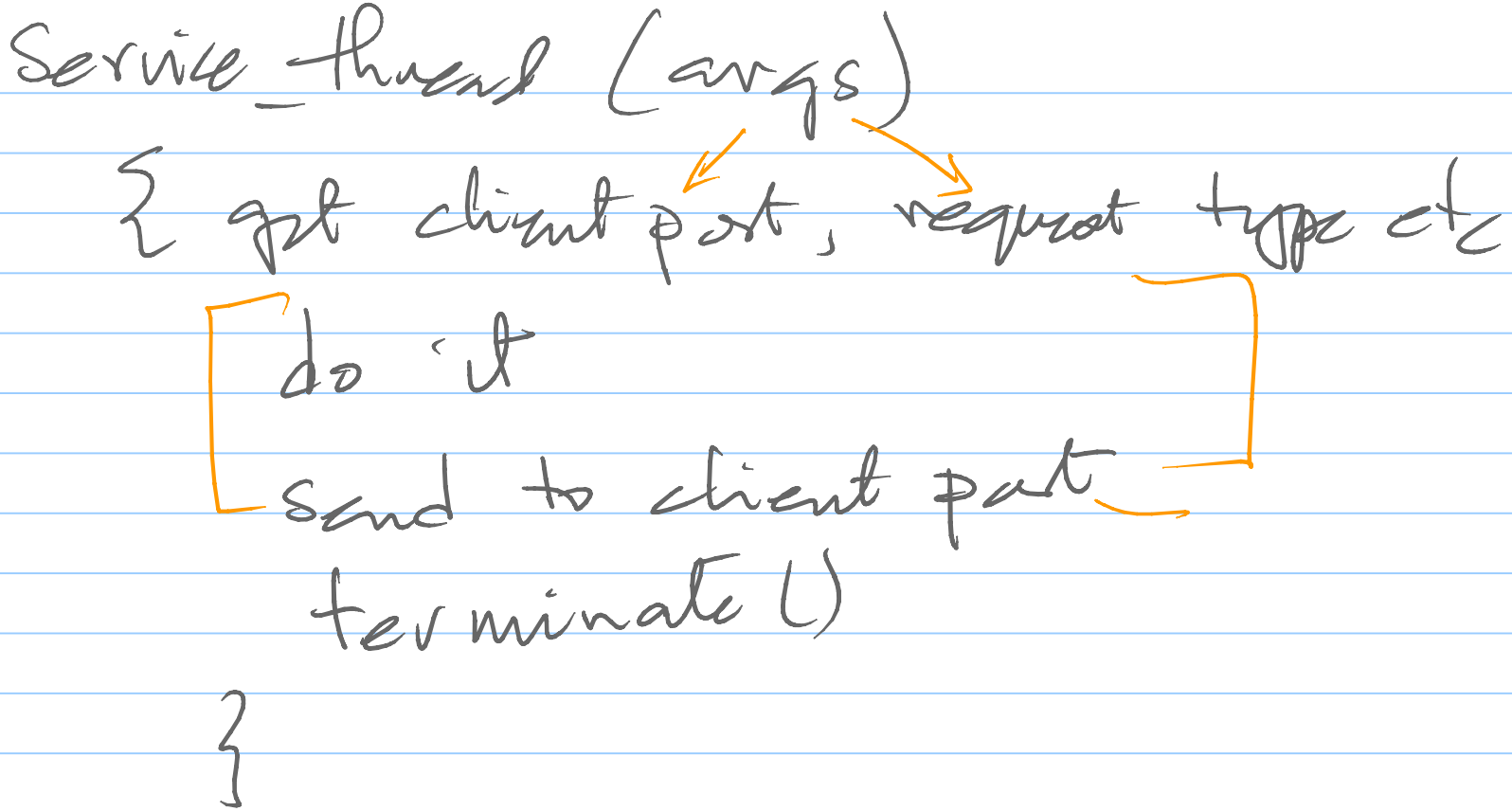
→ overlap CPU & I/O

→ scalable (?)

Multithreaded

∞ loop [rcv(sp, request)
createthread(service thread, arg)
↑ request

server stub


service_thread (args)
{ get client port, request type etc
do it
send to client port
terminate ()
}

multithreaded server → nesting

Servic thread

{ dedicated to one client ← createport → cp_2

Send (sp_2, req)

rcv (cp_2, response

}

prior scheme → dynamic multithreading

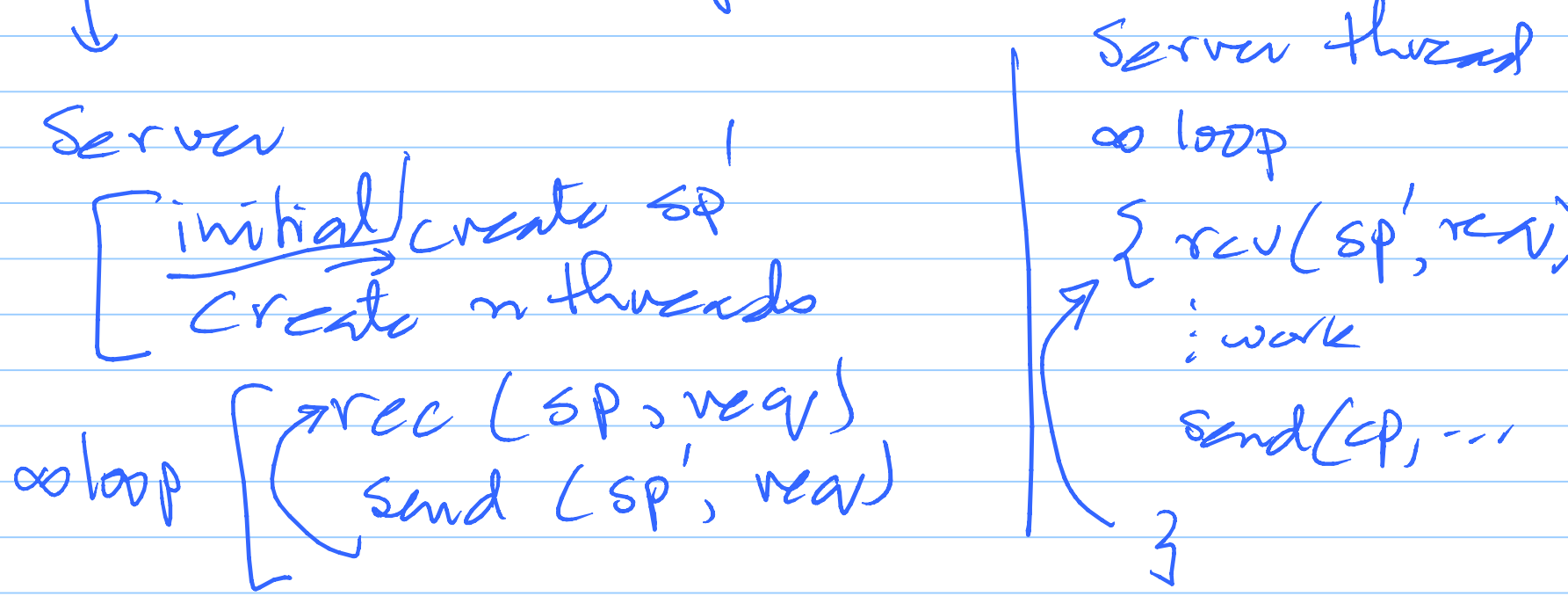
→ overhead

→ threads are created/deleted for every request

→ race conditions (global data)

↳ semaphores / monitors

Static multithreading → use a pool of threads



multithreaded server

→ use threads

has shared mem

& race conditions.

→ use processes.

No shared mem

(no race cond)

Static multithreading

→ lower overhead

→ no recursion →

→ better control of performance

How to implement ports

↳ port naming

→ global & unique

→ atomic → two sends → port does not double
two recvs → port, get diff msgs

→ Send/recv works on any port

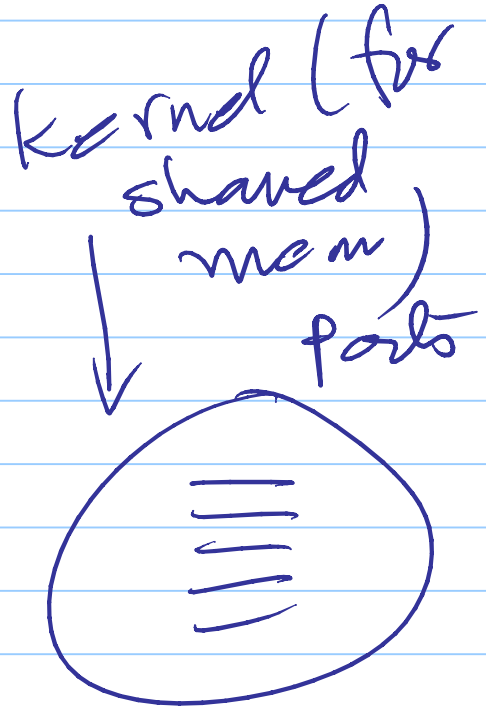
Simplified

→ all processes (server, clients)
& ports are on 1 machine

→ centralized

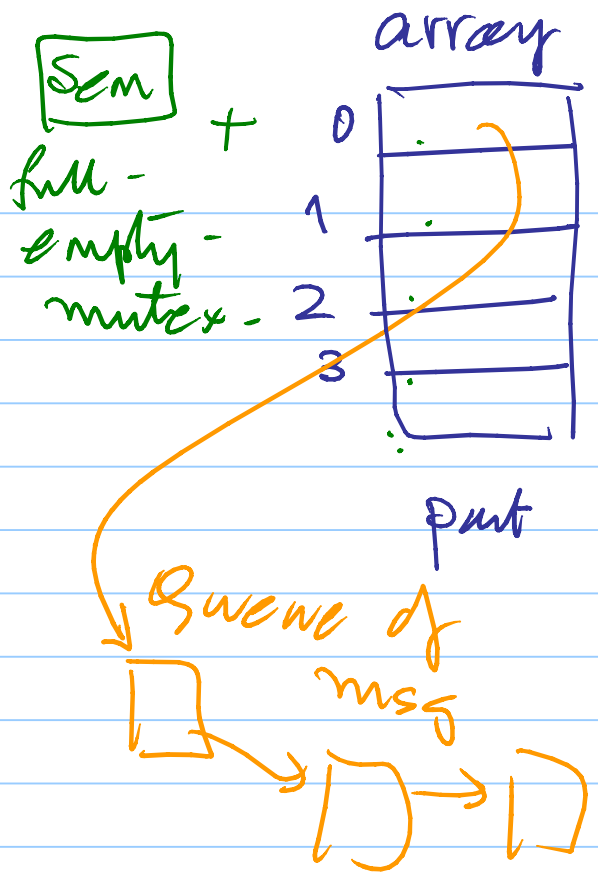
Process

- Send (p, msg)
- rcv (p, msg)
  - ↳ kernel calls



Kernel

```
Send (p, msg)
  { P(full[p])
    P(mutex[p])
      addQ(Q[p], msg)
    V, V.
rcv (p, msg)
```



prod

{ P(full)

P(mutex)

add

V(mutex)

V(empty

N

Cons

{ P(empty)

P(mutex)

delete

V(mutex)

V(full)

}

0

create port( ) — P(global mutex)

- { find unused entry → change to used
  init Q, Sem,
  return port # }
  V(global_mutex)

delete port(p) → • mark p unused
• clean