UNIVERSITY of WASHINGTON

LECTURE 20 · CSE 333 · Summer 2026

Intro to Concurrency

Discussion: Do you have more trouble starting a project? Or finishing a project?

Reminders

Assessments:

- Exercise 8 due Friday at 11:59pm
- Homework 4 released soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Final Exam
  - CSE2 G10 from 1:30-3:30pm on Tuesday, August 18
  - Makeup exam: 5-7pm on different days. Email coming soon!

Review

Full server code!

```
1  addrinfo hints = {};
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  hints.ai_flags = AI_PASSIVE;
5  addrinfo* results = nullptr;
6  getaddrinfo(nullptr, port, &hints, &results);
7  int listen_fd = socket(results->ai_family,
8                         results->ai_socktype,
9                         results->ai_protocol);
10 bind(listen_fd, results->ai_addr, results->ai_addrlen);
11 listen(listen_fd, SOMAXCONN);
12 while (true) {
13   int client_fd = accept(listen_fd, nullptr, nullptr);
14   HandleClient(client_fd);
15   close(client_fd);
16 }
```

```
$ ./server &

$ nc localhost 3333

hello, networks

HELLO, NETWORKS
```

One copy but any number of users

- The uppercase logic never changed
- netcat is somebody else's client
- Your own client comes next

listen_fd and client_fd

`accept()`

`listen_fd`

the doorway

`no client yet`

One socket exists. It is bound and listening, and it carries no request bytes.

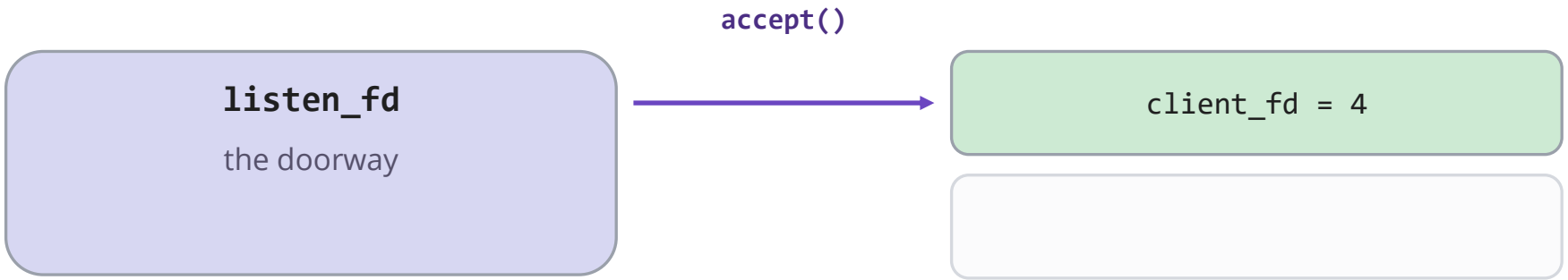
file descriptor table

`0-2`	std streams	`terminal`
`3`	socket	`listening :3333`

What listen_fd is for

- It was created, bound, and marked listening
- It is the address clients dial
- read() on it never returns request data

listen_fd and client_fd



accept() did not change listen_fd. It returned a second descriptor, and now both are open.

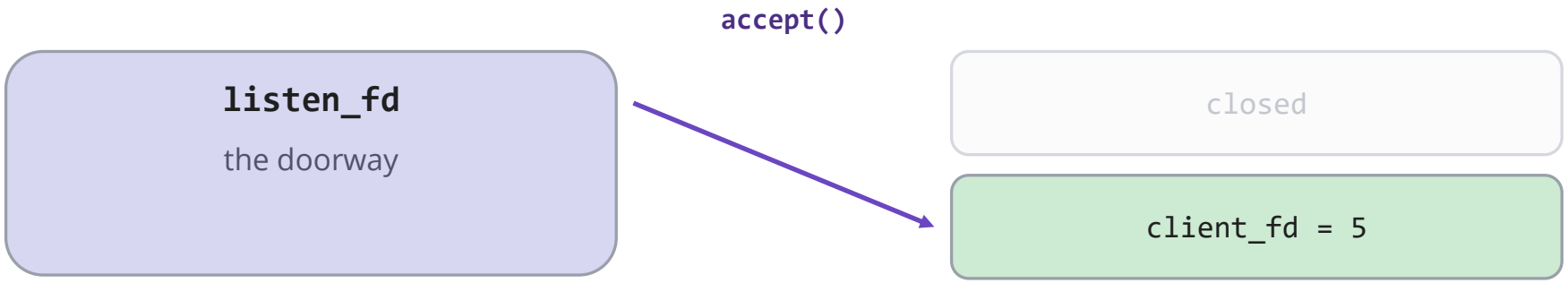
file descriptor table

0-2	std streams	terminal
3	socket	listening :3333
4	socket	client 10.0.0.7

What accept() actually did

- listen_fd is untouched and still listening
- client_fd is a new row in the same table
- Only client_fd carries this client's bytes
- HandleClient wanted this one

listen_fd and client_fd



close(client_fd) ends one conversation. listen_fd survives, and the next accept() returns a fresh descriptor.

file descriptor table

0-2	std streams	terminal
3	socket	listening :3333
4	closed	was 10.0.0.7
5	socket	client 10.0.0.9

Two lifetimes

- listen_fd lasts as long as the server
- Each client_fd lasts exactly one conversation
- close(listen_fd) is what shuts the service down
- Forgetting to close client_fd runs you out of descriptors

Finished client!

```
addrinfo hints = {};
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
addrinfo* results = nullptr;
getaddrinfo(host, port, &hints, &results);
int fd = socket(results->ai_family,
                results->ai_socktype,
                results->ai_protocol);
connect(fd, results->ai_addr, results->ai_addrlen);
freeaddrinfo(results);
WriteAll(fd, "hello\n", 6);
string reply = ReadLine(fd);
close(fd);
```

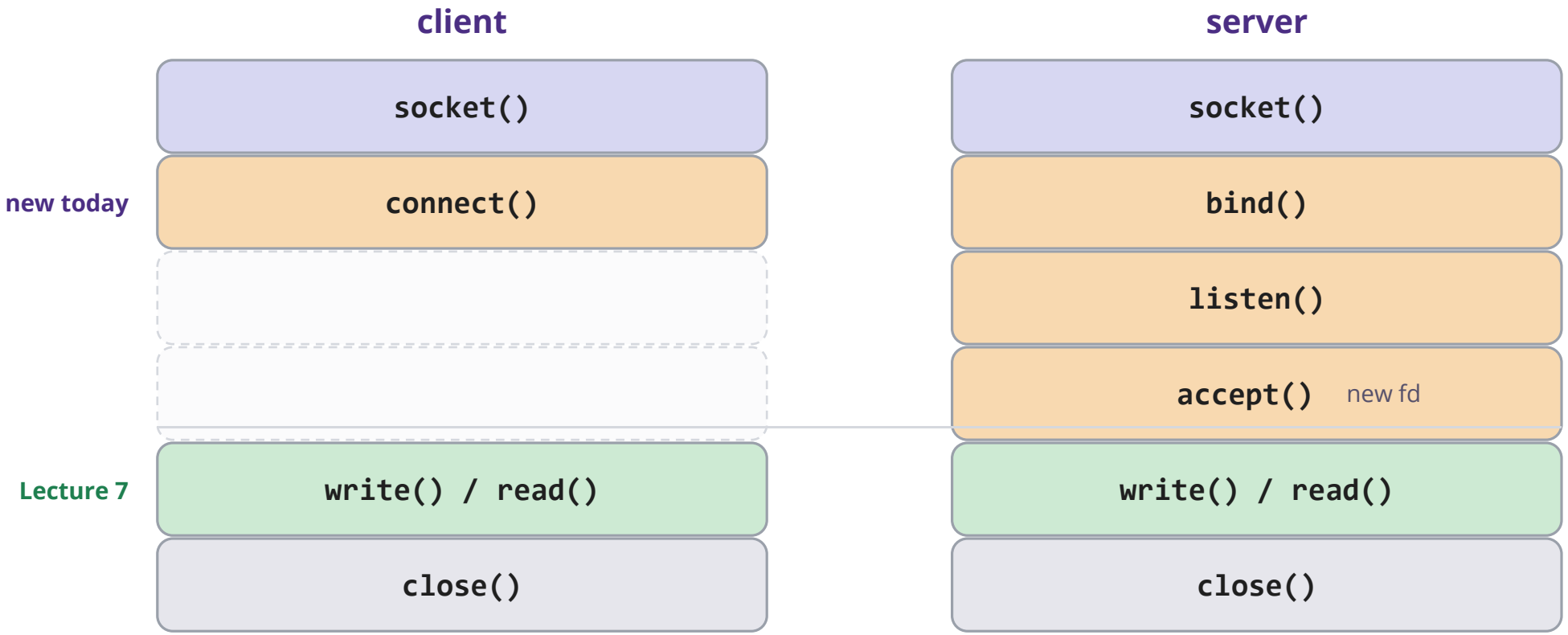
```
$ ./client attu 3333

hello, networks

HELLO, NETWORKS
```

Where we started

- argv became a descriptor we read
- stdout became a descriptor we write
- The uppercase line never changed

REVIEW

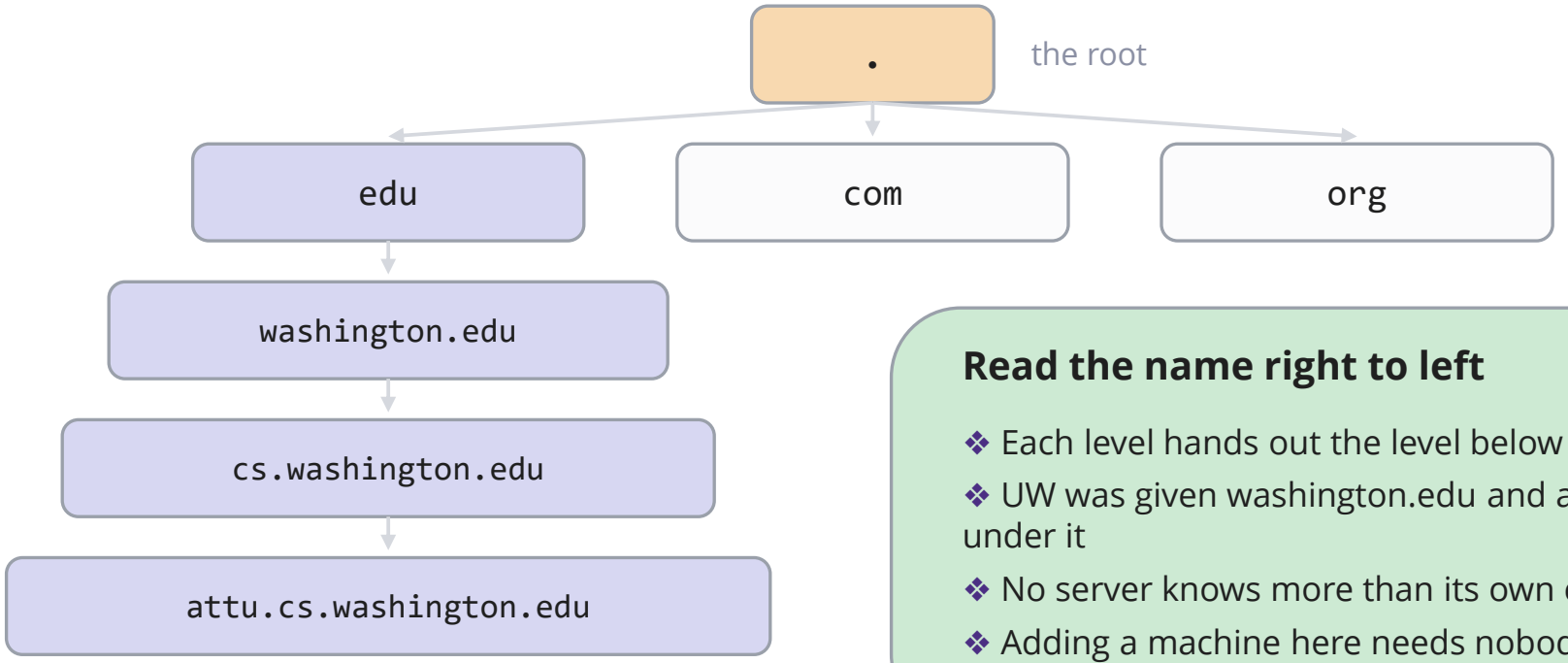
Client and server, side by side



Both sides call socket(). The client names who it wants; the server claims a port and waits. Below the line, both sides are doing Lecture 7 file I/O.

REVIEW

DNS



Read the name right to left

- Each level hands out the level below it
- UW was given washington.edu and answers for everything under it
- No server knows more than its own children
- Adding a machine here needs nobody else's permission

DNS and HTTP

Caching and TTL

```
$ dig +noall +answer attu.cs.washington.edu
attu.cs.washington.edu. 3600 IN A 128.208.1.138

$ dig +noall +answer attu.cs.washington.edu
attu.cs.washington.edu. 3512 IN A 128.208.1.138
```

3600 is the time to live (in seconds)
The second lookup came from a cache and the counter is what is left

Why the root survives

- Every answer carries a TTL
- Resolvers cache it and reuse it until it expires
- Most lookups never leave your building
- The root is asked about edu, not about attu
- **The cost:** a changed address takes a TTL to be believed everywhere.

HTTP: Request

```
$ nc www.example.com 80
GET /index.html HTTP/1.1
Host: www.example.com
Connection: close
```

The last line is empty on purpose. That blank line indicates the headers are done

Three parts

- **Request line:** method, path, version
- **Headers:** name, colon, value, one per line
- **A blank line:** the headers are over
- Lines end with carriage return and newline

HTTP: Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

<html><body>hello</body></html>
```

Similar to a request, but now with a status line on top and a body underneath the blank line.

How you know when to stop

- Read until the blank line to get the headers
- Then read exactly **Content-Length** more bytes
- Without it you cannot tell a slow body from a finished one
- 200, 404, 500: the number is for the program, the text is for you

HTTP in HW4

your server does	using
accept a connection	the socket calls from the review
read until a blank line	the readN loop, because read() can return 0 bytes
parse the request line	split on spaces, then look the query up in your index
write the response	WriteAll, because write() is insufficient too

Concurrency

One message at a time

You have three questions, and every answer must come from a different person.
You send the first one, watch your phone until the reply lands an hour later, then send the second one.

`task`	`waiting`	`worker`	`sharing`
one question you need answered	the reply is out of your hands	someone who can wait on your behalf	one notebook everybody writes in

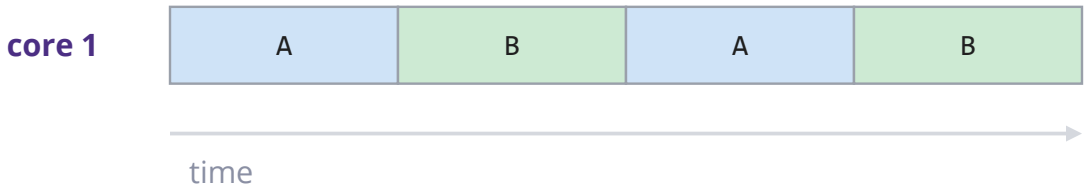
Four parts of the problem

- Three hours went by and you did about a minute of typing
- A blocked worker uses no CPU, so many tasks can be waiting
- Two people writing in one notebook can corrupt the notes

Concurrency and parallelism

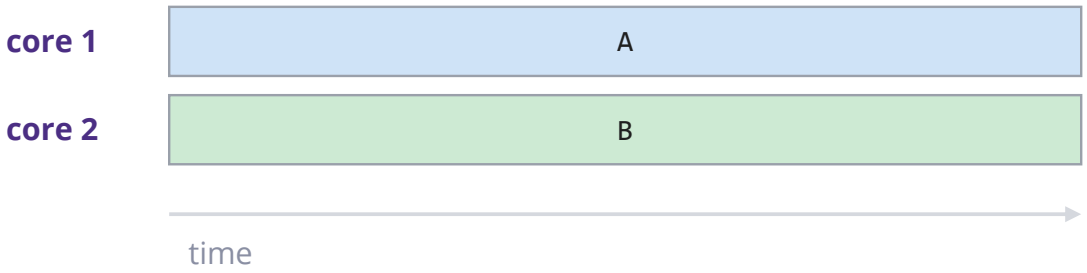
concurrency

Several tasks make progress during the same interval.



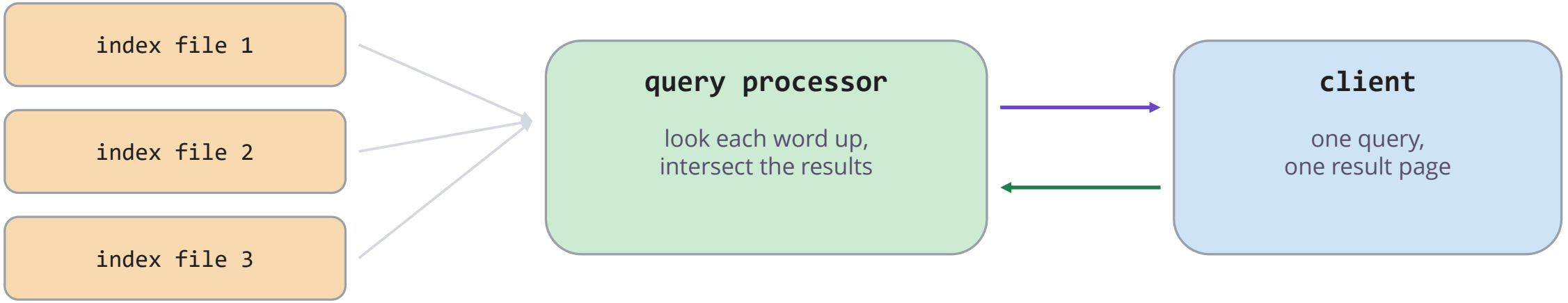
parallelism

Several instructions execute at the same instant.



Concurrency can improve an I/O-bound server on one core. Parallelism needs multiple cores and helps CPU-bound work.

The search server



Same shape as our uppercase server, more work per request

- An index maps a word to the documents that contain it
- It is sharded across files, because it does not fit in one
- A query is several words, so it is several lookups, then a merge

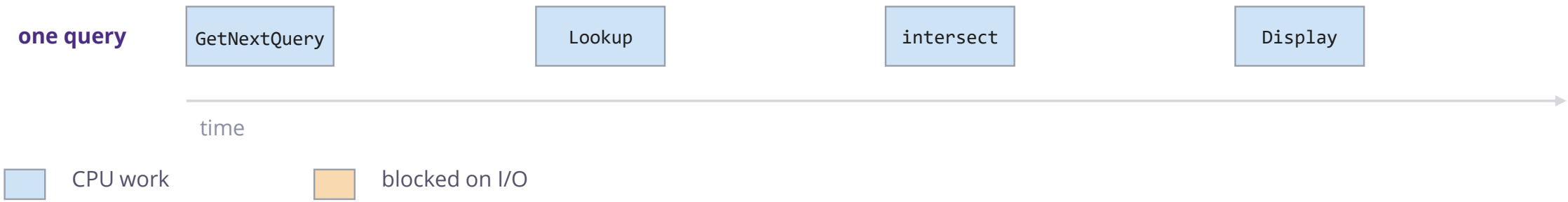
The sequential server

```
1  doclist Lookup(string word) {
2    hitlist = file.read(hash(word));   // disk
3    foreach (hit in hitlist)
4      doclist.append(file.read(hit));  // disk
5    return doclist;
6  }
7
8  int main() {
9    SetupServerToReceiveConnections();
10   while (true) {
11     words = GetNextQuery();          // network
12     results = Lookup(words[0]);
13     foreach (word in words[1..n])
14       results = results.intersect(Lookup(word));
15     Display(results);                // network
16   }
17 }
```

Start with the sequential version

- It is short, and you can read it top to bottom
- It is easy to test and easy to reason about
- It gives the right answer every time
- The four highlights are the only lines that touch a device
- **Write this version first.** Then measure it.

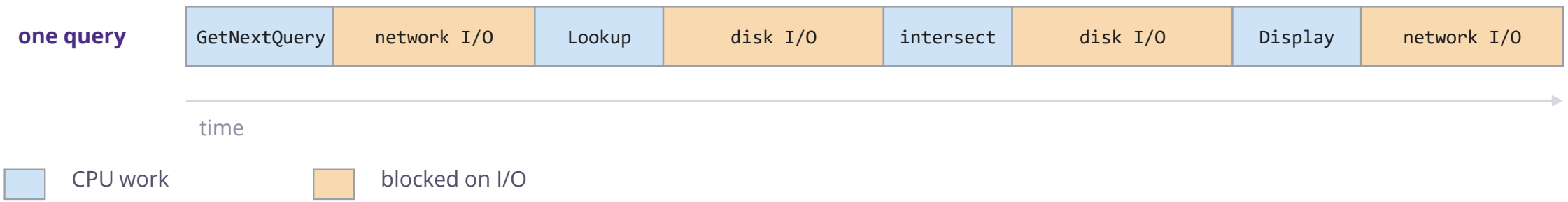
Where the time goes



CPU work

- Parse the query, hash a word, intersect two lists
- Format the results and hand them back
- This is every instruction the server executes for one query

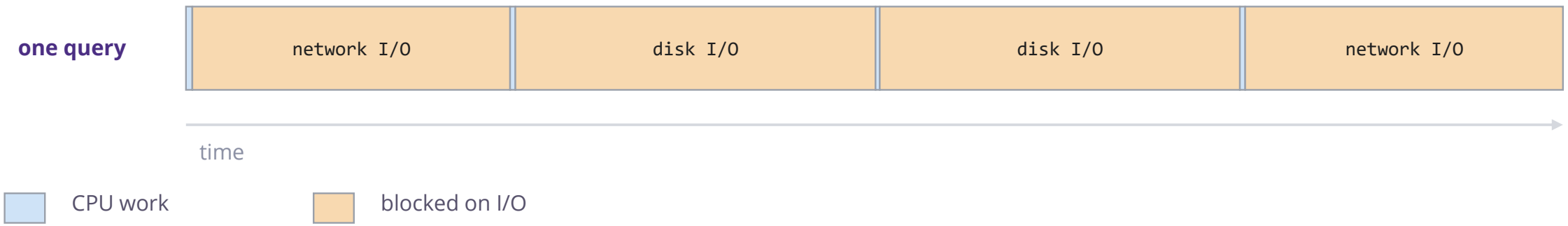
Where the time goes



Five of the eight blocks are waiting

- Reading the query waits for the client to send it
- Each index lookup waits for the disk
- Writing the result waits for the network to take it
- During all five, this program is doing nothing

The same query, to scale



The purple slivers are the CPU blocks from the last slide! They are puny!

Most of the request is waiting

- The CPU is idle for the overwhelming majority of a request
- Only one device is busy at a time, and usually none of them are
- Optimising the intersect loop barely changes response time

Latency numbers

operation	roughly	if one instruction took one second
one CPU instruction	1 ns	1 second
read from RAM	100 ns	under 2 minutes
read from an SSD	16 us	about 4 hours
seek on a spinning disk	2 ms	about 3 weeks
round trip inside a datacenter	0.5 ms	about 6 days
round trip across an ocean	150 ms	about 5 years

Numbers from Jeff Dean's "Numbers Everyone Should Know". The exact values age; the ratios do not.

Three queries

Three clients send a query at the same moment. Which resource is saturated?

```
1  // three clients, one thread of execution
2  while (true) {
3    words = GetNextQuery();  // ?
4    ...
5  }
```

A. the CPU

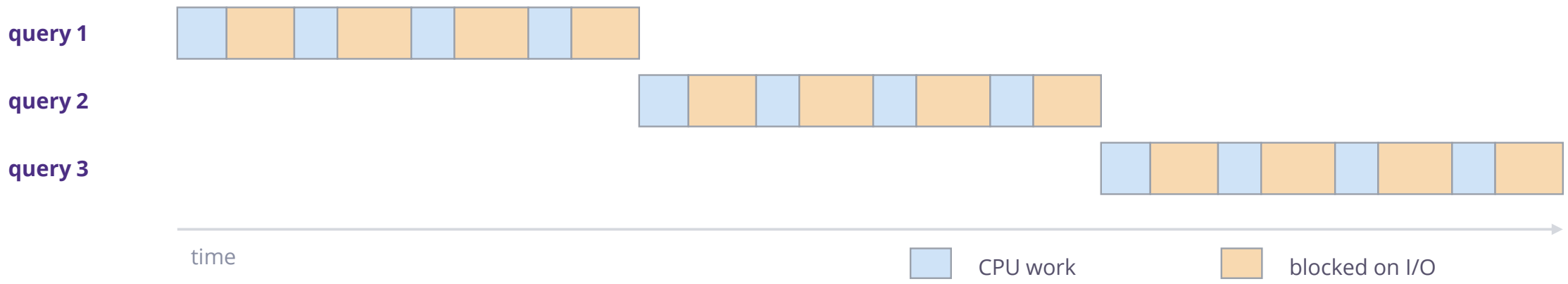
B. the disk

C. the network

D. none of them

Predict first: Slow is not the same as busy.

The resources are mostly idle



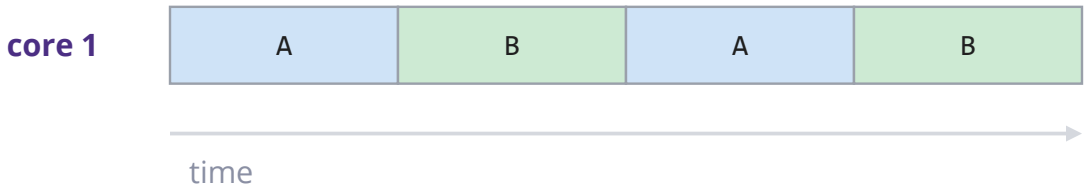
Three requests still run one at a time

- The CPU is idle for almost all of all three rows
- At most one device is in flight at any moment
- Query 3 has not started, and the machine has done nothing

Concurrency and parallelism

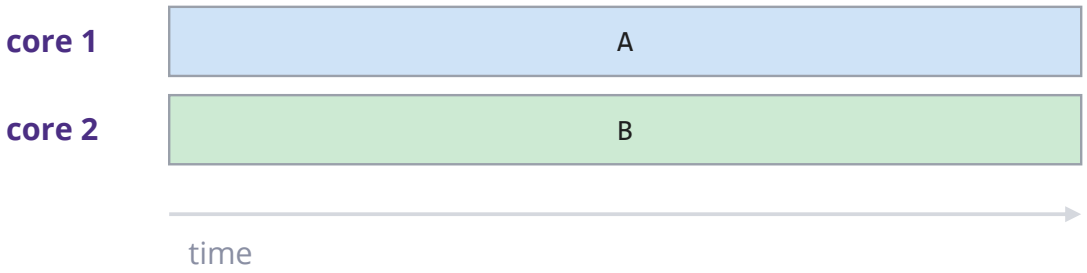
`concurrency`

Several tasks make progress during the same interval.



`parallelism`

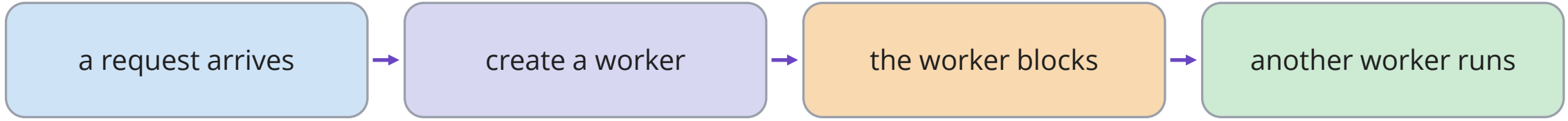
Several instructions execute at the same instant.



Concurrency can improve an I/O-bound server on one core. Parallelism needs multiple cores and helps CPU-bound work.

Workers

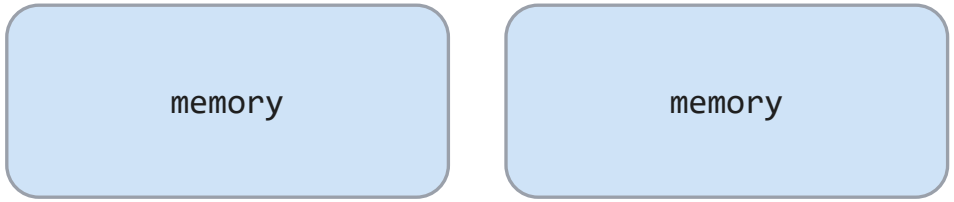
When a request arrives, hand it to a worker. The worker reads, looks up, and replies. When it blocks, the operating system runs a different worker.



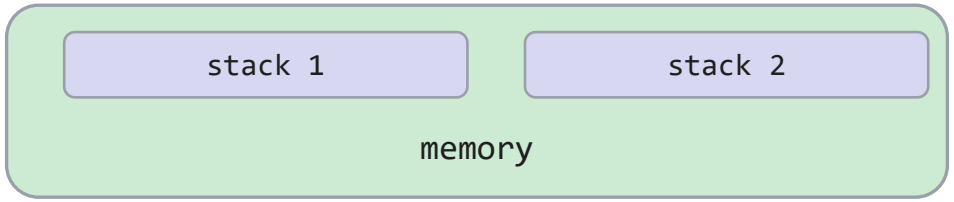
The request-handling code stays the same

- The lookup code, the intersect code, and the reply code stay put
- We are changing who runs them, not what they do
- **The remaining choice:** what is a worker made of?

Processes and threads



fork(): two of everything



pthread_create(): one extra stack

	process	thread
cost to create	high	low
shares the heap	no	yes
a crash takes down	one request	the server
needs locks	no	yes

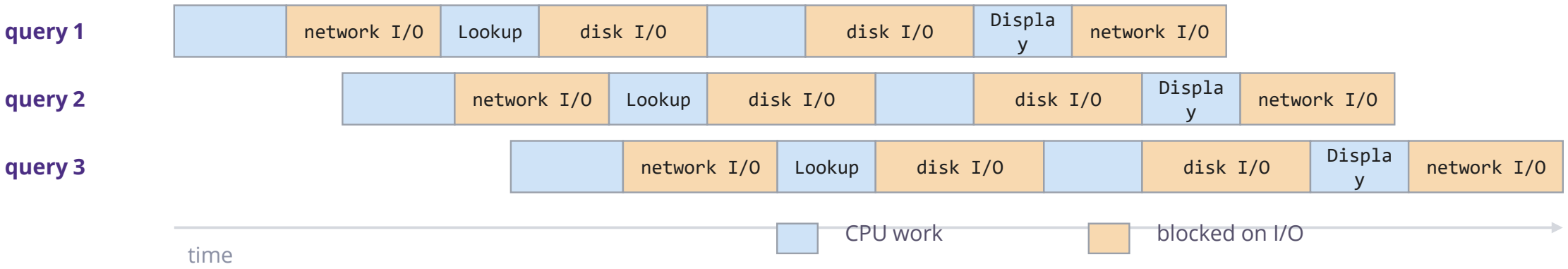
Event loops

```
1  fd_group.register(listen_fd, AcceptClient);
2  while (true) {
3    event = wait_for_any_ready_fd(fd_group);
4    event.callback(event.fd);      // never blocks
5  }
6
7  void AcceptClient(int listen_fd) {
8    int fd = accept(listen_fd, nullptr, nullptr);
9    fd_group.register(fd, HandleRequest);
10 }
```

One thread, non-blocking callbacks

- Put every descriptor in non-blocking mode
- Ask the kernel which ones are ready, with poll or epoll
- Run a small callback for each ready one, then loop
- **The cost:** sequential logic is split across callbacks

Overlapping the waiting



The same work, overlapped

- Query 3 finishes in a fraction of the time it used to
- Several I/O requests are in flight at once
- The operating system schedules all of this. We only had to create workers.

Threads and processes

Which of these statements are true?

A. each thread gets its own stack and its own heap

B. each thread runs on a separate core

C. threads in one process share open file descriptors

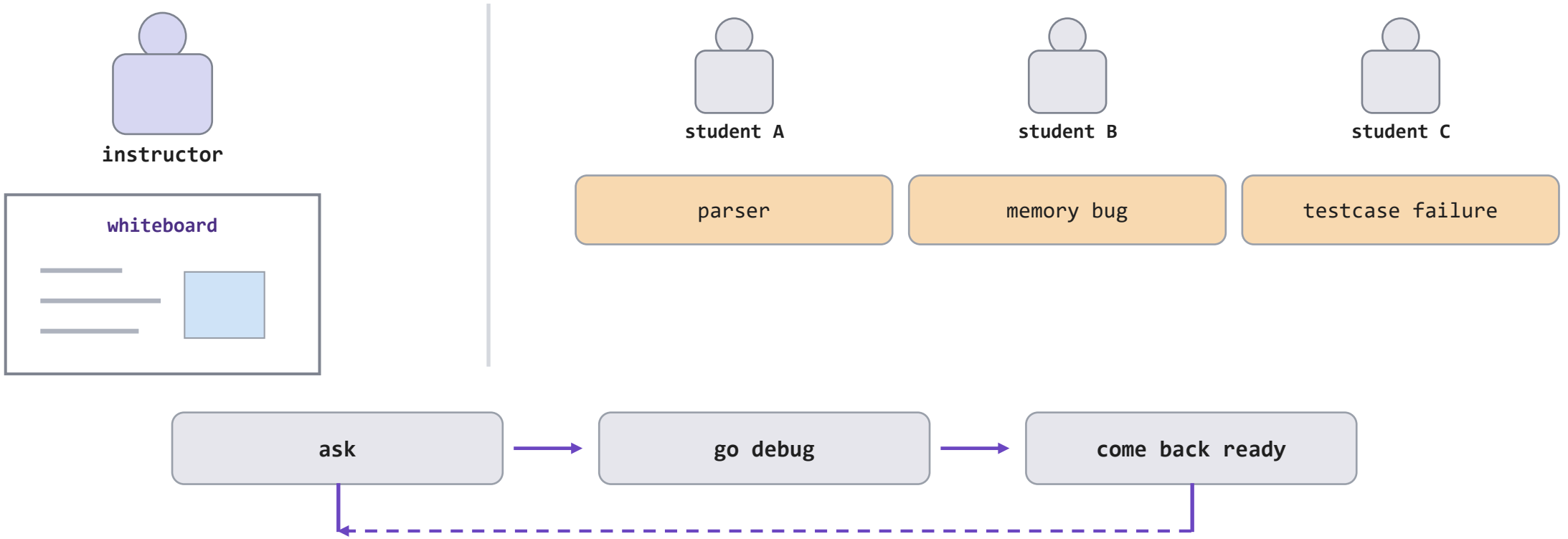
D. a parent and its forked child start with the same open descriptors

E. both C and D

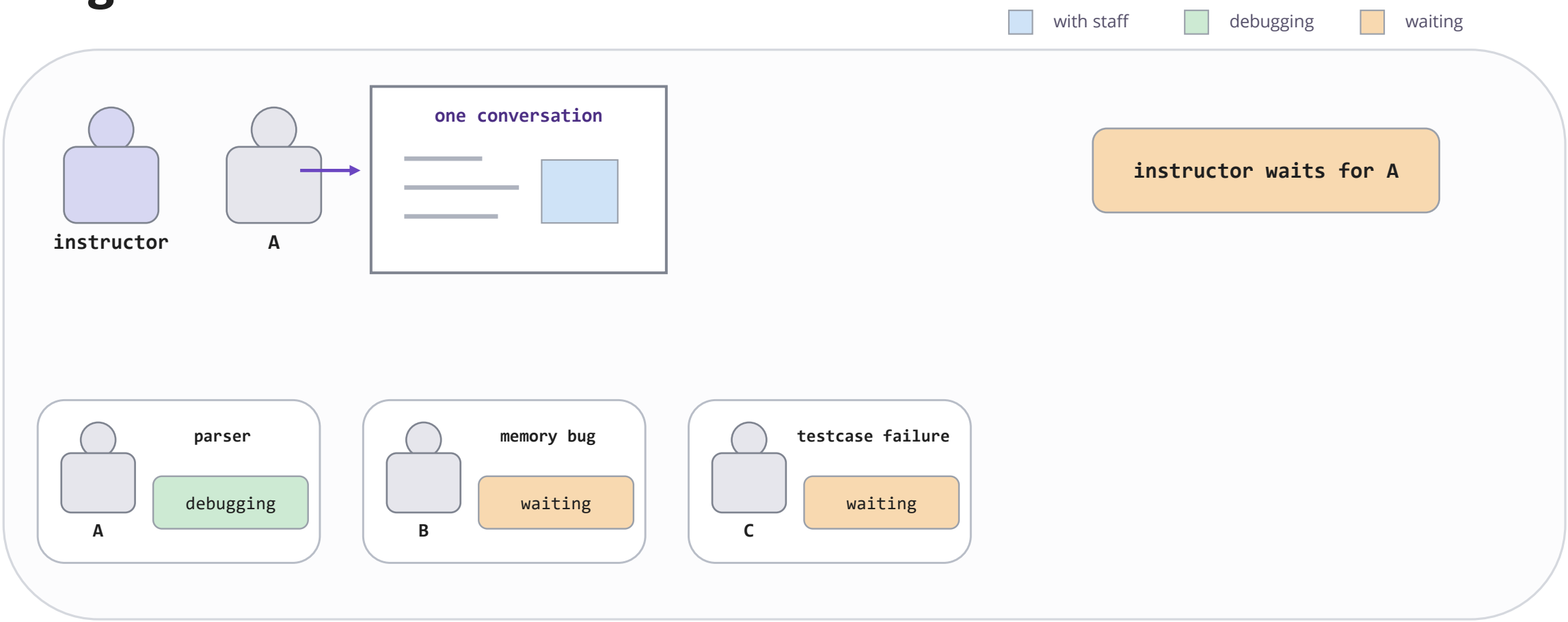
Predict first: Look back at the two diagrams before you answer.

Execution Models

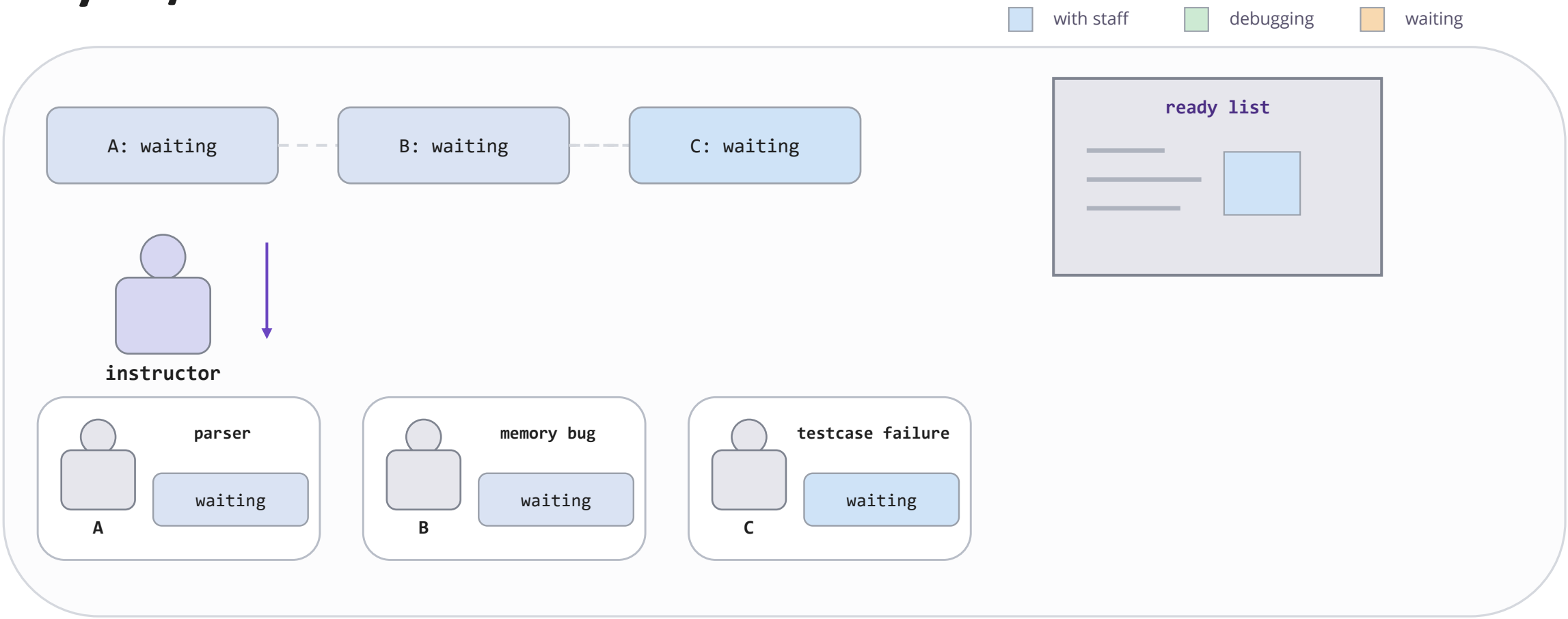
Office hours



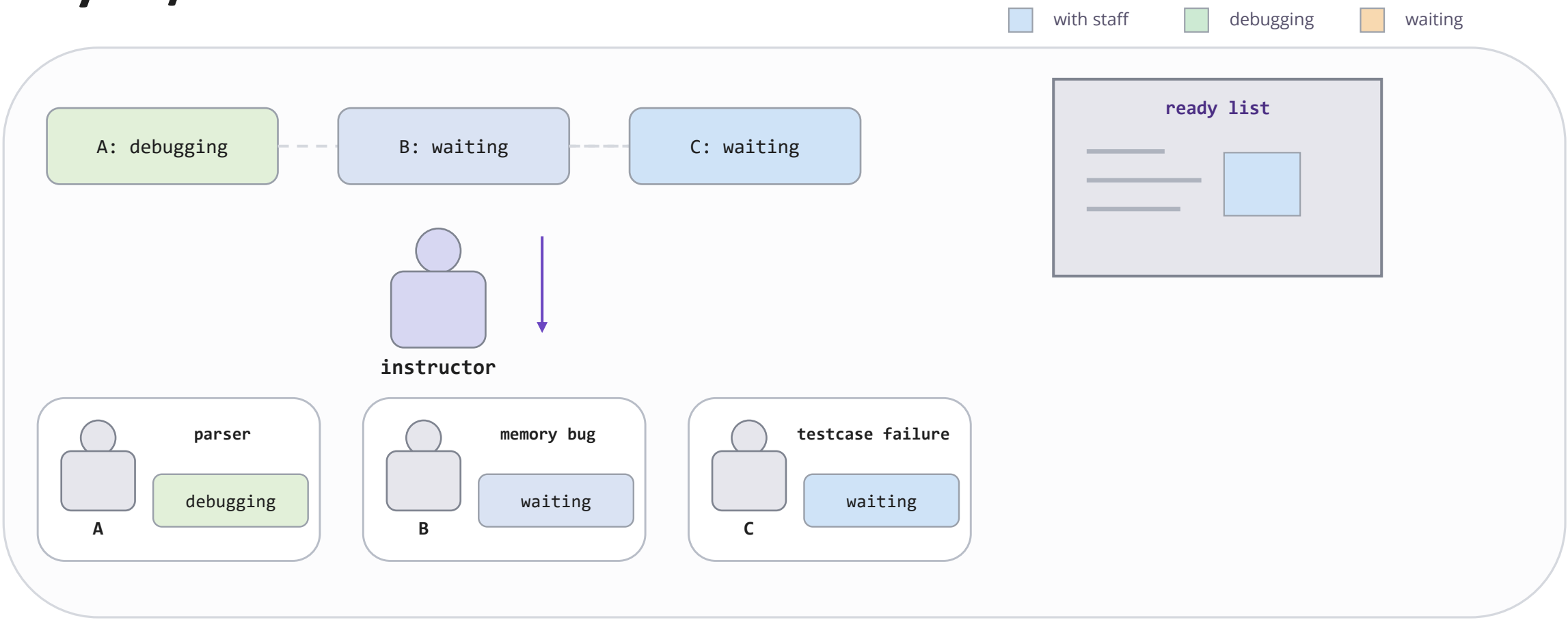
Single-threaded



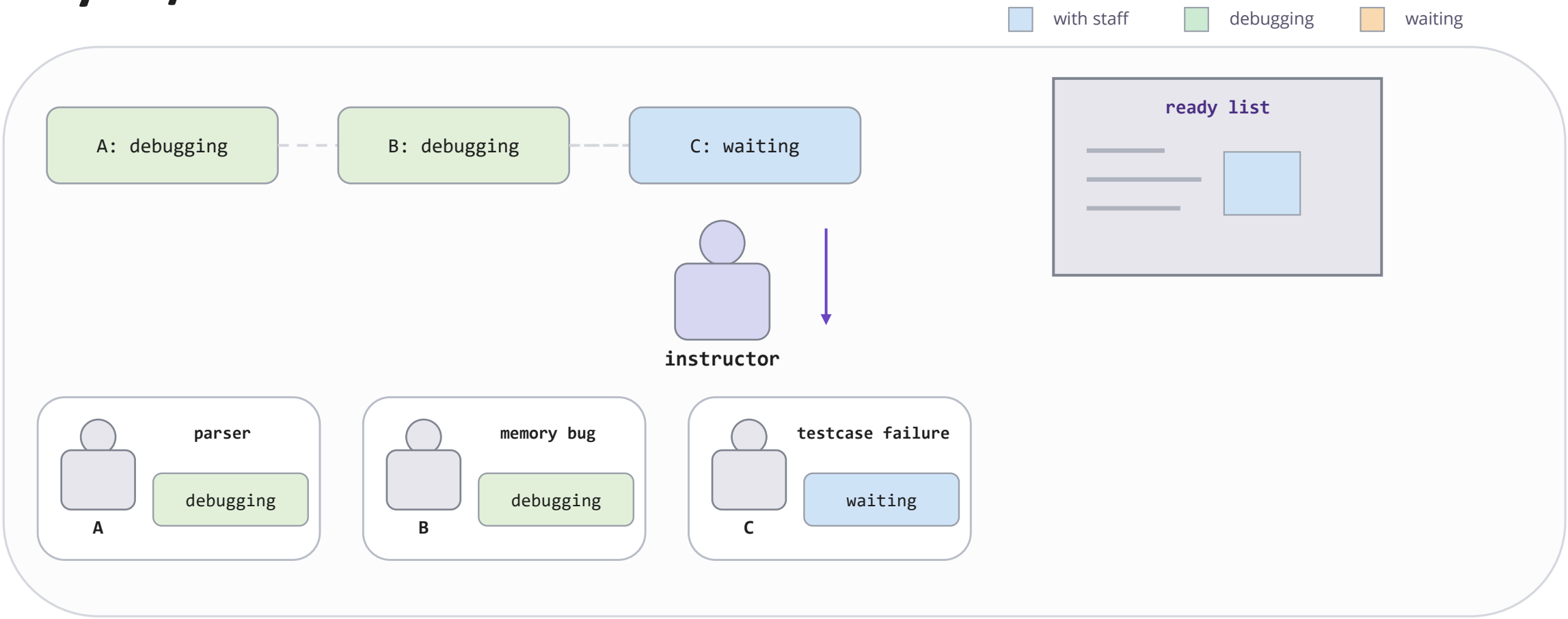
Async / event-driven



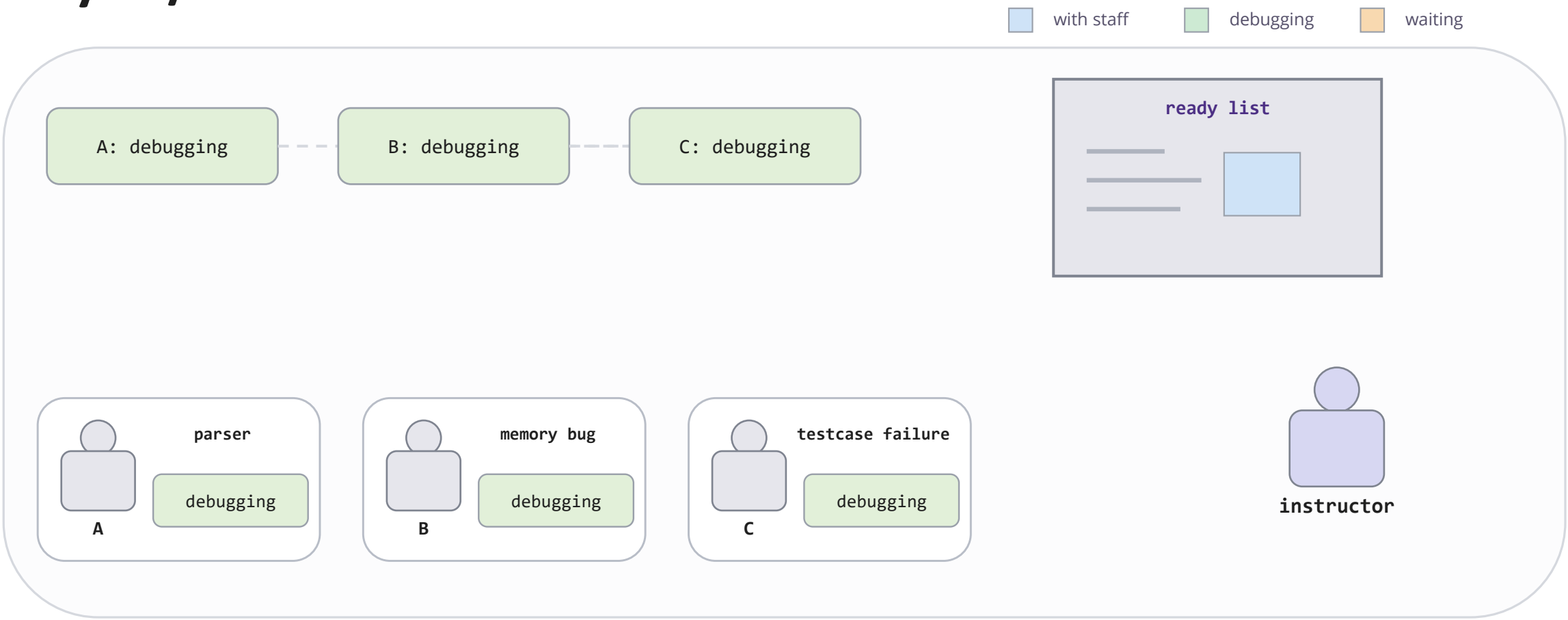
Async / event-driven



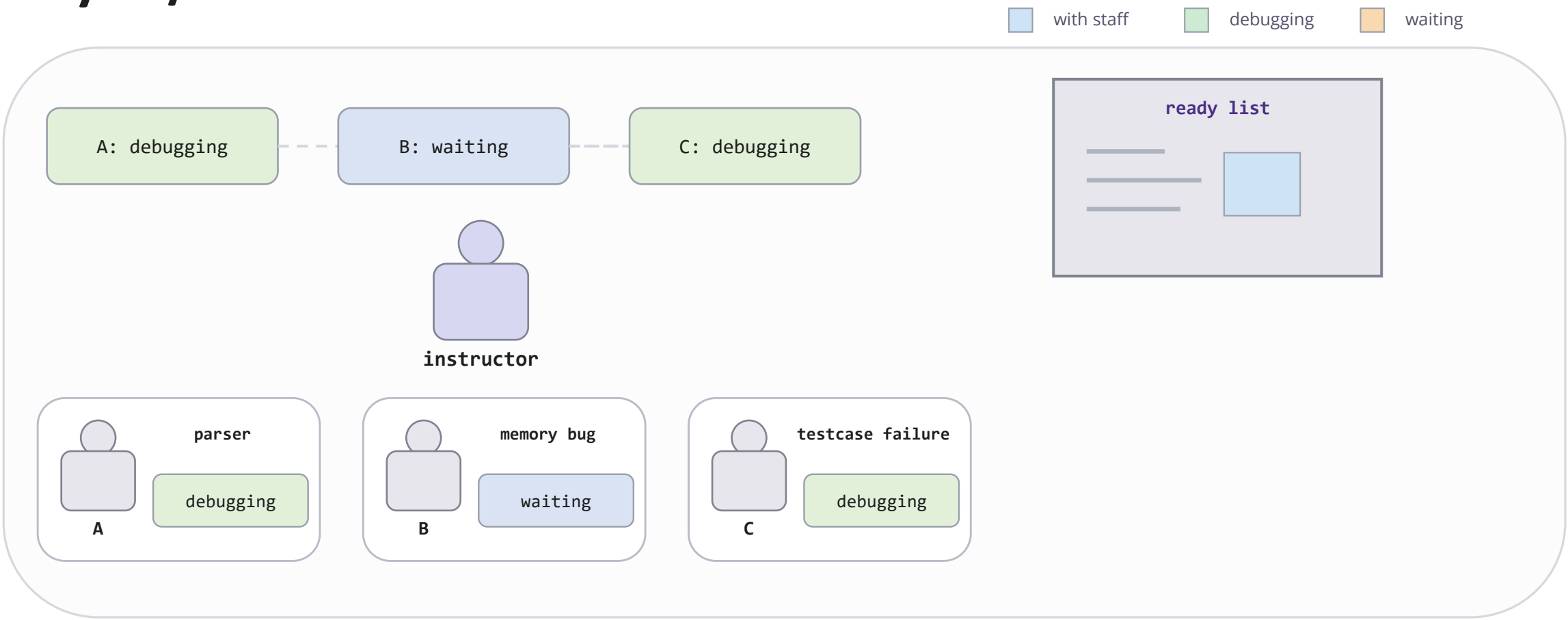
Async / event-driven



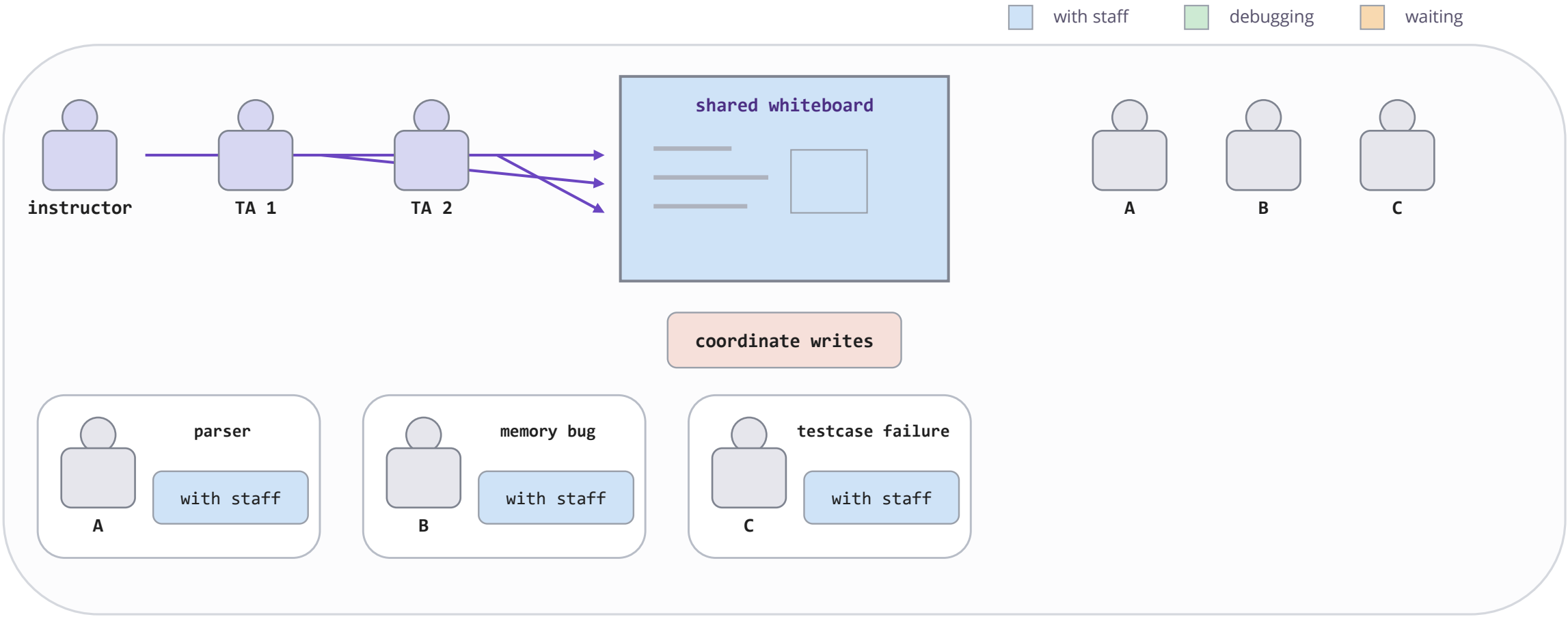
Async / event-driven



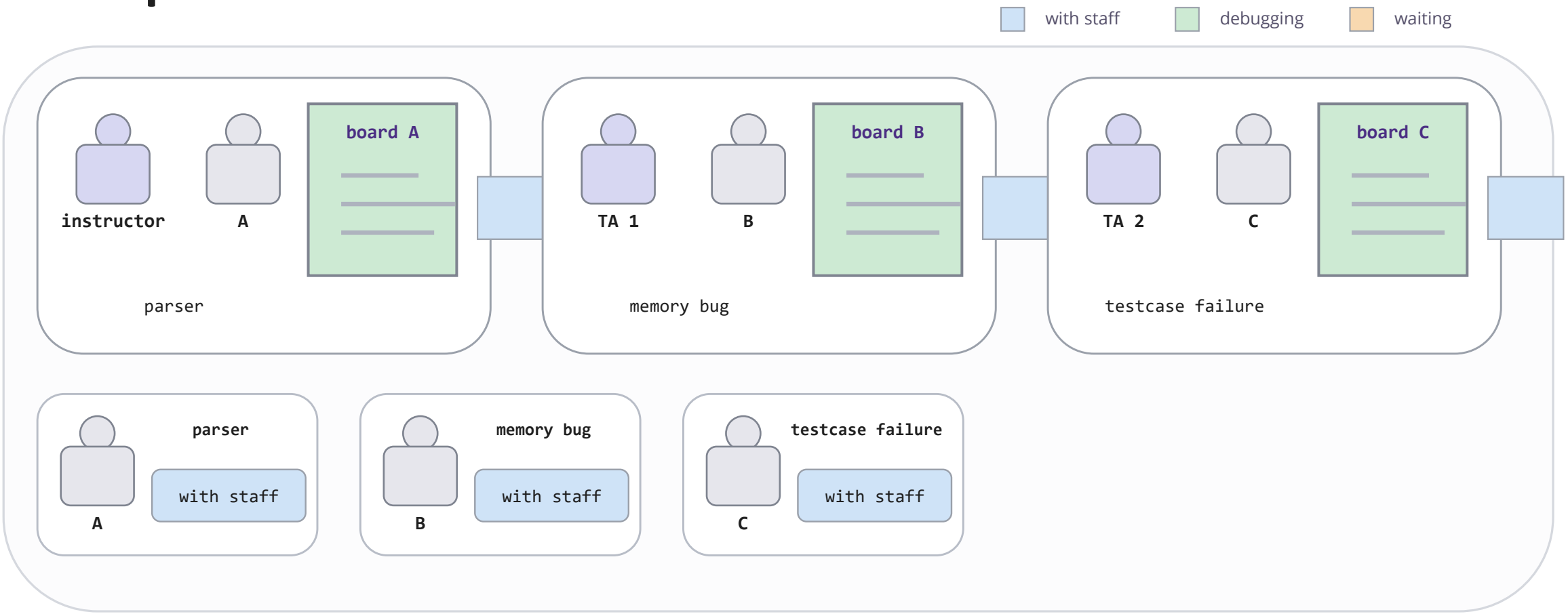
Async / event-driven



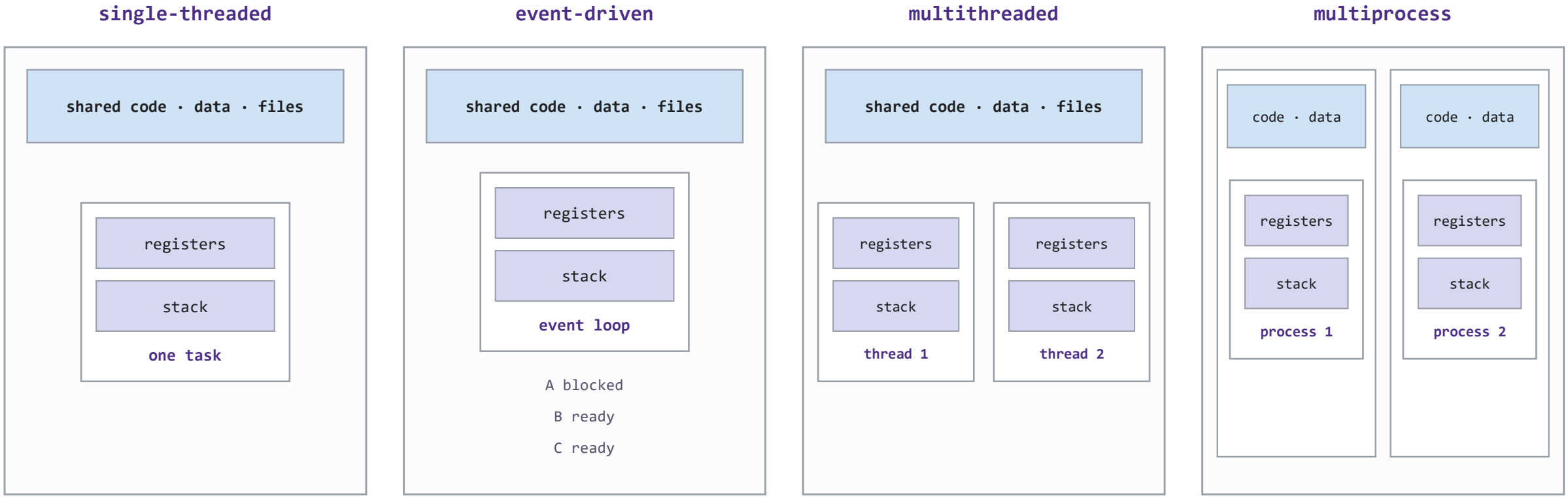
Multithreaded



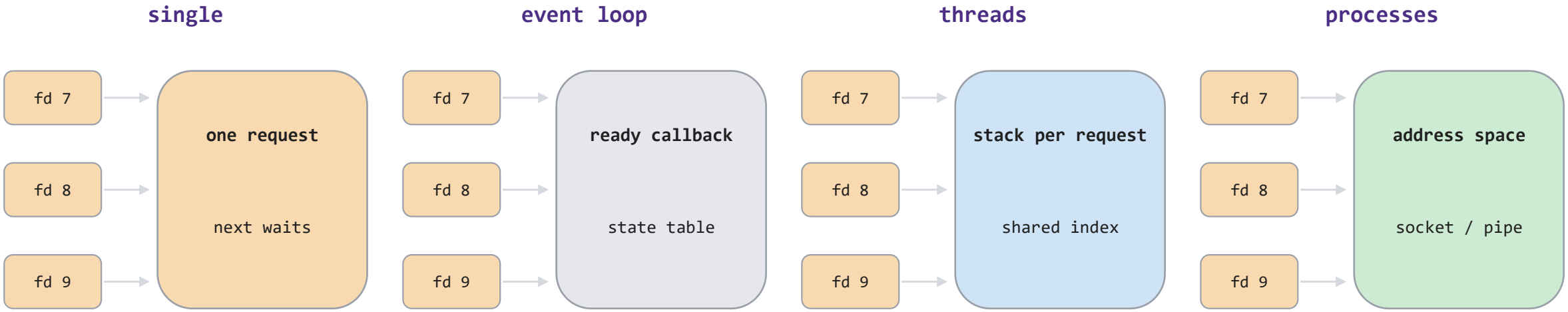
Multiprocess



Execution contexts and memory

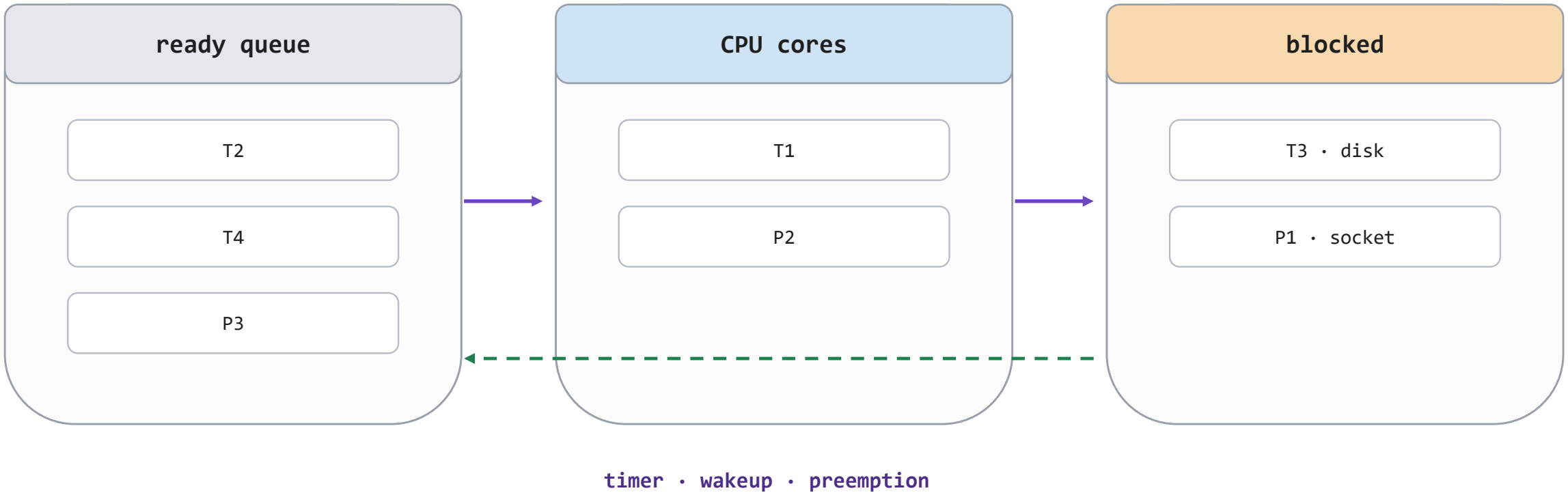


One server, four designs

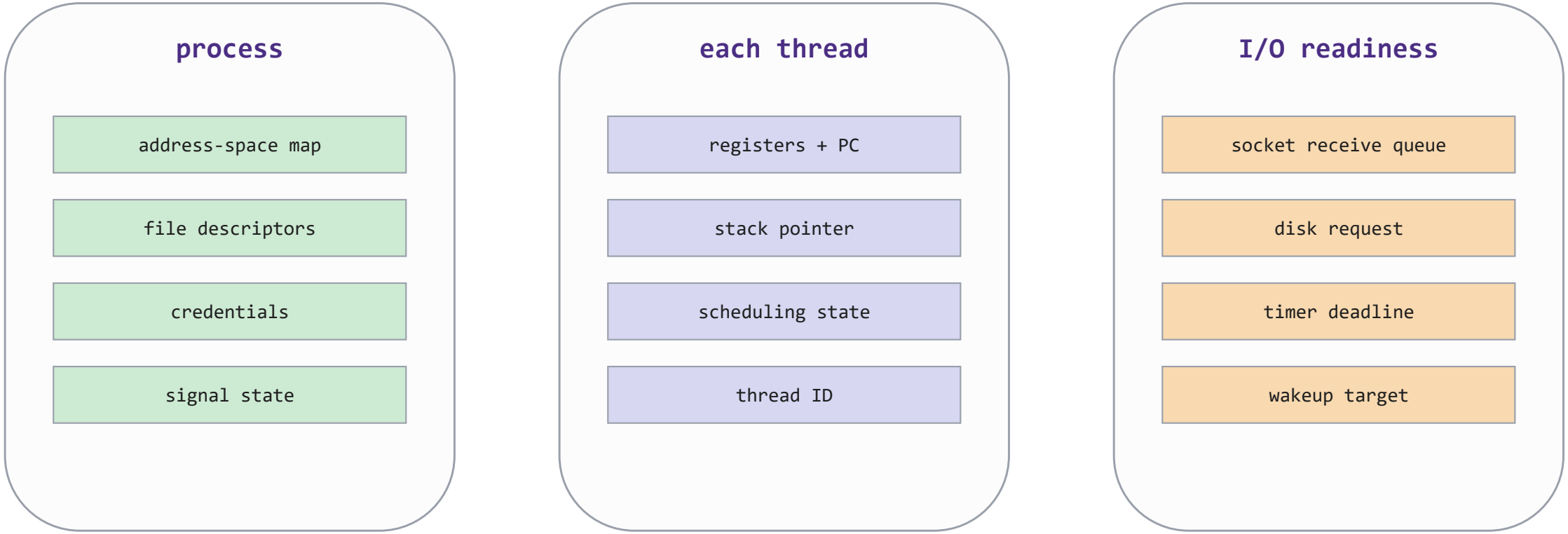


accept → read → compute → write

Ready, running, blocked



What the OS keeps



Synchronization

(with locks!)

The milk

The rule in the flat: if there is no milk, go and buy some.

```
1  if (!milk) {
2    buy milk
3  }
```

Living alone

- Check the fridge, see nothing, go to the shop
- Come back with one carton
- There is no concurrent roommate between check and purchase

The milk

The rule in the flat: if there is no milk, go and buy some.

```
1  if (!milk) {
2    buy milk
3  }
```

Living alone

- Check the fridge, see nothing, go to the shop
- Come back with one carton
- There is no concurrent roommate between check and purchase

Living with a roommate

- You check the fridge and leave
- They check the fridge and leave
- **Two cartons.** You both read the same state before either of you wrote to it.

The note

Does leaving a note on the fridge fix it?

```
1  if (!note) {
2    if (!milk) {
3      leave note;  buy milk;  remove note
4    }
5  }
```

A. yes, this is correct now

B. no, you could end up with no milk at all

C. no, you could still end up with two cartons

D. it works, but only if you are lucky

Predict first: Where can the other person be when you run line 1?

Checking and claiming

	you	your roommate	state
1	check note: none		no note, no milk
2		check note: none	no note, no milk
3	check milk: none		no note, no milk
4		check milk: none	no note, no milk
5	leave note, buy milk		milk
6		leave note, buy milk	two cartons

Checking the note and writing the note are two separate steps, so we solved the problem by adding a second copy of it.

Locks

```
1  // non-critical code
2  lock.acquire();      // wait here until it is mine
3    total++;           // inside the critical section
4  lock.release();      // let the next one in
5  // non-critical code
```

Mutex implementations combine hardware atomic operations with operating-system support for sleeping waiters.

Two guarantees

- ❖ At most one thread is between acquire and release
- ❖ Acquire is atomic, so there is no gap like the note had
- ❖ A blocked thread sleeps, it does not spin the CPU
- ❖ Release lets a waiting thread acquire the mutex

Reminders

Assessments:

- Exercise 8 due Friday at 11:59pm
- Homework 4 released soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Final Exam
  - CSE2 G10 from 1:30-3:30pm on Tuesday, August 18
  - Makeup exam: 5-7pm on different days. Email coming soon!

Questions?