LECTURE 20 · CSE 333 · Summer 2026

More 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:

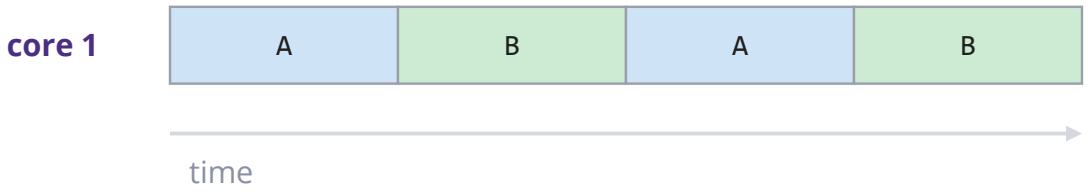
- Tim is doing a GPU lecture on Week 9 Friday
- Final Exam
  - CSE2 G10 from 1:30-3:30pm on Tuesday, August 18
  - Makeup exam: 5-7pm on different days. Email coming soon!

Review

Concurrency and parallelism

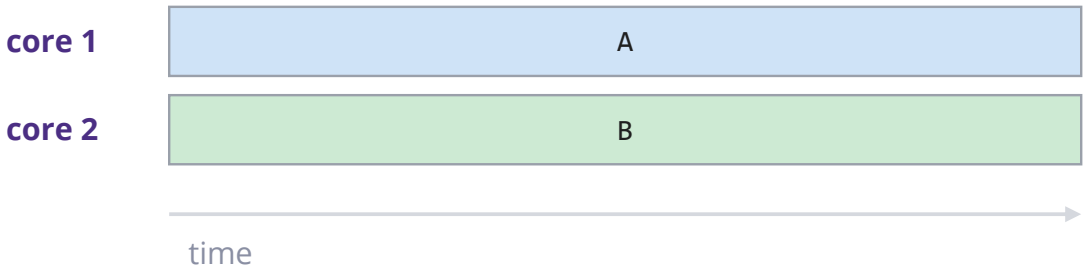
concurrency

Several tasks make progress during the same interval.



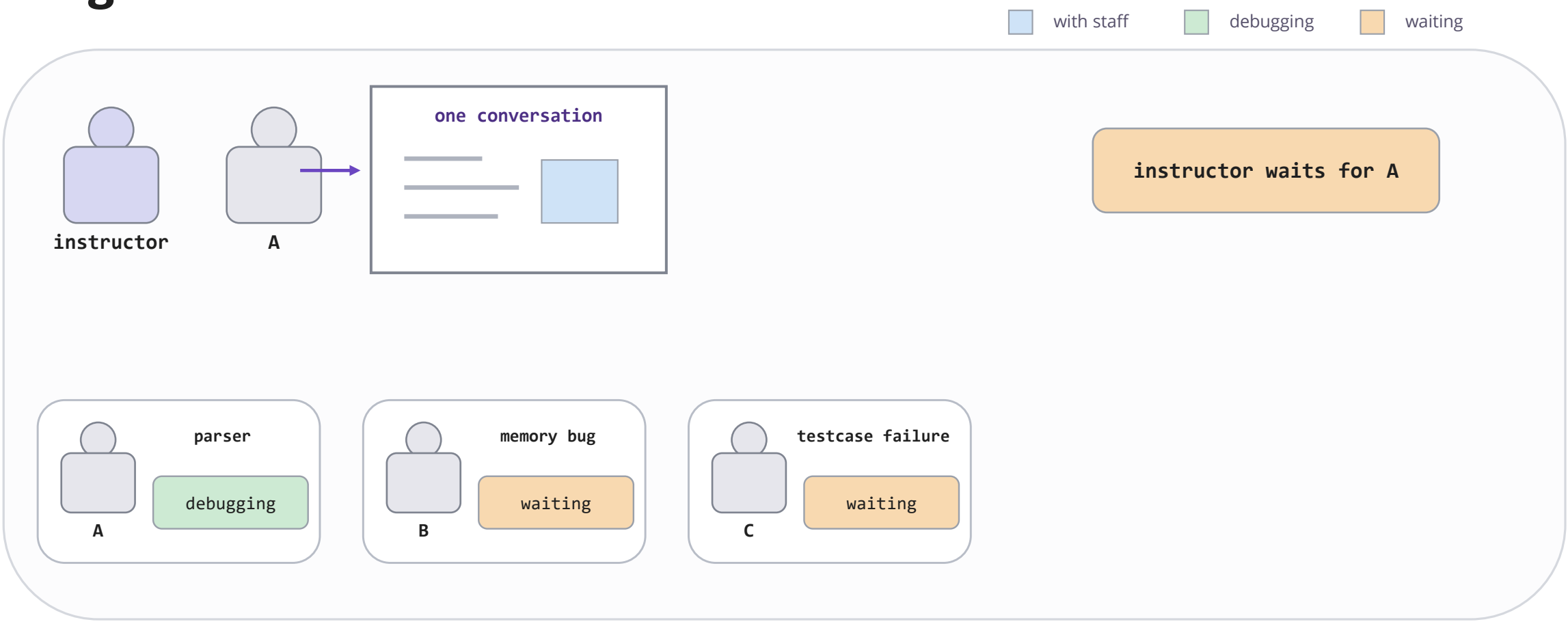
parallelism

Several instructions execute at the same instant.

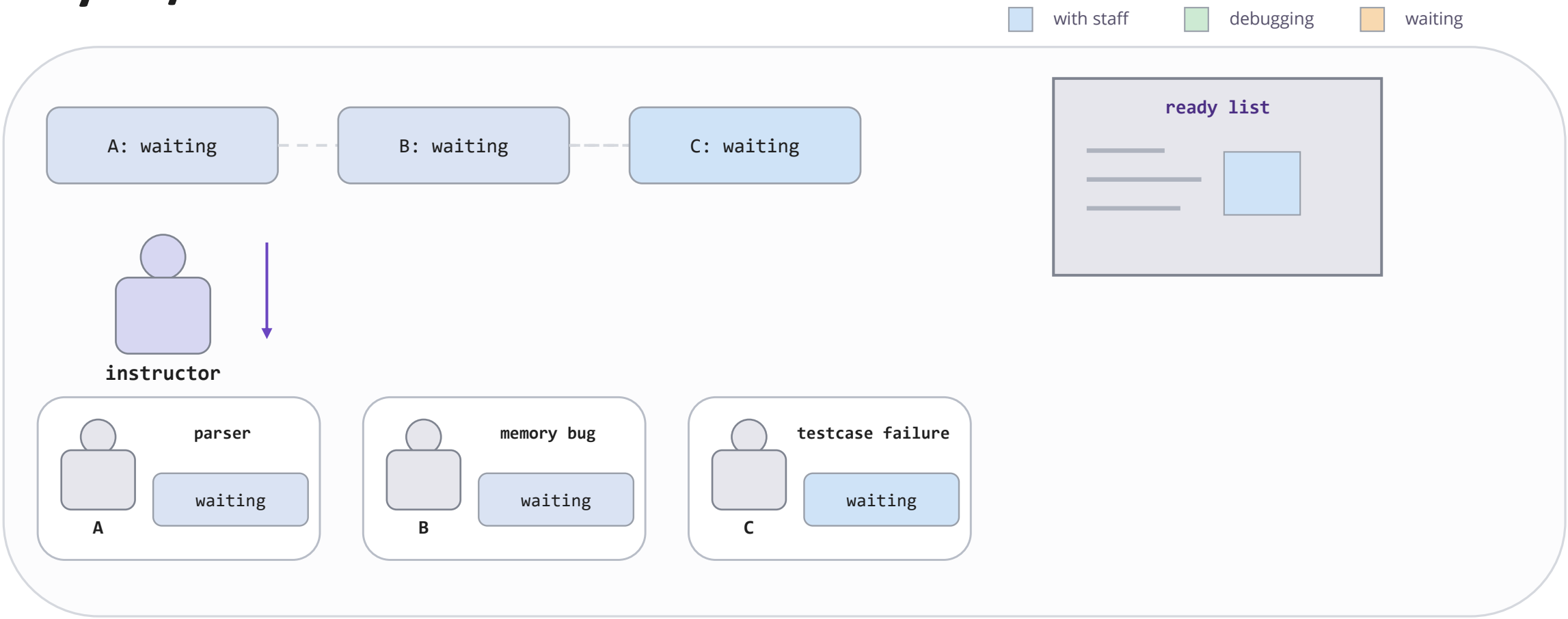


Concurrency can improve an I/O-bound server on one core. Parallelism needs multiple cores and helps CPU-bound work.

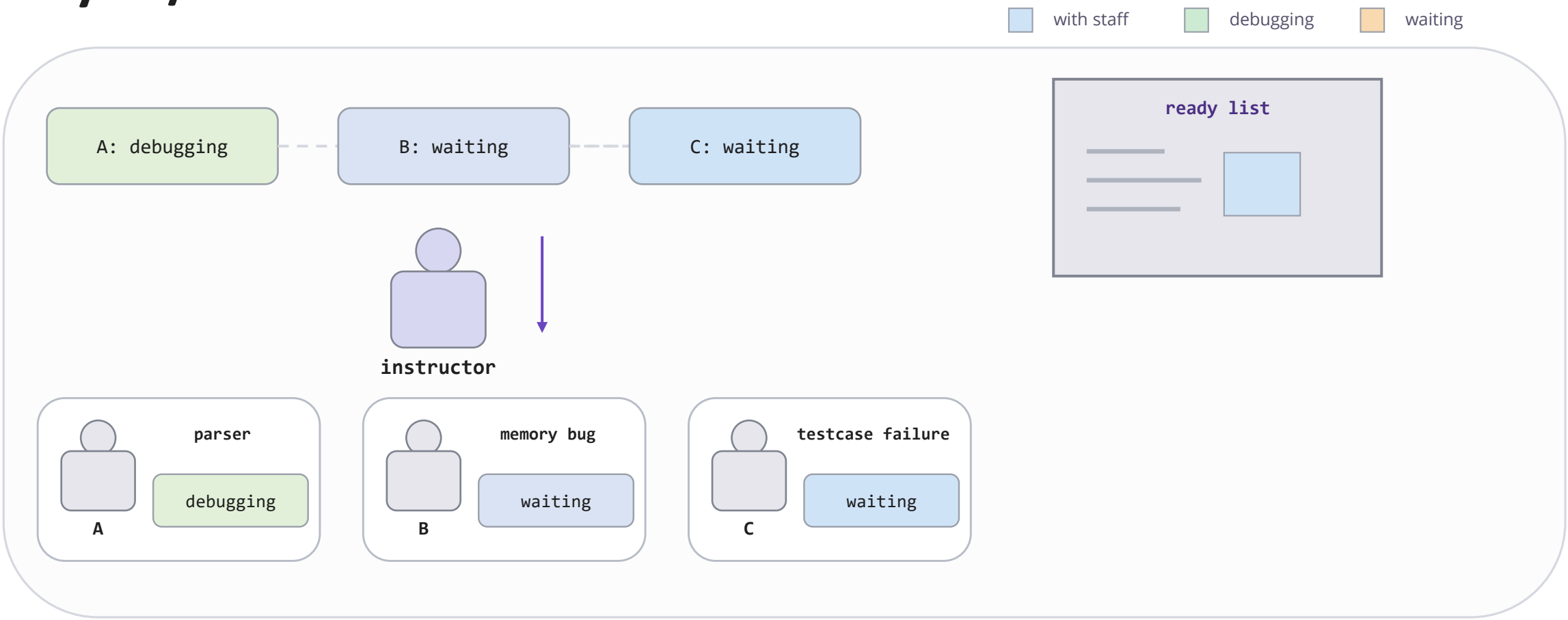
Single-threaded



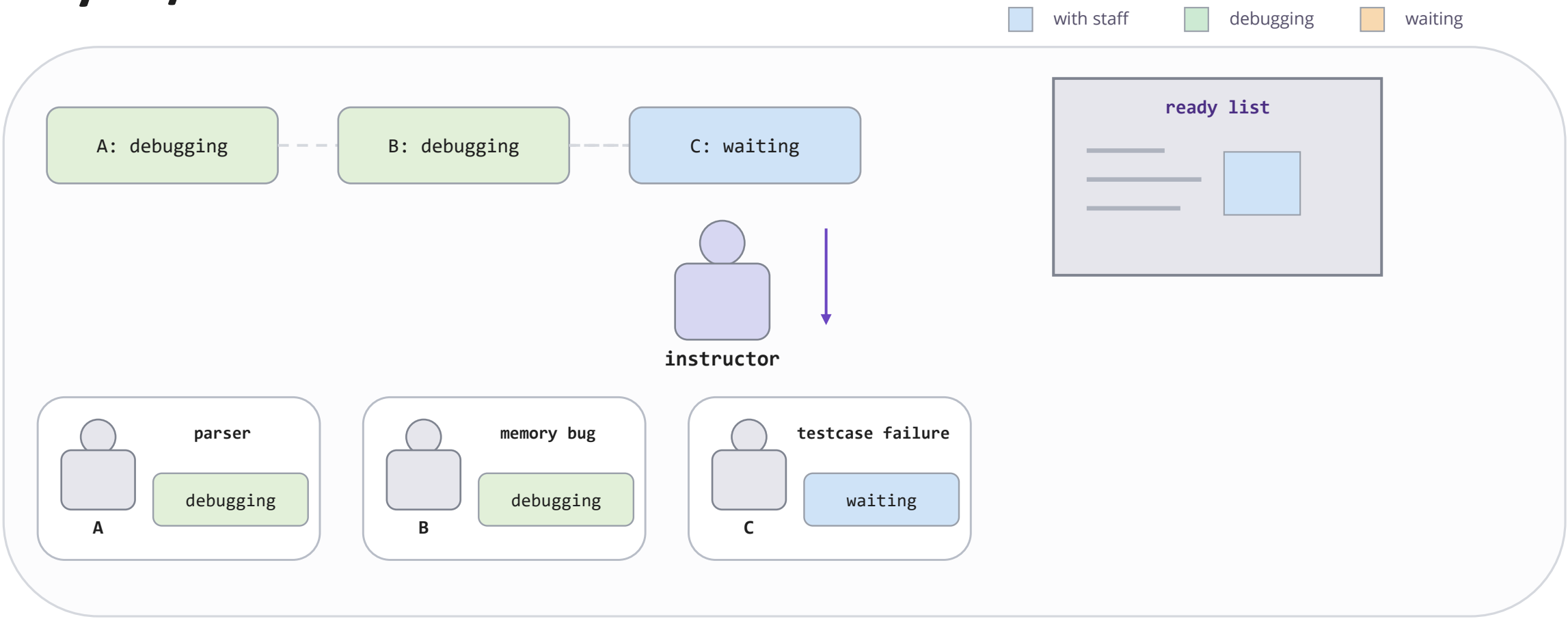
Async / event-driven



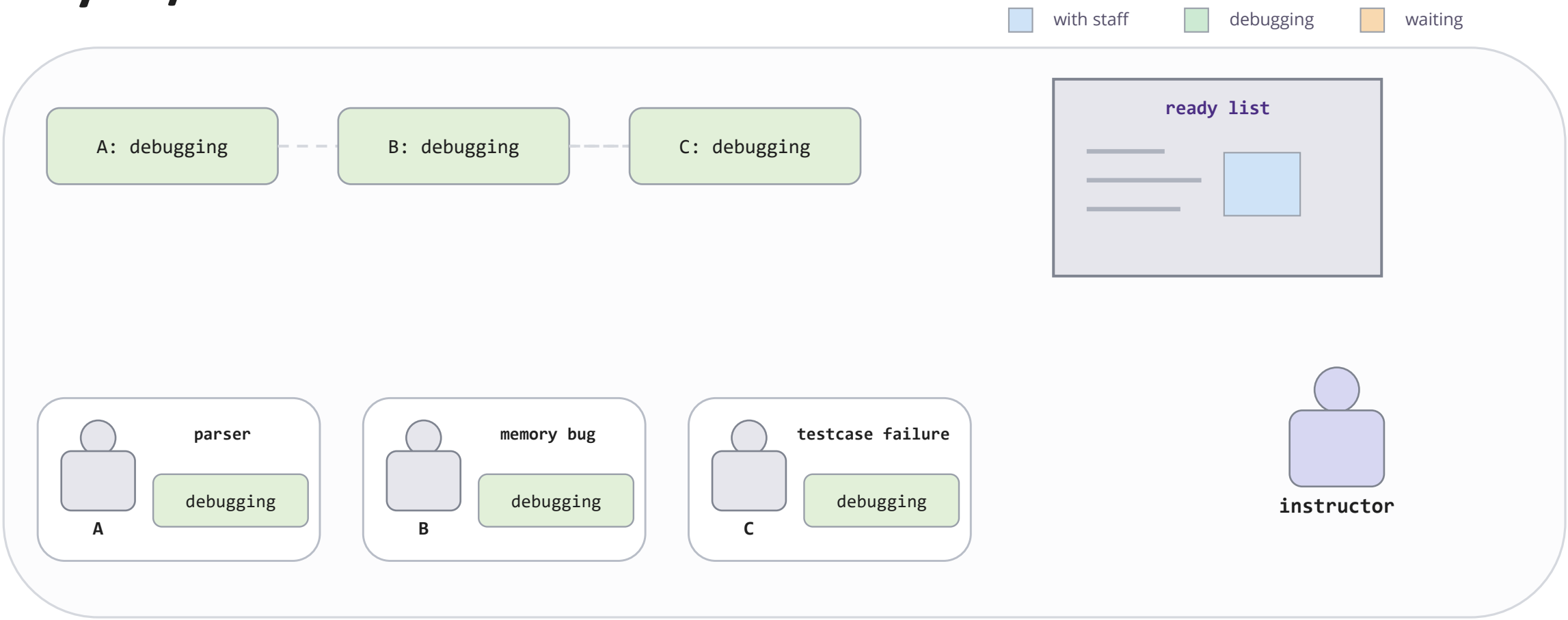
Async / event-driven



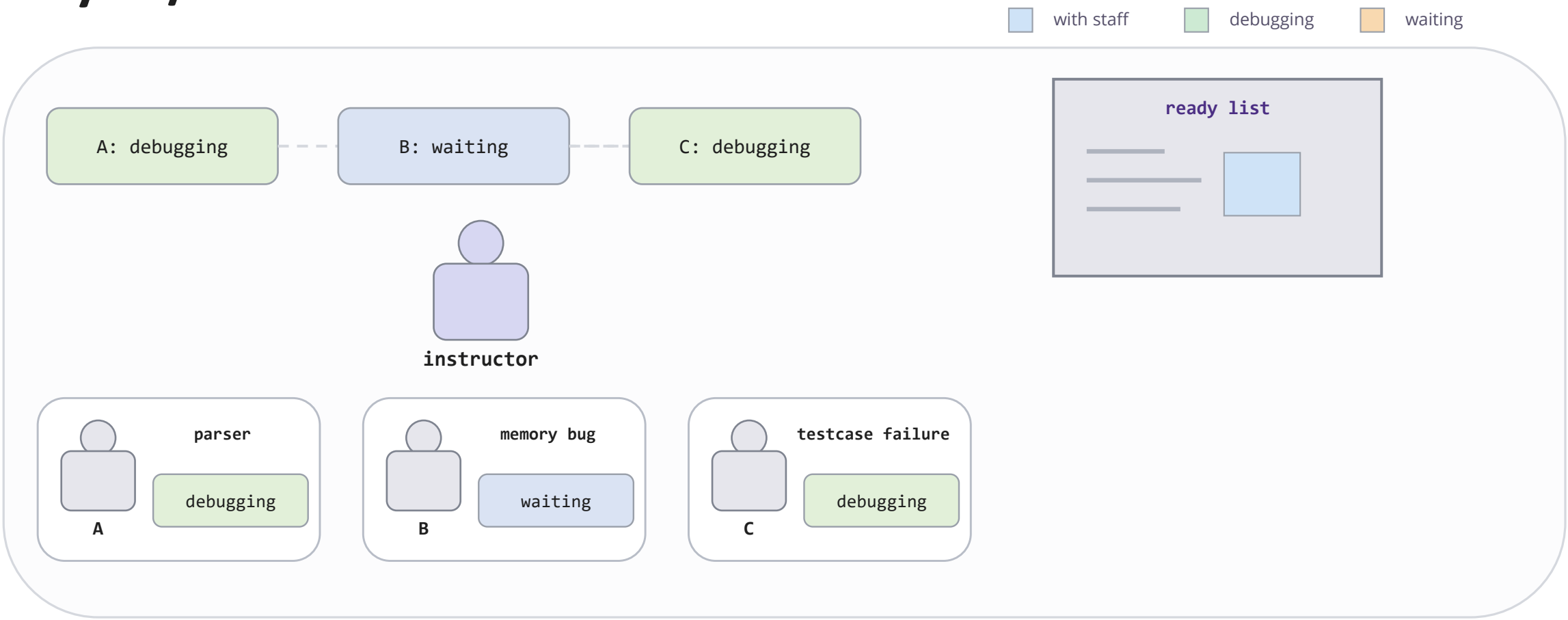
Async / event-driven



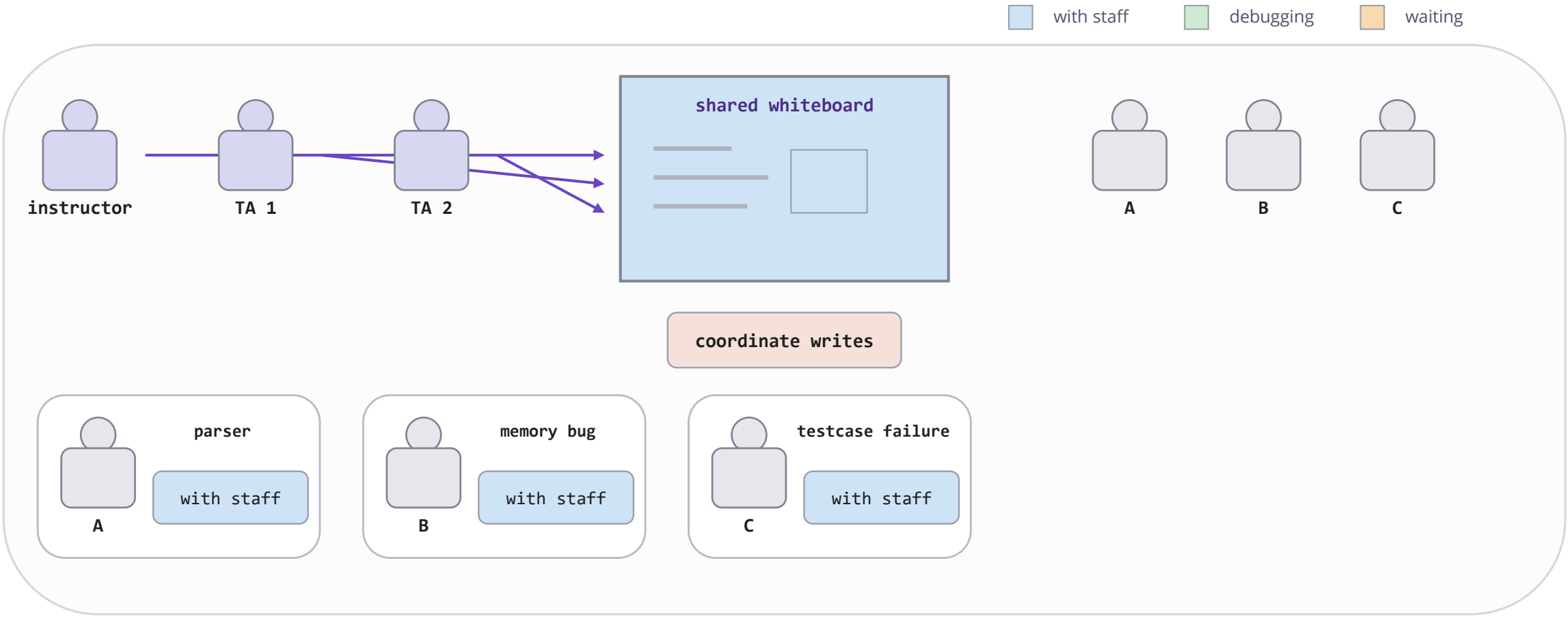
Async / event-driven



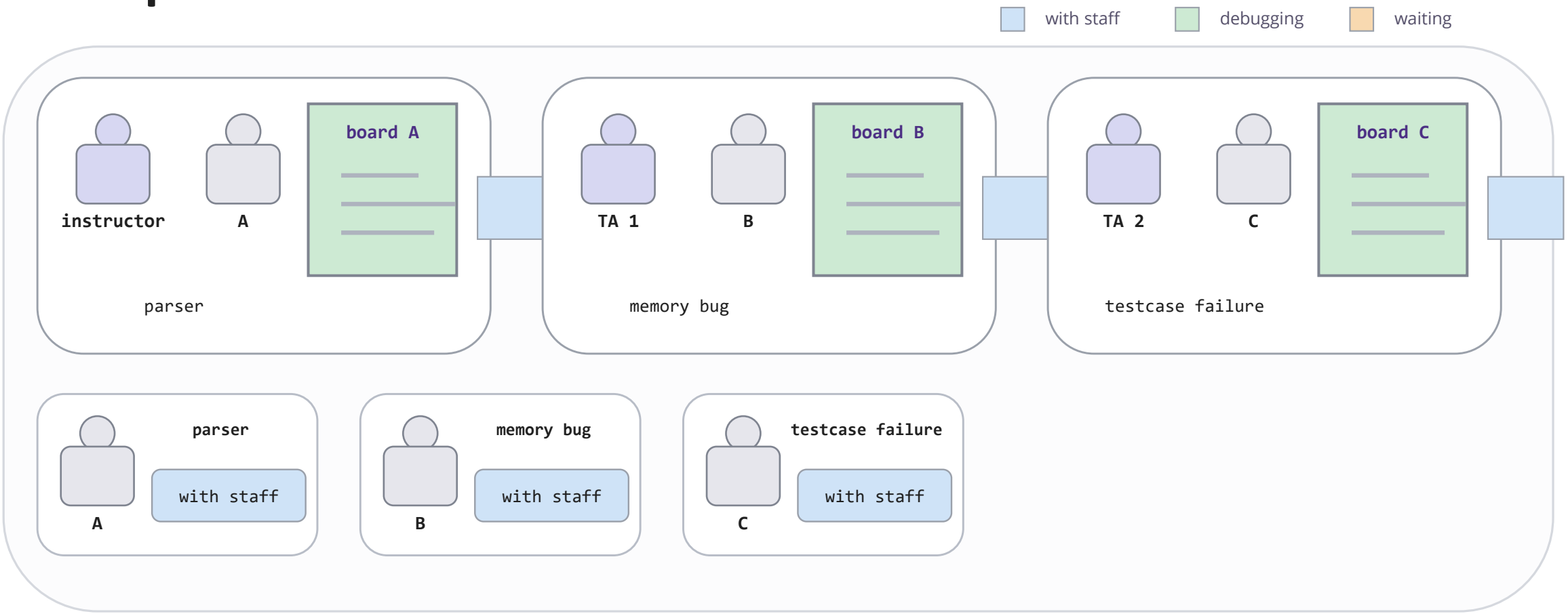
Async / event-driven



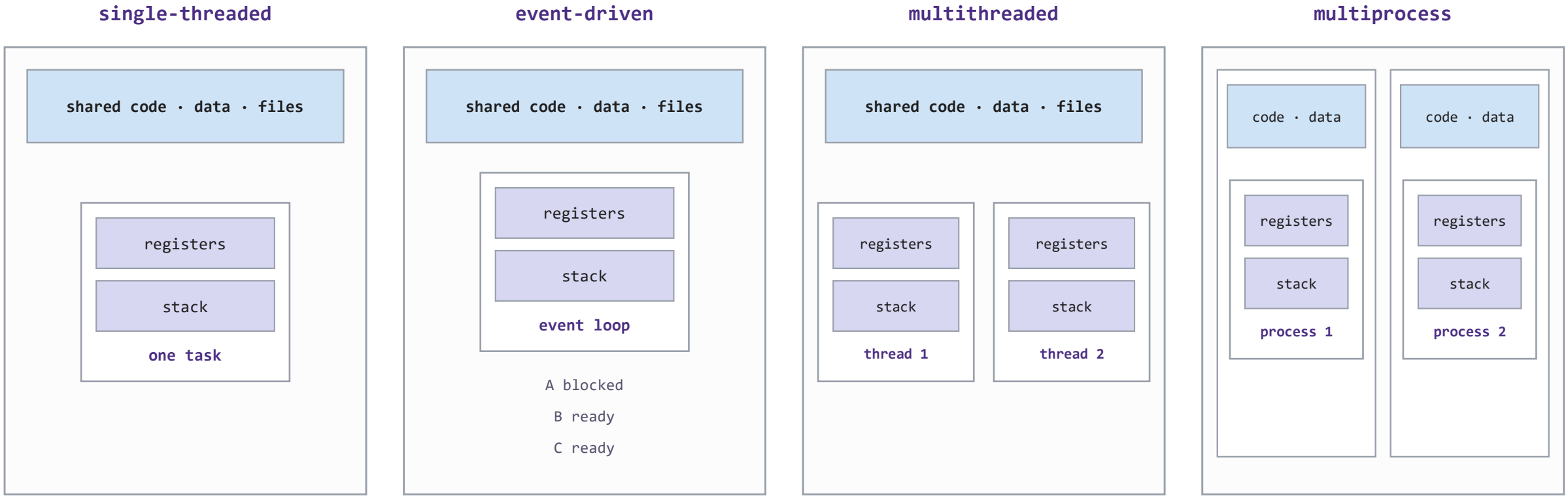
Multithreaded



Multiprocess

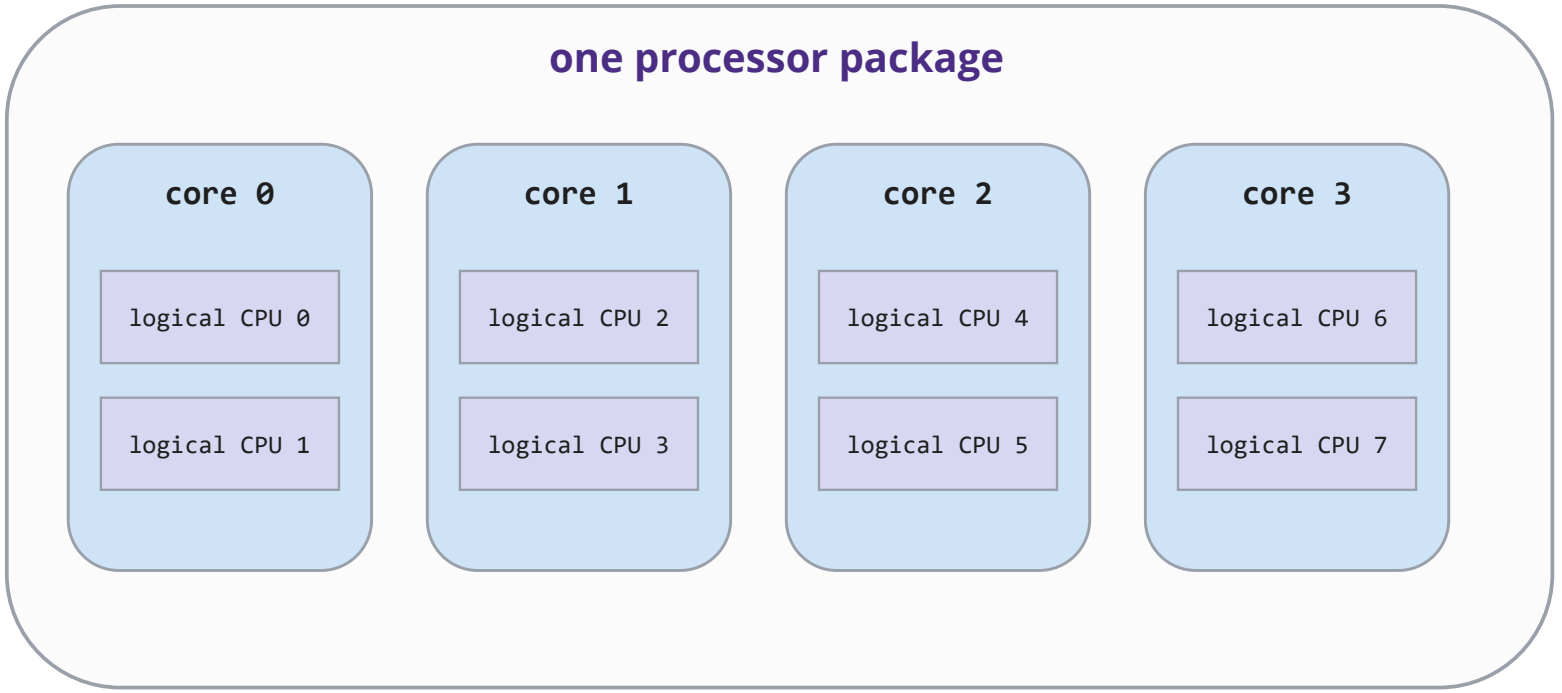


Execution contexts and memory



Execution Models

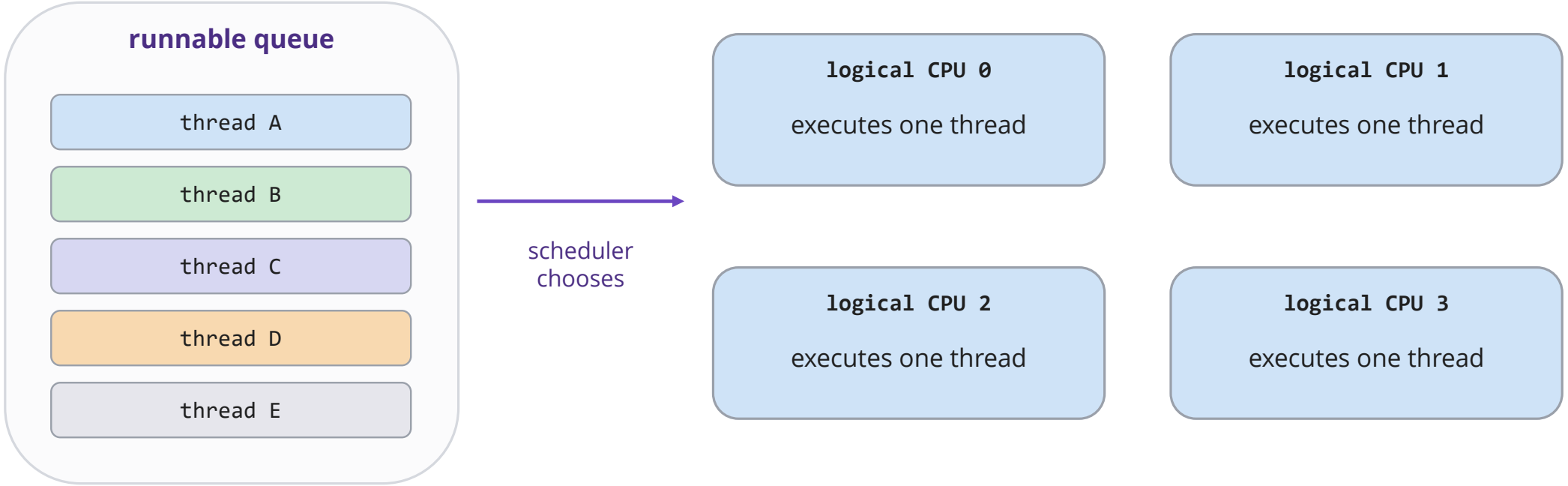
Packages, cores, and logical CPUs



Definitions

- A machine may have one or more processor packages
- Each package contains one or more physical cores
- A core may expose one or more logical CPUs
- Linux schedules software threads onto logical CPUs

The OS schedules runnable threads

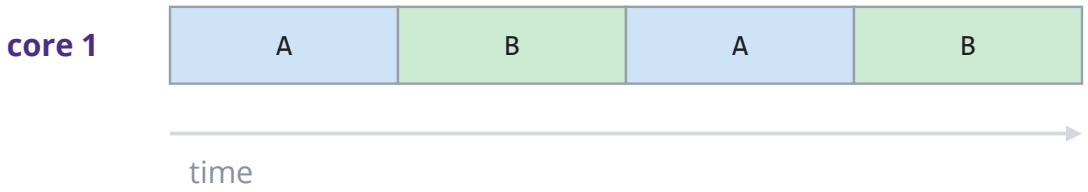


A timer interrupt can return a running thread to the queue. A blocked thread leaves the queue until its event completes.

Concurrency and parallelism

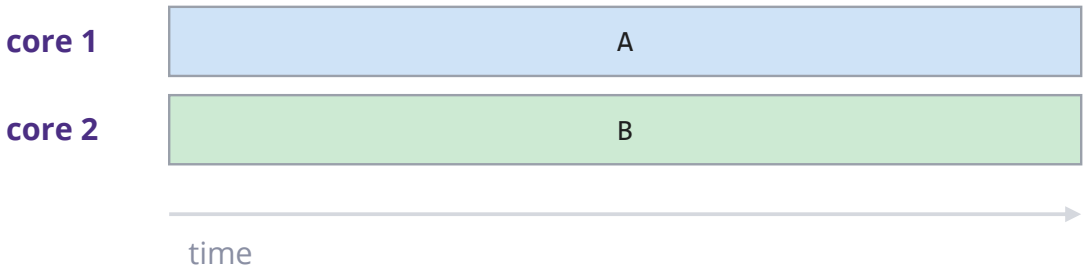
`concurrency`

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.

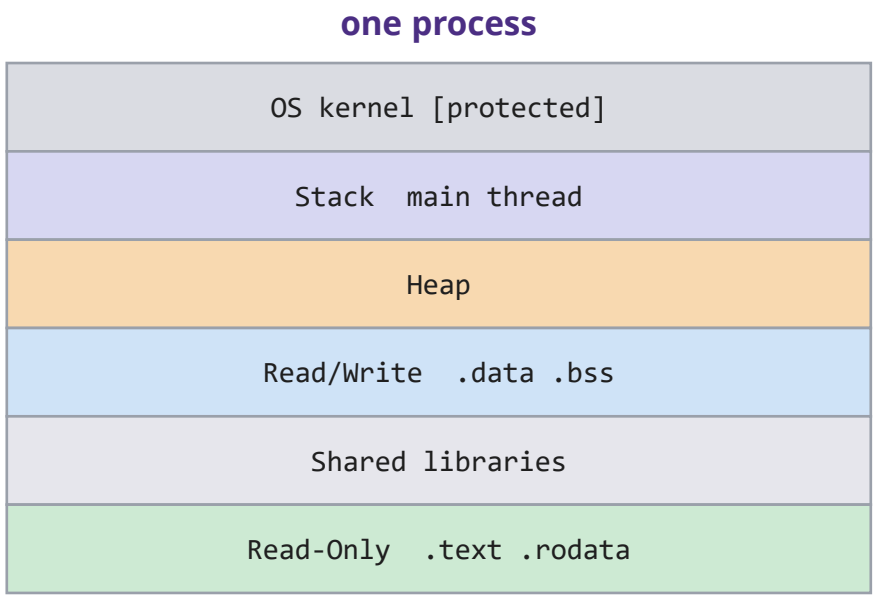
A software thread is execution state

```
1  void PrintOneToFive(void) {
2    for (int value = 1; value <= 5; value++) {
3      printf("%d\n", value);
4    }
5  }
```

program counter	next instruction
registers	current machine values
stack pointer	top of this thread's stack
stack	calls, locals, return addresses

A worker function is code.
A thread is the state required to execute that code and resume it after scheduling.

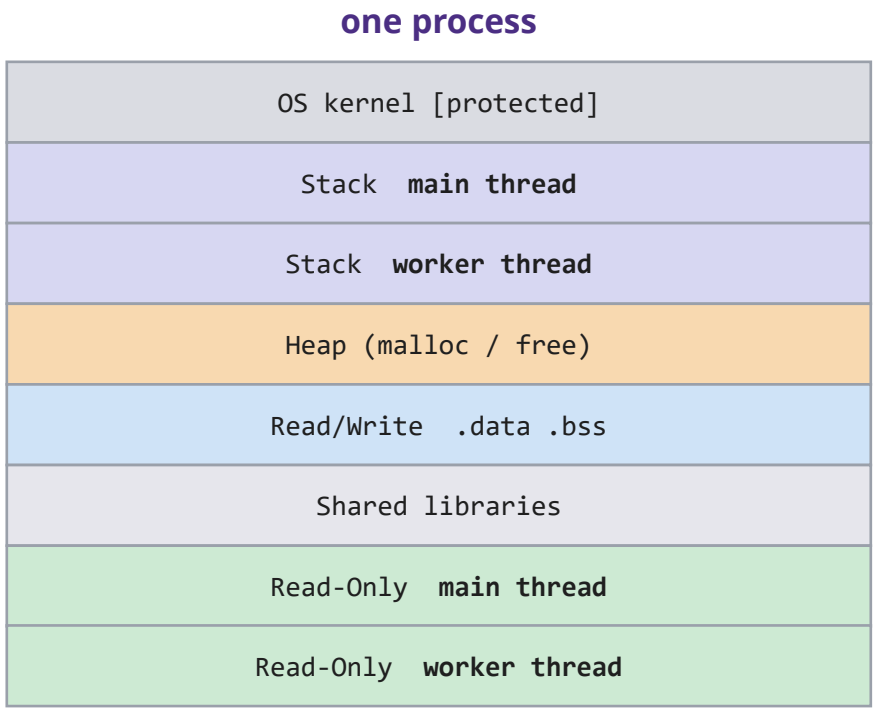
Before pthread_create



POSIX API for threads

- main begins on the process's initial thread
- The thread has one PC, register set, SP, and stack
- Code, globals, heap objects, and files belong to the process
- Calling a function adds a frame to the same stack

After pthread_create



What changed?

- The worker receives its own program counter, registers, and stack pointer, and stack!
- No second heap or global-data segment is created
- Both threads can reach the same heap objects and globals
- Both use the process's open file descriptors

POSIX threads are a library API

```
1  #include <pthread.h>
2
3  // compile and link
4  $ gcc -std=c17 -Wall -Wextra -pthread threads.c -o threads
```

Language and platform boundary

- pthread.h declares the POSIX thread functions and types
- pthreads is not part of the C language standard
- -pthread supplies compile and link settings

Version 1: Single-threaded

```c
#include <stdio.h>
#include <stdlib.h>

void PrintOneToFive(void) {
  for (int value = 1; value <= 5; value++) {
    printf("%d\n", value);
  }
}

int main(void) {
  PrintOneToFive();
  return EXIT_SUCCESS;
}
```

One call on a singlethread

- main runs on the process's initial thread
- PrintOneToFive is an ordinary function call
- value lives in a frame on the main thread's stack
- The call returns before main continues

Worker functions

```
1  void* PrintOneToFive(void* unused) {   // required signature
2    for (int value = 1; value <= 5; value++) {
3      printf("%d\n", value);
4    }
5    return NULL;  // exit the thread
6  }
7
```

Required worker function type

- The parameter type is void*: one untyped pointer
- No argument is needed yet, so the parameter is unused
- The return type is void*: one untyped result pointer
- Returning from the function terminates this thread

Version 2: Two-threaded

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* PrintOneToFive(void* unused) {
  for (int value = 1; value <= 5; value++) {
    printf("%d\n", value);
  }
  return NULL;
}

int main(void) {
  pthread_t thread;
  int rc = pthread_create(&thread, NULL, PrintOneToFive, NULL);
  if (rc != 0) return EXIT_FAILURE;
  rc = pthread_join(thread, NULL);
  if (rc != 0) return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

pthread_create

- &thread receives the new thread's handle
- NULL requests default thread attributes
- PrintOneToFive is the first function the worker executes
- NULL becomes the worker's unused argument

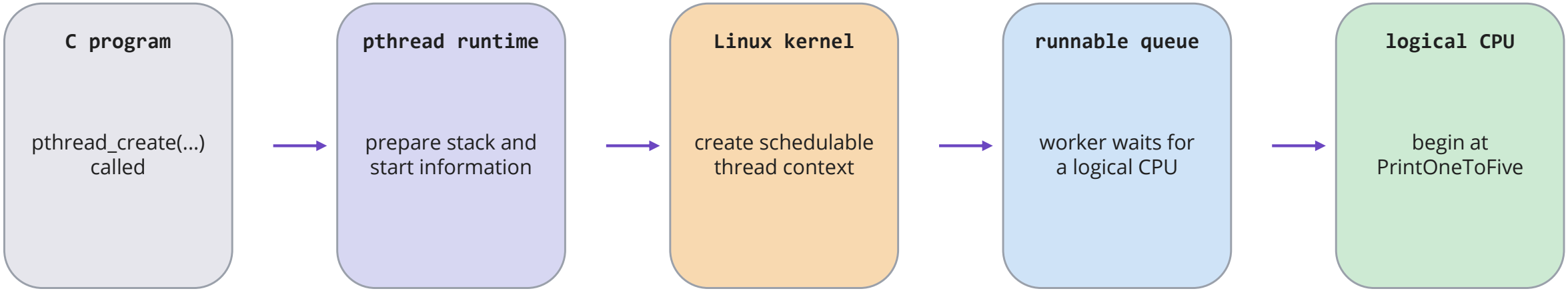
Version 2: Two-threaded

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* PrintOneToFive(void* unused) {
  for (int value = 1; value <= 5; value++) {
    printf("%d\n", value);
  }
  return NULL;
}

int main(void) {
  pthread_t thread;
  int rc = pthread_create(&thread, NULL, PrintOneToFive, NULL);
  if (rc != 0) return EXIT_FAILURE;
  rc = pthread_join(thread, NULL);
  if (rc != 0) return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

pthread_join

- Blocks the calling thread until this worker terminates
- Reclaims the worker's pthread bookkeeping
- NULL means main does not need the worker's result
- main returns only after the five lines are printed

What Linux does underneath pthread_create



On Linux, each pthread is normally a kernel-schedulable thread. It may run on any available logical CPU (which means the OS can move it between different logical CPUs as needed).

Why are threads tricky?

```
1  pthread_create(&thread, NULL, PrintOneToFive, NULL);
2  printf("main continues\n");
3  pthread_join(thread, NULL);
```

```
$ ./threads
main continues
1
2
3
4
5
```

Thread creation does not promise execution order.

The worker could print:

- before main
- after main or
- overlap main

```
$ ./threads
1
2
3
main continues
4
5
```

pthread_join constrains only what happens after joininig.

Problem #1

```
1  for (int i = 0; i < 5; i++) {
2    pthread_t thread;
3    int rc = pthread_create(&thread, NULL, PrintOneToFive, NULL);
4    if (rc != 0) return EXIT_FAILURE;
5    pthread_join(thread, NULL);
6  }
```

Why isn't this multithreading?

Problem #1: Joining immediately!

```
1  for (int i = 0; i < 5; i++) {
2    pthread_t thread;
3    int rc = pthread_create(&thread, NULL, PrintOneToFive, NULL);
4    if (rc != 0) return EXIT_FAILURE;
5    pthread_join(thread, NULL);
6  }
```

One worker at a time

- Iteration 0 creates a worker and then waits for it
- Iteration 1 cannot create its worker until join returns
- At most one worker from this loop exists at a time
- The code uses pthreads but gains no worker overlap

Version #3: Multi-threaded (no-args)

```
1  pthread_t threads[5];
2  int created = 0;
3
4  for (int i = 0; i < 5; i++) {
5    int rc = pthread_create(&threads[created], NULL,
6                            PrintOneToFive, NULL);
7    if (rc != 0) break;
8    created++;
9  }
10
11 for (int i = 0; i < created; i++) {
12   pthread_join(threads[i], NULL);
13 }
```

Five no-argument workers

- ❖ All five workers run the complete 1-through-5 loop
- ❖ The first loop creates without waiting
- ❖ The second loop eventually reclaims every worker
- ❖ Their output lines may interleave in many orders

Use the one pthread argument

```
1  void* PrintNumber(void* arg) {
2    int number = *(int*) arg;
3    printf("%d\n", number);
4    return NULL;
5  }
```

Pointer contract

- pthread_create passes one void* value
- This worker expects that pointer to address an int
- The dereference copies the integer into number
- number then lives on this worker's own stack

Use the one pthread argument

```
1  int main(void) {
2    int number = 3;
3    pthread_t thread;
4
5    int rc = pthread_create(&thread, NULL, PrintNumber, &number);
6    if (rc != 0) return EXIT_FAILURE;
7    rc = pthread_join(thread, NULL);
8    if (rc != 0) return EXIT_FAILURE;
9    return EXIT_SUCCESS;
10 }
```

Why does &number remains valid?

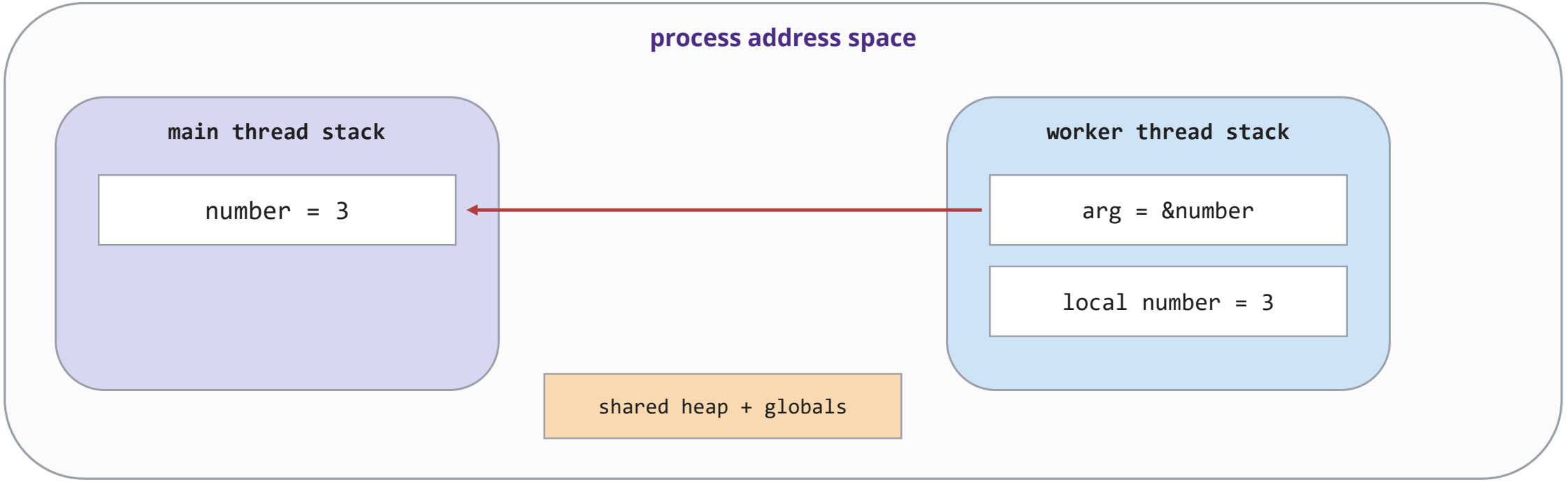
Use the one pthread argument

```
1  int main(void) {
2    int number = 3;
3    pthread_t thread;
4
5    int rc = pthread_create(&thread, NULL, PrintNumber, &number);
6    if (rc != 0) return EXIT_FAILURE;
7    rc = pthread_join(thread, NULL);
8    if (rc != 0) return EXIT_FAILURE;
9    return EXIT_SUCCESS;
10 }
```

Why &number remains valid

- ❖ number exists in main's stack frame
- ❖ main does not change number after creating the worker
- ❖ join completes before main returns
- ❖ The worker's read occurs during number's lifetime

Different stacks, but still same address space!



The pointer crosses stack regions but the copied integer does not.

Give each worker a distinct array element

```
1  int numbers[5] = {1, 2, 3, 4, 5};
2  pthread_t threads[5];
3  int created = 0;
4
5  for (int i = 0; i < 5; i++) {
6    int rc = pthread_create(&threads[created], NULL,
7                            PrintNumber, &numbers[i]);
8    if (rc != 0) break;
9    created++;
10 }
11
12 for (int i = 0; i < created; i++) {
13   pthread_join(threads[i], NULL);
14 }
```

The creation loop

- numbers contains five separate int objects
- &numbers[i] is a different address each iteration
- Each worker copies one element into its local number
- created tracks exactly which handles can later be joined

Join every worker after creating them all

```
1  int numbers[5] = {1, 2, 3, 4, 5};
2  pthread_t threads[5];
3  int created = 0;
4
5  for (int i = 0; i < 5; i++) {
6    int rc = pthread_create(&threads[created], NULL,
7                            PrintNumber, &numbers[i]);
8    if (rc != 0) break;
9    created++;
10 }
11
12 for (int i = 0; i < created; i++) {
13   pthread_join(threads[i], NULL);
14 }
```

The join loop

- No worker is joined inside the creation loop
- All successful handles remain available in threads
- Each join waits only if that worker is still running
- After the loop, every array access has completed

Five pointers, five worker-local copies

Problem #2

```
1  pthread_t threads[5];
2  int i;  // remains alive through the join loop
3
4  for (i = 1; i <= 5; i++) {
5    pthread_create(&threads[i - 1], NULL, PrintNumber, &i);
6  }
7
8  for (int index = 0; index < 5; index++)
9    pthread_join(threads[index], NULL);
```

One object changes repeatedly

- The loop contains one int object named i
- Every pthread_create copies the same address
- main writes a new value to i on each iteration
- A worker reads whatever value exists when it runs

Version #4: Multi-threaded (with args)

```
1  pthread_t threads[5];
2  int created = 0;
3
4  for (int value = 1; value <= 5; value++) {
5    int* number = malloc(sizeof(*number));
6    if (number == NULL) break;
7    *number = value;
8    int rc = pthread_create(&threads[created], NULL, PrintNumber, number);
9    if (rc != 0) { free(number); break; }
10   created++;
11 }
12
13 for (int i = 0; i < created; i++) {
14   pthread_join(threads[i], NULL);
15 }
```

Main thread owns

- ❖ malloc creates a distinct int object each iteration
- ❖ The object remains alive after the loop iteration ends
- ❖ Successful create transfers ownership to the worker
- ❖ Failed create leaves ownership with main, which frees it

Version #4: Multi-threaded (with args)

```
1  void* PrintNumberOwned(void* arg) {
2    int* number = arg;
3    printf("%d\n", *number);
4    free(number);
5    return NULL;
6  }
```

Worker thread owns

- The worker is the only owner after successful create
- It reads the object before freeing it
- It frees exactly once on every normal path
- No code dereferences number after free

Multiple arguments

```
1  typedef struct { int value; int repetitions; } PrintArgs;
```

main

```
1  PrintArgs* args = malloc(sizeof(*args));
2  if (args == NULL) return EXIT_FAILURE;
3  *args = (PrintArgs){7, 3};
4  int rc = pthread_create(&thread, NULL, PrintRepeated, args);
5  if (rc != 0) {
6    free(args);
7    return EXIT_FAILURE;
8  }
9  pthread_join(thread, NULL);
```

worker

```
1  void* PrintRepeated(void* raw) {
2    PrintArgs* args = raw;
3    for (int i = 0; i < args->repetitions; i++) {
4      printf("%d\n", args->value);
5    }
6    free(args);
7    return NULL;
   }
```

Returning values from threads

worker

```
1  void* Square(void* raw) {
2    int input = *(int*) raw;
3    int* result = malloc(sizeof(*result));
4    if (result == NULL) return NULL;
5    *result = input * input;
6    return result;
7  }
```

joining thread

```
1  int input = 7;
2  int rc = pthread_create(&thread, NULL, Square, &input);
3  if (rc != 0) return EXIT_FAILURE;
4
5  void* raw_result = NULL;
6  rc = pthread_join(thread, &raw_result);
7  if (rc != 0 || raw_result == NULL) return EXIT_FAILURE;
8  int* result = raw_result;
9  printf("%d\n", *result);
10 free(result);
```

The worker returns a heap pointer. After join, the joining thread owns that result and eventually frees it.

Every thread is joined or detached

```
1  pthread_join(thread, &result);
```

Join

- Another thread waits for completion
- The caller may collect a result
- Use for finite workers whose completion matters

```
1  pthread_detach(thread);
```

Detach

- No thread will join this worker
- pthread resources are reclaimed at termination
- Use for independent work with no returned result

A detached thread cannot later be joined. The process still ends all threads when main returns or exit is called.

Single-threaded server

```
1  while (1) {
2    int fd = accept(listen_fd, NULL, NULL);
3    if (fd < 0) continue;
4    HandleClient(fd);
5    close(fd);
6  }
```

One main thread

- ❖ accept returns one connected client descriptor
- ❖ main handles that client from start to finish
- ❖ main cannot call accept again while HandleClient blocks
- ❖ A second client waits even when the CPU is idle

Multi-threaded client

worker

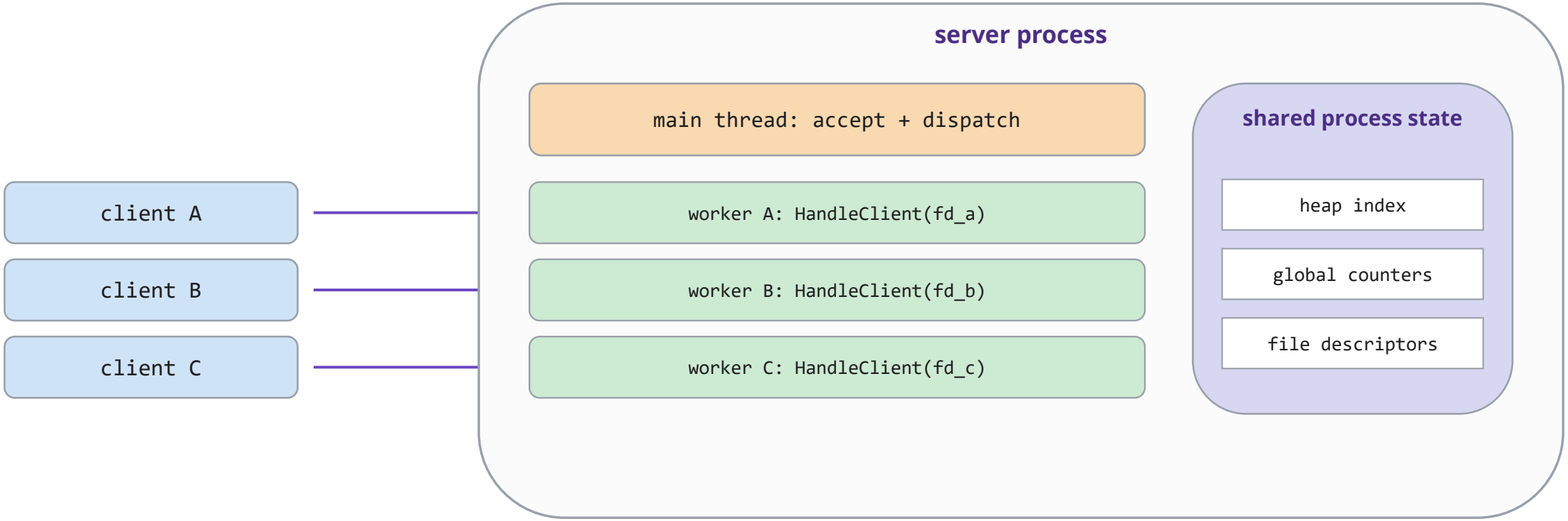
```
1  void* ClientThread(void* raw) {
2    int fd = *(int*) raw;
3    free(raw);
4    HandleClient(fd);
5    close(fd);
6    return NULL;
7  }
```

accept / dispatch loop

```
1  while (1) {
2    int accepted = accept(listen_fd, NULL, NULL);
3    if (accepted < 0) continue;
4    int* fd = malloc(sizeof(*fd));
5    if (fd == NULL) { close(accepted); continue; }
6    *fd = accepted;
7    pthread_t thread;
8    int rc = pthread_create(&thread, NULL, ClientThread, fd);
9    if (rc != 0) { close(*fd); free(fd); continue; }
10   rc = pthread_detach(thread);
11   if (rc != 0) pthread_join(thread, NULL);
12 }
```

The main thread returns to accept immediately. Each detached worker owns one descriptor argument and one client.

One process, several server threads



Why use threads

property	benefit	cost
creation	less state than a process	still allocates a stack
I/O waiting	another thread can run	more scheduling state
multiple cores	workers may run in parallel	contention can serialize them
shared memory	pass pointers directly	data races can corrupt state
failure boundary	one process to manage	one bad thread can damage all workers

Threads trade isolation for inexpensive communication and concurrent execution.

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.

A shared counter introduces a data race

```
1  int total = 0;
2
3  void* Count(void* unused) {
4    (void) unused;
5    for (int i = 0; i < 1000000; i++) {
6      total++;
7    }
8    return NULL;
9  }
10
11 // four threads run Count; main joins them and prints total
```

Conflicting accesses

- ❖ All four workers access the same global total
- ❖ Each access in total++ includes a write
- ❖ No mutex or other synchronization orders the accesses
- ❖ The C program therefore has undefined behavior

total++ is a read-modify-write sequence

```
1  total++;
2
3  // conceptual machine-level operations
4  register_value = total;
5  register_value = register_value + 1;
6  total = register_value;
```

Where values live

- total is one shared memory location
- register_value is private execution state
- A thread can be preempted between operations
- Another core can access total simultaneously

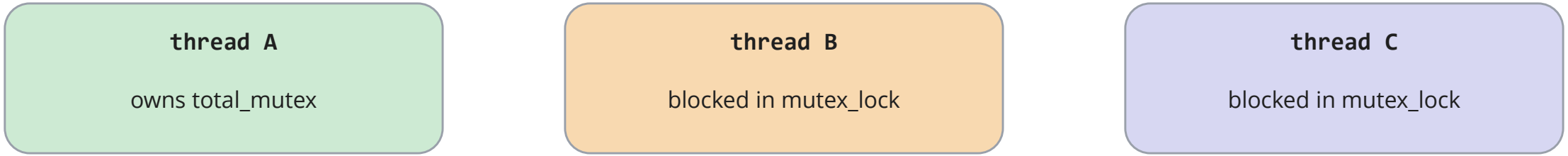
One C statement is not automatically one "atomic" hardware operation.

Two increments can produce one update

step	thread A	A register	thread B	B register	total
1	read total	0		-	0
2		0	read total	0	0
3	add 1	1		0	0
4		1	add 1	1	0
5	write total	1		1	1
6		1	write total	1	1

Both threads computed a locally correct 1. The second store overwrote the first store, so one increment was lost.

mutex




```
1  pthread_mutex_lock(&total_mutex);    // acquire or sleep
2  total++;                             // critical section
3  pthread_mutex_unlock(&total_mutex);  // release ownership
```

mutex

mutex lifetime

```
1  static pthread_mutex_t total_mutex;
2
3  int main(void) {
4    pthread_mutex_init(&total_mutex, NULL);
5    // create workers; join workers
6    pthread_mutex_destroy(&total_mutex);
7  }
```

critical section

```
1  void* Count(void* unused) {
2    (void) unused;
3    for (int i = 0; i < 1000000; i++) {
4      pthread_mutex_lock(&total_mutex);
5      total++;
6      pthread_mutex_unlock(&total_mutex);
7    }
8    return NULL;
9  }
```

mutex is a **mutual exclusion**. Only one thread can access through the lock at a time.

The C++17 equivalents

C++17 · standard library

typed thread arguments

```
1  void PrintNumber(int number) {
2    std::printf("%d\n", number);
3  }
4
5  std::vector<std::thread> threads;
6  for (int number = 1; number <= 5; number++)
7    threads.emplace_back(PrintNumber, number);
8  for (std::thread& thread : threads)
9    thread.join();
```

scope-based mutex ownership

```
1  std::mutex total_mutex;
2  int total = 0;
3
4  void Merge(int local) {
5    std::lock_guard<std::mutex> guard(total_mutex);
6    total += local;
7  }  // guard unlocks here
```

std::thread removes void* plumbing. std::lock_guard releases the mutex automatically when its scope ends.

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?