Asynchronous Programming Model for Concurrency

concurrency

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor.
A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

SO - What is the difference between concurrency and parallelism?
https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism

concurrency

- Parallelism can only take place when the hardware allows it
- If there are 8 virtual CPUs, then only 8 threads can run parallely
- However, concurrency can take place regardless of hardware capability as it is a model of execution
- So, theoretically, infinite threads can run concurrently on 8 virtual CPUs

concurrency

- Instead of executing one thread for a long time, breakdown the execution into units of time (slices) and execute *x slices* in some scheduling pattern. --> time slicing
- OS can decide which threads should have a higher priority and ask for them to be executed more often than other threads --> thread priority

Virtual threads

- Threads created as a software construct
- Do not directly map to hardware threads
- A virtual thread can be mapped to any hardware thread for execution
- OS/kernel maintain control of virtual thread

Concurrency in programming constructs

- Default option: do what the OS/kernel does
- Create threads (virtual)
- Save state
- Distribute work across virtual threads
- Set priority for work
- Execute

What if a task is taking time?

- Set its virtual thread to low priority
- Check for results after some (longer) duration
- Sleep the thread
- e.g. I/O operations

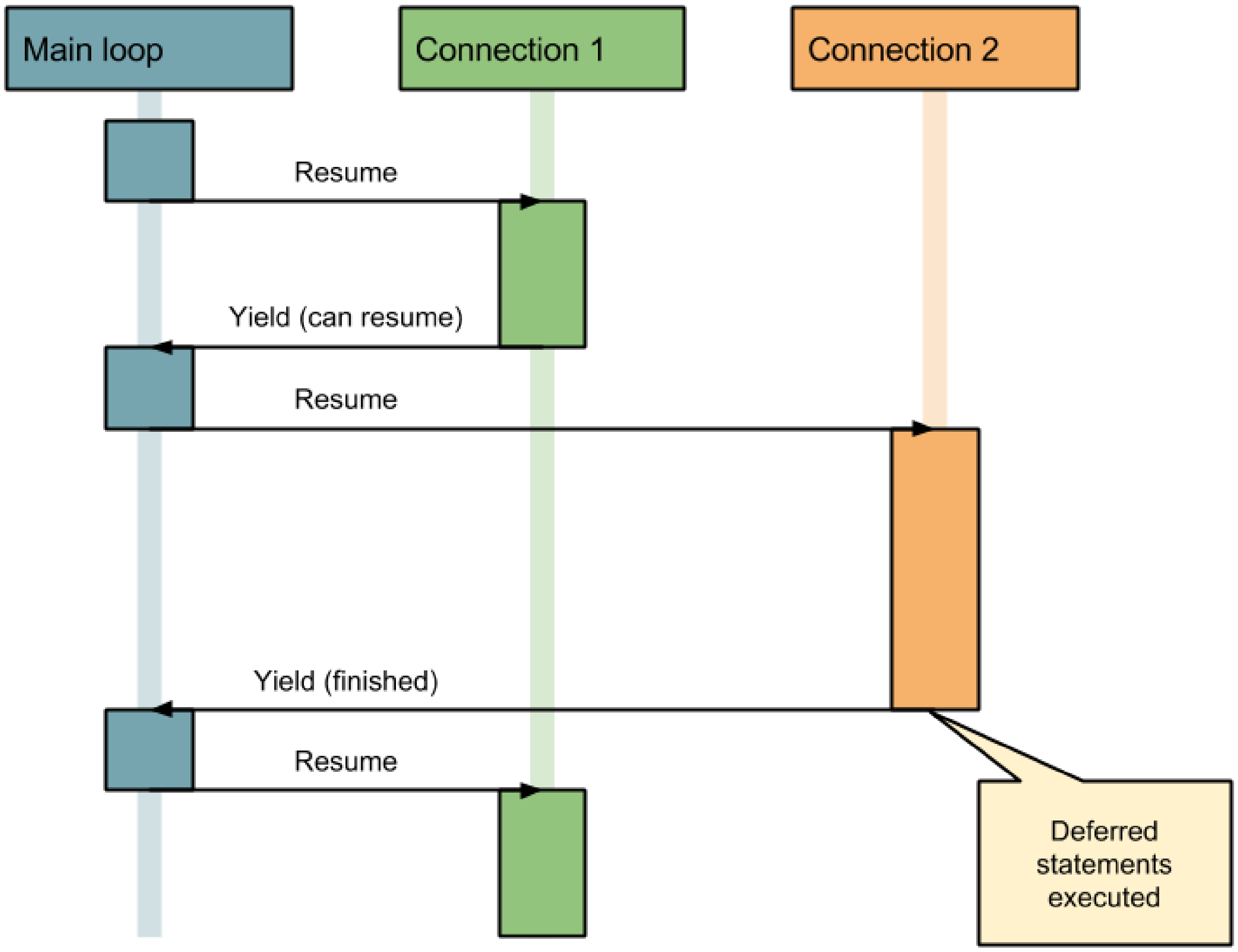
Do you like programming threads?

- Too much complexity
- Debugging is difficult
- Hardware dependant
- Compilers can’t optimise (much)
- OS has to provide decent constructs for threading

coroutines

- Multiple entry points
- Suspend and Resume
- Non-preemptive (very very very very important distinction)
  - Voluntary give up control
  - When idle, waiting for something, finished a task
  - Co-operative multi-tasking
  - Let other tasks run for a while!

coroutines


Main loop
Connection 1
Connection 2
Resume
Yield (can resume)
Resume
Yield (finished)
Resume
Deferred statements executed

coroutines

- Exceptions { try ... catch }
- Event Loops { on(event).do(foo) }
- Iterators { for x in list }
- Infinite series { 1, 2, 3 ... }
- Pipes { proc1 | proc2 | proc3 }

generators

- Coroutines for iterators
- Returns a list of items, but one at a time
- Return first value, give up control
- Next time control is passed, return second value, give up control
- ...
- Next time control is passed, return last value, expire state, give up control
- Next time control is passed, raise error!

generators

```
function foo(int n):
  // returns numbers 1 ... n
  int state = 1;
  while (state <= n) {
    return state;
    state += 1;
  }
```

asynchronous

- a-synchornous --> not synchronous
- Current work = A ; Other work = B
- Do B while A is still going on
- Possible when A is dependant on an event
- i.e. A is not executing, but *waiting*

asynchronous

- So A is seen to be executing in the background, but in fact it is waiting
- B is executing in foreground
- Periodically check if A’s event has happened. If not, continue executing B
- Once event happens, stop executing B, and continue executing A
- A *must* be a co-routine; B *can be* a co-routine

Implementing asynchronous

- How many threads do we need?
- What data should be stored to save state?
- Do we need hardware support?

Implementing asynchronous

- How many threads do we need? --> 1 minimum
- What data should be stored to save state?
  - function call stack
  - point of execution / interruption
  - function state: variables
- Do we need hardware support?
  - No! Because this is a software construct

Async / Await

- Usual keywords used to indicate asynchronous programming model
- Async is used to define a function (routine) as fit for asynchronous execution (co-routine)
- Await is used to indicate that something is waiting for an event to happen, therefore, control-break can happen here
- Terms are in theory; Actual programming languages have different keywords to reflect these concepts

Async / Await

```
async function foo(url):
   // download length of data from url
   char data[2048];
   // control break point
   await download(data);
   // something else may execute here
   // wait for download to complete
   return strlen(data);
```

Threads vs Async

- Two models of concurrency
- Let’s discuss! Which is better? For what tasks?
  - Hardware threads
  - Virtual threads
  - Co-routines (asynchronous execution)

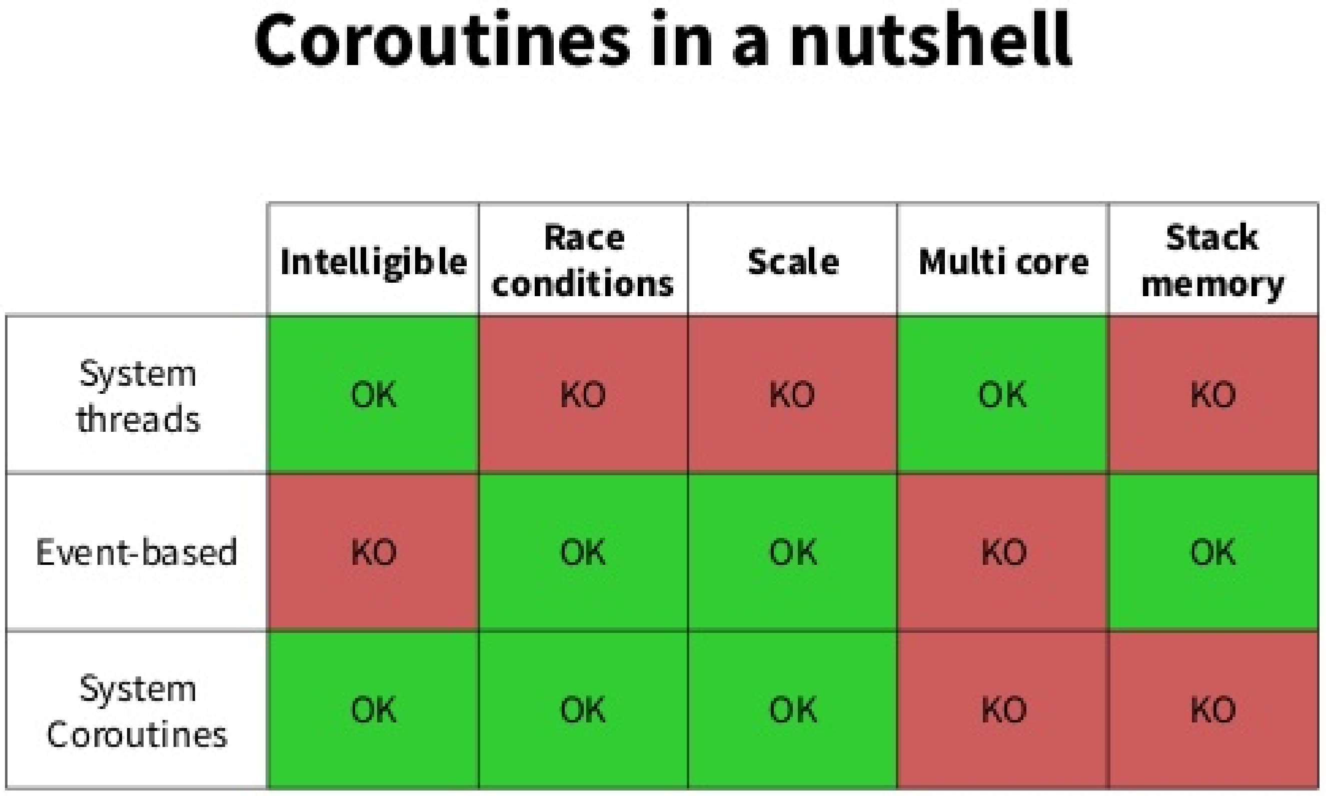
Threads are better for...

- Parallel execution
- Not worrying about execution control
- Performing same tasks multiple times
- Setting priority to repetitive tasks
- Handling different events simultaneously

Co-routines are better for...

- Event-based programming
- Having multiple points of execution and interruption
- Concurrent execution even on single threads
- Saving state of execution and resuming it later

Strengths vs Weakness



Coroutines in a nutshell

	Intelligible	Race conditions	Scale	Multi core	Stack memory
System threads	OK	KO	KO	OK	KO
Event-based	KO	OK	OK	KO	OK
System Coroutines	OK	OK	OK	KO	KO

Combine the two!

- Co-routines piggybacking on threads
- Multiple threads means multiple co-routines can execute simultaneously
- Create a pool of threads (T) to execute a pool of co-routines (C)
- Each thread gets some co-routine, executes it
- When interrupted, it grabs another co-routine from the pool and executes it