Discussions on Asynchronous Programming APIs

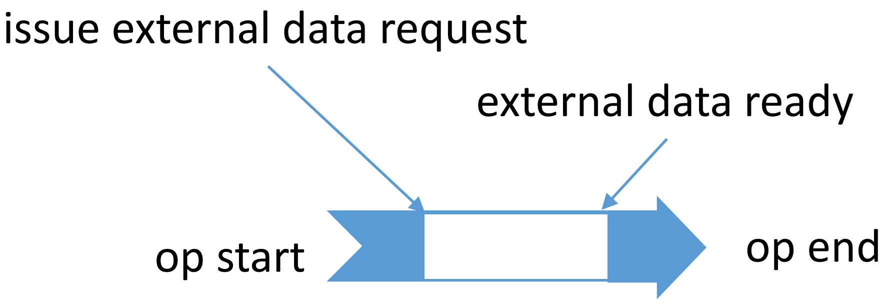
Xiao-Feng Li

xiaofeng.li@gmail.com

2019-10-1

What is asynchronous operation?

- An operation whose result depends on external data that is not computed by the processor, therefore does not prevent the processor from executing other task in parallel with it (the operation).
  - External data: timer interrupt, signal, and mostly the IO data
  - IO goes out of CPU’s control for local or remote data
  - Excluding cache access, which is handled by the processor, invisible to OS



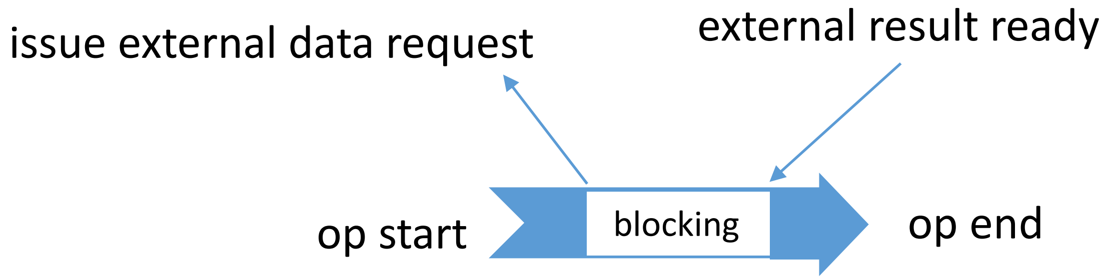
Async ops need system support

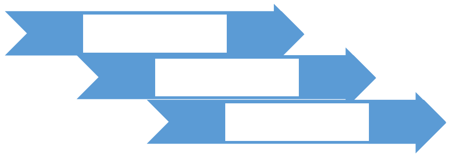
- To use external data requires:
  - Need to access system to issue IO request, to trigger timer, etc.
  - Need a way to know the external data ready
  - System needs to provide APIs to support async ops
- Semantics that has to be supported by the system API:
  - 1. To issue request, 2. to wait for the result, 3. to receive the result.

- A few factors to consider for API design
  1. Cost: Best use of processor resource
  2. Latency: Program receives data without delay
  3. Throughput: As many as possible async ops can be served

  (This is from async support point of view, omitting lots of other API design factors.)

Blocking single request

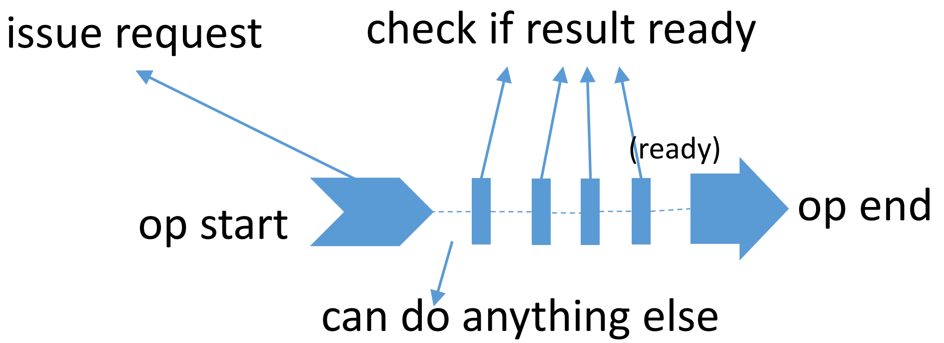


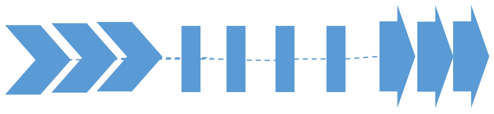


Multiple requests needs multithreads

- Integrate all three parts into a single API
  - Blocking API (issue request + wait for result + receive available result).
  - E.g., blocking write
- API design factors
  - Cost: When the thread is waiting, the OS can suspend it and schedule other threads. When the data is ready, the OS reschedule the thread.
  - Latency: Equals to scheduling latency.
  - Throughput: Equals to scheduling throughput.
- Discussions:
  - During the API execution, the thread can do nothing else.
  - To deal with multiple external data in parallel requires multiple threads, and data sync/share.
  - Thread scheduling in OS is usually heavy.

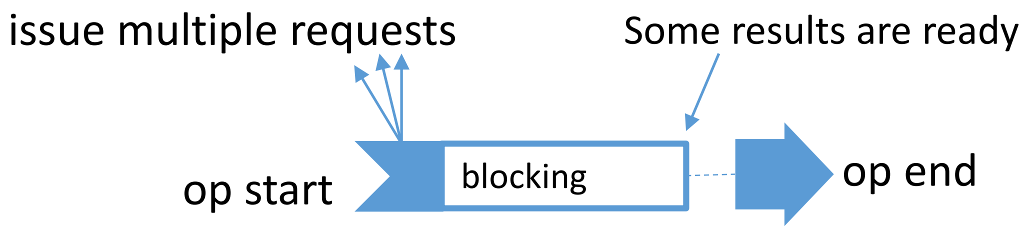
Non-blocking single request





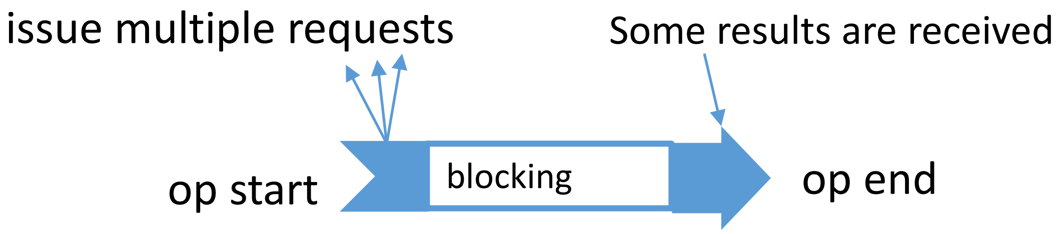
- Separates three parts into different APIs
  - Non-blocking APIs: issue request, poll, receive result.
  - E.g., non-blocking read, which actually uses same system call for polling and receiving
- API design factors
  - Cost: only the last polling is really useful, other pollings are redundant.
  - Latency: depends on how frequent the polling is, balancing latency vs. cost.
  - Throughput: one thread can multiplex over multiple async ops – which can run in parallel
- Discussions
  - The program needs to maintain states across three APIs.
  - Polling cost cannot be eliminated. For multiple async ops, the cost is multiplied too.
  - Normally needs polling loop.

Multi-requests readiness



- Merge multiple blocking ops into one, but only wait for certain # of results
  - Blocking API (issue multi-requests + wait for certain # of result), receiving API
  - E.g., select/epoll + read
- API design factors
  - Cost: Thread scheduling when blocking, and at least two syscalls.
    - (It is not our focus to compare implementations, such as select vs. epoll, who maintains FD sets, etc.)
  - Latency: Same as thread scheduling.
  - Throughput: one thread can process multiple parallel async requests. No thread data sync/share. Less thread scheduling.
- Discussions
  - Polling can be achieved with timeout or variants.
  - No easy way to conduct multiple parallel disk IOs: always ready, then blocking read.
  - To receive all the results, program normally needs a loop.

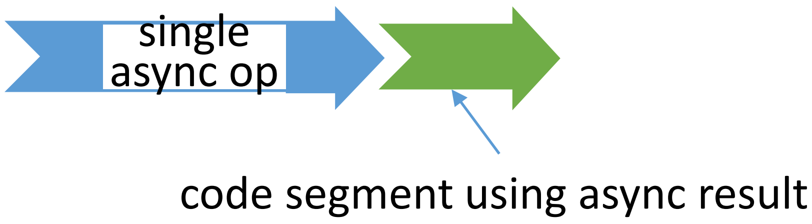
Multiple requests completion

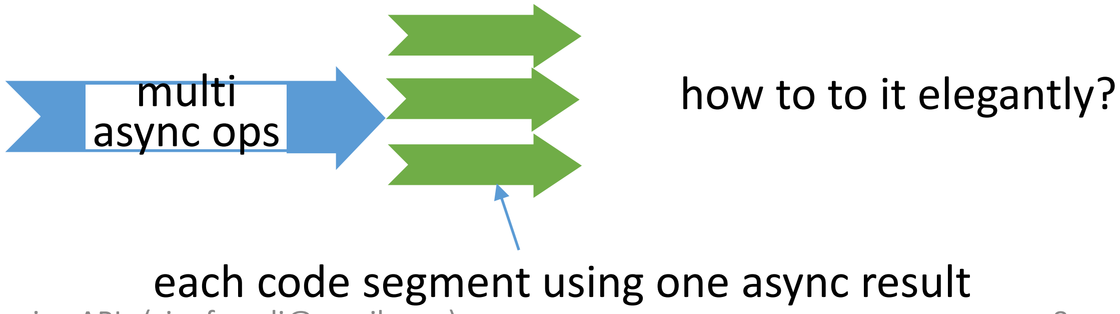


- Merge multiple async ops into one, and return after receiving some results
  - Blocking API: issue multi-requests + wait and receive certain # of results
  - E.g., kqueue, iocp, io_uring
- API design factors
  - Cost: merge two (or more) syscalls into one for (completion) result.
  - Latency: almost no additional latency besides incurred by the async op itself.
  - Throughput: support multiple concurrent async ops, with less scheduling and ring switches
- Discussions
  - Polling can be achieved with timeout or variants. Normally use a loop to receive all the results.
  - More flexible and scalable since "readiness" is not always well defined (e.g., disk IO). To some extent, memory access is similar – if not hidden by CPU, like distributed memory.
  - System manages the memory, probably resulting with some inflexibility.

Async ops can be hidden from programmer

- Programmers have different considerations than syscall designer
  - Syscall abstracts system max capability through min interface to programmer
    - Cost, latency, throughput
    - E.g., API for multiple requests completion does not wait for all results, because a single available result already triggers further computation. Low latency → Fast response.
  - Programmers do not necessarily care how the system works
    - They care to develop good applications: how to put together business logics
    - Productivity (and portability), responsiveness, scalability
    - Language and library developers can help bridge the gap between syscall and app
    - “Synchronous” API on top of asynchronous op syscalls

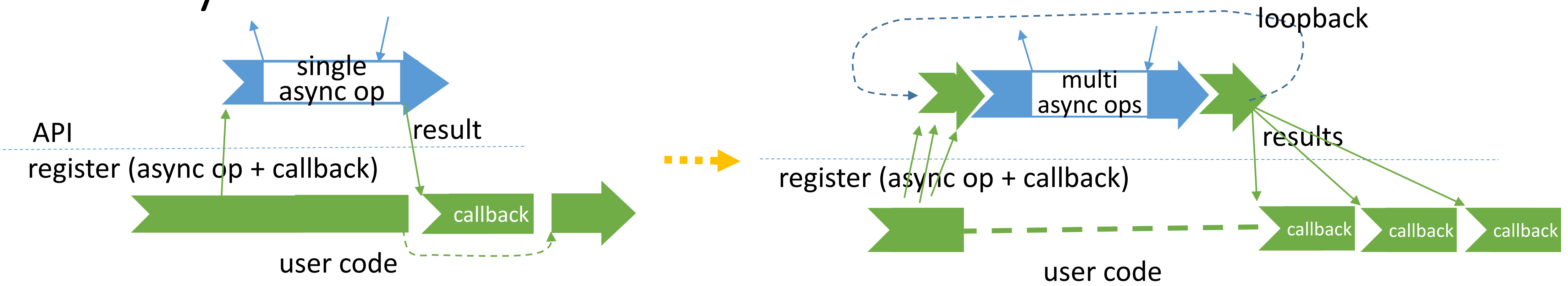




Asynchronous callback: signal

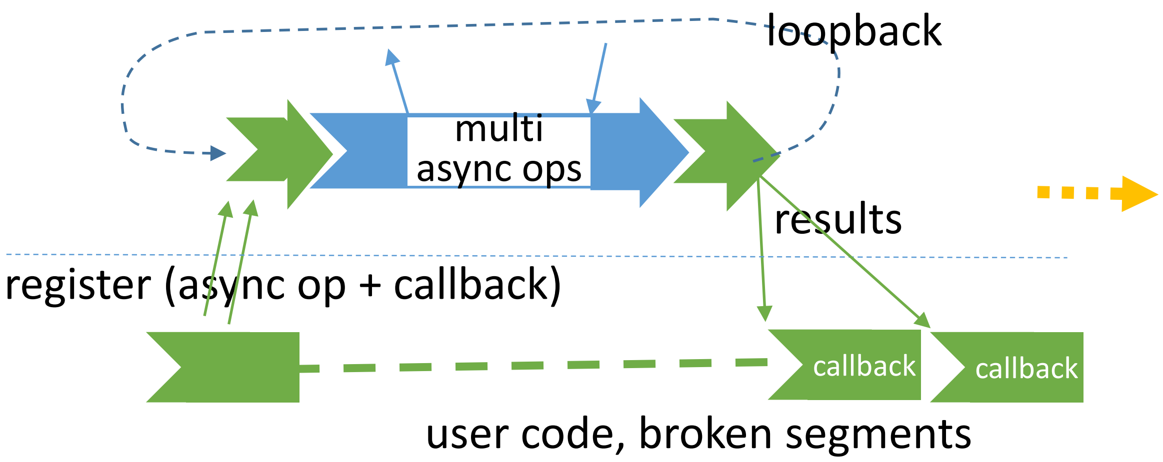
- API specifies the function to execute when async op finishes
  - Register the async op and the callback that consumes the result.
  - Then invoke syscall for the async op, and call the callback once result is available
  - E.g., signal handler, APC, or some user libs
- API design factors
  - Productivity: portable across *NIX.
  - Responsiveness: signal can only call the handler in a schedule point
  - Scalability: does not help very much if code registers only one pair of (async op + callback)
- Discussions
  - True asynchronous with signal as soft interrupt
  - Signal handler probably only used for very limited cases due to its specifics

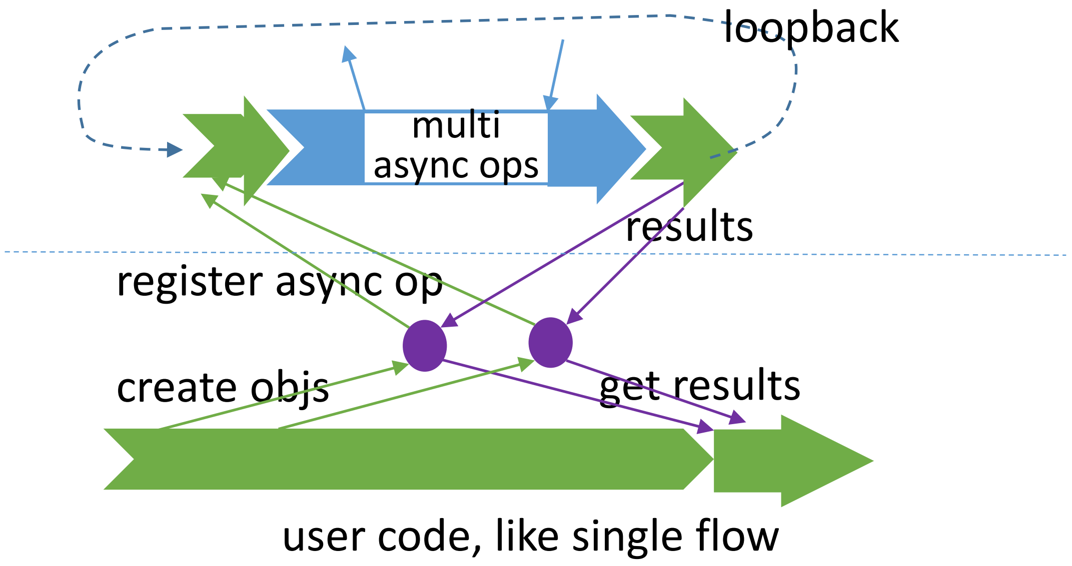
Asynchronous callback: event driven



- Decouple program API and system API for async ops
    - Program simply registers pair of (async op, callback) at anytime, leaving the logistics to runtime
    1. Runtime collects the registrations and uses one syscall for multiple requests
    2. Runtime dispatches async results by calling the callbacks one by one, then goto 1.
- API design factors
    - Productivity: hide the syscalls with unified API across async ops and OSes
    - Responsiveness: if implemented in single thread, callbacks are executed sequentially
    - Scalability: for IO intensive apps, single thread can achieve high throughput. For blocking IOs, thread pool can be used to support the same API.
- Discussions
    - Control flow is broken into async callbacks. Good for shallow nested callback tree, such as GUI.
    - Runtime for scheduling becomes part of the language. This is a fundamental change.
    - When nested callback tree is deep, counteract productivity: Callback hell.

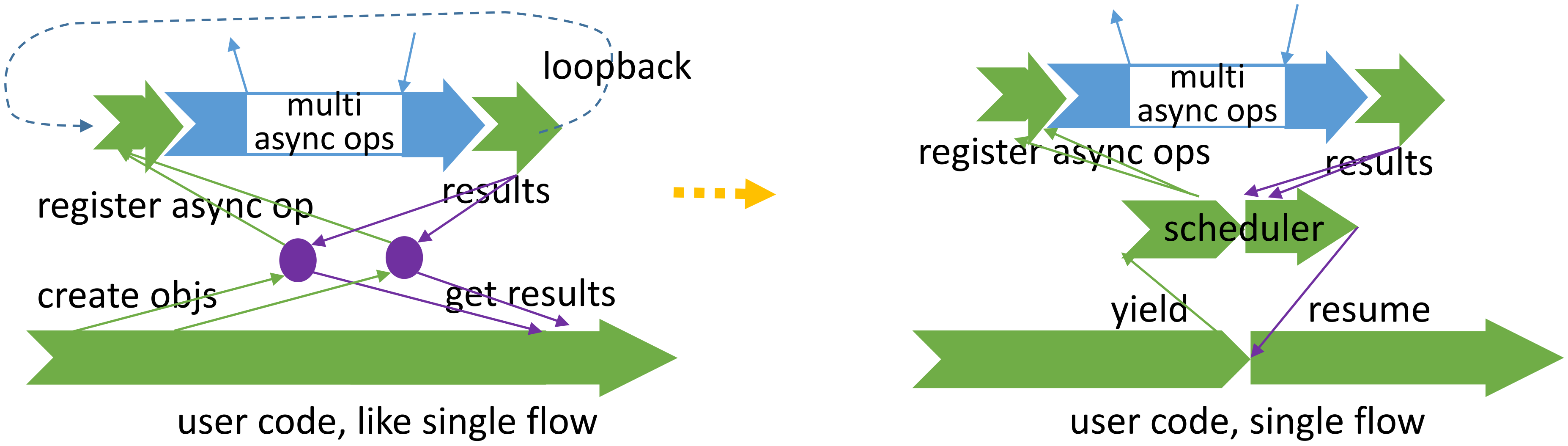
“Synchronous” callback





- Make the callback look like synchronously (or sequentially) executed
  - Wrap function for async op into an object (like FD), and get the result by blocking on the object (like reading FD) with callback to consume the async result.
  - Support multiple parallel async ops by creating an array or stream of independent objects.
  - E.g., Promise, Reactive extension
- API design factors
  - Productivity: one level up in API abstraction, simulating traditional intuitive “blocking call” without losing parallel async properties.
  - Responsiveness and scalability: same as asynchronous callback
- Discussions
  - Still rely on callbacks. Not very straightforward to integrate with traditional logics.

Resumable function (Coroutine)



- Decouple task scheduling from async op API with resumable function
  - Function yields to scheduler at the point of async function execution, resumes with result.
  - The scheduler maintains data dependence among tasks including issuing async ops.
  - The code segment following the yield point is like a callback consuming the yield result.
  - E.g., generator, await
- API design factors
  - Productivity: back to traditional intuitive synchronous programming with async benefits.
  - Responsiveness and scalability: same as asynchronous callback for async ops.
- Discussions
  - The API is a result of convergence of green threads and async op APIs, with the best of both.
  - Like in green threads, the scheduler can schedule any ready tasks with coroutines

Summary

- Asynchronous programming API design aims to improve both productivity and scalability
  - Productivity is achieved largely through “synchronous” programming constructs
  - Scalability is achieved by overlapping async operations with processor execution
  - The key of the API design is to allow a single thread to issue multiple outstanding async operations so that any ready result can enable certain code execution, hence minimal processor idle time
  - This enables pure user level solution that is platform agnostic, and then thread pool is largely an orthogonal multiplier
  - The APIs have evolved from synchronous system call to synchronous user call while maintaining all the async benefits
- Two levels of scheduling: user coroutines and kernel threads
  - Exploits the best of green threads and async APIs