# Communicating Sequential Processes in C++

John Skaller

July 28, 2021

# Contents

# Chapter 1

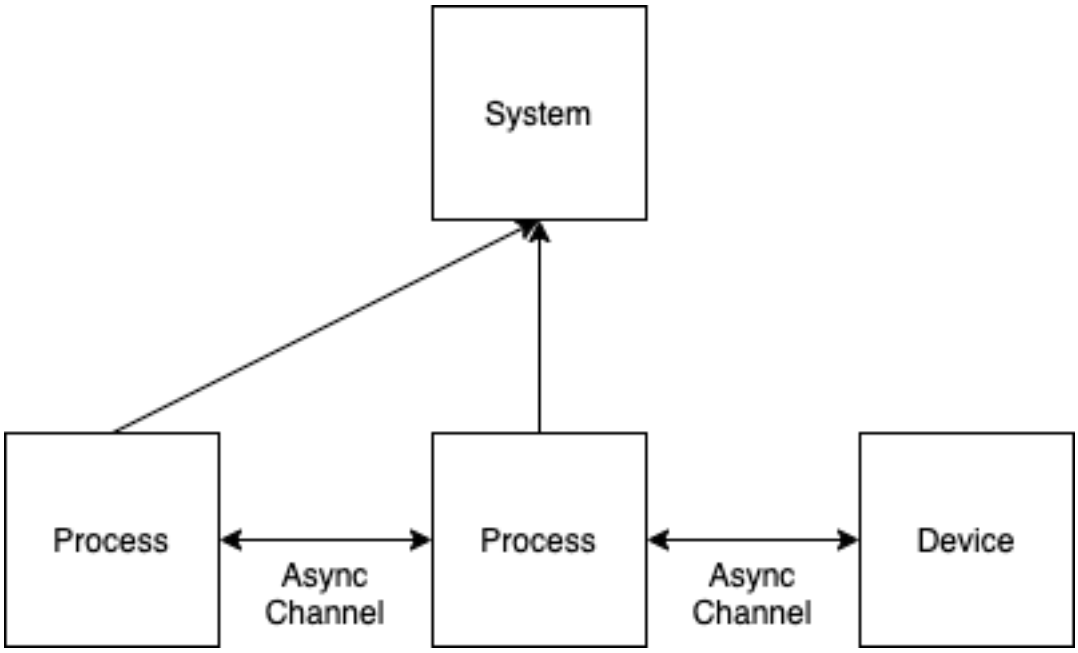
# CSP model

The CSP kernel supports a simple architecture.

## 1.1 Systems

A *system* consists of several *processes* which communicate using *asynchronous channels*. Systems are used to manage core *internal* resources: processing elements (CPUs), program code, time, and data memory.

A *device* is an arbitrary object and associated service thread which is used to manage *external* resources, including network and file system access, and user interfaces.




Processes within the same system can communicate with each other or with devices. Processes in distinct systems must use a device as a middleman to communicate.

## 1.2 Processes

The execution of a process is effected by one or more *threads*. If there is only one, the process is *single threaded,* otherwise if there are two or more, *multi-threaded.* Every process is started with a single *initial* thread.

A *fibre* is a logical thread of control and associated execution context. Each process consists of a collection of fibres.

Fibres in a process are *running* if a thread is elaborating the fibre's program code, otherwise the fibre is *suspended*.

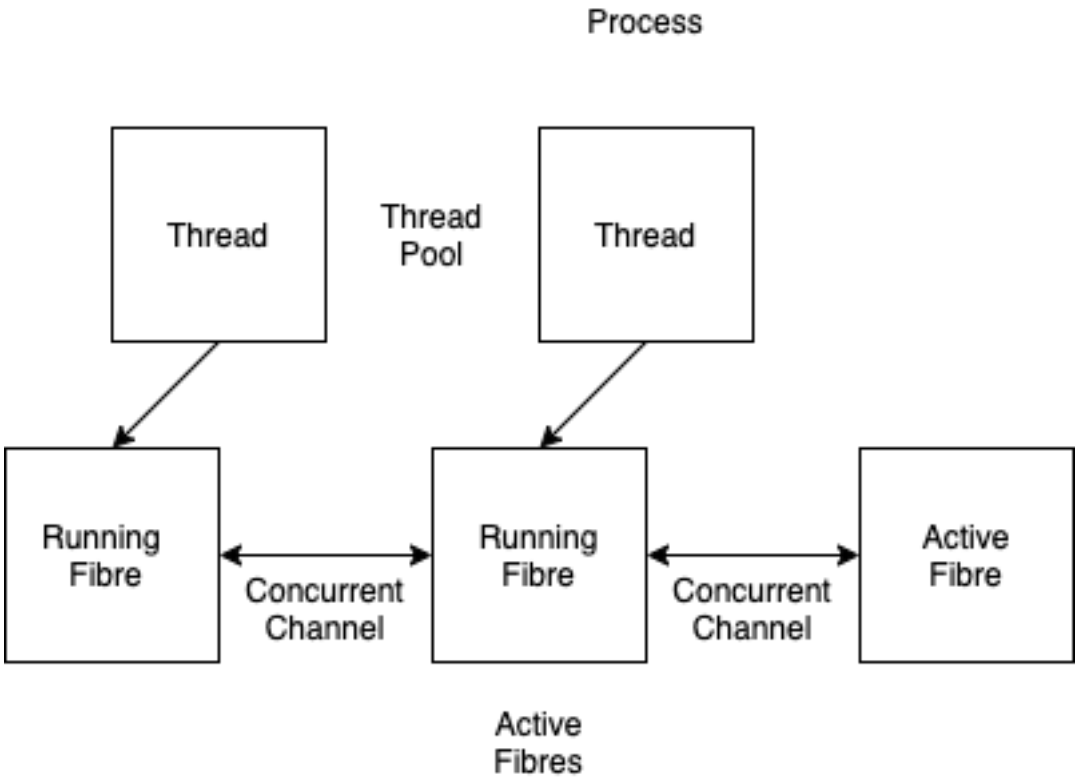
A suspended fibre can either be *ready* or *waiting*. The collection of ready and running fibres are said to be *active*.

All the ready fibres of a processes are maintained in a set and are owned by their process. Running fibres are owned by the thread running them which are owned by the thread's process.

When a thread is out of work, it attempts to locate a ready fibre then run it. If there are no ready fibres, no fibres are running, and there are no fibres waiting on asynchronous I/O from an external device or another system, then

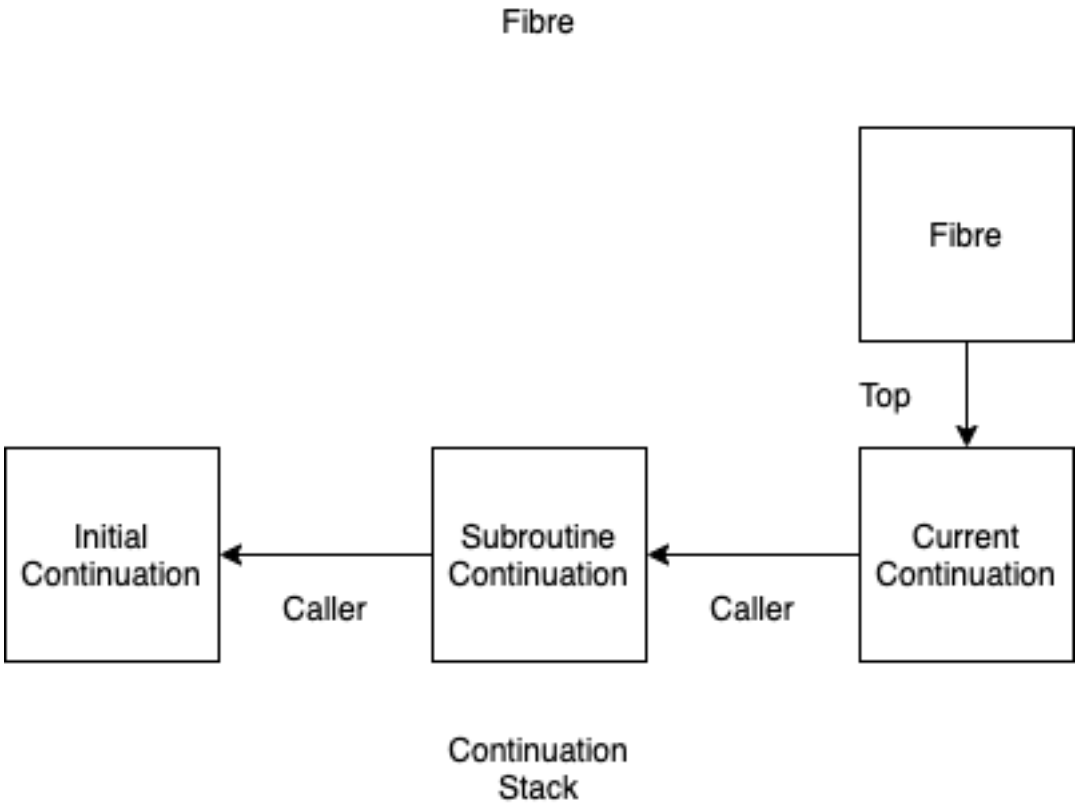
the process terminates and all associated memory is released, all but the initial thread are terminated, and the initial thread returns control.

A process is initiated by a single C++ function call which is passed a pointer to the initial continuation to invoke and a pointer to the owning system. A single threaded processes is then constructed with a single fibre owning the initial continuation.




## 1.3 Fibres

A *fibre* consists of a stack of continuations.

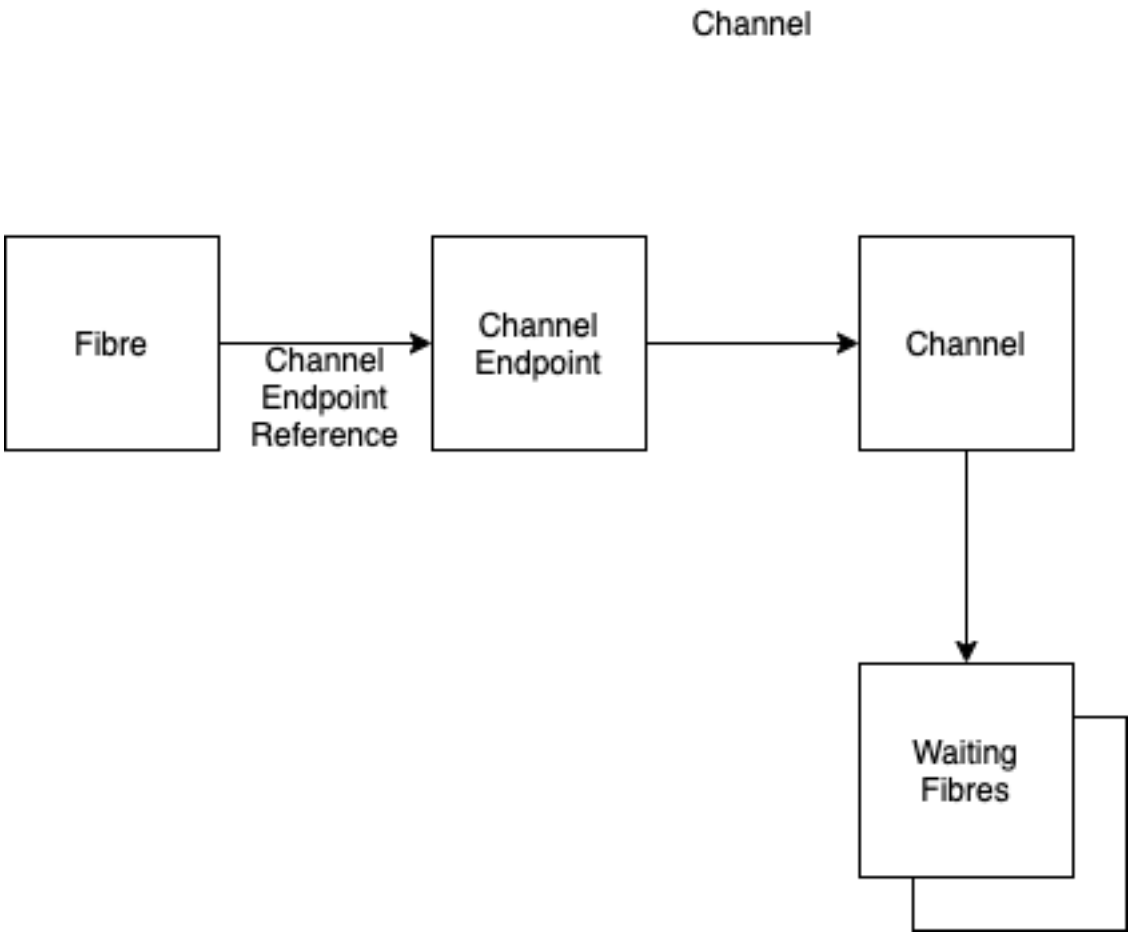



## 1.4 Channels

A *channel* is a synchronisation primitive which also allows transmission of data.

Channels are accessed via *channel endpoints*, using *channel endpoint references.*

A particular channel enpoint may be accessed only from the continuations of one fibre; that is, each endpoint must be uniquely owned by a single fibre. This invariant ensures the correct termination of fibres making I/O requests which cannot be satisfied.


Channel
Fibre
Channel Endpoint Reference
Channel Endpoint
Channel
Waiting Fibres

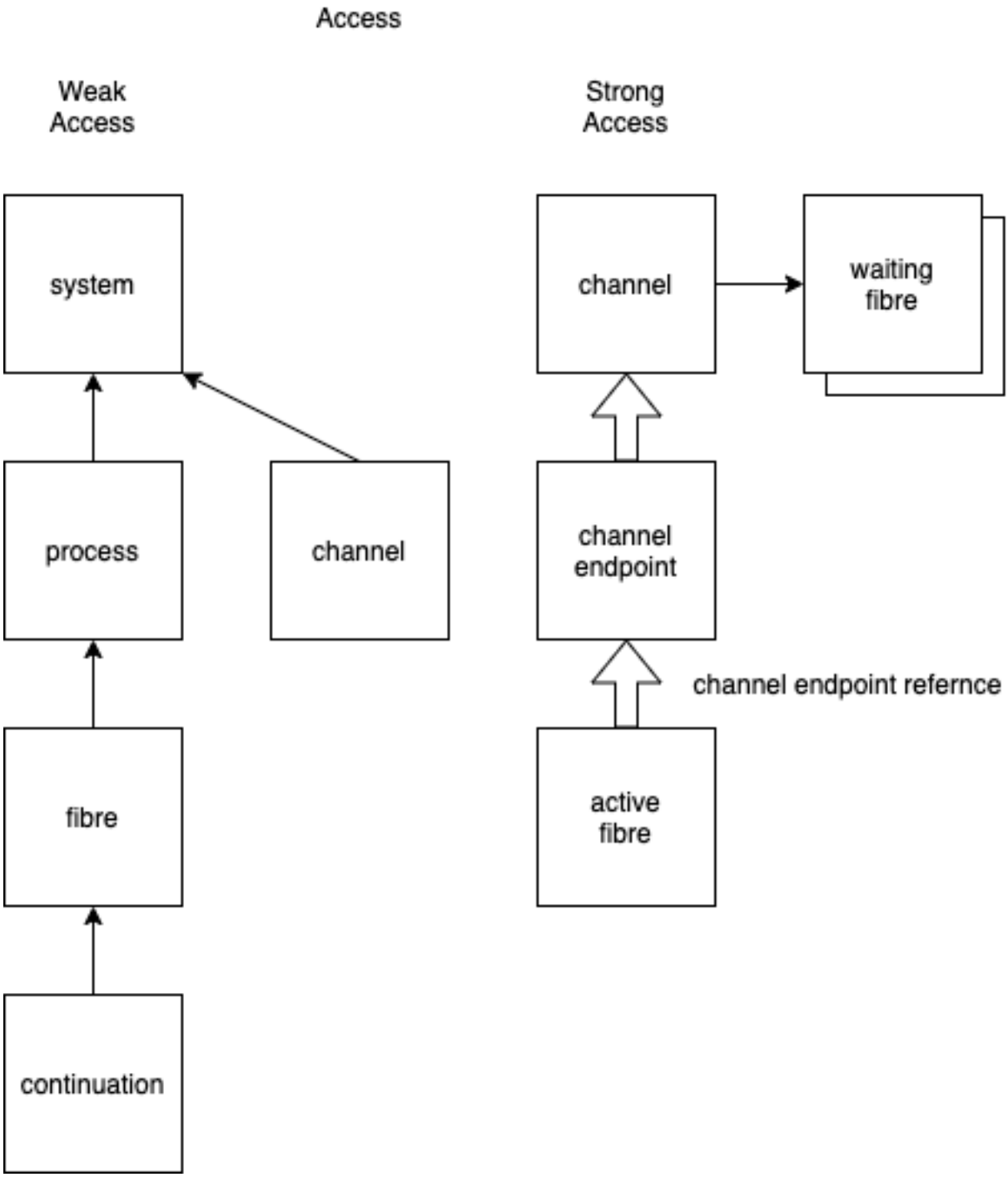
# Chapter 2

# Access

Objects in the system can access others using either pointers or reference counting smart pointers. A variable containing a pointer which expresses ownership is called a *strong pointer variable*. Reference counting pointer variables are always strong.

Ordinary pointers can also be strong. In this case, the object referred to must be manually deleted before the object containing the variable is destroyed.

A weak pointer variable is a variable containing a pointer which provides access to an object which owns, directly or indirectly, the containing variable. Such objects must be deleted before the owning object is deleted so that the lifetime of the weak pointer variable is contained temporally in the owner lifetime.


Access
Weak Access
Strong Access
system
process
channel
fibre
continuation
channel
waiting fibre
channel endpoint
channel endpoint refernce
active fibre

# Chapter 3

# Operational Semantics

The core semantic operations of the system are construction of processes, threads, and fibres, and channel I/O.

Reads and writes on channels are performed with special service calls which are given a channel endpoint reference. The semantic rules are simple enough.

A channel may be in three states. It may be empty, consist of a set of readers, or consist of a set of writers.

When a write is performed then, if the channel is empty or consists of a set of writers, the fibre performing the write is added to the channel. Since this fibre we the currently running fibre of some thread, and the fibre is now suspended, the thread no attempts to find another fibre to execute.

If the channel contains a reader then, instead, it is removed from the channel. Data is transfered from the writer to the reader. Nominally, both fibres then become active fibres of their respective processes.

# Chapter 4

# Allocators

An *allocator* is an object with two methods, `allocate` and `deallocate`, used to manage memory blocks. The allocate method retrieves an available block of a particular size, and the deallocate method returns to the free store.

Each system has a *system allocator* which must be used to construct process objects, channel objects, and channel endpoint objects.

Each process has a *process allocator* which must be used to construct fibres and continuations of that process.

We provide a standard system allocator of class `system_allocator_t`. This allocator is thread safe and uses spinlocks to protect the allocation and deallocation methods.

The management objects and the user memory blocks are all suballocated from a single large memory block retrieved from another allocator, its *bootstrap allocator.* We provide a convenience allocator which can be used as the initial bootstrap for other allocators, of class `malloc_free_t` allocator.

The standard system allocator requires that only blocks retrieved from it by `allocate` be returned to it, by method `deallocte`. In addition, no more than the available number of blocks of any particular size may be allocated or the behaviour is undefined. The system allocator must be constructed with the maximum number of blocks required.

Requests for a particular size are mapped to requests for the size which is the innitially specified size which is the least upper bound of the requested size. Requests for blocks larger than any specified size have undefined behaviour.

The model is based on the fact that all programs are ultimately finite state machines. This requires all scalable programs be bounded so that the maximum number of blocks of any particular size ever required at any time can be calculated and thus pre-allocated.

## 4.1 Allocator Structures

All allocators are used via a reference counting handle `alloc_ref_t`. Every non-root allocator has constructors whose first argument is a handle to its parent, which is the allocator that constructed it. This ensure the parent remains live whilst its child does, and provides access to the parent which must be used to deallocate it.

A root allocator is passed a `nullptr` and is created by `C++` standard `new` and deallocated by `delete`. Therefore, allocators must form a directed acyclic graph with respect to parentage.

In addition many allocators delegate to others, often one other, but many others could be used. The delegation graph must also be a directed acyclic graph.

# Chapter 5

# Continuations

The heart of the system are continuation objects. These are user defined classes that contain the application code. The code must be presented in a heavily styalised way in order for the system to operate correctly.

A translator for a higher level language will generate conformant continuations.

The heart of a continuation contains three methods:

1. The constructor is responsible for binding the lexical context
2. The `call` method is responsible for assigning the arguments
3. The `resume` method is responsible for executing a step of code