# Introduction to Asynchronous Programming

In this document we introduce an asynchronous model for concurrent programming. For certain applications, an asynchronous model may yield performance benefits over traditional multithreading. Much of the material presented in this document is taken from Dave Peticola's excellent introduction to Twisted[1], a Python framework for asynchronous programming.

## 1 The Models

We will start by reviewing two (hopefully) familiar models in order to contrast them with the asynchronous model. By way of illustration we will imagine a program that consists of three conceptually distinct tasks which must be performed to complete the program. Note I am using task in the non-technical sense of something that needs to be done.

The first model we will look at is the single-threaded synchronous model, in Figure 1 below:

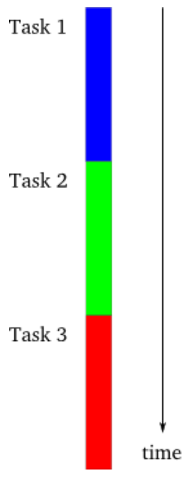

Figure 1: The single-threaded synchronous model

This is the simplest style of programming. Each task is performed one at a time, with one finishing completely before another is started. And if the tasks are always performed in a definite order, the implementation of a later task can assume that all earlier tasks have finished without errors, with all their output available for use — a definite simplification in logic.

We can contrast the single-threaded synchronous model with the multi-threaded synchronous model illustrated in Figure 2.

In this model, each task is performed in a separate thread of control. The threads are managed by the operating system and may, on a system with multiple processors or multiple cores, run truly concurrently,

[1]`http://krondo.com/?page_id=1327`

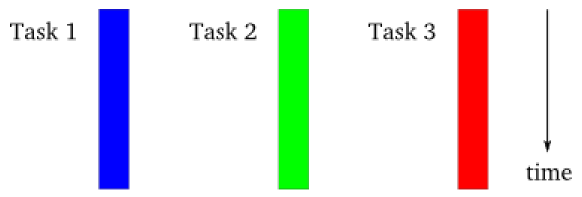

Figure 2: The threaded model

or may be interleaved together on a single processor. The point is, in the threaded model the details of execution are handled by the OS and the programmer simply thinks in terms of independent instruction streams which may run simultaneously. Although the diagram is simple, in practice threaded programs can be quite complex because of the need for threads to coordinate with one another. Thread communication and coordination is an advanced programming topic and can be difficult to get right.

Some programs implement parallelism using multiple processes instead of multiple threads. Although the programming details are different, conceptually it is the same model as in Figure 2.

Now we can introduce the asynchronous model[2], depicted in Figure 3:

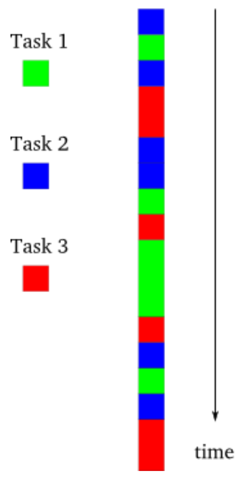

Figure 3: The asynchronous model

In this model, the tasks are interleaved with one another, but in a single thread of control. This is simpler than the threaded case because the programmer always knows that when one task is executing, another task is not. Although in a single-processor system a threaded program will also execute in an interleaved pattern, a programmer using threads should still think in terms of Figure 2, not Figure 3, lest the program work incorrectly when moved to a multi-processor system. But a single-threaded asynchronous system will always execute with interleaving, even on a multi-processor system.

[2]The multi-threaded synchronous model is also referred to as “preemptive multitasking,” with the asynchronous model called “cooperative multitasking.”

There is another difference between the asynchronous and threaded models. In a threaded system the decision to suspend one thread and execute another is largely outside of the programmer's control. Rather, it is under the control of the operating system, and the programmer must assume that a thread may be suspended and replaced with another at almost any time. In contrast, under the asynchronous model a task will continue to run until it explicitly relinquishes control to other tasks.

## 2 The Motivation

We've seen that the asynchronous model is in some ways simpler than the threaded one because there is a single instruction stream and tasks explicitly relinquish control instead of being suspended arbitrarily. But the asynchronous model clearly introduces its own complexities. The programmer must organize each task as a sequence of smaller steps that execute intermittently. If one task uses the output of another, the dependent task must be written to accept its input as a series of bits and pieces instead of all together.

Since there is no actual parallelism, it appears from our diagrams that an asynchronous program will take just as long to execute as a synchronous one. But there is a condition under which an asynchronous system can outperform a synchronous one, sometimes dramatically so. This condition holds when tasks are forced to wait, or block, as illustrated in Figure 4:

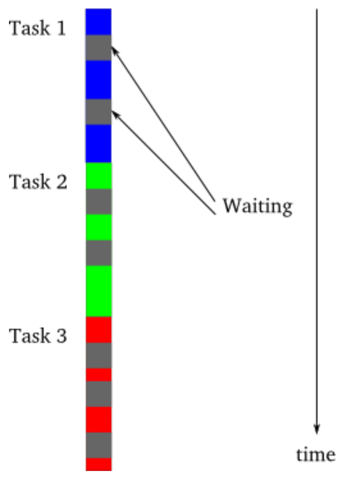

Figure 4: Blocking in a synchronous program

In the figure, the gray sections represent periods of time when a particular task is waiting (blocking) and thus cannot make any progress. Why would a task be blocked? A frequent reason is that it is waiting to perform I/O, to transfer data to or from an external device. A typical CPU can handle data transfer rates that are orders of magnitude faster than a disk or a network link is capable of sustaining. Thus, a synchronous program that is doing lots of I/O will spend much of its time blocked while a disk or network catches up. Such a synchronous program is also called a blocking program for that reason.

Notice that Figure 4, a blocking program, looks a bit like Figure 3, an asynchronous program. This is not a coincidence. The fundamental idea behind the asynchronous model is that an asynchronous program, when faced with a task that would normally block in a synchronous program, will instead execute some other task that can still make progress. So an asynchronous program only blocks when no task can make progress (which is why an asynchronous program is often called a non-blocking program). Each switch from one task to another corresponds to the first task either finishing, or coming to a point where it would have to block. With a large number of potentially blocking tasks, an asynchronous program can outperform a synchronous one by spending less overall time waiting, while devoting a roughly equal amount of time to real work on the individual tasks.

Compared to the synchronous model, the asynchronous model performs best when:

- There are a large number of tasks so there is likely always at least one task that can make progress.
- The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running.
- The tasks are largely independent from one another so there is little need for inter-task communication (and thus for one task to wait upon another).

These conditions almost perfectly characterize a typical busy network server (like a web server) in a client-server environment. Each task represents one client request with I/O in the form of receiving the request and sending the reply. A network server implementation is a prime candidate for the asynchronous model, which is why Twisted and Node.js, among other asynchronous server libraries, have grown so much in popularity in recent years.

You may be asking: Why not just use more threads? If one thread is blocking on an I/O operation, another thread can make progress, right? However, as the number of threads increases, your server may start to experience performance problems. With each new thread, there is some memory overhead associated with the creation and maintenance of thread state. Another performance gain from the asynchronous model is that it avoids context switching — every time the OS transfers control over from one thread to another it has to save all the relevant registers, memory map, stack pointers, FPU context etc. so that the other thread can resume execution where it left off. The overhead of doing this can be quite significant. (For more on this, we suggest taking CSCI1670).

## 3 Event-Driven Programming

We've seen that asynchronous programming allows us to spend less time waiting while utilizing only a single thread. But how exactly do we reap that benefit? Consider a program that blocks waiting to read input from stdin.

Listing 1: Reading from standard in

```
void monitor_stdin(void) {
    char input[500];
    while(1) {
        memset(input, '0', 500);
        scanf("%s", &input);
```

```
            printf("You typed %s\n", input);
        }
}
```

scanf() blocks until data is available on the stdin file descriptor. This means that the thread executing this function is not able to perform any other work until scanf() returns.

Most asynchronous programming libraries get around this problem in the following way:

- Rather than block on scanf(), continue executing the next line of code.
- Provide a chunk of code to run when scanf() actually finds data available on stdin. This **callback** block of code will execute when scanf() **would have** returned in the synchronous case [3].

If you've used AJAX before, this pattern will sound familiar. It's the same pattern employed by many GUI frameworks — "When the button is clicked, call this function, but don't just continually wait for the button to be clicked because that would block us from doing any other work." If you experiment with Twisted or Node.js, you'll again see this pattern — providing a callback to be performed when some event occurs in the future.

**libevent** is an asynchronous event notification library. While we won't be using it explicitly in this course, it serves as the foundation for many asynchronous programming frameworks. The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Currently, libevent supports /dev/poll, kqueue, select, poll, and epoll, all of which are different I/O event notification mechanisms. libevent provides a higher level abstraction around these mechanisms, choosing the best one based on the OS. In this course, we'll be using select() and epoll() directly, but libevent may be useful for future indulgences in asynchronous programming.

## 4 select() to the rescue

One traditional way to write network servers is to have the main server block on accept(), waiting for a connection. Once a connection comes in, the server fork()s or spaws a new thread, and the child process/thread handles the connection while the main server is able to service new incoming requests.

With select(), instead of having a process/thread for each client, there is usually only one thread that multiplexes all requests, servicing each client as much as it can.

So one main advantage of using select() is that your server will only require a single process/thread to handle all requests. Thus, your server will not need shared memory or synchronization primitives for different 'tasks' to communicate.

[3] Providing these callback functions (sometimes called "event handlers") can introduce some complexity to the programming logic. The programmer is responsible for maintaining state between when a blocking operation begins and when the callback is invoked. In the synchronous model, this state is automatically maintained on the stack. In the asynchronous model, the programmer must ensure the data is available for later use when the future event occurs. Luckily many languages provide closures to deal with this transparently, which you can read up on if you're curious. Debugging is also a bit more complicated in an asynchronous environment. If you've registered an event handler and then set a breakpoint inside of the callback code, the stack will always show something to the extent of "event loop called event handler." You need extra support to see where the handler was actually registered, i.e. what code caused the callback to be invoked.

select() works by blocking until something happens on a file descriptor (which can represent an actual file, a pipe, or a network socket). What's 'something'? Data coming in, being able to write to the file descriptor, or a timeout — you tell select() what you want to be woken up by.

Most select()-based servers are structured around a event loop consisting of the following:

- Fill up a fd_set structure with the file descriptors you want to know when data comes in on.
- Fill up a fd_set structure with the file descriptors you want to know when you can write on.
- Call select() and block until something happens.
- Once select() returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to.
- Repeat this process forever.

To get a comprehensive overview of select() with sample code, read sections 7.1 and 7.2 of Beej's guide at: `http://beej.us/guide/bgnet/output/html/multipage/advanced.html`. This should be one of your primary resources as you're implementing non-blocking I/O for the Snowcast server. If you have any questions about how to use select(), or about asynchronous programming in general, be sure to contact the TAs or stop by their office hours.