Programming Coroutines in C++

John Skaller

March 22, 2021

Contents

Chapter 1

Overview

The Felix programming language uses a novel architecture which is related to Tony Hoare's communicating sequential process (CSP) model. We will document the design and implementation but our primary goal here is to allow the C++ programmer to write code conforming to the run time library (RTL) requirements to implement the abstractions.

However we will first demonstrate how the system works by progressively introducing features in pure C++, without reference to the Felix RTL. When the API is finally introduced the programmer will then be familiar with the motivation and design issues. The problem is all such architectures involve complex interactions between components which cannot be introduced sequentially; but humans learn in steps.

1.1 Continuations

A *continuation* is an object which encapsulates the future of a program. What has already happened is called the history, it is represented by a sequence of significant events called a *trace*.

In a traditional system, when a subroutine is called, the caller is suspended with its state saved on the machine stack, and a value containing the address of the next instruction to be executed after the subroutine being called returns, is passed to that subroutine. This is called the *return address*.

When the subroutine is finished its work, it sets the program counter to the return address and the stack pointer back to the stack frame of the caller, thereby resuming the caller.

The callers data frame, together with the return address, is a *continuation,* and in particular, the *current continuation* at the point of the subroutine call.

We are going to develop a technique in which a heap allocated frame is used to hold local data instead of the machine stack, these frames will be linked with pointers forming a so-called *spaghetti stack*. These objects will be continuations. The reason for doing this will become apparent later. Suffice it to say that context switching between threads of control is done by stack swapping, and user space switching between threads using a spaghetti stack can be done extremely fast by simply swapping pointers. By contrast, swapping machine stacks during pre-emptions of hardware threads is extremely expensive, the number of allocable threads is strictly limited by the operating system, and the amount of memory used for a thread is high, even if it is not actually used, but the cost in address space reserved for each pthread is extreme. This is because with linear addressing, the address space for the maximum possible stack size must be pre-allocated, even if the actual memory is only assigned to it on demand. User space threads, called *fibres* using spaghetti stacks, impose no such demands: one pays, instead, an extra cost allocating and deallocating heap frames, in return for lightning fast context switching and minimal use of both memory and address space.

1.1.1 Emulating Subroutines

Our first task is to emulate subroutine operation. Consider the following class representing the base of a continuation:

```
struct con_t {
  con_t *caller; // caller continuation
  int pc;        // program counter
};
```

Now we will make a simple subroutine:

```
struct hello : con_t {
  int num;

  con_t *call(con_t *cc, int n) {
    caller = cc;
    num = n;
    return this;
  }

  con_t * resume() {
    ::std::cout << "Hello " << num << ::std::endl;
    auto tmp = caller;
    delete this;
    return tmp;
```

```
    }
};
```

It is important to note that we have split control flow into two parts.

In the `call` function we initialise the object with the callers current continuation and the argument of the call. Passing the callers current continuation explicitly and saving it is essential to the construction of a spaghetti stack. We have created subroutine but we have *suspended* execution until later.

In the `resume` method we actually execute the suspension, resuming it where it left off, or, in this case where it needs to start. When we're finished, we return the callers continuation.

Now lets write another procedure which calls our `hello` routine. We're going to call it 10 times, passing a counter than ranges from 1 to 10.

```
struct doit: public con_t {
  int counter;

  con_t *call(con_t *cc) {
    caller = cc;
    pc = 0;
    return this;
  }

  con_t *resume() {
    switch(pc){

    case 0:
      counter = 1; // init local variable

    case 1:
      pc = 2;
      return (new hello) -> call(this,counter);

    case 2:
      ++counter;
      if(counter <= 10) {
        pc = 1;
        return this;
      }
      {
        auto tmp = caller;
        delete this;
        return tmp;
```

```
      }
    } // end switch
  } // end resume
} // end doit
```

The `call` function implements a subroutine call to the `doit` procedure but leaves it in a suspended state, ready to run, starting at its entry point. The entry point is always given by setting the program counter `pc` to 0. The caller's continuation `cc` is passed in, it will be `nullptr` if this is the top level procedure.

Local variables such as `counter` are always represented by a non-static member. Function local automatic variables are not used because these cannot be preserved when the procedure is suspended.

To actually use this system we need a driver. So here's out mainline:

```
#include <iostream>
// put code here
int main() {
  doit *program  = new doit;
  con_t *p = program->call(nullptr);
  while(p)p=p->resume(); // driver
  return 0;
}
```

We first construct a continuation object for our top level procedure `doit` and then initialise it with its `call` method by passing the current continuation, which is `nullptr` because this is the top level procedure. The `while` loop which is running our code will terminate when a `nullptr` is returned.

Inside the `doit::resume` method we want to start at the entry point, `case 0` by initialising the loop control variable `counter` to 1.

Then we call the `hello` subroutine, passing it the counter. We also have to pass it our `this` pointer and set our `pc` value to the next code to execute at `case 2`.

So the steps in a subroutine call require:

1. Set our `pc` to the code we want to execute after the call
2. Construct the target subroutine object
3. Initialise the target routine by passing it our `this` pointer as the caller continuation
4. Also pass any arguments of the call
5. Return the pointer of the subroutine suspension we just create to the driver

The driver now resumes the subroutine at its entry point and steps through it.

To return from a subroute we simply return the saved `caller` to the driver which resumes stepping through it at the case it saved in its `pc` variable, and also delete the object.

Importantly all local variables of the caller are preserved because they're all non-static members of the C++ structure.

In the code above we repeatedly heap allocate the subroutine, and delete it again, on each loop iteration. This emulates the pushing and popping of frames on the machine stack, using the heap allocated frames a spaghetti stack uses.

The performance overhead in this case can be reduced by only allocating the subroutine frame once, and that will work .. in this case. But then, you could also just use a standard C++ procedure on the machine stack, in this case.

The Felix compiler performs these optimisations by static analysis, the C++ programmer using the Felix run time can perform them by hand. Felix also use a garbage collector so that pointers to variables in frames of objects cannot dangle, even if the procedure has completed.

But by now you're asking, why should we program this way, ever?

1.1.2 Interleaving

In order to grasp the advantages of our system, suppose we would like to run 100 copies of our `doit` procedure concurrently. If you use the system Operating System machinery you can do this by creating 100 pre-emptive threads all running the code at once.

If you have only, say, 4 CPUs, the OS can only execute 4 routines simultaneously, so it occasionally pre-empts one of the running threads, saves its state, and starts the CPU running the code of some other thread from where it was previously pre-empted. This is called *pre-emptive multi-tasking*.

The state of a conventional thread is stored in the machine stack, machine registers, and heap objects for which there are pointers in these objects (recursively). So the OS has to save a machine stack pointer and other registers, and set new values to restart another thread.

This may be OK but such control exchanges are slow and expensive, and they occur at random times outside program control, and so require special synchronisation devices such as locks, which are also expensive. However by far the biggest problem is that machine stacks are allocated in pages which are typically 4K bytes. With linear addressing the problem is much worse: the maximum possible size of a machine stack must be calculated and address space reserved for all of it. This is fine for 100 threads.

But what if you have 100 million threads? Your operating system will probably crash because it cannot handle that number of threads, and your virtual memory

system will be madly paging memory off disk, to reload the data of a machine stack on every swap. Although 64 bit machines have a lot of address space, a lot is wasted due to the 4K granularity.

By contrast, swapping spaghetti stacks can be done in user space by swapping a single pointer, and stacks only use the amount of space they need because they're allocated on the heap, which typically has 16 byte granularity, a lot less than 4K bytes. In addition with coroutine architecture, interleaving only occurs cooperatively, eliminating the need for a lot of expensive synchronisation devices.

We are going to first demonstrate interleaving in a rather silly way, as a prelude to introducing a real scheduler. Our silly scheduler will just execute things in a round robin order. We do a bit of each active fibre of control in turn until all are completed.

We will use a C++ queue object to do this.

```
int main() {
  auto q = ::std::queue<con_t*>;

  // initialise queue
  for (int i=0; i<100; ++i){
    doit *program  = new doit;
    con_t *p = program->call(nullptr);
    q.push(p);
  }

  // execute
  while (!q.empty()) {
    con_t *p = q.pop();
    p = p->resume();
    if(p) q.push(p);
  }
  return 0;
}
```

Wow! We just implemented a *cooerative multi-tasking* scheduler! It runs steps of a set of fibres until there is no work left to do. It uses a round-robin scheduling strategy which attempts to be "fair", executing one "step" of each thread in turn.

1.1.3 Faster Please

The round robin scheduler above using a C++ standard library queue is very slow. The reason is the library routine will allocate nodes in a doubly linked list whenever you push onto the queue, and delete one, whenever you pop.

To fix this problem we will add a new object representing a fibre:

```
struct fibre_t {
  con_t *cc;
  fibre_t *next;
  fibre_t() : cc(nullptr), next(nullptr) {}
  fibre_t(con_t *ccin) : cc(ccin), next (nullptr) {}
};
```

Now here is our new mainline:

```
int main() {
  fibre_t *current  = nullptr;

  // initialise queue
  for (int i=0; i<100; ++i) {
    doit *program  = new doit;
    con_t *p = program->call(nullptr);
    current = fibre_t (p, current);
  }

  // execute
  while (current) {
    con_t *p = current->cc;
    while(p)p=p->resume();
    fibre_t *tmp = current;
    delete current;
    current = tmp;
  }
  return 0;
}
```

Our code uses a stack protocol for the active fibres rather than the queue our previous example used. Why? The reason is simple: we don't care about ordering but we care about speed. Less instruuctions are required to push and pop from a singly linked list.

We have paid a small price, we have to allocate an extra object, the fibre, and find the top continuation of the fibre with an extra level of indirection. But then the processing loop is the same.

Lets encapsulate the fibre driver in a method:

```
struct fibre_t {
  con_t *cc;
```

```
    fibre_t *next;
    fibre_t() : cc(nullptr), next(nullptr) {}
    fibre_t(con_t *ccin) : cc(ccin), next (nullptr) {}
    void run_fibre() { while(cc)cc=cc->resume(); }
  };

  Lets encapsulate the operation in a new object, a scheduler:

  \(\begin\){minted}{c++}
  struct sync_sched {
    fibre_t * current;
    fibre_t *active;

    scheduler() : current(nullptr), active(nullptr) {}

    void push(fibre_t *fresh) {
      fresh->next = active;
      active = fresh;
    }
    fibre_t *pop() {
      fibre_t *tmp = active;
      active = active->next;
      return tmp;
    }

    void sync_run() {
      while(active) {
       current = active;
       active = active-> next; // pop
       current->run_fibre();
       delete current;
     }
   }
  };
```

1.2 Channels

Finally we're about to discover the whole reason for the model we've developed: the use of a device called a synchronous channel for communication. A channel is nothing more than a list of fibres which is either empty, or all the fibres are waiting to read, or all waiting to write.

```
  struct channel_t {
    fibre_t *next;
```

```
    channel_t () : next (nullptr) {}
    ~channel_t() {
      while (next) {
        fibre_t *f = (fibre_t*)*(unitptr_t)next & ~(unitptr_t)1u);
        fibre_t *tmp = f->next;
        delete f;
        next = tmp;
      }
    }

    void push_reader(fibre_t r) {
      r->next = next;
      next = r;
    }
    void push_writer(fibre_t w) {
      w->next = next;
      next = (fibre_t*)((uintptr_t)w & (uintptr_t)1u);
    }

    fibre_t *pop_reader() {
      fibre_t *tmp = next;
      if((uintptr_t)tmp & (unitptr_t)1u) return nullptr;
      next = next->next;
      return tmp;
    }
    fibre_t *pop_writer() {
      fibre_t *tmp = next;
      if(!((uintptr_t)tmp & (unitptr_t)1u)) return nullptr;
      next = (fibre_t*)next->next;
      return (fibre_t*)((uintptr_t)tmp & ~(uintptr_t)1u);
    }
  };
```

The casts there are because we're doing a hack! We're stealing the low bit of the fibre pointer and using it as a flag to determine if the channel is holding readers or writers. If the pointer is not null, and the low bit is 0, the channel holds readers, if the low bit is 1, the channel holds writers.

If we try to pop a reader from a channel containing writers, we get a nullptr back. Similarly if we try to pop a writer from a channel containing readers we get a nullptr back. Of course if the channel is empty we also get a nullptr back.

Now, we have to actually code the I/O operations. Unfortunately whilst the rules are simple, the implementation is a bit tricky!

We need to reserve a slot in a continuation for the I/O request: there are two

requests, one is to read, the other to write.

The operation is that, if a write is requested, the data in the data slot we added will be written to a reader on the channel, if there is one, otherwise the writer suspended and is added to the channel.

If a read is requested, the data from the data slot of a writer on the channel is moved to the reader, if there is one, otherwise the reader is suspended and is added to the channel.

If the I/O request is satisfied, the reader or writer on the channel is removed from the channel and now both the reader and writer are added to the active list of the scheduler, because they're ready to proceed. The scheduler then picks a new suspended fibre to run, removing it from the active list.

In principle the scheduler can pick any active fibre to proceed with. As an optimisation and to make it more intuitive, we will always pick the reader to continue after an I/O operation. Since the reader is only getting a single machine word, it has a chance to cast it to a pointer to the actual data to be transfered and copy it if necessary, before the writer can clobber it with new data.

Now we need to design the service requests. We will use an enum to specify the request types:

```
enum svc_code_t {
  read_request_code_e,
  write_request_code_e,
  spawn_fibre_request_code_e
};
```

I have thrown in a third request: spawn. This is a request to push a new fibre on the scheduler active list. We need this request because the code in a continuation does not have direct access to the scheduler! In fact, it does not have access to the fibre it is top of either!

Now we need the data packets for the requests. An I/O request needs to tell the system two things: the channel on which to perform the request, and, the location into which to put, or from which to get, the word to be transfered:

```
struct io_request_t {
  svc_code_t svc_code;
  channel_t *chan;
  void **pdata;
};
```

For the spawn:

```
struct spawn_fibre_request_t {
  svc_code_t svc_code;
  fibre_t *tospawn;
};
```

Finally we have to write up these data structures into a variant:

```
union svc_req_t {
  io_request_t io_request;
  spawn_fibre_request_t spawn_fibre_request;
  svc_code_t get_code () const { return io_request.svc_code; }
};
```

Note: technically, the `get_code` method could violate the C++ Standard rules by accessing the wrong component. It would be more correct to separate the code out from the I/O request packet types, and make a union of them, then make a struct with the code as the first member and the union as the second. The problem is it is hard to construct such an object. It is easier to cheat, and construct a simple request object and cast it to the union type, which is again breaking the rules.

We have to extend the continuation to hold a request:

```
// continuation
struct con_t {
  con_t *caller; // caller continuation
  int pc;        // program counter
  union svc_req_t *svc_req; // request
  virtual con_t *resume()=0;
  virtual ~con_t(){}
};
```

and now the fibre has to handle it by returning it:

```
// fibre
struct fibre_t {
  con_t *cc;
  fibre_t *next;

  // default DEAD
  fibre_t() : cc(nullptr), next(nullptr) {}

  // construct from continuation
  fibre_t(con_t *ccin) : cc(ccin), next (nullptr) {}
```

```
  // immobile
  fibre_t(fibre_t const&)=delete;
  fibre_t& operator=(fibre_t const&)=delete;

  // destructor deletes any remaining continuations in spaghetti stack
  ~fibre_t() {
    while(cc) {
      con_t *tmp = cc->caller;
      delete cc;
      cc = tmp;
    }
  }

  // run until either fibre issues a service request or dies
  svc_req_t *run_fibre() {
    while(cc) {
      cc=cc->resume();
      if(cc && cc->svc_req) return cc->svc_req;
    }
    return nullptr;
  }
};
```

Here's our new scheduler:

```
// scheduler
struct sync_sched {
  fibre_t *current; // currently running fibre, nullptr if none
  fibre_t *active;  // chain of fibres ready to run

  sync_sched() : current(nullptr), active(nullptr) {}

  // push a new active fibre onto active list
  void push(fibre_t *fresh) {
    fresh->next = active;
    active = fresh;
  }
  // pop an active fibre off the active list
  fibre_t *pop() {
    fibre_t *tmp = active;
    if(tmp)active = tmp->next;
    return tmp;
  }
  void sync_run();
```

```
  void do_read(io_request_t *req);
  void do_write(io_request_t *req);
  void do_spawn_fibre(spawn_fibre_request_t *req);
};
```

Now the scheduler has to handle the service request.

```
// scheduler subroutine runs until there is no work to do
void sync_sched::sync_run() {
  current = pop(); // get some work
  while(current) // while there's work to do
  {
    svc_req_t *svc_req = current->run_fibre();
    if(svc_req)  // fibre issued service request
      switch (svc_req->get_code())
      {
        case read_request_code_e:
          do_read(&(svc_req->io_request));
          break;
        case write_request_code_e:
          do_write(&(svc_req->io_request));
          break;
        case spawn_fibre_request_code_e:
          do_spawn_fibre(&(svc_req->spawn_fibre_request));
          break;
      }
    else // the fibre returned without issuing a request so should be dead
    {
      assert(!current->cc); // check it's adead fibre
      delete current;       // delete dead fibre
      current = pop();      // get more work
    }
  }
}
```

The new methods now need to be written. Here's the read operation:

```
void sync_sched::do_read(io_request_t *req) {
  fibre_t *w = req->chan->pop_writer();
  if(w) {
    *current->cc->svc_req->io_request.pdata =
      *w->cc->svc_req->io_request.pdata; // transfer data

    // null out svc requests so they're not re-issued
```

```
      w->cc->svc_req = nullptr;
      current->cc->svc_req = nullptr;

      push(w); // onto active list
      // i/o match: reader retained as current
    }
    else {
      req->chan->push_reader(current);
      current = pop(); // active list
      // i/o fail: current pushed then set to next active
    }
  }
```

The write operation is similar but differs because we always want the reader to be active:

```
  void sync_sched::do_write(io_request_t *req) {
    fibre_t *r = req->chan->pop_reader();
    if(r) {
      *r->cc->svc_req->io_request.pdata =
        *current->cc->svc_req->io_request.pdata; // transfer data

      // null out svc requests so they're not re-issued
      r->cc->svc_req = nullptr;
      current->cc->svc_req = nullptr;

      push(current); // current is writer, pushed onto active list
      current = r; // make reader current
    }
    else {
      req->chan->push_writer(current); // i/o fail: push current onto channel
      current = pop(); // reset current from active list
    }
  }
```

and finally the spawn is simple, we will make the spawned fibre current to emulate the behaviour of a subroutine call:

```
  void sync_sched::do_spawn_fibre(spawn_fibre_request_t *req) {
    current->cc->svc_req=nullptr;
    push(current);
    current = req->tospawn;
  }
```

1.2.1 Contracts

in all the above code the methods have pre-conditions and post-conditions and the objects have public invariants; we have not stated these. These things can be said to be the *terms* of a *contract*.

In production code, we should always state the contract terms, but we should go further: we should *prove*, or at least outline a proof, that the terms of the contract are adhered to, where we have written both parties of the contract.

For a subroutine call, we should prove the caller meets the pre-condition, for an object, that the invariants are maintained. Sometimes this is trivial: for example, the `pop_reader` function may return a `nulptr`. Did we handle both the null and non-null cases? A quick inspection shows `pop_reader` is only called in one place in `do_write`, and we have indeed checked if it has returned null, and handled both the null and non-null cases. In turn, the dereference of the returned pointer is safe, because it is inside the scope of the branch handling the non-null case.

It is less obvious that `current` cannot be null, and if it is, that the current continuation `cc` of the indicated fibre must also be non-null.

How can we be sure we have an exhaustive proof? Our code is simple enough to elaborate all possible control flow paths, and annotate each point on the path with the state of the important variables.

There is a problem with contract annotations: they make the code more verbose, which can make it harder to understand. On the other hand the specification of the contract terms can also aid comprehension.

The canonical exemplar of this issue is the tension between dynamically typed languages like Python, in which arguments, return values, and variables do not carry type annotations, and languages like C++ where static type annotations can be used. In particular the use of the `auto` keyword in C++ for local variables is common: it improves comprehensibility by removing the need to write messy type annotations, at the expense of needing to calculate from non-local information what the type would be. For the compiler, this also removes a source of error checking.

We have not used auto in our code, but this is because our types are all fairly simple and it's useful to see exactly what they are: in general it is reasonable to use it for local variables where the type is easily deduced from lexically close code.

1.3 Using channels

Since we have now established all the code needed to service I/O requests we need to discover how into perform the requests and how to use the I/O as a synchronisation device. A continuation with a `resume` method that performs,

directly or indirectly, a service request is said to be a *coroutine*. If no service requests can be performed executing the resume method, the continuation is a trivial coroutine or subroutine. All the routines we have presented so far are procedures, that is, none of them return a value, instead they rely on effects for their utility.

A routine can perform a service call indirectly if it invokes a coroutine, or a routine which invokes a coroutine .. and so on. Therefore being a proper coroutine depends on the transitive closure of all routines that a given routine can call.

There is a cruicial distinction between proper coroutines and procedures: mere procedures can be optimised to put their objects on the machine stack, because they cannot be suspended. In fact, an ordinary C procedure can be used, which can be even more efficient.

Now we need to design a little demo which uses channels. We will start with something apparently simple which can easily be done in C++ already: we will map a list of integers to a map of their squares. This will involve three coroutines and two channels.

The first coroutine accepts a C++ list of integers and writes them down a channel.

```
struct producer : con_t {
  ::std::list<int> *plst;
  ::std::list<int>::iterator it;
  channel_t *chan;
  union {
    void *iodata;
    int value;
  };
  io_request_t w_req;

  con_t *call(
    con_t *caller_a,
    ::std::list<int> *plst_a,
    channel_t *outchan)
  {
    caller = caller_a;
    plst = plst_a;
    pc = 0;
    w_req.chan = outchan;
    return this;
  }

  con_t *resume() override {
    switch (pc) {
```

```
        case 0:
          it = plst->begin();
          pc = 1;
          w_req.svc_code = write_request_code_e;
          w_req.pdata = &iodata;
          return this;

        case 1:
          if(it == plst->end()) {
            auto tmp = caller;
            delete this;
            return caller;
          }
          value = *it++;
          svc_req = (svc_req_t*)(void*)&w_req; // service request
          return this;
        default: assert(false);
      }
    }
  };
```

It is intersting to note that, because the pc is not reset at the end of case 2, it remains at the value 1, therefore each time the resume method is called after first entering case 1 branch, it will be called again from the case 1 entry point, effecting a loop.

Note also we did not delete the object after it returns. We will see how this is handled and the constraints that imposes when we write the mainline.

I have used a union to hold the int to be transfered, aliased with the actual type required. The assumption here is that the int is not larger than a `void*`. This saves heap allocating the integer and passing a pointer, which avoids a memory management hassle. Again, technically, our code is ill formed because we're moving a different union component from the one we stored but it will work.

The consumer code is similar, but it constructs a list from its inputs instead.

```
  struct consumer: con_t {
    ::std::list<int> *plst;
    union {
      void *iodata;
      int value;
    };
    io_request_t r_req;

    con_t *call(
```

```
      con_t *caller_a,
      ::std::list<int> *plst_a,
      channel_t *inchan_a)
    {
      caller = caller_a;
      plst = plst_a;
      r_req.chan = inchan_a;
      pc = 0;
      return this;
    }

    con_t *resume() override {
      switch (pc) {
        case 0:
          pc = 1;
          r_req.svc_code = read_request_code_e;
          r_req.pdata = &iodata;
          return this;

        case 1:
          svc_req = (svc_req_t*)(void*)&r_req; // service request
          pc = 2;
          return this;

        case 2:
          plst->push_back(value);
          pc = 1;
          return this;
        default: assert(false);
      }
    }
  };
```

You will see something very interesting here: this code is an infinite loop! This is correct. It is the usual case for coroutines! You may wonder how the loop terminates. We will soon see, but the answer briefly is that it does not terminate, it simply fails to resume.

Now we will implement a transducer which reads an integer and writes its square. There's a bit more housekeeping because it uses both an input and output channel.

```
  struct transducer: con_t {
    union {
      void *iodata;
```

```
    int value;
  };
  io_request_t r_req;
  io_request_t w_req;

  con_t *call(
    con_t *caller_a,
    channel_t *inchan_a,
    channel_t *outchan_a)
  {
    caller = caller_a;
    r_req.chan = inchan_a;
    w_req.chan = outchan_a;
    pc = 0;
    return this;
  }

  con_t *resume() override {
    switch (pc) {
      case 0:
        pc = 1;
        r_req.svc_code = read_request_code_e;
        r_req.pdata = &iodata;
        w_req.svc_code = write_request_code_e;
        w_req.pdata = &iodata;
        return this;

      case 1:
        svc_req = (svc_req_t*)(void*)&r_req; // service request
        pc = 2;
        return this;

      case 2:
        value = value * value; // square value
        svc_req = (svc_req_t*)(void*)&w_req; // service request
        pc = 1;
        return this;
      default: assert(false);
    }
  }
};
```

Now we need to set up the system to test it.

```
int main() {
  // create the input list
  ::std::list<int> inlst;
  for (auto i = 0; i < 20; ++i) inlst.push_back(i);

  // output list
  ::std::list<int> outlst;

  // create scheduler
  sync_sched sched;

  // create channels
  channel_t chan1;
  channel_t chan2;

  // create fibres
  fibre_t *prod = new fibre_t ((new producer)->call(nullptr, &inlst, &chan1));
  fibre_t *trans = new fibre_t ((new transducer)->call(nullptr, &chan1, &chan2));
  fibre_t *cons = new fibre_t ((new consumer)->call(nullptr, &outlst, &chan2));

  // push initial fibres onto scheduler active list
  sched.push(prod);
  sched.push(trans);
  sched.push(cons);

  // run it
  sched.sync_run();

::std::fflush(stdout);

  // the result is now in the outlist so print it
  ::std::cout<< "List of squares:" << ::std::endl;
  for(auto v : outlst) ::std::cout << v << ::std::endl;
}
```

The key thing here is to note that the `sync_run` function only returns when there is no work to do. Any fibre that terminated is deleted by the scheduler. Those fibres that remain hanging on channels will be deleted by the channel destructors.

When a produce is connected to a series of transducers and terminated by a consumer, the structure is called a *pipeline*. Pipelines have some nice properties, including the fact that connection is associative. What this means is that, for example, a producer connected to a transducer is a producer, two transducers connected together are a transducer, and a transducer connected to a

consumer is a consumer. This means you can defined new pipeline components by connecting arbitrary components in the sequence into new components, so the system is highly modular.

A coroutine which depends only on data it reads and writes from channels, and does not modify its environment, is said to be *pure*. Pure coroutines are referentially transparent. We can count the producer as pure, assuming the input list bound to it is not modified, and the transducer is pure, however the consumer is not.

It is useful to note that a pipeline is semantically equivalent to a monad. You can see from even this small example coroutines excel at handling streams: the producer in the example is converting a spatial list to a temporal stream.

1.4 Control inversion and peers

The most important thing about using coroutines is that they dispense with the master/slave relationship enforced by subroutine calling and stacks. Instead, coroutines are peers. This can only be done by fast context switching which is precisely what our system provides.

It is vital to understand why peer to peer relations are better. Consider the methods for mapping a list of integers to a list of its squares. There are two established methods for doing this with subroutines.

The first method is to use a higher order function called map that accepts the list and a per element processing function, and returns a new list. The processing function is a callback slave of the master map function. The map has to scan the list and construct a new one, which is easy to do because it is the master and gains the advantage of structured programming, that is, the implicit coupling of parameter passing and control flow. But the slave callback is hard to write in complex applications because it keeps losing its state.

The second method is to use an iterator as a slave to scan the list, and possibly construct a new one, and write a loop to drive it that does the application dependent work. This is much better for the client programmer than writing a callback, because structured programming tracks the state. However the iterator is a callback, and is therefore hard to write for complex data structures.

What we actually want is that *both* the iteration and client computations think they're masters, so they are both easy to program, and this is precisely what our system provides: all the components are coroutines which read and write data.

When a master is converted to a slave, or vice-versa, this is called *control inversion*. The master/slave relation is also known as client/server or push/pull. It is the bane of a programmers existence to have to take a master routine and control invert it into a callback. The reason is that the structured programming

advantage of implicit coupling of control flow and data stacking is lost in a callback, and has to be manually put back. In our system, all routines are masters in effect, but we have to use disciple to implement the patterns for operations like subroutine calling and returning. This is the price, along with the need to use heap allocations instead of the much faster machine stack, in order to support high speed context switching.

Although our system works and is extremely fast, it retains some of the problems of a typical stack machine which also has pointers: pointer to data in a returned continuation will dangle, because the continuation has been deleted. We will look later at some ways to solve this problem. One method is obvious: use a garbage collector. The current Felix run time system does precisely that and ensures no pointers can dangle. Garbage collection is fast and achieves better thruput than other methods, and is fully general. However naive collectors are not compatible with real time operation.

1.5 Thread Safety

The code we have presented so far is not thread safe. There are two things we want to do. The first is to ensure two threads running separate schedulers allow fibres to communicate with channels. The second thing we want to do is allow multiple threads to service the same active list, in effect providing a thread pool to execute queued jobs.