Appendix of Submission #46

Stackful Coroutine Made Fast

ACM Reference Format:
Stackful Coroutine Made Fast. 2023. Appendix of Submission #46. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 In-stack generator

```
struct GCTX {
    uint64_t _ip, _fp;
};

// used in the caller
class Generator : public NoCopyNoMove {
    GCTX _ctx;
    uint64_t _val;
public:
    template<typename F, typename... Ts>
    __attribute__ ((always_inline))
    Generator(F&& f, Ts&&... xs) {
        auto fp = __builtin_frame_address(0);
        _val = f((GCTX*)fp, std::forward<Ts>(xs)...);
        register uint64_t callee_ip asm("rdx");
        register uint64_t callee_fp asm("rsi");
        asm volatile ("" : "=r"(callee_ip), "=r"(callee_fp) : :
            "rbx", "rcx", "rsp", "r8", "r9", "r10", "r11",
            "r12", "r13", "r14", "r15");
        _ctx = {callee_ip, callee_fp};
    }

    __attribute__ ((always_inline))
    void resume() {
        register uint64_t& _ip asm("rdx") = _ctx._ip;
        register uint64_t& _fp asm("rsi") = _ctx._fp;
        register uint64_t retval asm("rax");
        __asm__ volatile (R"(
            lea 1f(%%rip), %%rdi
            xchg %%rbp, %0
            jmp *%1
        1:
        )" : "+r"(_fp), "+r"(_ip), "=r"(retval) : :
            "rbx", "rcx", "rsp", "rdi", "r8", "r9",
            "r10", "r11", "r12", "r13", "r14", "r15");
        _val = retval;
    }
    uint64_t value() { return _val; }
    operator bool() { return _ctx._ip; }
    ~Generator() { while(unlikely(*this)) resume(); }
};

// the promise object of the generator, should be first
// constructed on entry, and gets destructed last on exit
class GPromise: public NoCopyNoMove {
    GCTX _ctx;
public:
    GPromise(GCTX* fp) {
        _ctx._ip = (uint64_t)__builtin_return_address(0);
        _ctx._fp = (uint64_t)fp;
    }
    __attribute__ ((always_inline))
    void yield(uint64_t x) {
        register uint64_t& retval asm("rax") = x;
        register uint64_t& _caller_ip asm("rdi") = _ctx._ip;
        register uint64_t& _caller_fp asm("rsi") = _ctx._fp;
        __asm__ volatile (R"(
            lea 1f(%%rip), %%rdx
            xchg %%rbp, %1
            jmp *%0
            1:
        )" : "+r"(_caller_ip), "+r"(_caller_fp), "+r"(retval) :
           : "rbx", "rcx", "rdx", "rsp", "r8", "r9",
             "r10", "r11", "r12", "r13", "r14", "r15");
    }
    ~GPromise() {
        auto frame = (uint64_t*)__builtin_frame_address(0);
        frame[1] = _ctx._ip;
        register uint64_t _callee_ip asm("rdx");
        __asm__ volatile ("xor %0, %0" : "=r"(_callee_ip));
    }
};
```

2 Sum of Sequence (using in-stack generator)

```
__attribute__ ((noinline))
uint64_t seq_gen(GCTX* fp, uint64_t c) {
    // create promise obj on entry, destructed on exit
    GPromise g(fp);
    while(c)
        g.yield(c--);
    return 0;
}

__attribute__ ((noinline))
uint64_t sum_seq(uint64_t c) {
    uint64_t sum = 0;
    for (Generator g(&seq_gen, c); g; g.resume())
        sum += g.value();
    return sum;
}
```

3 Hanoi (using in-stack generator)

```
__attribute__ ((noinline))
void _Hanoi(GPromise& g, char n, char f, char t, char a) {
    if (n == 0) return;
    _Hanoi(g, n-1, f, a, t);
    g.yield(n + (f << 8) + (t << 16));
    _Hanoi(g, n-1, a, t, f);
}

__attribute__ ((noinline))
uint64_t Hanoi(GCTX* fp, char n) {
    GPromise g(fp);
    _Hanoi(g, n, 'a', 'b', 'c');
    return 0;
}
```

```

__attribute__ ((noinline))
uint64_t test_hanoi(uint64_t c) {
    uint64_t sum = 0;
    for (Generator g(&Hanoi, (char)c); g; g.resume()) {
        auto n = g.value() % 256;
        auto f = g.value() / 256 % 256;
        auto t = g.value() / 256 / 256;
//          printf("move disk %d from '%c' to '%c'\n", n, f, t);
        sum++;
    }
    return sum;
}
```

4 Write_fully

```
__attribute__ ((noinline))
void wait_for_ready() { proton::thread_yield(); }

__attribute__ ((noinline))
ssize_t write_some(void *buf, size_t count) {
  wait_for_ready();
  // auto some = std::min((size_t)(1 + rand() % 16), count);
  auto some = 800;
  total += some;
  return some;
}

__attribute__ ((noinline))
ssize_t write_fully(void *buf, size_t count) {
  size_t size = 0;
  while (count) {
    ssize_t s = write_some(buf, count);
    if (s < 0) return s;
    else if (s == 0) return size;
    assert(s <= count);
    count -= s;
    size += s;
    (char*&)buf += s;
  }
  return size;
}
```