Lecture 12: User Level Threads

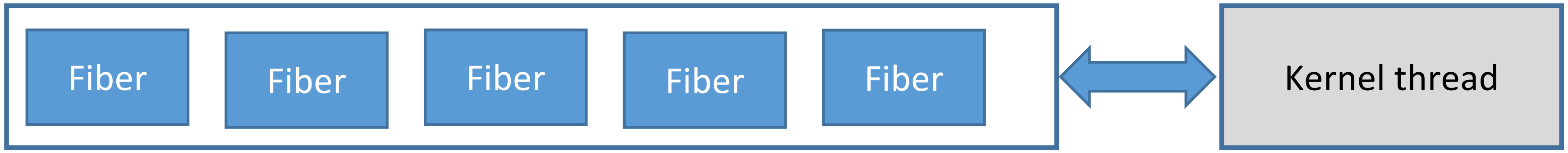
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- Boost fiber library
- Quiz-3

boost::fibers::fiber

- A fiber is a userland thread unlike the kernel thread (e.g., pthread maps **1x1** with kernel thread in Linux)
  - Several fibers can map with single pthread (**M x N** threading)



- Fiber emulates much of the std::thread
  - Extends the concurrency taxonomy
    - On a single computer, multiple processes can run
    - Within a single process, multiple threads can run
    - Within a single thread, multiple fibers can run
- Builds on top of boost::context
  - Each fiber has its own stack, registers, instruction pointer..
    - It means they can scheduled cooperatively
- It is super easy to create a fiber

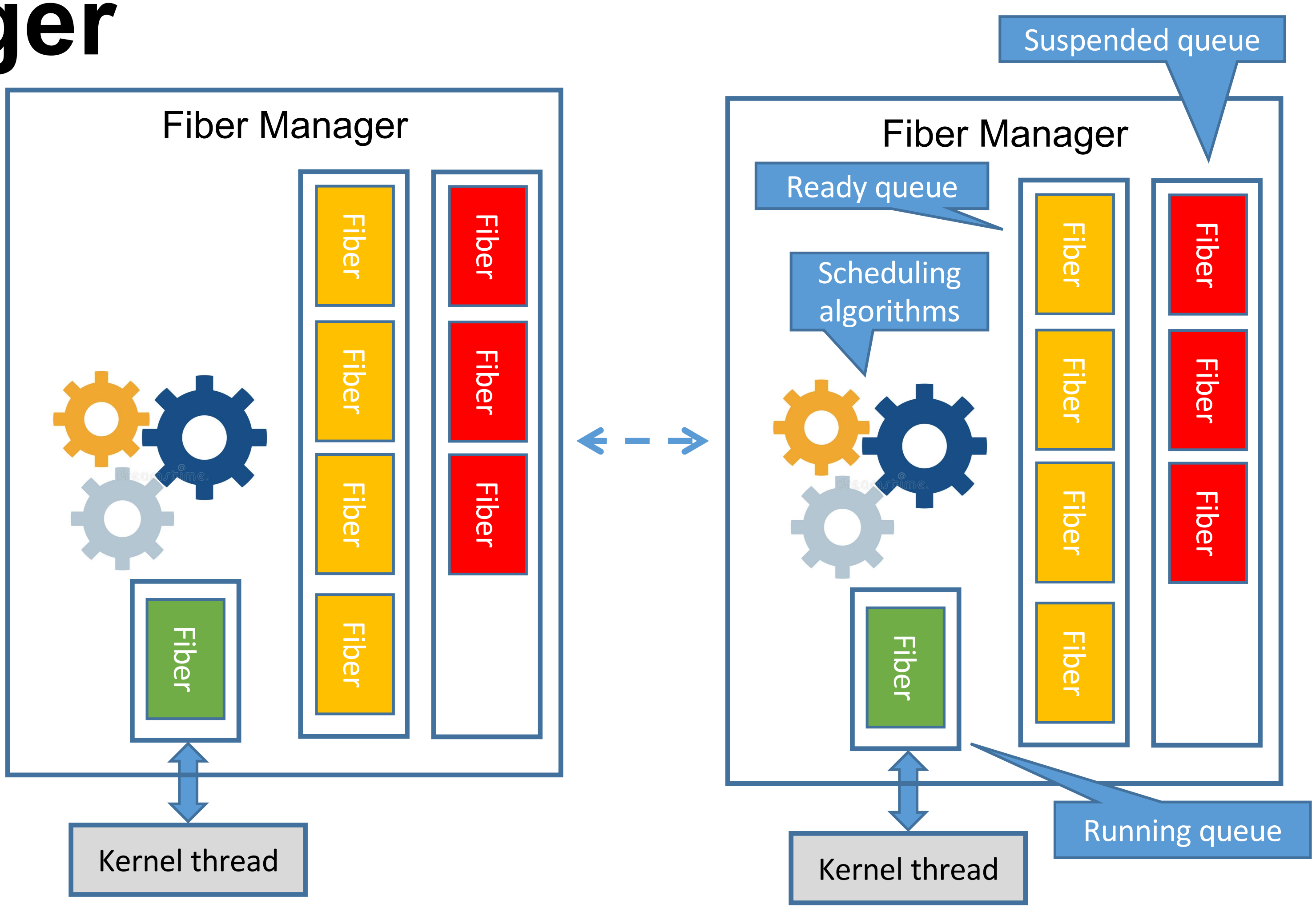
```
boost::fibers::fiber (F, [=]() { /*Do something*/ }); // Spawns a fiber F
```

Fiber v/s Thread

- A thread can run only one fiber at a time
  - Although several fibers can be queued up for execution at a thread at any given time
- Creating several fibers by a single thread doesn’t imply parallelism unlike creating several threads
  - By default fibers created by a thread will run by that thread only, but it can be detached to allow its execution at any other thread

Fiber Manager

- The fibers in a thread are coordinated by a fiber manager
  - The manager created/managed silently by the fiber library

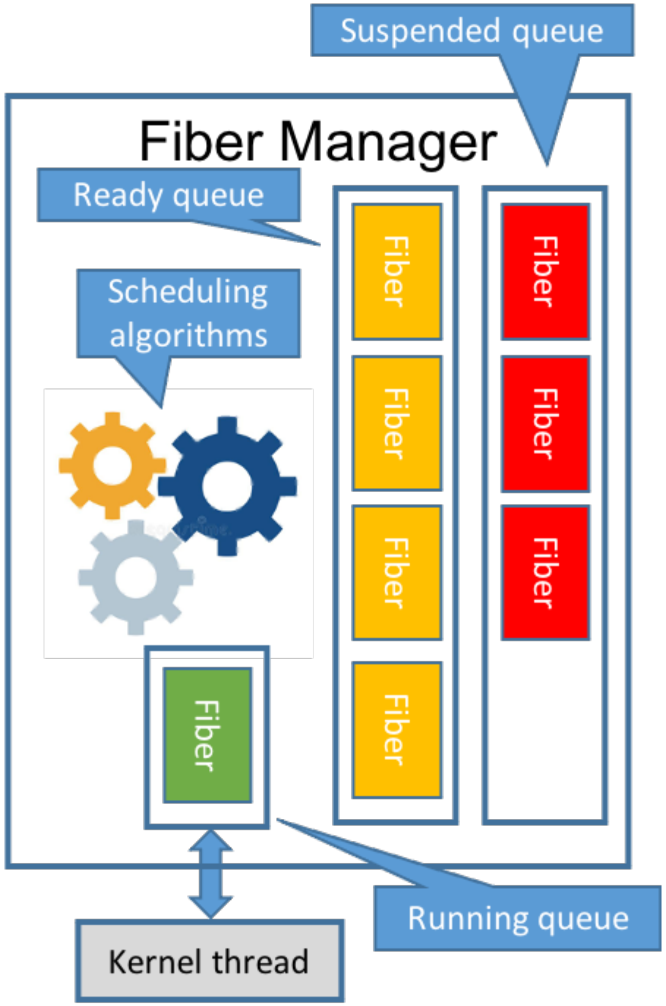


Fiber Manager

- Similar to threads, a fiber can be in the running, suspended or ready state
- Fibers trade control with the manager in a cooperative way
  - boost::this_fiber::yield();
  - boost:this_fiber::sleep_for
  - boost:this_fiber::sleep_until
  - boost:fibers::mutex
  - boost:fibers::condition_variable
  - some_fiber.join()
  - ….

These operations will land the fiber into which queue (ready/suspended)?

- Manager uses a scheduling algorithm to select a ready fiber to run (any similarity with Linux kernel?)
- Manager carries out the context switch to swap between the fibers
  - Kernel thread blocks if there are no ready fibers



Fiber Scheduler

- Manager uses a default round-robin scheduler
  - Scheduling within a thread
- Boost fibers provides shared_work and work_stealing as alternative schedulers to round_robin
  - Scheduling across the threads
- Boost fibers also allow creation of a custom scheduler

```
void thread_function() {
  boost::fibers::use_scheduling_algorithm<my_own_fiber_scheduler>();
}
```

Fiber Context Switching is Extremely Fast

Table 1.3. time per thread (average over 10,000 - unable to spawn 1,000,000 threads)

pthread	`std::thread`	`std::async`
54 μs - 73 μs	52 μs - 73 μs	106 μs - 122 μs

Table 1.4. time per fiber (average over 1.000.000)

fiber (16C/32T, work stealing, tcmalloc)	fiber (1C/1T, round robin, tcmalloc)
0.05 μs - 0.09 μs	1.69 μs - 1.79 μs

Source: https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/fiber/performance.html

Question

- Is there any difference(s) between calling std::future.get() / std::future.wait() on a std::thread v/s a boost::fiber?

Creating Fibers

```
#define millisleep(x) std::this_thread::sleep_for(std::chrono::milliseconds(a))
.....
millisleep(500);
millisleep(100);
```

```
#include <boost/fiber/all.hpp>
#define millisleep(x) boost::this_fiber::sleep_for(std::chrono::milliseconds(a))
.....
boost::fibers::fiber f1 ([=]() { millisleep(500); }); // Fiber F1 launched
boost::fibers::fiber f2 ([=]() { millisleep(100); }); // Fiber F2 launched
f1.join(); // Wait for termination of F1
f2.join(); // Wait for termination of F2
```

Method call can be made directly instead of passing lambda, e.g. f1(foo, p1, p2, p3), where ‘p’ are parameter to method `foo’

- What would be the execution time of these two programs?
  - Note that it’s a single thread execution in each case
- Note that both programs are using different implementations of sleep
  - Fiber manager handle its own sleep, but not the std::sleep

Fiber Futures

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f1 (std::async([=](){ return fib(n-1); }));
    std::future<uint64_t> f2 (std::async([=](){ return fib(n-2); }));
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    boost::fibers::future<uint64_t> f1 (boost::fibers::async([=](){ return fib(n-1); }));
    boost::fibers::future<uint64_t> f2 (boost::fibers::async([=](){ return fib(n-2); }));
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

For n=40:

- Which of these programs would be faster?
- Which of these programs is a parallel program?

Yielding Fibers

```
boost::fibers::fiber f1([=]() {
  cout << “A ”;
  boost::this_fiber::yield();
  cout << “B ”;
  boost::this_fiber::yield();
  cout << “C ”;
});

boost::fibers::fiber f2([=]() {
  cout << “D ”;
  boost::this_fiber::yield();
  cout << “E ”;
  boost::this_fiber::yield();
  cout << “F ”;
});

f1.join();
f2.join();
```

- yield() saves the context of currently running fiber, and places it inside the **ready** queue
  - Manager can schedule it again based on the scheduling algorithm
- What will be the output of this program?

Producer-Consumer using Fibers

```
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([&]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([&]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  str = “Hello Fiber”;
  cnd.notify_one();
});

f1.join();
f2.join();
```

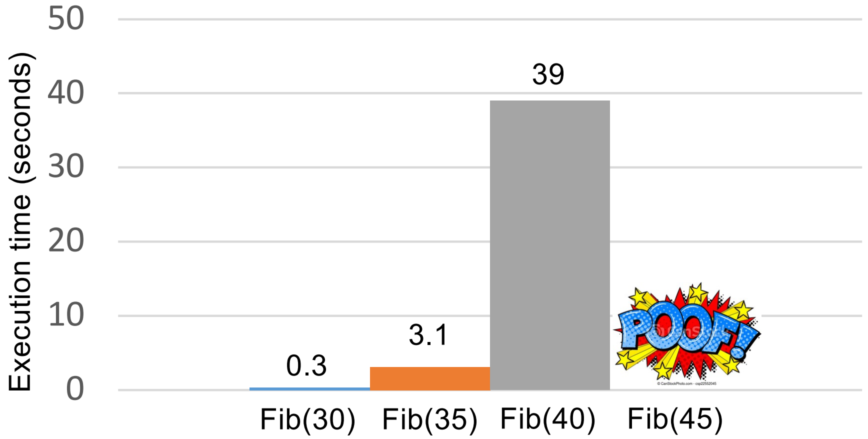
- Fiber F1 moving into suspended queue, and then back into ready queue after a notify from F2
  - Single thread execution!

Fiber Pitfalls

```
std::mutex mtx;
std::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  str = “Hello Fiber”;
  cnd.notify_one();
});

f1.join();
f2.join();
```

- Can you spot the difference?
  - What effect it would cause, and why?

Fiber Overheads / Limitations



- Language restriction
  - Fiber library requires C++11
  - Cannot be used in C-based HPC libraries/programs
- (Serious) Runtime overheads
  - Graph shown for calculating recursive Fibonacci that spawns detached fiber for every recursive call until threshold reached (n<10)
    - Total nested fibers created
      - Fib [30, 57K], Fib [35, 635K], Fib [40, 7049K]
  - Single worker used!
  - Platform details
    - AMD EPYC 7551 32-core processor
    - Ubuntu 18.04.3 LTS
    - GCC version 7.5.0
      - -O3 flag used
    - Boost version 1_80_0

Reading Materials

- Fibers
  - https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/index.html

Installing Boost Context and Fiber Library

- Install Boost
  - wget https://boostorg.jfrog.io/artifactory/main/release/1.80.0/source/boost_1_80_0.tar.gz
  - tar xvfz boost_1_80_0.tar.gz
  - cd ~/boost_1_80_0/
  - ./bootstrap.sh --prefix=/absolute/path/to/boost-install --with-libraries=fiber,context
  - ./b2 install
- Compile programs
  - g++ -O3 -I/absolute/path/to/boost-install/include -L/absolute/path/to/boost-install/lib Program.cpp –lboost_fiber –lboost_context -lpthread
- Execute programs
  - export LD_LIBRARY_PATH=/absolute/path/to/boost-install/lib:$LD_LIBRARY_PATH
  - ./a.out

Next Lecture (L #13)

- Mid semester review