Lecture 04: Context Switching Inside the User Space

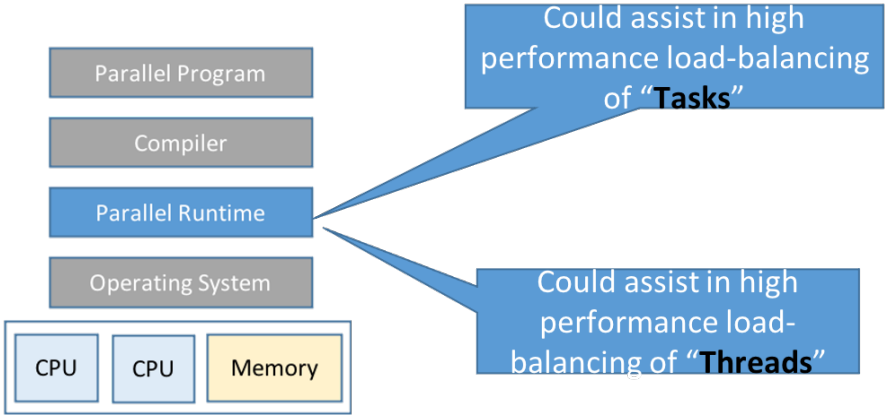
Vivek Kumar

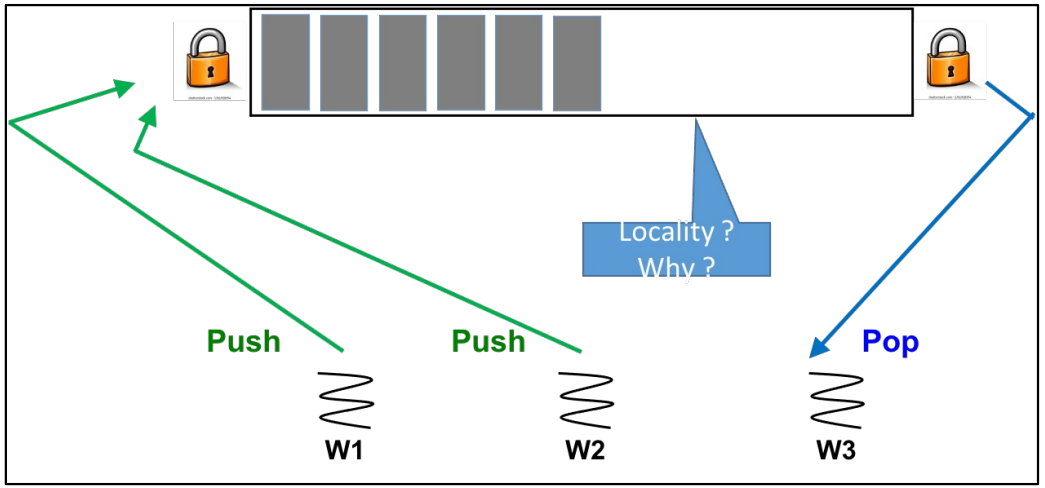
Computer Science and Engineering

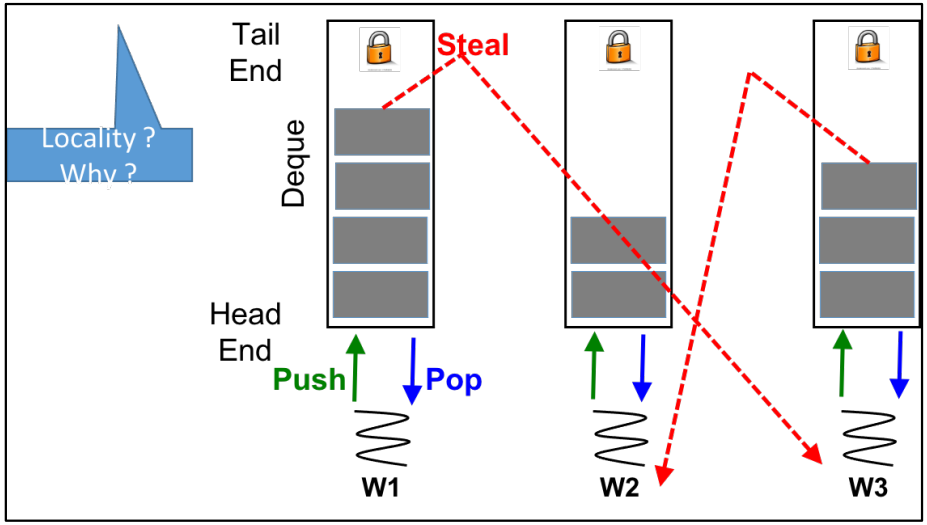
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)







- Parallel runtime system for task-scheduling
  - Work-sharing
  - Work-stealing

Today's Class

- Threading models
- Boost C++ libraries for concurrency
  - Context
  - Introduction to Fibers

Multicore Parallel Programming Landscape

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64
    std::future<uint64
    //get will block
    return f1.get() + f2.get();
  }
}
```

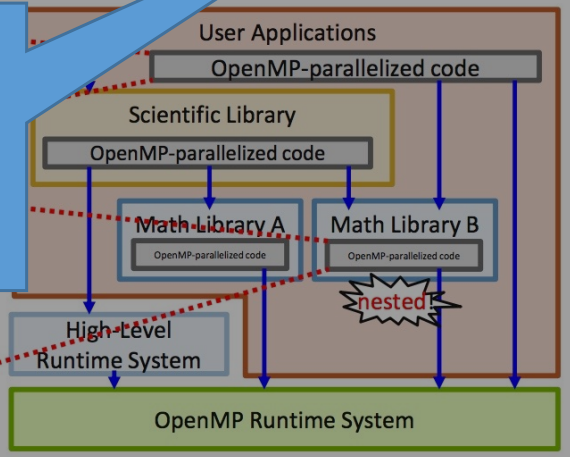
Requirement: Parallel runtime system that could schedule "tasks"

```
#pragma omp par
for (i = 0; i <
  dgemv(matrix[

// BLAS library
void dgemv(...)
  #pragma omp p
  for (i = 0; i < n; i++)
    dgemv_seq(data[n], i);
}
```

Code Example

Focus of this lecture: Parallel runtime system that could schedule "threads"



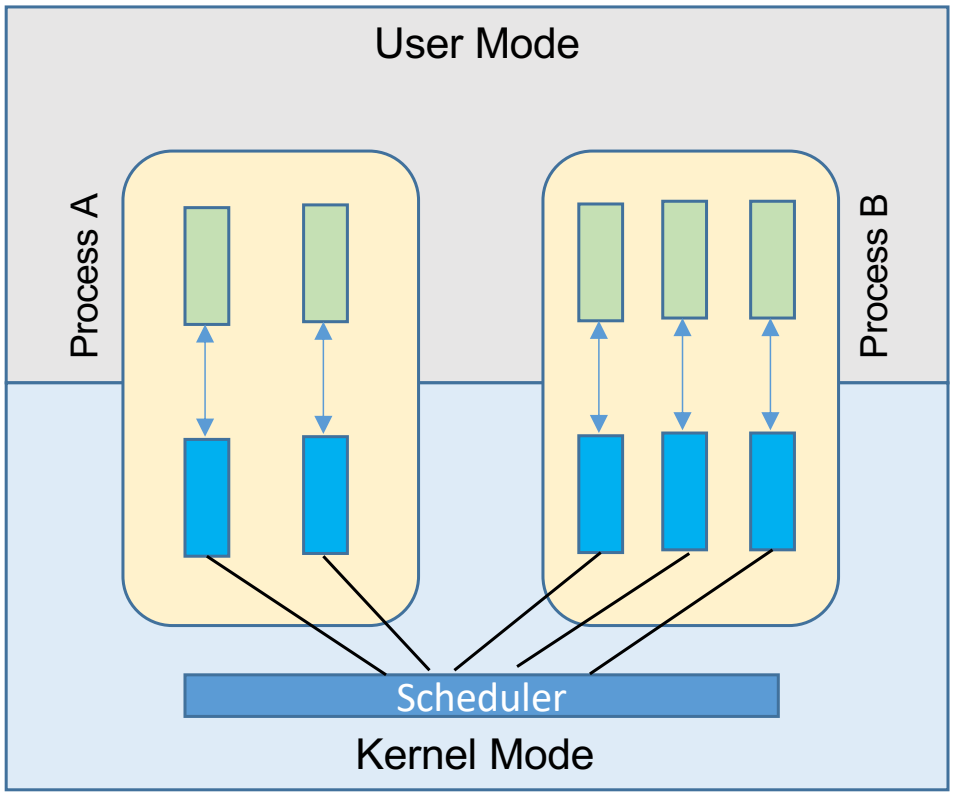
OpenMP picture source: https://www.mcs.anl.gov/~iwasaki/pdfs/papers/PACT2019_slides.pdf

- Newer tasks-based programming models
  - OpenMP tasks, HClib, TBB, HPX, etc.
- Thread-based programming model
  - OpenMP work-sharing loops
    - Most popular
    - Being used extensively in several real world applications
      - Some of them were written really long ago
        - Hard to change to newer programming models
      - Nested **thread** creation
        - Creates threads exponentially

Threading Model

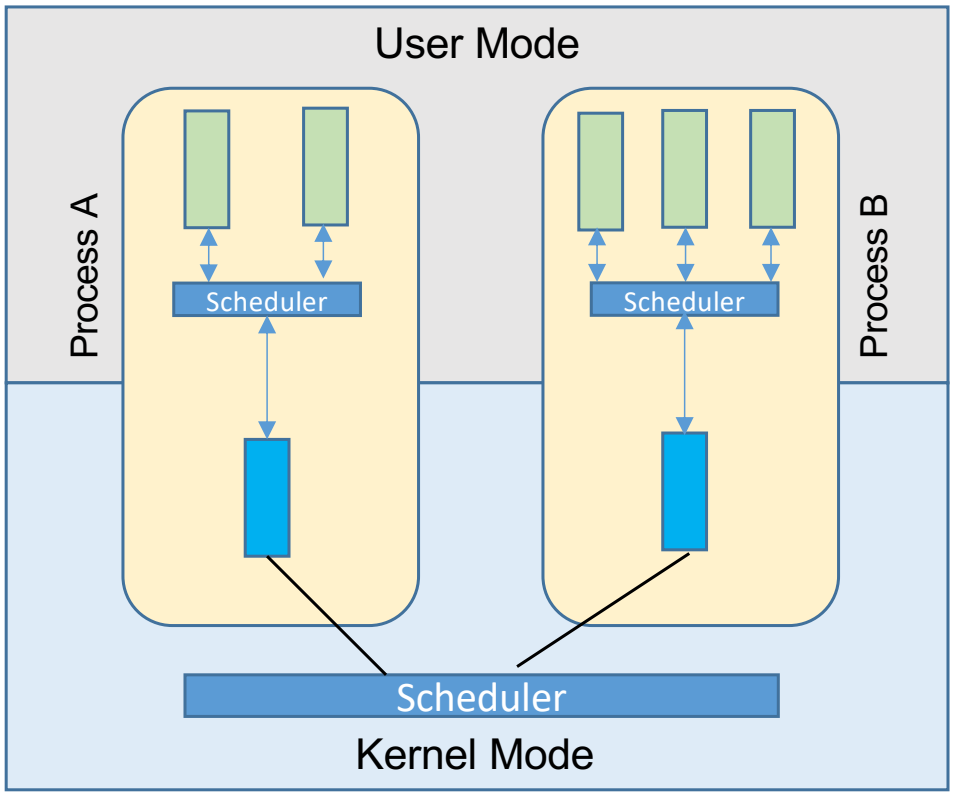
- **1x1** threading Model (Kernel Level Threads)
- **MxN** threading model (User Level Threads)

1x1 Threading Model



- Every thread created by the user has 1x1 mapping with the kernel thread
  - E.g., pthread library on Linux
- OS manages all thread operations
  - Heavyweight operations
    - Thread creation
    - Context switches
  - Scheduling policy solely managed by the kernel

MxN Threading Model



- User gets to create several threads, but each of these threads can be mapped to a single kernel level thread
  - Some JVMs have been doing it
- Runtime library (in user space) manages all thread operations
  - Lightweight operations (OS is totally unaware of user level thread operations)
    - Thread creation
    - Context switches
  - Flexible scheduling policies can be implemented

Today's Class

- Threading models
- Boost C++ libraries for concurrency
  - Context
  - Introduction to Fibers

Boost C++ Libraries

WELCOME TO BOOST.ORG!

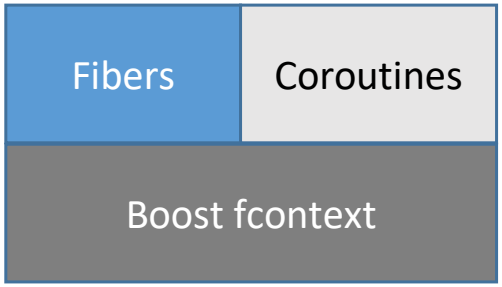
Boost provides free peer-reviewed portable C++ source libraries.

We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages the use of Boost libraries for all users with minimal restrictions.

We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Beginning with the ten Boost Libraries included in the Library Technical Report (TR1) and continuing with every release of the ISO standard for C++ since 2011, the C++ Standards Committee has continued to rely on Boost as a valuable source for additions to the Standard C++ Library.

Boost Context Library

- Provides a sort of cooperative multitasking on a single thread
- By providing an abstraction of the current execution state in the current thread, a *fcontext_t* instance represents a specific point in the application's execution path
  - stack (with local variables)
  - stack pointer
  - all registers and CPU flags
  - instruction pointer
- Provides the means to suspend the current execution path and to transfer execution control, thereby permitting another *fcontext_t* to run on the current thread
  - Helps in extremely low latency context switching of execution inside userspace (around 19 CPU cycles on x86_64 platform [1])
- Disadvantage
  - Not supported on all platforms as based on assembly code



Documentation: https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/index.html

[1] https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/context/performance.html#performance

Boost Context: Only Two Low-level Core APIs

- Create new context

```
fcontext_t make_context( /* pointer to top of new stack */,
                         /* size of the new stack */,
                         /* function to call when starting new context */);
```

- Jump to new context

```
void* jump_fcontext( /*current context */,
                     /* new context */,
                     /* some more arguments … */);
```

How to Handle Blocking Task?

```
/* thread local variable */
fcontext_t* steal_loop_context;

void* worker_routine(void* args) {
  steal_loop_context = make_context( /* Method steal_task_from_victim */);
  jump_context(/* current context */, steal_loop_context);
}
steal_task_from_victim(int wid) {
  while( /* thread pool is active */) {
    /* find and execute tasks */
  }
}
get() {
  future* f = get_current_future();
  if(f->is_not_ready()) {
    /* create/save current context and switch to steal_loop_context */
  }
  else return f->value;
}
```

```
/* User application */
.......
void foo() {
  future_t* future =
                async(.......);
  future.get();
}
```

Boost Context C++11 Library

- Two primary operations
  - callcc
    - Call with current **continuation**
    - Captures current continuation and triggers a context switch
  - Resuming a saved continuation
    - resume()
      - Can be used to switch across different continuations

Boost Context Library: Example

```
#include <boost/context/all.hpp>
void A() {
  cout<< “IN-A” << endl;
  /* Do something */
  cout<< “OUT-A” << endl;
}
void B() {
  cout<< “IN-B” << endl;
  /* Do something */
  cout<< “OUT-B” << endl;
}
void C() {
  cout<< “IN-C” << endl;
  /* Do something */
  cout<< “OUT-C” << endl;
}
int main() {
  A();
  B();
  C();
}
```

Figure-1

```
#include <boost/context/all.hpp>
ctx::continuation A(ctx::continuation cont) {
  cout<< “IN-A” << endl;
  cont = cont.resume();
  /* Do something */
  cout<< “OUT-A” << endl;
  return std::move(cont);
}
/* Methods B & C rewritten as A above */
int main() {
  ctx::continuation a = ctx::callcc(A);
  ctx::continuation b = ctx::callcc(B);
  ctx::continuation c = ctx::callcc(C);
  a.resume();
  b.resume();
  c.resume();
}
```

Figure-2

- Figure-1

IN-A
OUT-A
IN-B
OUT-B
IN-C
OUT-C

- Figure-2

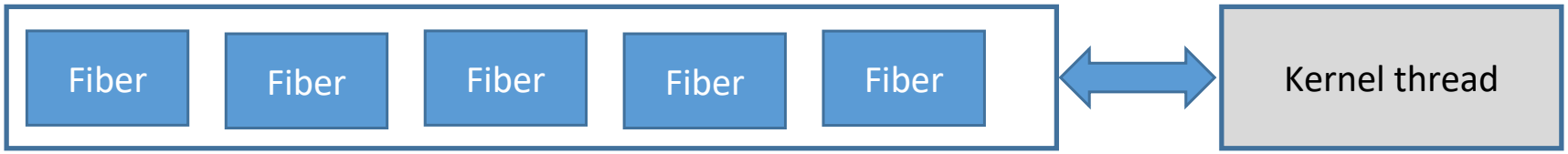
IN-A
IN-B
IN-C
OUT-A
OUT-B
OUT-C

Today's Class

- Threading models
- Boost C++ libraries for concurrency
  - Context
  - Introduction to Fibers

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

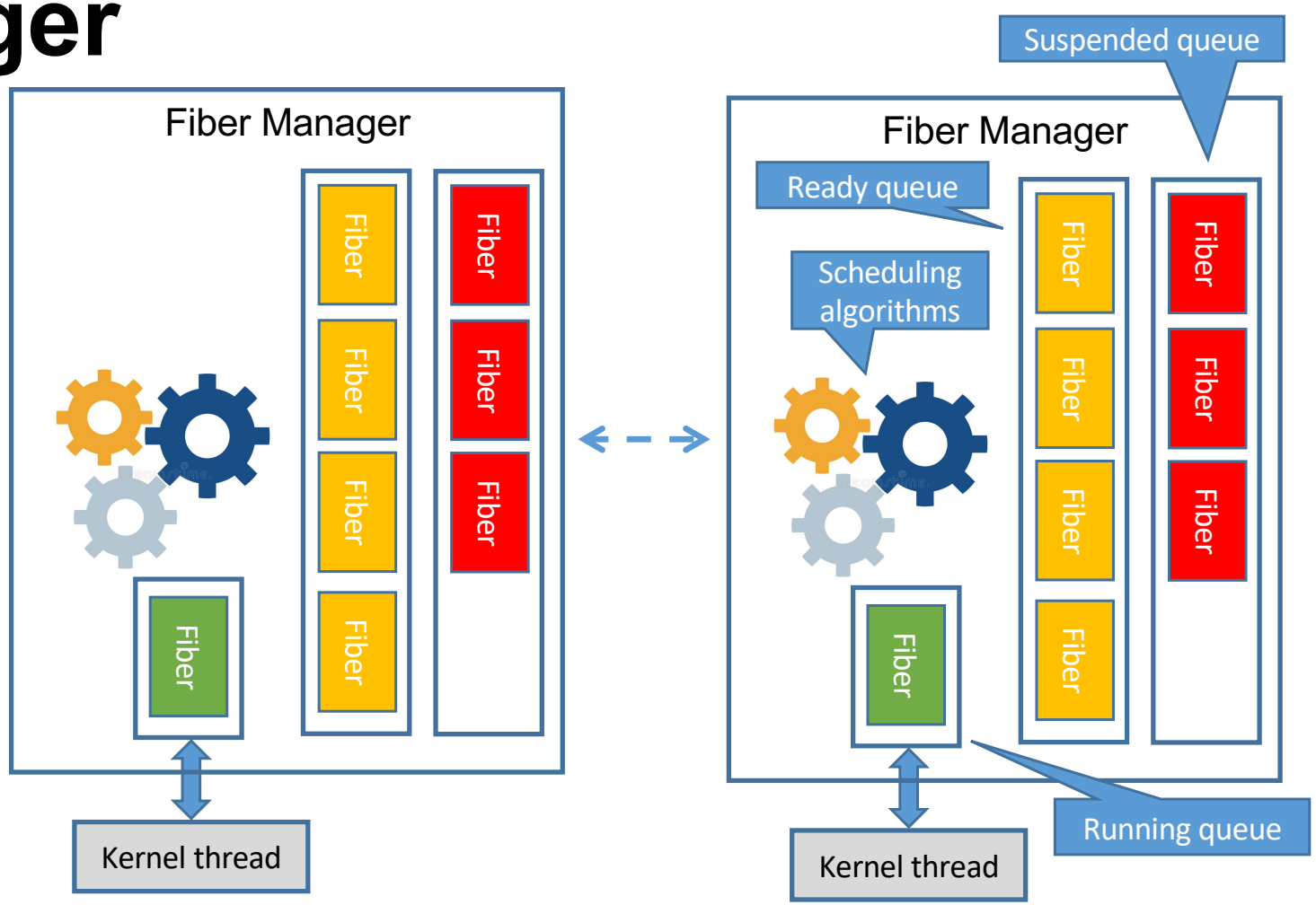
```
boost::fibers::fiber (F, [=]() { /*Do something*/ }); // Spawns a fiber F
```

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

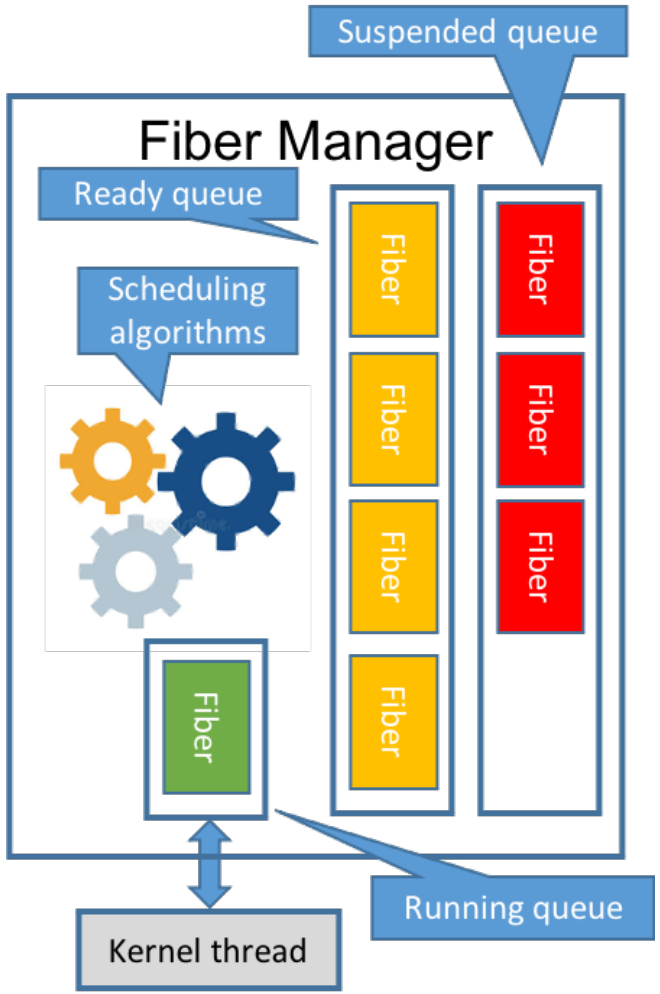
- The fibers in a thread are coordinated by a fiber manager
  - The manager created/managed silently by the fiber library



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()
  - ….
- 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

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



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 future.get() / future.wait() on a std::thread v/s a 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
```

- 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

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

Reading Materials

- Context
  - https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/index.html
- 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 (#05)

- Boost library for concurrency (contd.)
- Argobots runtime for User Level Threads (ULTs)