Mutex and Synchronization

Drs. Kropp and Youssfi

Mutual Exclusion

- To avoid race conditions between threads, use C++17 `mutex`
- `mutex` = MUTual EXclusion

```
// Declared a mutex to be locked
std::mutex my_mutex;

void critical_func () {

  // Start of critical section
  std::lock_guard<std::mutex> my_lock (my_mutex);

  // Safe access to shared data by multiple threads
  // ...
  // end of critical section
  // my_lock is automatically unlocked and destroyed

}
```

Mutual Exclusion

- A C++17 `std::lock_guard` can be used to lock the `mutex` over the critical section or a local scope
- `std::lock_guard` is automatically unlocked and destroyed at the end of a local scope (i.e. critical section)
- Example: incrementing a global variable by multiple threads

```
// Declared a mutex to be locked
std::mutex my_mutex;

void critical_func () {

  // Start of critical section
  std::lock_guard<std::mutex> my_lock (my_mutex);

  // Safe access to shared data by multiple threads
  // ...
  // end of critical section
  // my_lock is automatically unlocked and destroyed

}
```

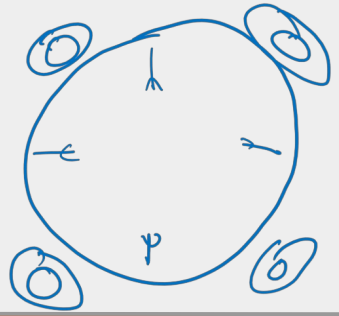
Mutex versus Sephaphore

- Mutex is specifically for ensuring mutual exclusion
- Semaphore is a more general term for communicating between threads

Avoiding Deadlock

- C++17 can lock two or more `mutex` objects at same time with a call to `std::scope_lock` constructor
- Example: dining philosophers
  - Each philosopher will try lock both the left and right forks at the same time.
  - If one fork is already locked, the constructor will release the other mutex (i.e. fork).
  - Constructor will return when both mutexes are locked

```
mutex forks[5];

void philosopher(int id) {

  // grab left and right fork at the same time!
  scoped_lock guard (forks[id], forks[(id + 1) % 5]);

  PrintThread{} << "Philosopher " << id;
  PrintThread{} << " took left & right forks" << endl;

  // Once a philosopher has two forks, he/she will
  // eat. If deadlock occurs no philosopher will eat!
  PrintThread{} << "Philosopher " << id
  PrintThread{} << " had a good meal" << endl;

}
```



Synchronizing Threads

- Previously, we’ve been simply managing access to resources
- What if we want to synchronize actions between threads?
- Threads can wait for events triggered by conditional variables

Synchronizing

- `std::condition_variable`
- The wait has to be on a `unique_lock` so that the waiting thread can unlock the `mutex`
  - `unique_lock` is like `lock_guard` with a bit more flexibility
- Threads can also notify other threads waiting on events associated with conditional variables so that they can proceed.

```
std::condition_variable cv;
// thread waits for event

// "lock" protects access to predicate
std::unique_lock<std::mutex> lock(mtx);
while (!predicate) cv.wait(lock);
...
...
// thread notifies other thread to stop waiting
cv.notify_all();
```

Flavors of wait and notify

Two flavors for condition variable notify:

- `condition_variable::notify_one()`
  - Unblocks one of the threads currently waiting for this condition.
- `condition_variable::notify_all()`
  - Unblocks all threads currently waiting for this condition.

3 flavors for condition variable wait:

- `condition_variable::wait()`
  - The execution of the current thread (which shall have locked mutex) is blocked until notified.
- `condition_variable::wait_until()`
  - The execution of the current thread (which shall have locked mutex) is blocked either until notified or until abs_time, whichever happens first.
- `condition_variable::wait_for()`
  - The execution of the current thread (which shall have locked mutex) is blocked during rel_time, or until notified (if the latter happens first).

Synch Threads Example

```
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id (int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (!ready) cv.wait(lck);
  // ...
  std::cout << "thread " << id << '\n’;
}

void go() {
  ready = true;
  cv.notify_all();
}

int main (){
  std::thread threads[10]; // Spawns 10 threads
  for (int i=0; i<10; ++i)
    threads[i] = std::thread(print_id,i);

  std::cout << "10 threads ready to race...\n";
  go();

  for (auto& th : threads)
    th.join();

  return 0;
}
```