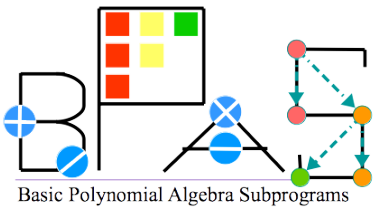
Building Object-Oriented (and Optional) Parallelization from C++ Primitives

Alexander Brandt

Ontario Research Center for Computer Algebra
Department of Computer Science
University of Western Ontario, Canada

March 23, 2022



ORCCA
Ontario Research Centre for Computer Algebra

Introduction

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) {
3          return;
4      }
5      int k = i + (j-1)/2;
6      mergeSort(A, i, k);
7      mergeSort(A, k, j);
8      merge(A, i, k, j);
9  }
```

→ Where recursive calls are independent, those function calls can be executed **concurrently**

→ **Fork** the execution control flow and then **join** or **sync** them

→ Hardware must support parallelism with multi-core or multi-processors

Compiler-Level Automatic Parallelization

→ CILK and OPENMP provide automatic parallelization though compiler extensions

→ Very easy but flexibility more challenging

```
void mergeSort(int* A, int i, int j) {
  //... base case, k
  cilk_spawn mergeSort(A, i, k);
  mergeSort(A, k, j);
  cilk_sync
  merge(A, i, k, j);
}
```

```
void mergeSort(int* A, int i, int j) {
  //... base case, k
  #pragma omp parallel num_threads(2)
  {
    #pragma omp sections {
      #pragma omp section {
        mergeSort(A, i, k);
      }
      #pragma omp section {
        mergeSort(A, k, j);
      }
    }
  }
  merge(A, i, k, j);
}
```

Fork-Join Parallelism with BPAS

→ Object-oriented

→ Standard C++, no compiler extensions

```
1  void mergeSort(int* A, int i, int j) {
2      //... base case, k
3      threadID id;
4      ExecutorThreadPool& pool =
5          ExecutorThreadPool::getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12
13     merge(A, i, k, j);
14 }
```

Outline

What is Multithreading?

Multithreading is **not** multi-core or multiprocessor.

↳ But together they give us **performance**.

Multithreading is a programming model which allows for multiple, concurrent **threads** of execution, each with their own **context**.

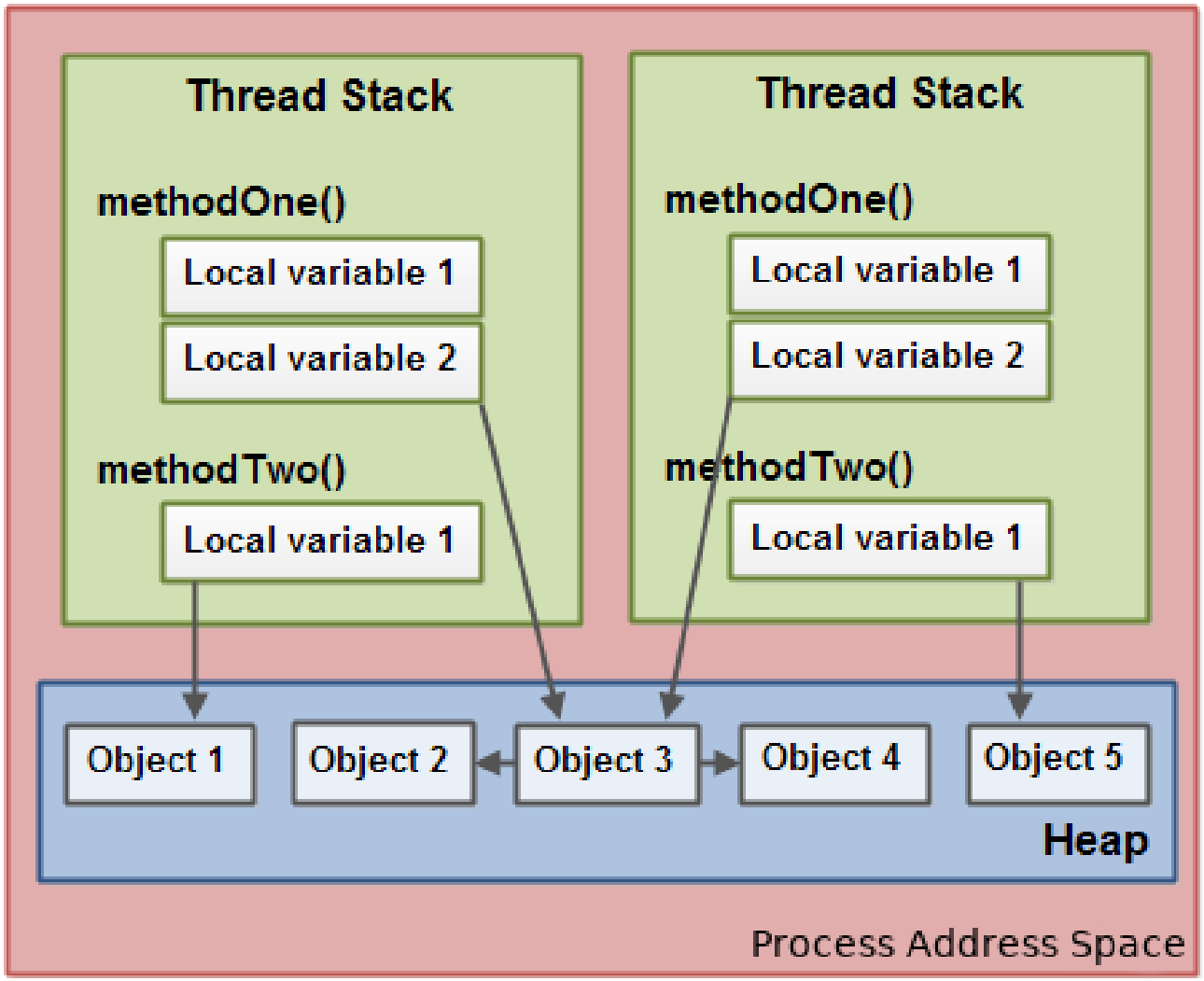
→ Thread: the smallest processing unit that can be scheduled by the OS.

→ Context: A thread's own and unique local variables, PC, register values, stack.

→ But, threads within the same process share an address space (heap).

Process: An instance of a program being executed. Usually handled by the operating system and requires a lot of overhead to instantiate and set-up properly. A process can contain many threads.

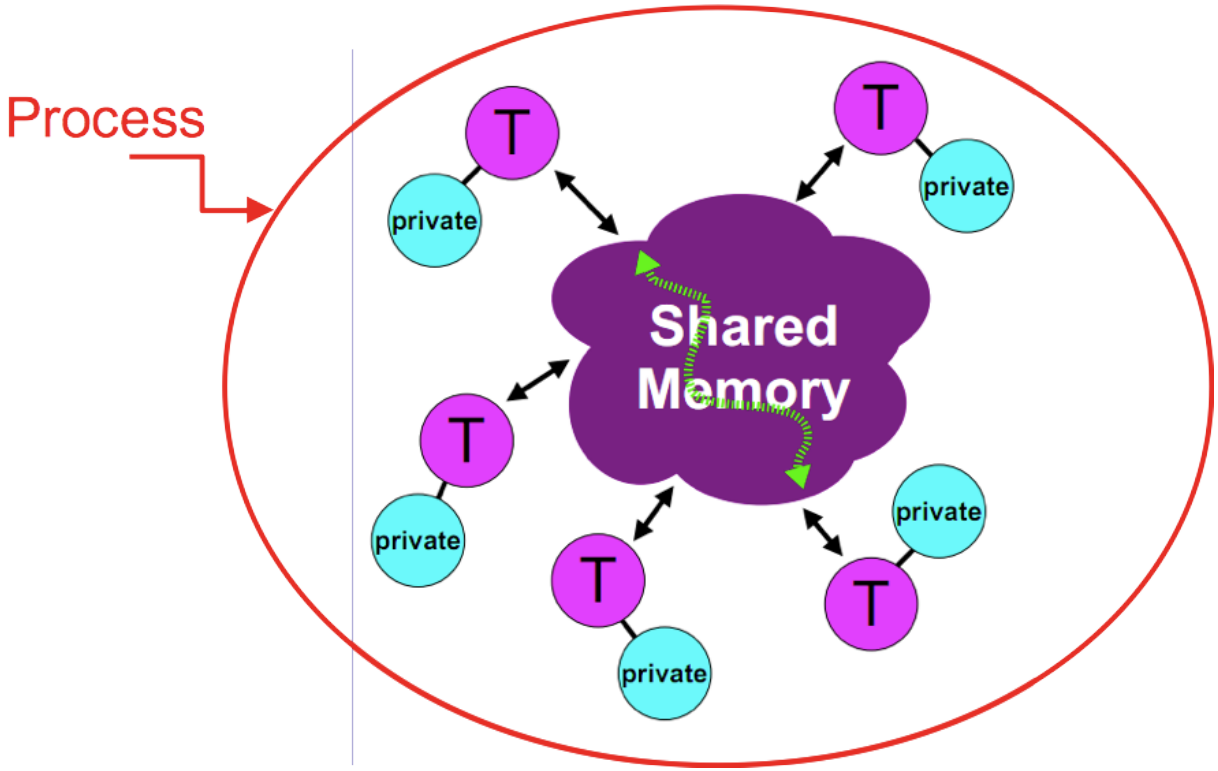
Multithreading Memory Model (1/2)



Multihreading Memory Model (2/2)

It is also possible to separate the heap into “thread-local” or private memory and shared memory.

Different programming languages allow this construct.



Multithreading and Multi-core

Using multiple threads does not require multiple cores (processors).

- → A single processor can handle multiple threads via **time-division multiplexing**: the sharing of time on the datapath between threads.
  - ↳ This requires **context switching**—updating the state of the processor's register file, PC, stack pointer, etc. to match the thread's context.
- → With multiple cores (processors), threads can run simultaneously.
  - ↳ Each core/processor has its own registers, etc. ⇒ no context switching.
  - ↳ If the number of threads exceeds the number of processors (cores) then multiple threads must run on one processor (core) via context switching.
- → **Thread scheduling** is *hard*. The OS and hardware typically handle it
  - ↳ Preemptive scheduling: threads are interrupted and context switches are forced.
  - ↳ Non-Preemptive scheduling: a.k.a cooperative scheduling, threads yield themselves to allow others to run. Threads are not interrupted.

Context Switching

- → To switch contexts, the context/state being switched from must first be saved in some way.
- → Generally, this occurs by storing all the values of the registers, PC, etc. in some special data structure (e.g. a process control block) and storing that somewhere in memory.
  - ↳ Usually in the operating system's memory address space.
- → Context switching is expensive, particularly if threads are not from the same processes.
  - ↳ Of course, an operating system can switch between multiple processes.

Data Races

- → Via MESI (CS3350), we know cache coherency is a problem.
- → If two threads both attempt to write to same memory location at the same time, one must be first.
- → Recall: transaction serialization.
- → But, what order do the writes occur?
- → **Non-Determinism**

```
void setAddress(int* addr, int val) {
    *addr =val;
}

int main(int argc, char** argv) {
    int* p =new int[1];
    *p = 0;

    std::thread t1(setAddress, p, 1);
    std::thread t2(setAddress, p, 2);
    t1.join();
    t2.join();
    std::cerr << "p: " << *p << "\n";
    return 0;
}
```

Data Races In Action

```
alex@niflheim:~/foo$ g++ NonDetEx.cpp -lpthread
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 1
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
```

More Data Races

- What happens if multiple threads reading and writing?
  - Need synchronization between threads.
- What are the possible values of p that could be printed?
- 1 *or* 2 $\Rightarrow$ **non-determinism**.
- Context switch could occur between reading from address and writing back updated result.

```
void incrAddr(int* address) {
    int val =*address;
    val++;
    *address =val;
}

int main(int argc, char** argv) {
    int* p =new int[1];
    *p = 0;

    std::thread t1(incrAddr, p);
    std::thread t2(incrAddr, p);
    t1.join();
    t2.join();
    std::cerr << "p: " << *p << "\n";
    return 0;
}
```

Scheduling Multiple Threads

→ The interleaving of instructions for multiple threads being executed simultaneously is non-deterministic.

Case 1:

Time Step	Thread 1	Thread 2
1	`val = *addr`	
2	`val++`	
3	`*addr = val`	
4		`val = *addr`
5		`val++`
6		`*addr = val`

The final value is 2.

Case 2:

Time Step	Thread 1	Thread 2
1	`val = *addr`	
2	`val++`	
3		`val = *addr`
4	`*addr = val`	
5		`val++`
6		`*addr = val`

The final value is 1.
Address was read from twice before it was ever updated. Both threads read 0 from `*addr` and both write 1 to `*addr`.

Fixing Data Races

- → To fix data races we need thread **synchronization**.
  - ↳ Only one thread can execute some **critical section** at one time.
  - ↳ This is called **mutual exclusion**.
- → We generally use **locks** whose "ownership" allows a thread to access a critical section.
- → If a thread tries to "lock" (a.k.a take/own/capture) a lock that is already locked by another thread, it waits for the lock to be unlocked and then tries to lock it again.

```
std::mutex mutex;
void incrAddr(int* address) {
    mutex.lock(); //wait here until successful lock
    int val =*address;
    val++;
    *address =val;
    mutex.unlock();
}
```

Implementing a Lock

Semaphore: a counter used to control access to a critical section.

→ Locks usually use a specialized **binary semaphore**: its value is 0 or 1.

The simplest lock is a **spinlock**.

→ It waits by "spinning " until the lock can be acquired.

→ A bad, non-working, but simple example:

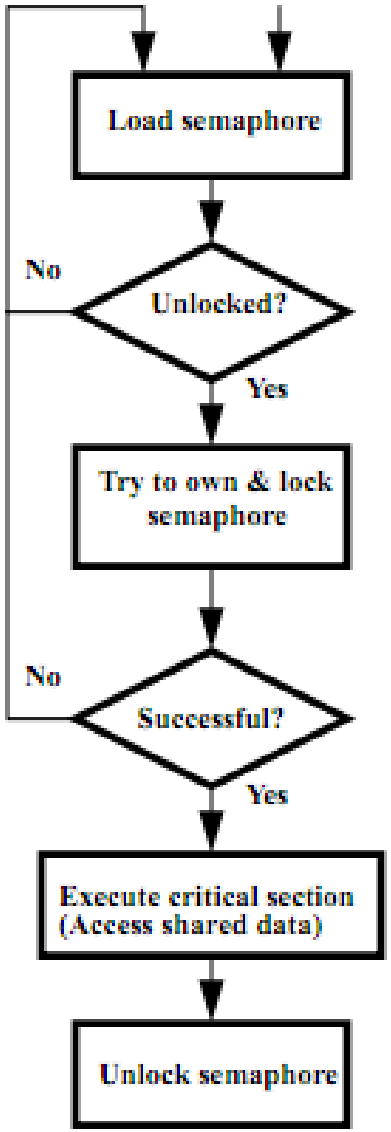
```
void spinlock::lock() {
    int spins =0;
    while(this.semaphore ==1) {
        ++spins;
    }
    this.semaphore =1;
}
```

→ If multiple threads spinning on same spinlock, which one is first to set the semaphore to 1? (i.e. which one owns the lock?) Which one goes back to spinning?

Better Lock Implementation: "Test and Set"

- → "Test and set" is an **atomic** operation that sets a variable's value, returning its old value.
- → Atomic: an operation (many instructions) which is viewed as happening instantly across all threads. Cannot be interrupted by a context switch.
- → `this.setSemaphore()` is atomic.

```
void spinlock::lock() {
    int spins =0;
    while(this.semaphore ==1) {
        ++spins;
    }
    int oldVal =this.setSemaphore(1);
    if (oldVal ==1) {
        //another thread got there first
        this.lock();
    }
}
```



Outline

Threads

- → Within a process, **threads** are independent control flows
- → Programs can **spawn** threads as needed
- → In C++, thread object creation automatically spawns thread; thread exits once assigned code segment finishes executing

```
1  std::thread t( []() -> void {
2      std::cout << "A thread started.\n"
3      int n = 0;
4      for (int i = 0; i < 1000; ++i) {
5          n += i;
6      }
7      std::cout << "A thread finished.\n"
8  });
9
10 doSomethingElse();
11
12 t.join();
```

Threading Primitives

C++11 introduced the *Thread Support Library*

- `std::thread`
  - ↳ C++ class encapsulating a thread (often a `pthread`) and its low-level spawn and join
- `std::mutex`
  - ↳ shared object between threads to indicate *mutual exclusion* to a **critical region**.
  - ↳ `mutex` is *locked* or *owned* by at most one thread at a time.
- `std::lock_guard`, `std::unique_lock`
  - ↳ temporary object wrapping a `mutex` whose object lifetime automatically locks and unlocks the `mutex`.
  - ↳ the constructor **blocks** and only returns once the shared `mutex` is successfully owned by the calling thread.
- `std::condition_variable`
  - ↳ blocks the current thread and temporarily releases a lock
  - ↳ receives notification from another thread to awaken the blocked thread

Parallel Overheads

Creating and managing multiple threads of execution can be expensive

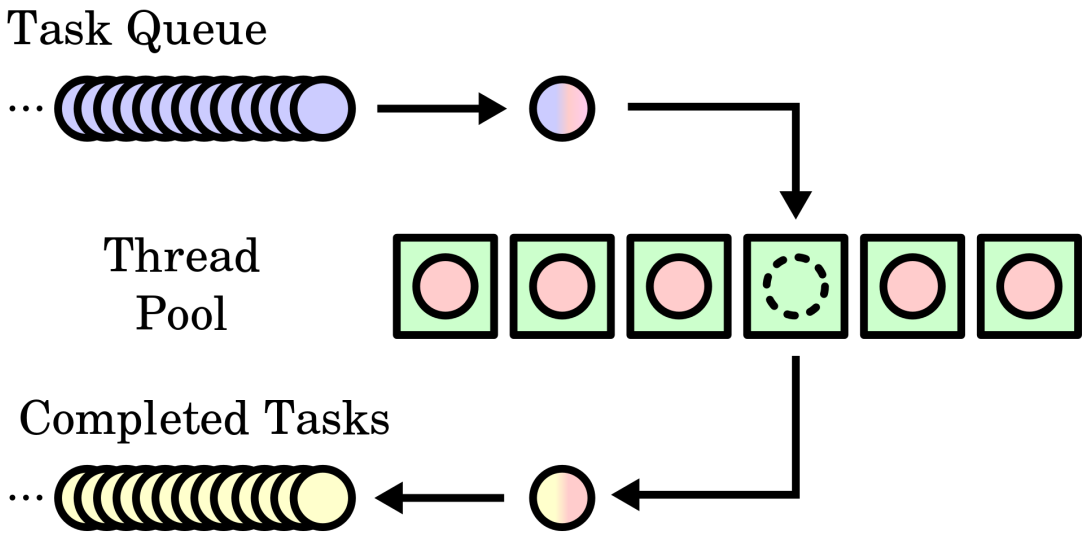
- → Every thread spawn requires non-insignificant amount of time
- → If more threads are active than the hardware supports, **over-subscription** occurs and repeated **context switching** slows down the program
- → Thread synchronization, locking `mutexs`, accessing critical regions require special care

Thread pools mitigate the first two, by supplying a fixed number of long-running threads.

Parallel programming patterns are algorithmic designs for efficient thread scheduling and minimizing locking

Thread Pools

- → Highly parallel programs benefit greatly for a thread pool
- → A fixed number of threads are spawned, only once, at the beginning of the program
- → Threads remain active for the program lifetime
- → Threads receive *tasks*, code blocks or functions, to execute as needed
- → Threads return to the pool upon completing their task



Outline

Long-Running Threads

Threads typically terminate once their assigned function/code block finishes

We require a mechanism which allows threads to:

1. Remain active until explicitly told to exit (or the entire program exits)
2. Receive new code blocks to execute on demand

std::function

Functors, **function objects**, **callable objects**

- → First-class objects which are callable using normal function syntax
- → Are often constructed by passing function names, function pointers
- → `std::bind` binds arguments to a function or function object, returning a function object which requires fewer arguments

```
1  void printInteger(int a) {
2      std::cout << a << std::endl;
3  }
4
5  //Function object from function name
6  std::function<void(int)> f_printInt(printInteger);
7  f_printInt(12);
8
9  //Function object binding arguments to function name
10 std::function<void()> f_print42( std::bind(printInteger,42) );
11 f_print42();
```

Lambda Expressions

Creates an anonymous function using a **closure** and returns a function object

→ Can capture variables in the enclosing scope

→ Can define the body of function object at point of creation

```
1  //Lambda expression with two parameters
2  std::function<int(int,int)> f_addInts( [](int a, int b) -> int {
3      return a + b;
4  });
5  f_addInts(4, 6);
6
7  int x = 12, y = 27;
8  //Lambda expression capturing variables in scope by reference
9  std::function<void()> f_printXY( [&]() -> void {
10     std::cout << "x: " << x << ", y: " << y << std::endl;
11 });
12 f_printXY();
```

Function Executor Thread

`FunctionExecutorThread`

- → A class encapsulating a long-running thread that receives function objects as tasks to execute asynchronously
- → Spawns an internal `std::thread` on object creation, joining thread on destruction
- → `sendRequest(std::function<void()>)`: execute a task, store task in internal queue if thread currently busy
- → `waitForThread()`: useful helper function which blocks until all tasks are complete
- → Results available through passed objects or pointers

Function Executor Thread: Usage

```
1  int A[N];
2  int* ret = new int();
3  FunctionExecutorThread t;
4
5  t.sendRequest( [=]() void -> {
6      int s = 0;
7      for (int i = 0; i < N; ++i) {
8          s += A[i];
9      }
10     *ret = s;
11 });
12
13 doSomethingElse();
14
15 //make sure result is available before continuing
16 t.waitForThread();
17
18 std::cout << "sum: " << *ret << std::endl;
```

Function Executor Thread: Implementation

```
1  class FunctionExecutorThread {
2
3      AsyncObjectStream<std::function<void()>> requestQueue;
4      std::thread m_worker;
5
6      std::mutex m_mutex;
7      std::condition_variable m_cv;
8
9      FunctionExecutorThread() {
10         //member functions require pointer to member
11         m_worker = std::thread(
12             &FunctionExecutorThread::eventLoop, this);
13     }
14
15     //NOTE: copy constructor and copy operator are deleted
16
17     void eventLoop();
18
19     void sendRequest(std::function<void()>);
20
21     void waitForThread();
22 }
```

Function Executor Thread: Implementation Details

```
1  class FunctionExecutorThread {
2      void eventLoop() {
3          std::function<void()> task;
4          while(requestQueue.getNextObject(task)) {
5              task();
6              std::unique_lock<std::mutex> lk(m_mutex);
7              bool notify = requestQueue.streamEmpty();
8              lk.unlock();
9              if (notify) m_cv.notify_all();
10         }
11     }
12
13     void sendRequest(std::function<void()> f) {
14         std::lock_guard<std::mutex> lk(m_mutex);
15         requestQueue.addResult(f);
16     }
17
18     void waitForThread() {
19         std::unique_lock<std::mutex> lk(m_mutex);
20         while (!requestQueue.streamEmpty()) {
21             m_cv.wait(lk);
22         }
23     }
24 }
```

Object Streams

The `AyncObjectStream` class provides:

1. a queue for tasks, or any object, and
2. a blocking mechanism to keep the `FunctionExecutorThread` alive and idle when waiting for tasks

→ Actually a class template for any kind of object being passed between two threads

→ Implements a queue satisfying the **producer-consumer problem** (explained later)

→ A **std::queue** combined with a mutex and condition variable

AsyncObjectStream Interface

```
1  template <class Object>
2  class AsyncObjectStream {
3
4      std::queue<Object> retObjs;
5      std::mutex m_mutex;
6      std::condition_variable m_cv;
7      bool finished; //is the stream still open?
8
9      //Producer: add an object to the queue
10     void addResult(Object&& res);
11
12     //Producer: close the producer end of stream,
13     //          no more objects to produce
14     void resultsFinished();
15
16     //Consumer: pop an object from the queue, return true
17     //          iff stream is open and objects available
18     bool getNextObject(Object& res);
19
20     //Consumer: determine if queue is currently empty
21     void streamEmpty();
22 };
```

AsyncObjectStream: getNextObject

```
1  bool getNextObject(Object& res) {
2      std::unique_lock<std::mutex> lk(m_mutex);
3      if (finished && retObjs.empty()) {
4          lk.unlock();
5          return false;
6      }
7
8      //Wait in a loop in case of spurious wake ups
9      while (!finished && retObjs.empty() {
10         m_cv.wait(lk);
11     }
12
13     if (finished && retObjs.empty()) {
14         lk.unlock();
15         return false;
16     } else {
17         res = retObjs.front();
18         retObjs.pop();
19         lk.unlock();
20         return true;
21     }
22 }
```

Recap

`std::function`

→ First-class objects encapsulating functions to be used as tasks

`FunctionExecutorThread`

→ A long-running thread that receives and asynchronously executes function objects

`AsyncObjectStream`

→ A blocking queue used inside `FunctionExecutorThread`

→ Passes function objects to worker thread; keeps thread alive while waiting for new tasks

Outline

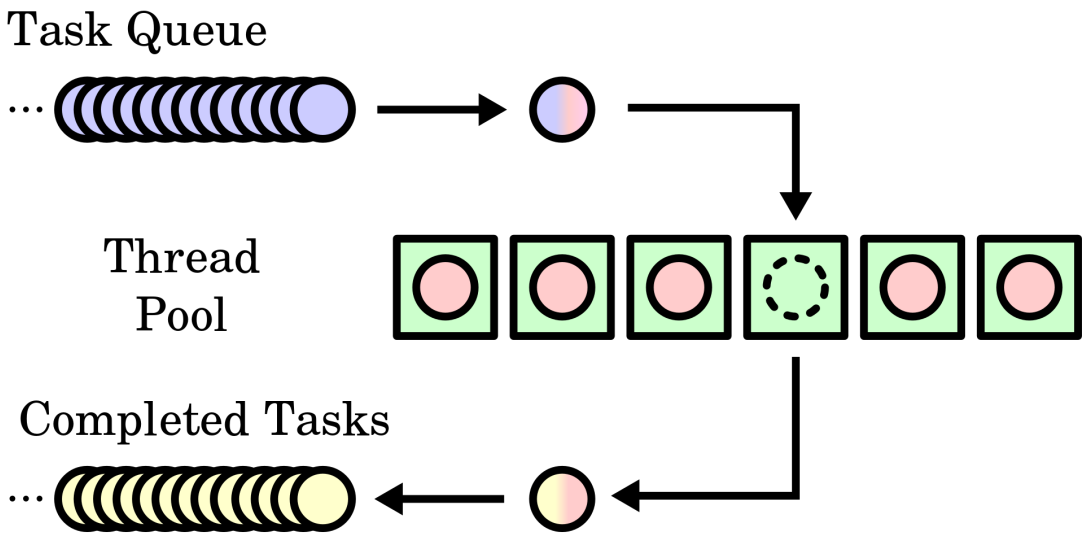
Thread Pools

A **thread pool** manages a collection of long-running threads and a queue of tasks

→ spawn all threads once at the beginning of program

→ idle threads receive and execute tasks as required

→ if all threads busy, tasks are added to queue



ExecutorThreadPool

- → A thread pool built using **FunctionExecutorThreads**
- → An internal queue of tasks and queue of threads
- → When threads are busy, they are temporarily removed from the pool
- → When all threads busy, tasks are added to task queue

```
1  class ExecutorThreadPool {
2
3  private:
4      std::deque<FunctionExecutorThread*> threadPool;
5      std::deque<std::function<void()>> taskPool;
6      std::mutex m_mutex;
7      std::condition_variable m_cv; //used in waitForThreads
8
9      void tryPullTask();
10     void putBackThread(FunctionExecutorThread* t);
11
12 public:
13     void addTask(std::function<void()> f);
14     void waitForThreads();
15 }
```

ExecutorThreadPool: addTask

```
1  void addTask(std::function<void()> f) {
2      std::unique_lock<std::mutex> lk(m_mutex);
3      taskPool.push_back(f);
4      lk.unlock();
5      tryPullTask();
6  }
7
8  void tryPullTask() {
9      std::unique_lock<std::mutex> lk(m_mutex);
10
11     if (!taskPool.empty() && !threadPool.empty()) {
12         FunctionExecutorThread* worker = threadPool.front();
13         threadPool.pop_front();
14
15         std::function<void()> f = taskPool.front();
16         taskPool.pop_front();
17         worker->sendRequest(f);
18     }
19
20     lk.unlock();
21 }
```

FunctionExecutorThread Callback (1/2)

How does a worker thread notify the thread pool that it has become idle?

→ A callback function inserts the thread itself back into the pool

```
1  ExecutorThreadPool(int nthreads) {
2      //...
3      FunctionExecutorThread* t = new FunctionExecutorThread();
4      t->setCallback(std::bind(
5          &ExecutorThreadPool::putBackThread, this));
6  }
7
8  void putBackThread(FunctionExecutorThread* t) {
9      std::unique_lock<std::mutex> lk(m_mutex);
10     if (!taskPool.empty()) {
11         std::function<void()> f = taskPool.front();
12         taskPool.pop_front();
13         worker->sendRequest(f);
14     } else {
15         threadPool.push_back(t);
16     }
17     lk.unlock();
18     m_cv.notify_all(); //notify waitForThreads()
19 }
```

FunctionExecutorThread Callback (2/2)

How does a worker thread notify the thread pool that it has become idle?

→ A callback function inserts the thread itself back into the pool

```
1  class FunctionExecutorThread {
2
3      std::function<void(FunctionExecutorThread*)> cb;
4
5      void eventLoop() {
6          std::function<void()> task;
7          while(requestQueue.getNextObject(task)) {
8              task();
9
10             if (cb) cb((FunctionExecutorThread*) this);
11
12             std::unique_lock<std::mutex> lk(m_mutex);
13             bool notify = requestQueue.streamEmpty();
14             lk.unlock();
15             if (notify) m_cv.notify_all();
16         }
17     }
18 }
```

ExecutorThreadPool: Flexible Usage (1/2)

- → In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using `addTask`
- → Abstract away actual threads through **thread IDs**
- → Once thread obtained, repeat Steps 2–3 as often as necessary

```
1  class ExecutorThreadPool {
2    //Storage for threads removed from pool by obtainThread
3    std::vector<FunctionExecutorThread*> occupiedThreads;
4
5    //Step 1: obtain a thread's ID, removing it from the pool
6    void obtainThread(threadID& id);
7
8    //Step 2: execute a task on a particular thread
9    void executeTask(threadID id, std::function<void()>& f);
10
11   //Step 3 (optional): wait for thread to become idle
12   void waitForThread(threadID id);
13
14   //Step 4: return thread to pool (waits before returning)
15   void returnThread(threadID id);
16 }
```

ExecutorThreadPool: Flexible Usage (2/2)

→ In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using `addTask`

→ Can obtain one thread at a time (previous slide), or multiple threads at a time

```
1  class ExecutorThreadPool {
2
3      //Step 1: obtain threadIDs, removing them from the pool
4      void obtainThreads(std::vector<threadID>& ids);
5
6      //Step 2: execute a task on a particular thread
7      void executeTask(threadID id, std::function<void()>& f);
8
9      //Step 3 (optional): wait for threads to become idle
10     void waitForThreads(std::vector<threadID>& ids);
11
12     //Step 4: return threads to pool (waits before returning)
13     void returnThreads(std::vector<threadID>& ids);
14 }
```

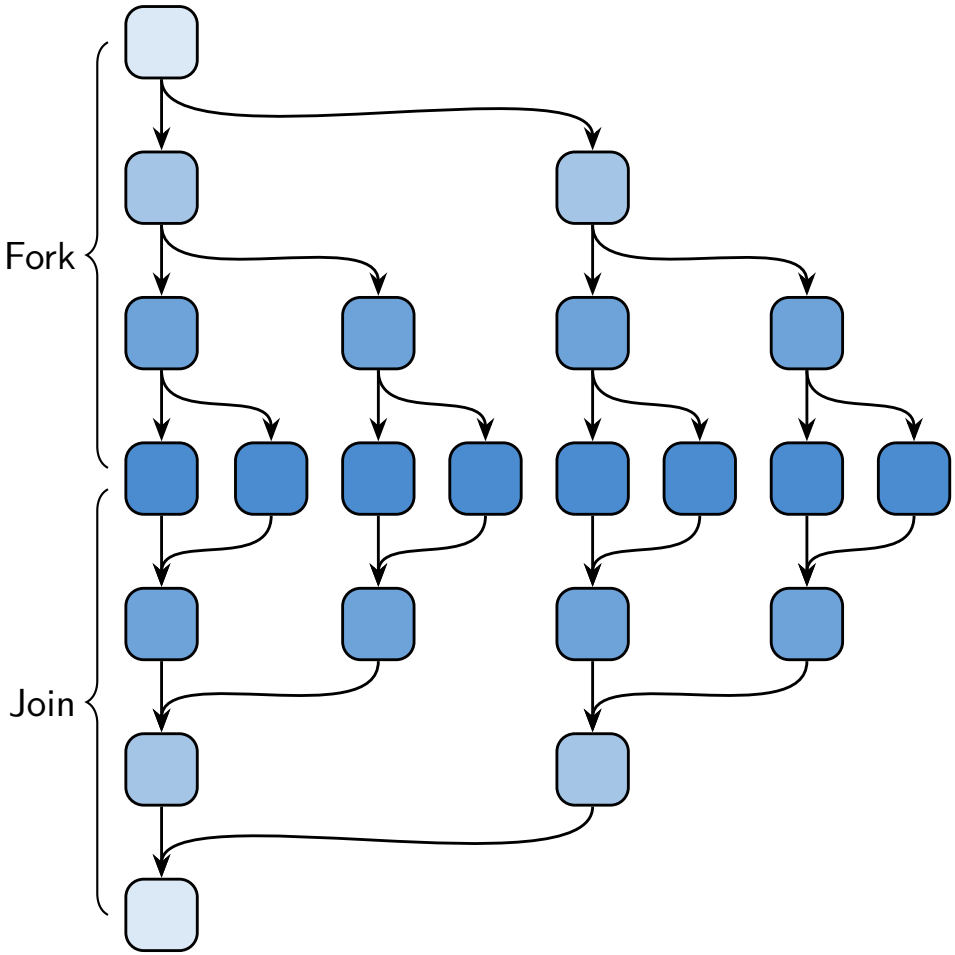
ExecutorThreadPool Singleton

- → To avoid *over-subscription*, a program should not use multiple thread pools
- → All areas of code should share the same thread pool
- → Use a classic singleton pattern

```
1  class ExecutorThreadPool {
2
3  private:
4      //pool size defaults to 1 less than hardware allows,
5      //the main thread counts as 1
6      ExecutorThreadPool(int nthreads =
7          std::thread::hardware_concurrency() - 1;);
8
9  public:
10     static ExecutorThreadPool& getThreadPool() {
11         static ExecutorTreadPool pool;
12         return pool;
13     }
14 }
```

Outline

Fork-Join



→ **Fork**: divide problem and execute separate calls in parallel

→ **Join**: merge parallel execution back into serial

→ Recursively applying fork-join can easily parallelize a divide-and-conquer algorithm

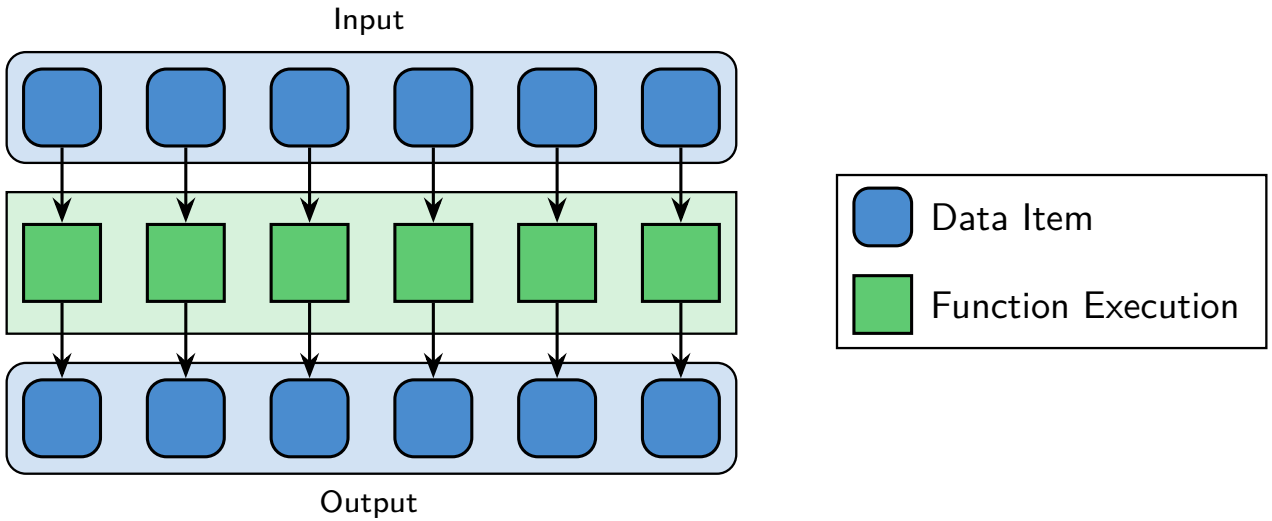
Fork-Join with ExecutorThreadPool

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      mergeSort(A, i, k);
5      mergeSort(A, k, j);
6      merge(A, i, k, j);
7  }
```

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      threadID id;
5      ExecutorThreadPool& pool = getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12     merge(A, i, k, j);
13 }
```

Map

- → Simultaneously execute a function on each data item in a collection
- → If more data items than threads, apply the pattern block-wise: partition the collection and apply one thread to each partition
- → Often simplified as just a `parallel_for` loop
- → Where multiple map steps are performed in a row, they must operate in lockstep



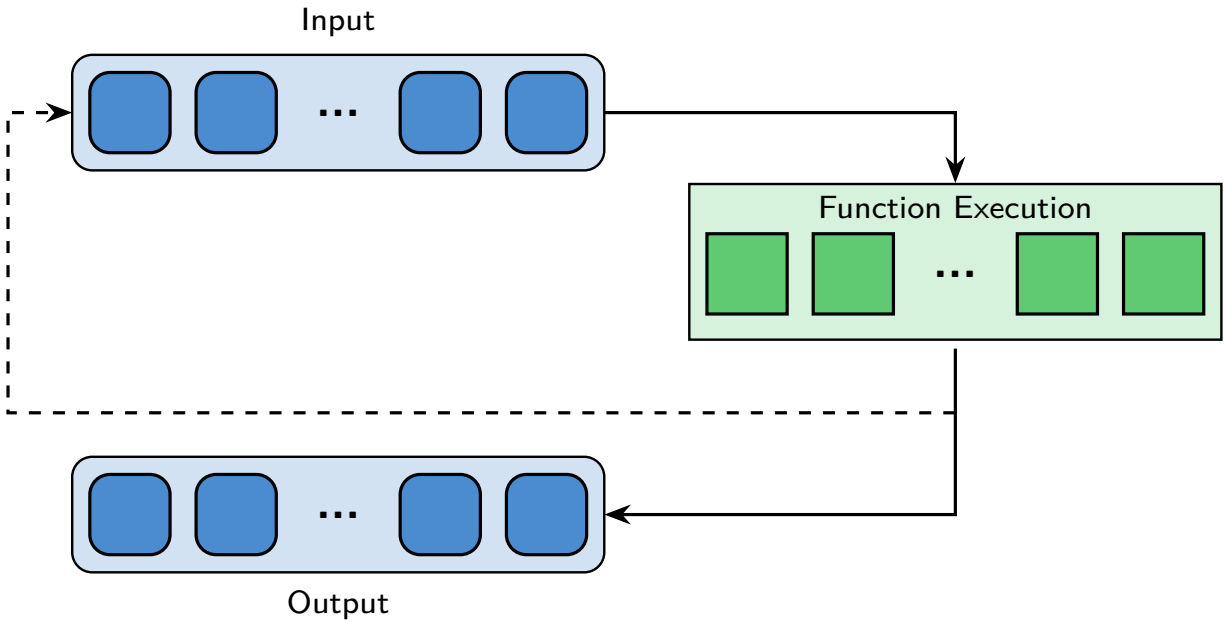
Map with ExecutorThreadPool

```
1  //Apply f to each item of A, returning results in B
2  void MapExample(int* B, int* A, int n,
3                  std::function<void(int*,int*)> f) {
4
5      for (int i = 0; i < n; ++i) {
6          f(&B[i], &A[i]);
7      }
8  }
```

```
1  void MapExample(int* B, int* A, int n,
2                  std::function<void(int*,int*)> f) {
3
4      ExecutorThreadPool& pool = getThreadPool();
5      std::vector<threadID> ids;
6      pool.obtainThreads(n-1, ids); //assume n-1 threads avail.
7
8      for (int i = 0; i < n-1; ++i) {
9          pool.executeTask(ids[i], std::bind(f, &B[i], &A[i]));
10     }
11     f(&B[n-1], &A[n-1]); //use main thread for one call
12
13     pool.returnThreads(ids); //also waits for threads
14 }
```

Workpile

- → Workpile generalizes map pattern to a *queue* of tasks
- → Tasks in-flight can add new tasks to input queue
- → Threads take tasks from queue until it is empty
- → Very similar in structure to a thread pool
- → Can be seen as a `parallel_while` loop

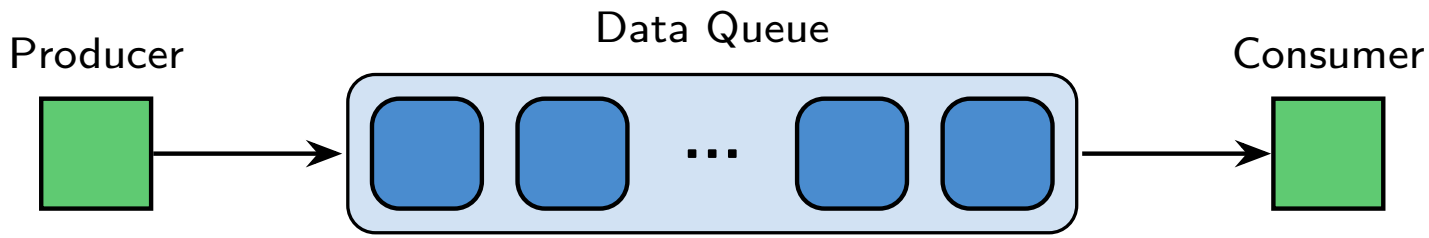


Workpile with ExecutorThreadPool

```
1  void processInt(std::queue<int> B, int a) {
2      a -= 10;
3      if (a > 0) {
4          getThreadPool().addTask(std::bind(processInt, B, a));
5      } else {
6          B.push(a);
7      }
8  }
9
10 void WorkpileExample(std::queue<int> B, std::queue<int> A) {
11     ExecutorThreadPool& pool = getThreadPool();
12     while (!A.empty()) {
13         pool.addTask( std::bind(processInt, B, A.front()) );
14         A.pop();
15     }
16     pool.waitForAllThreads();
17 }
```

Producer-Consumer

- → Two functions connected by a queue
- → The producer produces data items, pushing them to the queue
- → The consumer processes data items, pulling them from the queue
- → Producer and consumer execute simultaneously; at least one must be active at all times $\Longrightarrow$ no **deadlock**



- → In some circumstances, the producer may be considered as an **iterator** or **generator**

Generators

→ Generators are special kinds of coroutines which **yield** data items one at a time, rather than many as a collection

→ A yield pauses execution of the function, and allows computations to resume from that point at the next function call

```
1  void FibonacciGen(int n) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          yield Fn_1;
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9  }
```

→ Where the generation of data items is itself expensive, generators may execute asynchronously, following the producer-consumer pattern

AsyncGenerator and AsyncObjectStream

We want an *object-oriented* approach to create and use generators.

`AsyncObjectStream` already solves the producer-consumer problem.

→ It provides a queue which blocks and notifies the consumer as data is produced, implemented using a condition variable

→ As a class template, can be used within `AsyncGenerator` to yield any type of object

```
1  template <class Object>
2  class AsyncObjectStream {
3      void addResult(Object&& res); //Producer
4
5      void resultsFinished(); //Producer
6
7      bool getNextObject(Object& res); //Consumer
8
9      void streamEmpty(); //Consumer
10 };
```

AsyncGenerator

AsyncGenerator is itself a class template, templated by Object, the type of object to generate.

→ The **AsyncGenerator** acts as interface between producer and consumer

→ The consumer constructs the **AsyncGenerator**, passing the constructor the producer's function and arguments

→ The producer's signature should be:

```
1 void producerFunction(..., AsyncGenerator<Object>&);
```

→ The AsyncGenerator being constructed inserts itself into the producer's list of arguments so that it has reference to the generator object

AsyncGenerator Example

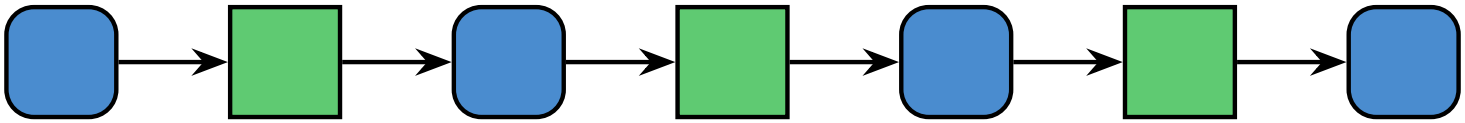
```
1  void FibonacciGen(int n, AsyncGenerator<int>& gen) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          gen.generateObject(Fn_1); //yield Fn_1 and continue
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9      gen.setComplete();
10 }
11
12 void Fib() {
13     int n;
14     std::cin >> n;
15     AsyncGenerator<int> gen(FibonacciGen, n);
16
17     int fib;
18     //get one integer at a time until generator is finished
19     while (gen.getNextObject(fib)) {
20         std::cerr << fib << std::endl;
21     }
22 }
```

AsyncGenerator Implementation

```
1  template <class Object>
2  class AsyncGenerator {
3      AsyncObjectStream<Object> stream;
4      FunctionExecutorThread t;
5
6      //Create a generator from a function and its arugments
7      template <class Function, class... Args>
8      AsyncGenerator(Funcion& f, Args&...args) {
9          std::function<void()> boundF =
10                     std::bind(f, args..., std::ref(*this));
11         t.sendRequest(boundF);
12     }
13
14     //Create a dummy generator which yields items in sequence
15     //from a pre-computed collection
16     AsyncGenerator(std::vector<Object>& A) {
17         for (Object obj : A) { stream.addResult(obj); }
18         stream.resultsFinished();
19     }
20
21     //delegate to stream's methods
22     void generateObject(Object& obj) //addResult(obj)
23     void setComplete();              //resultsFinished()
24     bool getNextObject(Object& obj); //getNextObject(obj)
25 }
```

Pipeline

- → A sequence of stages where the output of one stage is used as the input to another
- → Two consecutive stages form a producer-consumer pair
- → Internal stages are both producer and consumer
- → Typically, a pipeline is constructed statically through code organization
- → Pipelines can be created dynamically and implicitly with `AsyncGenerators` and the callstack



Pipelines with AsyncGenerator

```
1  void intSequence(AsyncGenerator<int>& prevStage, AsyncGenerator<int>&
       nextStage) {
2      int i;
3      while(prevStage.getNextObject(i)) {
4          nextStage.generateObject(i);
5      }
6  }
7
8  std::vector<int> A = {1,2,3,4,5,6,7,8,9};
9  AsyncGenerator<int> stageOne(A);
10
11 AsyncGenerator<int> stageTwo(intSequence, stageOne);
12 AsyncGenerator<int> stageThree(intSequence, stageTwo);
13 AsyncGenerator<int> stageFour(intSequence, stageThree);
14
15 //consume from last stage of pipeline
16 int i;
17 while(stageFour.getNextObjext()) {
18     std::cout << i << std::endl;
19 }
```

Outline

ExecutorThreadPool Optional Parallelism

Since **ExecutorThreadPool** contains a finite number of threads, `obtainThread(id)` may not be able to obtain an idle thread.

- → In this case, the `threadID` returned is a special ID which indicates "not a thread"
- → Then, `executeTask(id, task)`, `returnThread(id)`, `waitForThread(id)` behave serially
- → Hence, all calls to `executeTask` are merely a *suggestion* for parallelism, depending on the current state of the thread pool

```
1  class ExecutorThreadPool {
2      void obtainThread(threadID& id);
3
4      void executeTask(threadID id, std::function<void()>& f);
5
6      void waitForThread(threadID id);
7
8      void returnThread(threadID id);
9  }
```

ExecutorThreadPool Optional Parallelism (2/2)

```
1  void obtainThread(threadID& id) {
2      std::lock_guard<std::mutex> lk(m_mutex);
3      if (threadPool.empty()) {
4          id = ExecutorThreadPool::notAThread;
5      } else {
6          FunctionExecutorThread* t = threadPool.front();
7          threadPool.pop_front();
8          occupiedThreads.push_back(t);
9          id = t->get_id(); //a std::thread::id
10     }
11 }
12
13 void executeTask(threadID id, std::function<void()>& f) {
14     if (id == ExecutorThreadPool::notAThread) {
15         f();
16         return;
17     } else {
18         std::lock_guard<std::mutex> lk(m_mutex);
19         for (FunctionExecutorThread* t : occupiedThreads) {
20             if (t->get_id() == id) { t.sendRequest(f); }
21         }
22     }
23 }
```

AsyncGenerator Optional Parallelism

Rather than **AsyncGenerator** directly using a **FunctionExecutorThread**, use the **ExecutorThreadPool**.

→ If all threads in the pool are busy, execute the function serially instead

```
1  template <class Object>
2  class AsyncGenerator {
3      AsyncObjectStream<Object> stream;
4
5      //Create a generator from a function and its arugments
6      template <class Function, class... Args>
7      AsyncGenerator(Funcion& f, Args&...args) {
8          std::function<void()> boundF =
9                      std::bind(f, args..., std::ref(*this));
10
11         ExecutorThreadPool& pool = getThreadPool();
12         if (pool.allThreadsBusy()) {
13             boundF();
14         } else {
15             pool.addTask(f);
16         }
17     }
18 }
```

Cooperative Parallelism

With several simultaneous clients of `ExecutorThreadPool`, some tasks should be given priority.

- → Some tasks are more **coarse-grained**, offer more potential speed-up
- → Some tasks may expose more parallelism and should be executed first

Often, parallelism coming from Fork-Join or Map is preferred over Producer-Consumer.

- → **Goal**: allow Fork-Join and Map to access thread pool threads over Producer-Consumer while still keeping the latter possible when there are idle threads
- → **Solution**: **priority tasks**
- → `addTask()` vs `addPriorityTask()`

ExecutorThreadPool Priority Tasks

`pool.addPriorityTask()`

→ If there are idle threads, priority task behaves as normal task

↳ pull a thread from the pool and assign the task to it

→ If there are no idle threads:

1. temporarily allow over-subscription and spawn a new **priority thread**
2. assign the priority task to the new priority thread
3. the next thread returned to the pool is **retired** to recover a state without over-subscription

→ If the number of spawned priority threads equals the original number of threads in the pool, do not spawn any more

↳ instead, add the priority task to the *head* of the task queue so that it is the next executed task