THREADS CONT'D

LECTURE 12-2

JIM FIX, REED COLLEGE CS2-S20

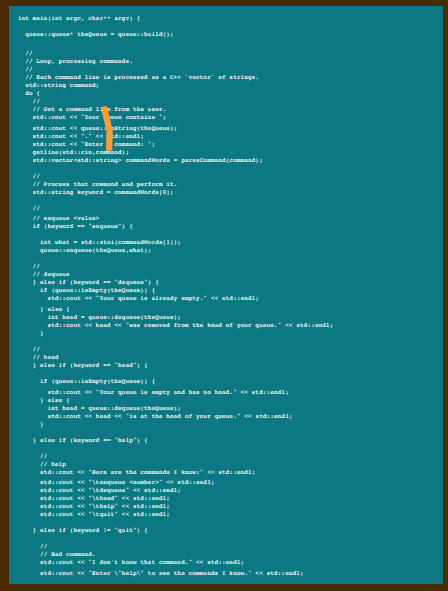
TODAY'S PLAN

- WALKCHEW.CC EXAMPLE
- TRIVIAL SPEEDUP EXAMPLE
- MORE ON THREADS
- COUNT EXAMPLE
- COUNT WITH MUTEX
- MUTEX-PROTECTED QUEUE

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter a command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter a command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

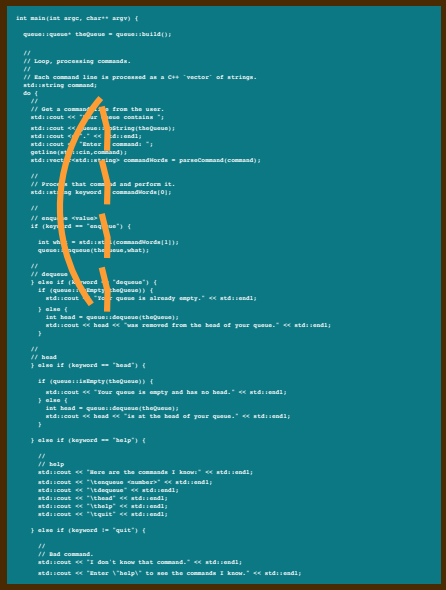
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command line from the user.
    std::cout << "Your queue contains ";
    std::cout << queue::toString(theQueue);
    std::cout << "." << std::endl;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

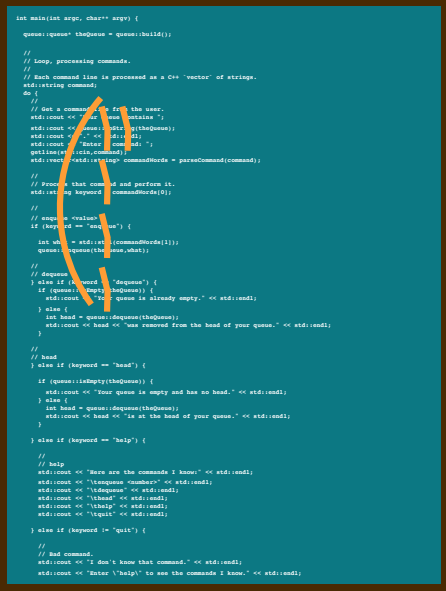


- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command from the user.
    std::cout << "\nWhat would you like to do?\n";
    std::cout << queue::toString(theQueue);
    std::cout << "> " << std::flush;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWords = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWords[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWords[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - It has its own instruction pointer into the code.
  - It has its own call stack.
  - It has its own copy of local variables.
  - *It "threads" its way through the code, executing it line-by-line.*

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

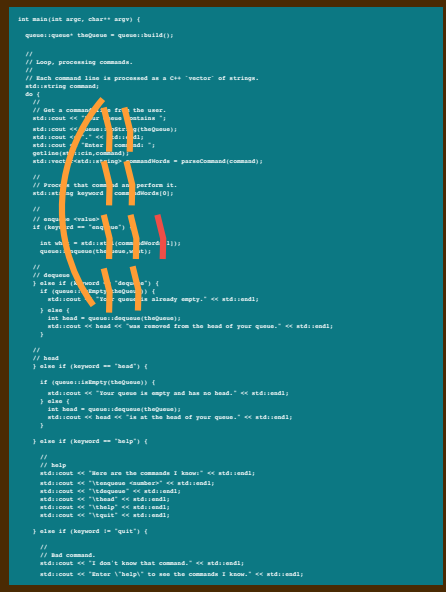
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command from the user.
    std::cout << "[" << queue::size(theQueue)
    std::cout << "] " << id;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWorde = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWorde[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWorde[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << " is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  -

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc



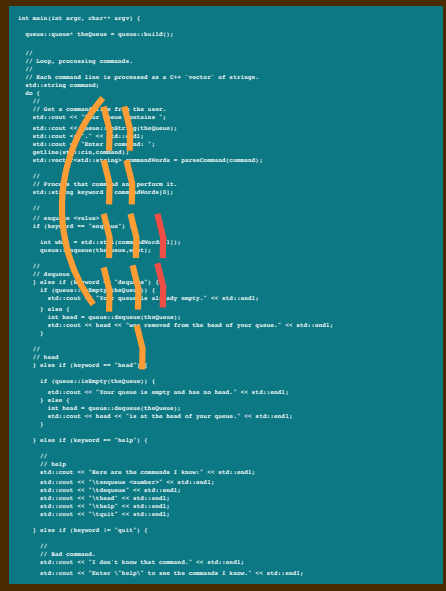
```
int main(int argc, char** argv) {

  queue::queue* theQueue = queue::build();

  //
  // Loop, processing commands.
  //
  // Each command line is processed as a C++ `vector` of strings.
  std::string command;
  do {
    //
    // Get a command from the user.
    std::cout << "\n[" << queue::size(theQueue)
    std::cout << "] " << id;
    std::cout << "Enter command: ";
    getline(std::cin,command);
    std::vector<std::string> commandWorde = parseCommand(command);

    //
    // Process that command and perform it.
    std::string keyword = commandWorde[0];

    //
    // enqueue <value>
    if (keyword == "enqueue") {

      int what = std::stoi(commandWorde[1]);
      queue::enqueue(theQueue,what);

    //
    // dequeue
    } else if (keyword == "dequeue") {
      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is already empty." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "was removed from the head of your queue." << std::endl;
      }

    //
    // head
    } else if (keyword == "head") {

      if (queue::isEmpty(theQueue)) {
        std::cout << "Your queue is empty and has no head." << std::endl;
      } else {
        int head = queue::dequeue(theQueue);
        std::cout << head << "is at the head of your queue." << std::endl;
      }

    } else if (keyword == "help") {

      //
      // help
      std::cout << "Here are the commands I know:" << std::endl;
      std::cout << "\tenqueue <number>" << std::endl;
      std::cout << "\tdequeue" << std::endl;
      std::cout << "\thead" << std::endl;
      std::cout << "\thelp" << std::endl;
      std::cout << "\tquit" << std::endl;

    } else if (keyword != "quit") {

      //
      // Bad command.
      std::cout << "I don't know that command." << std::endl;
      std::cout << "Enter \"help\" to see the commands I know." << std::endl;
```

- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

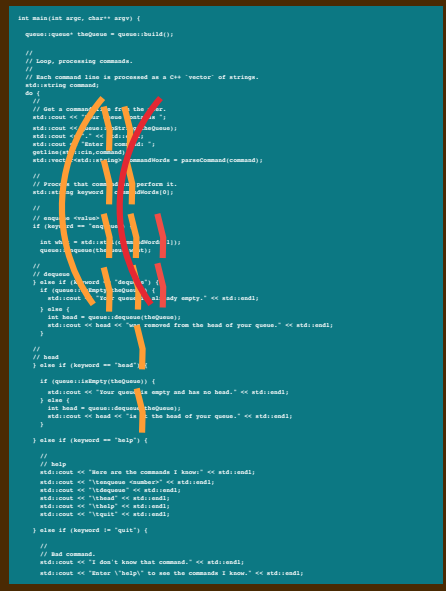


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

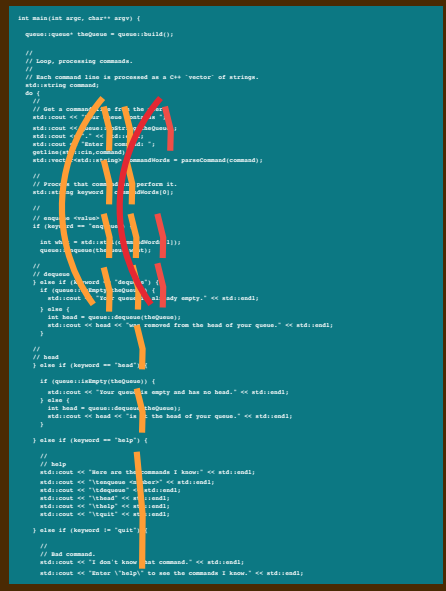


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

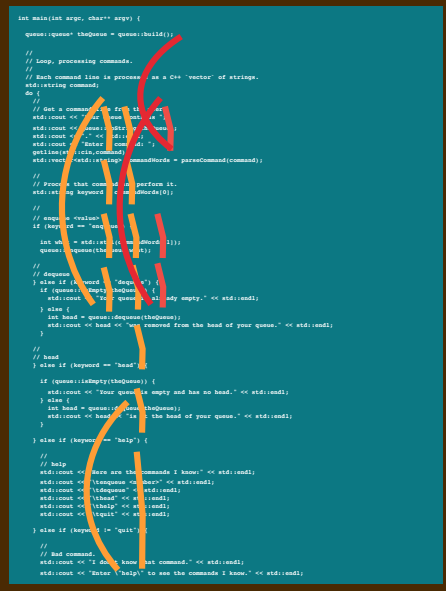


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

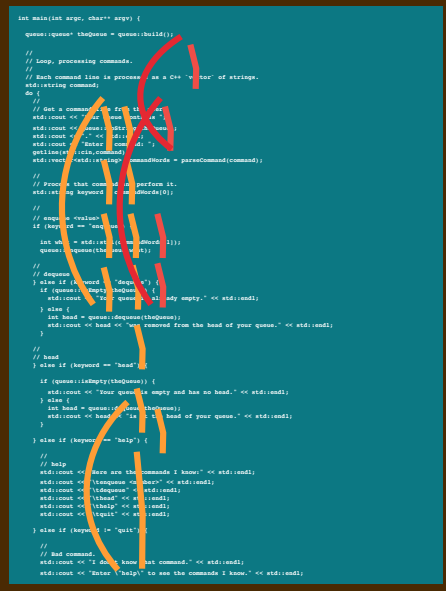


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

program.cc

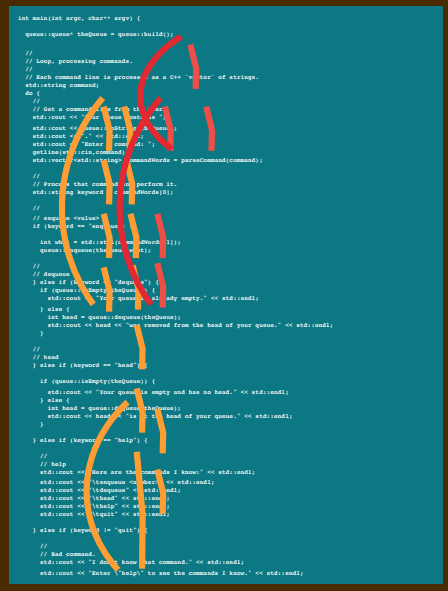


- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

THREAD LIBRARIES

- **Question:**
  - How do we write code to use several processor cores in parallel?
  - How do we structure one program's code to execute tasks concurrently?
- **Answer:**
  - Many language systems provide a ***threads*** library.

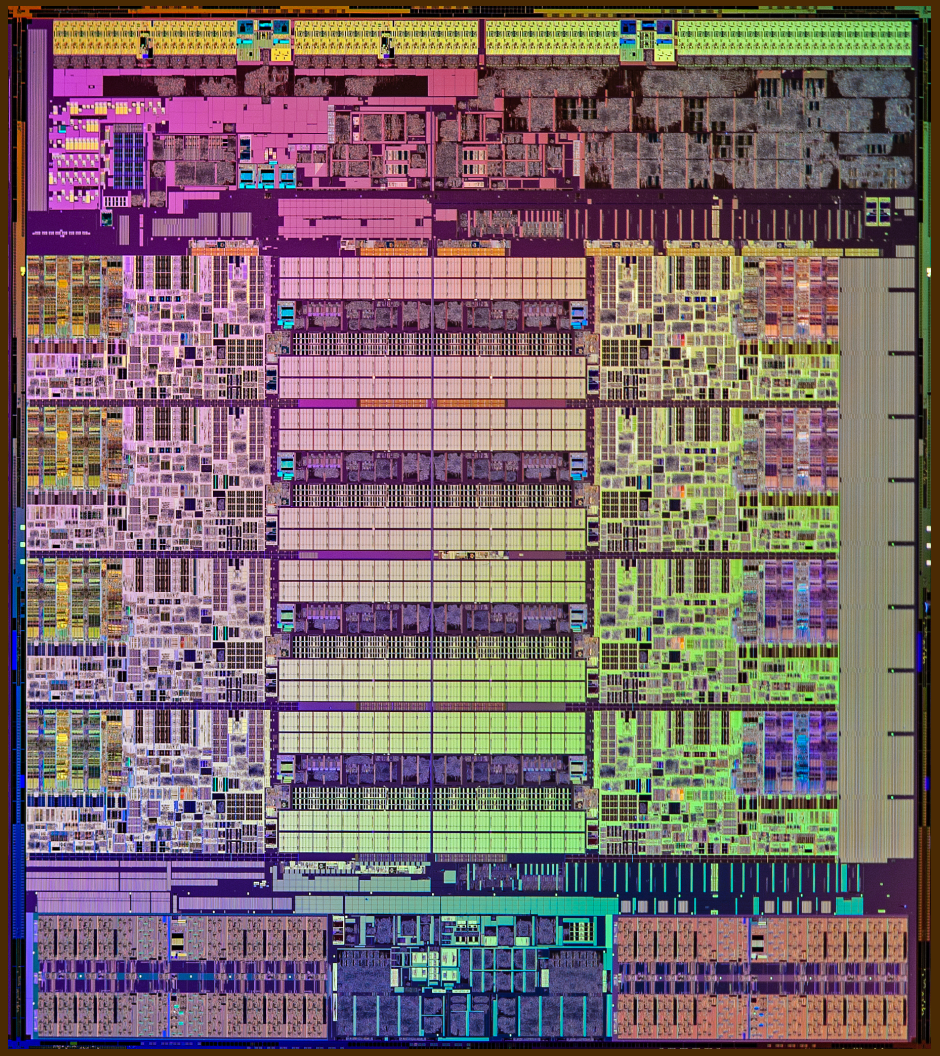
program.cc



- A ***thread*** is a code executing entity:
  - The libraries allow us to *create another thread* of execution.
  - It also *threads its way through the code line by line*, independently of the *other thread*.

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:



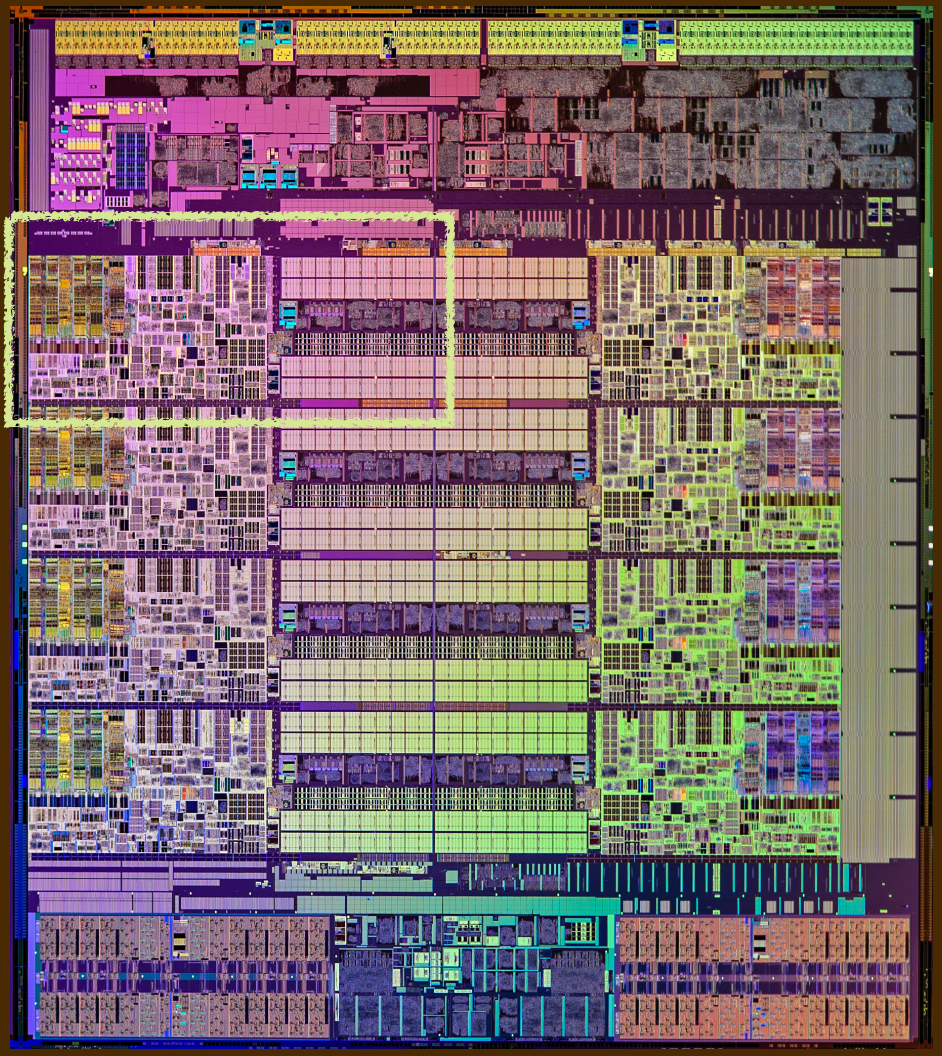
Intel Core i7-5960X Haswell-E 8-Core (2014)

- *https://www.anandtech.com/show/8426/the-intel-haswell-e-cpu-review-core-i7-5960x-i7-5930k-i7-5820k-tested*

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

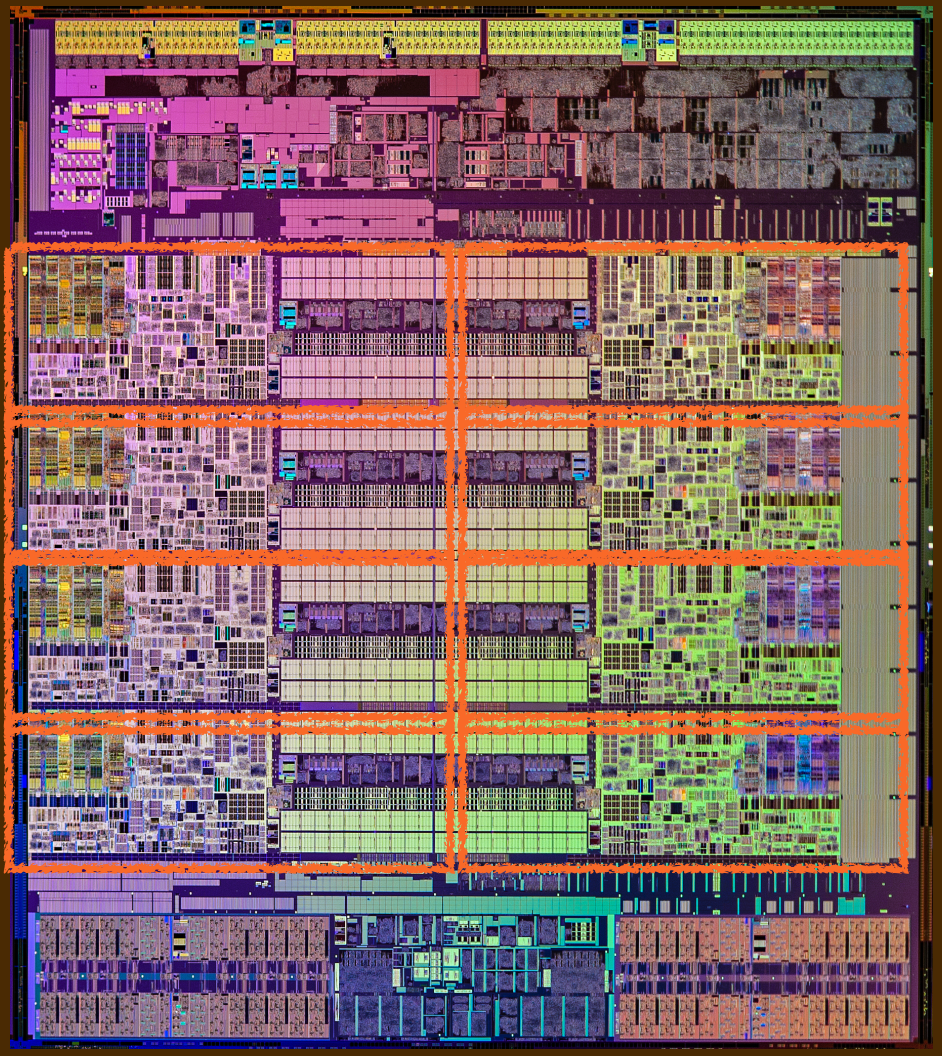
one CPU core



TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

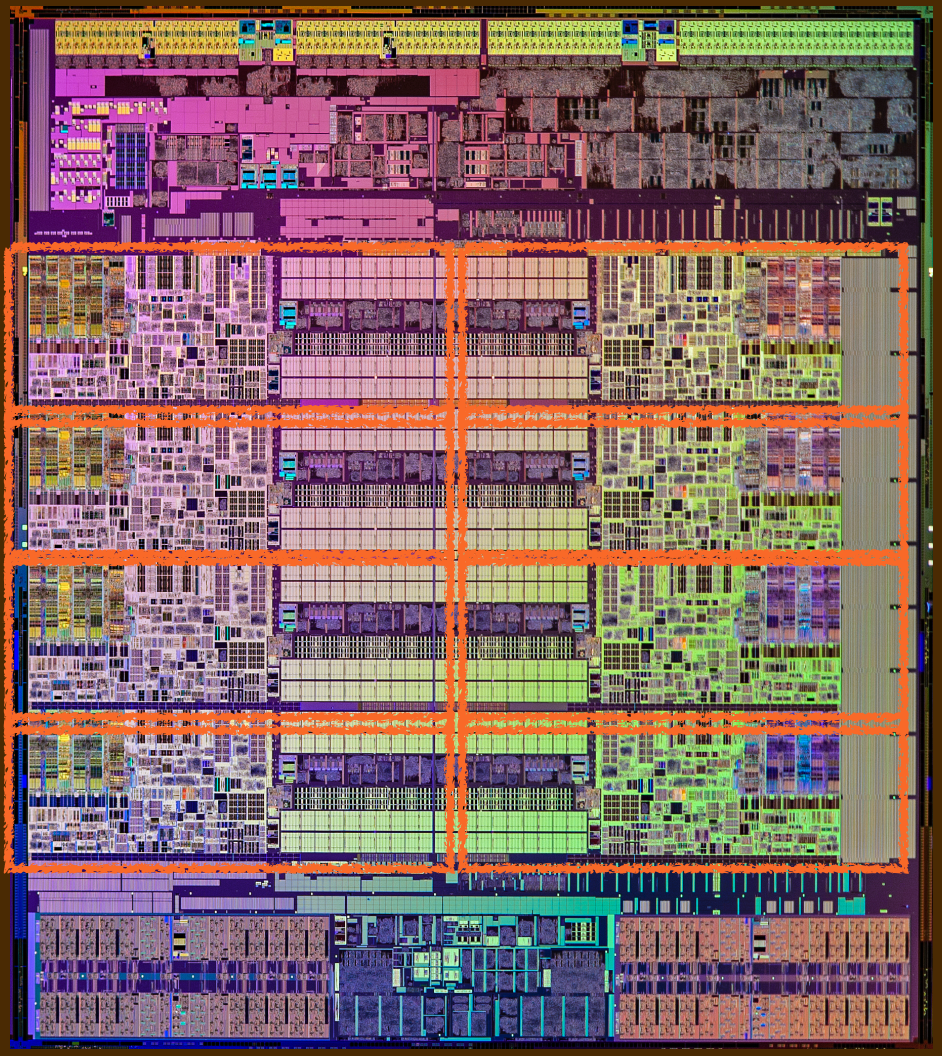
eight CPU cores



TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

eight CPU cores

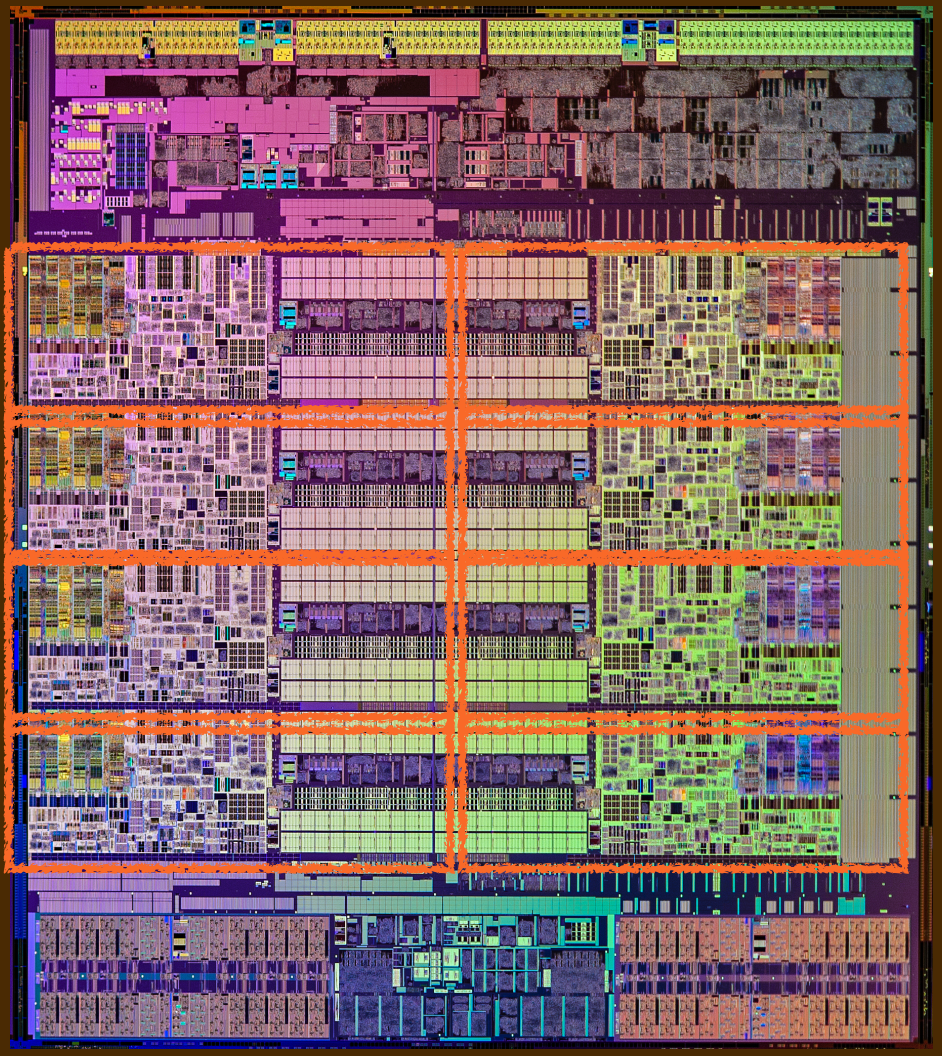


They share access to the same RAM memory, but not on this chip.

TODAY'S COMPUTERS HAVE MULTIPLE CORES

- A chip holds several processing cores on one die:

eight CPU cores



They each can be "fed" their own instruction stream.

They share access to the same RAM memory, but not on this chip.

MULTITHREADED PROGRAMS FOR PARALLELISM

- Using a threads library we can structure our code to create several threads.
  - Break a task's work into *N* pieces.
  - Create *N* threads to run each piece.

MULTITHREADED PROGRAMS FOR PARALLELISM

- Using a threads library we can structure our code to create several threads.
  - Break a task's work into *N* pieces.
  - Create *N* threads to run each piece.
- The OS might run them *in parallel* on separate cores.
- *E.g*. If we have 8 cores, then the task may get completed 8x faster!

MULTITHREADED PROGRAMS FOR PARALLELISM

- Using a threads library we can structure our code to create several threads.
  - Break a task's work into *N* pieces.
  - Create *N* threads to run each piece.
- The OS might run them *in parallel* on separate cores.
- *E.g*. If we have 8 cores, then the task may get completed 8x faster!
- Many application programs today (e.g. web browser) are multithreaded.
  - (console `top` demo)

CREATING TWO THREADS TO WALK AND CHEWGUM

- The **samples** folder contains C++ source **walkchew.cc** to perform two tasks.
- It uses the *POSIX Threads library* to create two threads:
  - One repeatedly outputs `step` #*xxxxxx*
  - One repeatedly outputs `chomp` #*yyyyyy*
    - ➡ Each with a count *xxxxxx* or *yyyyyy* of what step it's executing.
- The two threads run *concurrently*.
- The OS might run them *in parallel* on two separate cores.

CREATING TWO THREADS TO WALK AND CHEWGUM

```
#include <pthread.h> ...
typedef void *tmain(void *);
void *walk(long *v) {
  (*v) = 0;
  for (long i=100000;i>0;i--,(*v)++) std::cout<<"step #"<< *v <<"\n";
  return NULL;
}
void *chewGum(long *v) {
  long i;
  for (long i=100000;i>0;i--,(*v)++) std::cout<<"chomp #"<< *v <<"\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t id1, id2;
  long a1, a2, r1, r2;
  pthread_create(&id1,NULL,(tmain *)walk,   (void *)&a1);
  pthread_create(&id2,NULL,(tmain *)chewGum,(void *)&a2);
  pthread_join(id1,(void **)&r1);
  pthread_join(id2,(void **)&r2);
  std::cout << "Completed steps:" << a1 << " ...chomps:" << a2 << "\n";
  pthread_exit(NULL);
}
```

THEIR OUTPUT GETS INTERLEAVED

```
> g++ -o walkchew -lpthread walkchew.cc
> ./walkchew
cshtoempp  ##00

cshtoempp  ##11

sctheopm p# 2#
2s
tcehpo m#p3
#s3t
ecph o#m4p
 s#t4e
pc h#o5m
ps t#e5p
 c#h6o
mspt e#p6
#c7h
osmtpe p# 7#
8c
hsotmepp  ##89

cshtoempp  ##190...
```

CREATING TWO THREADS THAT COLLABORATIVELY COUNT

- The **samples** folder contains C++ source **count.cc**.
- It also uses the *PThreads library* to create two threads:
  - One repeatedly increments a value in memory.
  - The other repeatedly increments *the same location* in memory.
- These two collaborating threads also run *concurrently*.

TWO THREADS COLLABORATE ON COUNTING...

```
#include <pthread.h>
long increment(long *v) {
 long i;
 for (i=0; i<1000000; i++) {
    long tmp = (*v);
    tmp = tmp + 1;
    (*v) = tmp;
  }
  return i;
}
int main(int argc, char **argv) {
  pthread_t id1, id2;
  long r1, r2;
  long count = 0;
  pthread_create(&id1,NULL,(tmain *)increment,(void *)&count);
  pthread_create(&id2,NULL,(tmain *)increment,(void *)&count);
  pthread_join(id1,(void **)&r1);
  pthread_join(id2,(void **)&r2);
  std::cout << "Thread 1 incremented " << r1 << " times." << std::endl;
  std::cout << "Thread 2 incremented " << r2 << " times." << std::endl;
  std::cout << "The final count is " << count << "." << std::endl;
  pthread_exit(NULL);
}
```

WHAT HAPPENS?

```
> g++ -o count -lpthread count.cc
> ./count
```

????

THE FINAL COUNT FAILS TO REFLECT THE WORK DONE

```
> g++ -o count -lpthread count.cc
> ./count
Thread 1 incremented 1000000 times.
Thread 2 incremented 1000000 times.
The final count is 1283666.
```

- Why isn't the final count 2000000???

LET'S LOOK AT THE CRITICAL SECTION

```
1. long increment(long *v) {
2.  long i;
3.  for (i=0; i<1000000; i++) {
4.      long tmp = (*v);
5.      tmp = tmp + 1;
6.      (*v) = tmp;
7.   }
8.   return i;
9. }
```

- Both threads are repeatedly executing lines 4, 5, and 6.
  - **Line 4:** load the value in memory referenced by the pointer **`v`** into **`tmp`**.
  - **Line 5:** add 1 to **`tmp`**.
  - **Line 6:** store that updated **`tmp`** value back to the memory referenced by **`v`**.

INSTRUCTION STREAMS

You can think of thread 1 as executing this stream of instructions.

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
...
```

INSTRUCTION STREAMS

You can think of thread 1 as executing this stream of instructions.

But you should also imagine that thread 2 is executing a similar stream.

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
...
```

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
...
```

INSTRUCTION STREAMS

You can think of thread 1 as executing this stream of instructions.

But you should also imagine that thread 2 is executing a similar stream.

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
...
```

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
...
```

To reason about this, imagine their execution is interleaved.

- But you can't predict the interleaving. It depends...

INSTRUCTION STREAMS

Like this maybe:

```
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
```

INSTRUCTION STREAMS

But it probably won't be so nice/regular a pattern. Like this:

```
long tmp = (*v);
tmp = tmp + 1;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
(*v) = tmp;
long tmp = (*v);
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
tmp = tmp + 1;
(*v) = tmp;
```

INSTRUCTION STREAMS

Or this:

```
long tmp = (*v);
tmp = tmp + 1;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
tmp = tmp + 1;
(*v) = tmp;
```

INSTRUCTION STREAMS

Suppose the shared count is 10 before this interleaving executes. Then:

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
long tmp = (*v); // fetch 12
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
long tmp = (*v); // fetch 13
tmp = tmp + 1;   // set to 14
(*v) = tmp;      // store 14
long tmp = (*v); // fetch 14
tmp = tmp + 1;   // set to 15
(*v) = tmp;      // store 15
long tmp = (*v); // fetch 15
tmp = tmp + 1;   // set to 16
(*v) = tmp;      // store 16
```

Six increments,

so 10 --> 16.

INSTRUCTION STREAMS

But, working from 10, this interleaving goes awry

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
(*v) = tmp;
long tmp = (*v);
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
tmp = tmp + 1;
(*v) = tmp;
```

INSTRUCTION STREAMS

But, working from 10, this interleaving goes awry

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
(*v) = tmp;      // store 12
long tmp = (*v); // fetch 12
long tmp = (*v); // fetch 12
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
```

INSTRUCTION STREAMS

But, working from 10, this interleaving goes awry

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
(*v) = tmp;      // store 12
long tmp = (*v); // fetch 12
long tmp = (*v); // fetch 12
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
```

The work of three increments are lost!

INSTRUCTION STREAMS

Same with this...

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
(*v) = tmp;
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
long tmp = (*v);
long tmp = (*v);
tmp = tmp + 1;
(*v) = tmp;
tmp = tmp + 1;
(*v) = tmp;
```

INSTRUCTION STREAMS

Same with this...

```
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
long tmp = (*v); // fetch 10
tmp = tmp + 1;   // set to 11
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
(*v) = tmp;      // store 11
long tmp = (*v); // fetch 11
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 12
long tmp = (*v); // fetch 12
long tmp = (*v); // fetch 12
tmp = tmp + 1;   // set to 12
(*v) = tmp;      // store 13
tmp = tmp + 1;   // set to 13
(*v) = tmp;      // store 13
```

Again, three are lost!

LET'S LOOK AT THE CRITICAL SECTION

```
1. long increment(long *v) {
2.   long i;
3.   for (i=0; i<1000000; i++) {
4.     long tmp = (*v);
5.     tmp = tmp + 1;
6.     (*v) = tmp;
7.   }
8.   return i;
9. }
```

- The threads are fighting over access to a shared location in memory.
- It is critical that only one thread executes lines 4,5, and 6 at a time.
- They need *exclusive access* to that *critical section* of code.
- Thread systems provide a ***lock*** or a ***mutex variable***, to *synchronize* access.

LET'S LOOK AT THE CRITICAL SECTION

```
1. long increment(shared_counter* sc) {
2.     ...
3.     pthread_mutex_lock(&sc->mutex);
4.     long tmp = (*sc->ptr);
5.     tmp = tmp + 1;
6.     (*sc->ptr) = tmp;
7.     pthread_mutex_unlock(&sc->mutex);
8.     ...
9. }
```

- The threads are fighting over access to a shared location in memory.
- It is critical that only one thread executes lines 4,5, and 6 at a time.
- They need *exclusive access* to that *critical section* of code.
- Thread systems provide a ***lock*** or a ***mutex variable***, to *synchronize* access.

THE FINAL COUNT NOW REFLECTS THE WORK DONE

```
> g++ -o count_mutex -lpthread count_mutex.cc
> ./count_mutex
Thread 1 incremented 1000000 times.
Thread 2 incremented 1000000 times.
The final count is 2000000.
```

- But the code runs *a lot slower*.

SUMMARY AND CHALLENGES

- Today's processors have several cores.
- Multithreading allows you to run program components concurrently.
  - ➡ Operating system might schedule on several cores to enable parallelism and thus better task throughput.
  - Care must be taken to synchronize access to shared resources.
  - Synchronization takes time, task coordination slows things down.
  - ➡Getting speedups with parallelism is tricky.
- Scheduler isn't predictable. Resource sharing is hard to manage,
  - Debugging single threaded code is tricky.
  - Debugging multithreaded code is even trickier.

USING STL <THREAD> FOR WALKCHEW

```
#include <thread>
#include <iostream>
#include <functional>

int main(void) {

  int steps  = 10000;
  auto walk = [steps](void)->void {
    for (int i=0;i<steps;i++) std::cout << "step #" << i << "\n";
  };

  int chomps = 10000;
  auto chewGum = [chomps](void)->void {
    for (int i=0;i<chomps;i++) std::cout << "chomp #" << i << "\n";
  };

  // Start both threads and wait for them to each complete.
  std::thread t1 {walk};
  std::thread t2 {chewGum};
  t1.join();
  t2.join();
  std::cout << "Number of steps: " << steps;
  std::cout << " ...chomps: " << chomps << std::endl;
}
```

USING STL <THREAD> AND <MUTEX> FOR COUNT

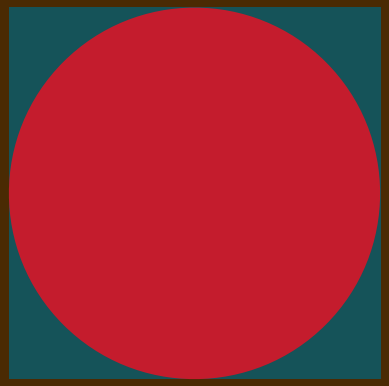
```
#include <thread>
#include <mutex>
#include <iostream>
#include <functional>
...
int main(int argc, char **argv) {

  // Set up their shared data.
  std::shared_ptr<std::mutex> mutex {new std::mutex {}};
  std::shared_ptr<long> counter {new long{0}};

  // Start both threads and wait for them to each complete.
  long count1;
  long count2;
  std::thread t1 {incrementWith(100000,counter,mutex,count1)};
  std::thread t2 {incrementWith(100000,counter,mutex,count2)};
  t1.join();
  t2.join();

  // Report the results.
  std::cout << "Thread 1 incremented " << count1 << " times." << std::endl;
  std::cout << "Thread 2 incremented " << count2 << " times." << std::endl;
  std::cout << "The final count is " << (*counter) << "." << std::endl;
}
```

USING STL <THREAD> AND <MUTEX> FOR COUNT

```
#include <thread>
#include <mutex>
...
std::function<void(void)> incrementWith(int num_times,
                                        std::shared_ptr<long> counter,
                                        std::shared_ptr<std::mutex> mutex,
                                        long& mycount) {
  return [num_times, counter, mutex, &mycount](void) -> void {
    long i;
    for (i=0; i<num_times; i++) {
      mutex->lock();
      long tmp = *counter;
      (*counter) = tmp + 1;
      mutex->unlock();
    }
    mycount  = i;
  };
}
...
int main(int argc, char **argv) {
  ...
  std::thread t1 {incrementWith(100000,counter,mutex,count1)};
  ...
}
```

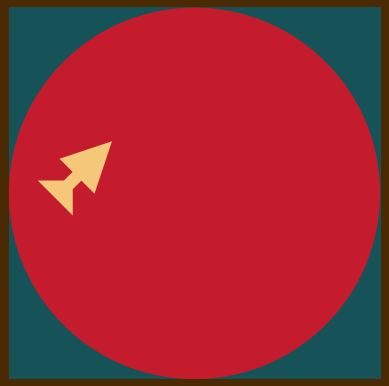
EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit.

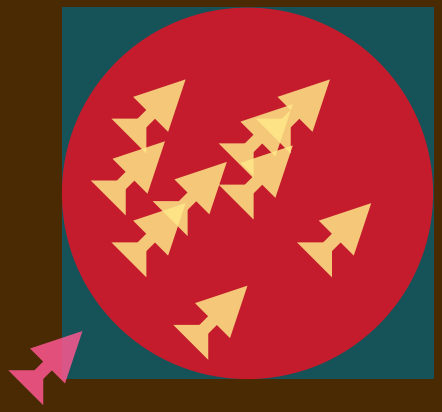
EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



- Some hit. Others miss.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit. Others miss.

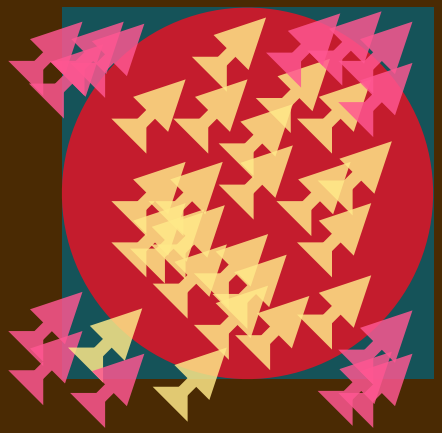
EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.

- Some hit. Others miss.

EXAMPLE: COMPUTING PI

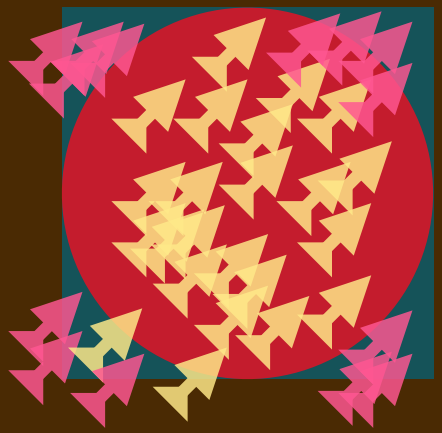
- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



- Some hit. Others miss.
- Ratio of hits to throws is π/4.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



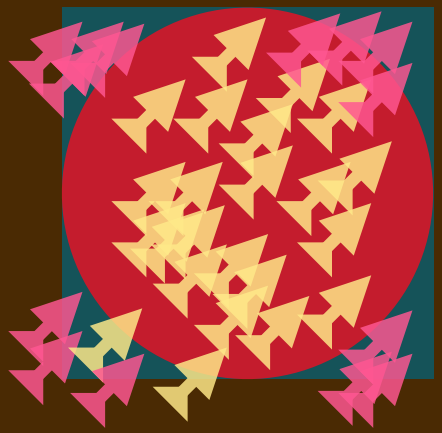
- Some hit. Others miss.
- Ratio of hits to throws is π/4.
- When run with

```
./pi 1000000 4
```

the program throws 1000000 darts and counts hits with 4 threads.

EXAMPLE: COMPUTING PI

- The **samples** folder contains **pi.cc**
- It throws tons of darts at a virtual dartboard of radius 1.



- Some hit. Others miss.
- Ratio of hits to throws is π/4.
- When run with

```
./pi 1000000 4
```

the program throws 1000000 darts and counts hits with 4 threads.

Check it out!