# CS 361: Introduction to Threads

Mark Boady

Department of Computer Science
Drexel University

September 28, 2022

# Threads

- Most computers can run multiple programs at the same time
- Threads allow one program to do multiple things at the same time
- On a single processor: OS switches between tasks
- Multiple Cores or Processors: True concurrent execution

# Threads

- Multiple things happening at once
- Threads can share memory
- Sometimes threads work independently
- Sometimes threads need to wait for other threads
- Sometimes threads need to access the same data at the same time

# Concurrency

- Concurrent Tasks
  - 1 thread for GUI
  - 1 thread for physics simulator
  - 1 thread for database updates
- Parallel Processing
  - Break one problem into many smaller problems
  - Smaller problems must be solved independently
  - Hadoop for big data

# Risks/Rewards

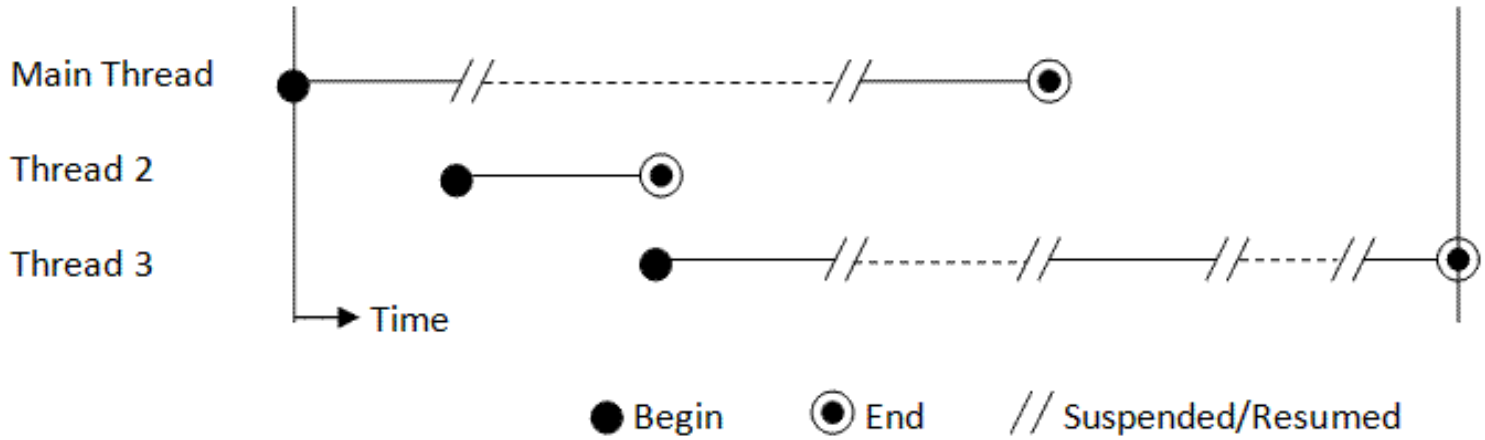
- If two threads never communicate or share data, they truly run in parallel
- In most real programs, we need to share data or wait for results
- This means sometimes one thread is waiting on another
- The key is to avoid waiting unless required
- **Race Conditions**: something bad happens because two threads change memory at the same time
- **Deadlock**: something bad happens because two threads wait on each other endlessly

# What is a thread?

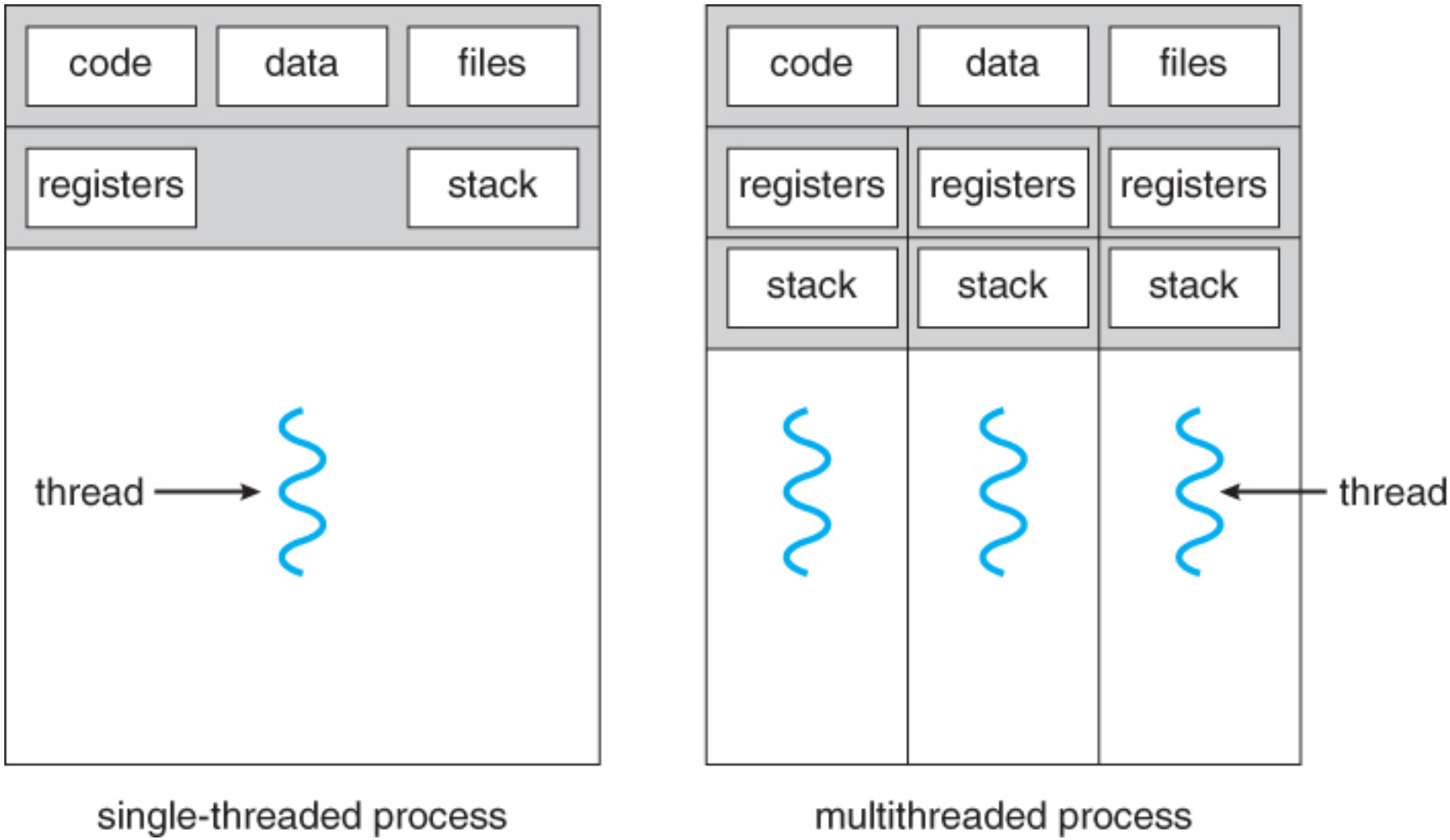
- Threads are partially independent and partially shared
- Shared:
  - All the program code loaded into memory
  - Optionally: any variable/object on the heap
  - We need to decide what variables/objects to share
- Independent:
  - Program Counter
  - Registers
  - Local Variables
  - Call Stack
- A thread should be **faster** than switching between two distinct programs due to the shared resources

# Multiple Threads

- One program and span many threads
- Threads can begin and end at an point
- `https://www3.ntu.edu.sg/home/ehchua/programming/java/images/Multithread.gif`



# Shared Memory




https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

# C++ Threads

- C++ added threads to the standard in C++11
- C++ threads further improved in C++17
- We will use the features of C++17
- Even more concurrent features in C++20

# Basics

- Part of the Standard Libary
- Use `#include <thread>`
- Create a new thread `std::thread t(someFunction)`
- Wait for the thread to finish with `t.join()`
- Compile with special flags
  `g++ -pthread --std=c++17 hello.cpp`
- *Note*: Not all OS need `-pthread`

# Parallel Hello

- Span a new thread to print a message
- Wait till that thread finishes to exit the program

# hello.cpp

code/hello.cpp

```
15 //Include I/O stream for printing
16 #include <iostream>
17 //Include thread library
18 #include <thread>
19
20 /**
21    This function prints a welcome message.
22  */
23 void welcome()
24 {
25     std::cout << "Welcome to CS361\n";
26 }
```

# hello.cpp

code/hello.cpp

```
33 int main(int argc, char** argv)
34 {
35     std::thread myThread(welcome);
36     //Wait for the thread to finish
37     myThread.join();
38     //Exit Program
39     return 0;
40 }
```

# Creating a Thread

- We create a new thread with a constructor
- Syntax:
  `std::thread threadName(functionToRun)`
- We pass in the name of the function we want the thread to execute
- We can also pass parameters to the function
- Syntax:
  `std::thread threadName(func, in1, in2, in3)`

# Passing Values

- There are multiple ways to share data between threads.
- Values are copied into the new thread by default
- Some Options to share data
  - Global Variables
  - Use Pointers
  - Use Explicit References

# Example: Sleep Sort

- Sleep Sort is s classic (silly) algorithm to show threads
- For each item, make a thread that sleeps that many seconds
- Each thread prints out its number when it wakes up.

# Sleep Sort Algorithm

**function** SLEEPTHREAD(value)
    Sleep for value seconds
    Print Value
    **return**
**end function**

# Sleep Sort Algorithm

**function** SLEEPSORT(Array A, int size)
    **for** i=0; i < size; i++ **do**
        sleepThread(A[i])
    **end for**
    Wait for all threads to end
**end function**

# Waiting for a Thread

- How do we wait for a thread?
- t.join() waits until this thread has exited
- What if we don't want to wait?
- t.detach() frees the thread to run, we won't wait for it
- A detached thread is still tied to main, it exits with the program

# Initial Setup

- We want to generate random arrays
- The size of the array comes from the command line
- We want to print the array so we can see it worked
- None of this need threads.

# sleep_01.cpp

code/sleep_01.cpp

```
18 /**
19  Generate an array of random numbers between 0
        and n*5
20  @param n is the size of the array to create
21  @return Pointer to the array created
22  */
23 int* randomArray(int n);
24 /**
25  Print out an array that is nicely formatted
26  @param A is a pointer to the array
27  @param size is the number of elements in the
        array
28 */
29 void printArray(int* A, int size);
```

# sleep_01.cpp

code/sleep_01.cpp

```
31 /**
32  Make an array with the given number of
        elements. Sort using sleepy sort.
33  @param argc is the number of command line
        arguments
34  @param argv is the text of the command line
        arguments
35  @return 1 on failure and 0 on success
36  */
```

# sleep_01.cpp

code/sleep_01.cpp

```
37 int main(int argc, char** argv)
38 {
39     //Check we got a command line argument
40     if(argc!=2)
41     {
42         std::cout <<
43             "Usage: sleepy [number elements]"
44             << std::endl;
45         return 1;
46     }
```

# sleep_01.cpp

code/sleep_01.cpp

```
47      //Use the command line argument as the
            size
48      int size = std::atoi(argv[1]);
49      //Initiate a random number generator
50      std::srand(std::time(NULL));
51      //Make an array
52      int* myArray = randomArray(size);
53      //Print the array to see what it looks
            like
54      printArray(myArray, size);
55      //Return and exit program
56      return 0;
57  }
```

# sleep_01.cpp

code/sleep_01.cpp

```
61 int* randomArray(int n){
62     int* A = new int[n];
63     for(int i=0; i < n; i++)
64     {
65         A[i] = std::rand()%(n*5);
66     }
67     return A;
68 }
```

# sleep_01.cpp

code/sleep_01.cpp

```
70 void printArray(int* A, int size)
71 {
72     std::cout << "[";
73     for(int i=0; i < size; i++)
74     {
75         std::cout << A[i];
76         if(i+1!=size)
77         {
78             std::cout << ", ";
79         }
80     }
81     std::cout << "]" << std::endl;
82 }
```

# Sleep Sort

- We need a function for the threads to run.
- Each thread will:
  1. Wait x seconds
  2. Print it's ID
  3. Print the value x
  4. End

# sleep_02.cpp

We need new includes and function prototypes

code/sleep_02.cpp

```
24 /**
25  Thread that sleeps for our sort
26  @param value is the number to sort
27 */
28 void sleepyThread(int value);
29 /**
30  Sort an array using Sleep Sort
31  @param array is the array to sort
32  @param size is the number of elements in the
        array
33 */
34 void sleepSort(int* array, int size);
```

Implementation of the Sleep Thread (Part 1)

code/sleep_02.cpp

```
105 void sleepyThread(int value){
106     //What is the Unique ID of this thread
107     std::thread::id myID =
108         std::this_thread::get_id();
109     //Determine how long to wait
110     std::chrono::seconds waitTime
111         = std::chrono::seconds(value);
```

Implementation of the Sleep Thread (Part 2)

code/sleep_02.cpp

```
112     //Fall Asleep
113     std::this_thread::sleep_for(waitTime);
114     //Print Value
115     std::cout << "Thread "
116         << myID << " says "
117         << value << std::endl;
118     //Exit
119     return;
120 }
```

# Sleep Sort Implementation

- We need to join all threads, otherwise the program will exit before we are done.
- We can't join in the loop we construct the threads
  - The loop won't continue until the first thread ended
  - No real parallel processing!
- Store threads in an array and join after all are started

Implementation of the Sleep Sort (Part 1)

code/sleep_02.cpp

```
123 void sleepSort(int* A, int size)
124 {
125     //Make an Array of threads to check
            against
126     std::thread* myThreads = new std::thread[
            size];
127     //Generate the threads
128     for(int i=0; i < size; i++)
129     {
130         myThreads[i] = std::thread(
                sleepyThread, A[i]);
131     }
```

Implementation of the Sleep Sort (Part 2)

code/sleep_02.cpp

```
132         //Wait for all the threads to end
133         for(int i=0; i < size; i++)
134         {
135             myThreads[i].join();
136         }
137         //Exit Function
138         return;
139     }
```

# Example Execution

```
1  [22, 28, 46, 38, 39, 10, 13, 39, 47, 47]
2  Thread 0x70000995c000 says 10
3  Thread 0x7000099df000 says 13
4  Thread 0x7000096cd000 says 22
5  Thread 0x700009750000 says 28
6  Thread 0x700009856000 says 38
7  Thread Thread 0x7000098d9000 says 0
       x700009a62000 says 39
8  39
9  Thread 0x7000097d3000 says 46
10 Thread Thread 0x700009ae5000 says 0
       x700009b68000 says 4747
```

# Race Condition

- The print statements overlapped with each other!
- Both threads were trying to use cout at the same time
- **Race Condition**: when two or more threads try to access the same memory at the same time
- Bad things can happen!

# Locks

- A **lock** is a data structure that ensures only one thread accesses something at a time
- We can have multiple **locks** to manage different resources
- New problems:
  - **Deadlock**: When a series of threads are all stuck waiting for each other to release a lock. It becomes impossible for any thread to move forward.
  - Slowdown is introduced any time a thread needs to wait its turn
- To much code protected by locks forces everything to run in sequence

# Locked Function

- We can lock an entire function
- Only one thread will be able to use this function at a time
- The lock is automatically released when the function returns
- Makes it easier to organize and manage locks
- We need shared lock for all the threads

Updated Include and global

code/sleep_03.cpp

```
19 //Needed for threading
20 #include <thread>
21 //Needed for sleep
22 #include <chrono>
23 //Needed for mutex lock
24 #include <mutex>
25 //Make our global lock
26 std::mutex coutLock;
```

New Print Function with Lock

code/sleep_03.cpp

```
154 void printLock(std::thread::id id, int num){
155     //Create a lock that lasts the
156     //life of this function
157     const std::lock_guard<std::mutex>
158         lock(coutLock);
159     //Print to Cout
160     std::cout << "Thread "
161         << id << " says "
162         << num << std::endl;
163 }
```

# Example Execution

```
[8, 7, 8, 0, 8, 3, 4, 1, 2, 3]
Thread 0x700002e17000 says 0
Thread 0x700003023000 says 1
Thread 0x7000030a6000 says 2
Thread 0x700002f1d000 says 3
Thread 0x700003129000 says 3
Thread 0x700002fa0000 says 4
Thread 0x700002d11000 says 7
Thread 0x700002c8e000 says 8
Thread 0x700002d94000 says 8
Thread 0x700002e9a000 says 8
```

# Sorting

- The values are sorted in order.
- They are not sorted in the array!
- We can have the thread store in the array instead of printing
- We need to know what index to store to
- The index variable needs to be locked, we don't equal numbers in the same index

# Array Index

- Each Thread will update a different index in the array
- The array does not need to be locked, no threads will overlap
- The index needs to be updates by each thread
- The index needs to be locked or two threads might look at the wrong index
- The index lock protects the whole array
- We lock inline to protect only the lines we need
- Counter is passed by reference

## Revised Includes

code/sleepsort.cpp

```
12 //cout and endl
13 #include <iostream>
14 //We need srand and rand
15 #include <cstdlib>
16 //We can use the time to set the random
       generator
17 #include <ctime>
18
19 //Needed for threading
20 #include <thread>
21 //Needed for sleep
22 #include <chrono>
23 //Needed for mutex lock
24 #include <mutex>
```

## Update to Sleep Sort Algorithm (Part 1)

code/sleepsort.cpp

```
141 void sleepSort(int* A, int size){
142     int index=0;
143     //Array of threads
144     std::thread* myThreads
145         = new std::thread[size];
146     //Generate the threads
147     for(int i=0; i < size; i++)
148     {
149         myThreads[i] = std::thread(
                sleepyThread,
150             A,//Array to change
151             A[i], //value to sleep on
152             std::ref(index)); //target
153     }
```

## Update to Sleep Sort Algorithm (Part 2)

code/sleepsort.cpp

```
154         //Wait for all the threads to end
155         for(int i=0; i < size; i++)
156         {
157             myThreads[i].join();
158         }
159         //Exit Function
160         return;
161     }
```

## Revised Sleep Thread (Part 1)

code/sleepsort.cpp

```
120 void sleepyThread(int * array, int value, int
      &index){
121   //Determine how long to wait
122   std::chrono::seconds waitTime
123       = std::chrono::seconds(value);
124   //Fall Asleep
125   std::this_thread::sleep_for(waitTime);
126   //Update Array
127   int myIndex;
```

## Revised Sleep Thread (Part 2)

code/sleepsort.cpp

```
128         //Lock so only this thread can change
129         counterLock.lock();
130         //Update Index
131         myIndex = index++;
132         //Free Lock
133         counterLock.unlock();
134         //Update array
135         array[myIndex]=value;
136         //Exit
137         return;
138 }
```

# Example Execution

```
1 [9, 7, 9, 8, 8, 5, 2, 8, 4, 4]
2 [2, 4, 4, 5, 7, 8, 8, 8, 9, 9]
```

# Summary

- Threads can be created
- We can join or detach threads
- Values can be shared between threads
- Race Conditions happen when threads use the same memory at the same time
- Locks and protect memory
- Locks can cause problems (more on this in future weeks)