Game Engineering: 2D

CS420-2013S-19

Introduction to Threading

David Galles

Department of Computer Science
University of San Francisco

9-0: Parallel Programming

- Xbox has 3 cores, each of which has 2 hardware threads
- Of these 6 hardware threads, XNA programs have access to 4
- Let's not let that processing power go to waste!

9-1: Thread vs. Process

- Different processes contain their own stack, address space, etc.
- Different processes can only communicate through some form of message passing
- Processes tend to be loosely coupled
- Easy example: Web browser and a word processor running simultaneously on your system, different processes

9-2: Thread vs. Process

- Threads are much lighter weight than processes
  - Threads share the same address space
  - Communicate through shared memory
  - Tend to be more tightly coupled than processes

9-3: Threading in C#

- Games that utilize parallelism are usually multi-threaded
- Main thread starts up the game, does initialization
- Main thread starts subthreads to do additional processing

9-4: C# Thread Basics

- Thread class
- Create an instace of this class, passing in the function that the thread will run
- Start the thread up, runs in parallel

9-5: C# Thread Basics

- Thread Basics program (see Lecture Notes page for code)
- What does this output?

9-6: C# Thread Basics

- Thread Basics program (see Lecture Notes page for code)
- What does this output?
  - Something like:

```
AAAAAAABBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBB
BBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAABBBB
BBBBBBBBBBB
```

9-7: C# Thread Basics

- All new threads need a function to start with
- The Delegate for function to start thread is:

```
void ThreadStart()
```

- Given any object, and any void method on that object that takes no parameters, we can start a thread to run that method

9-8: C# Thread Basics

```
ass MyClass

 ...
 void Method() { ... }
 ...



Class c = new MyClass();
read t = new Thread(new ThreadStart(c.Method));
Start();
```

9-9: C# Thread Basics

- Examples using Delegate.cs

9-10: C# Thread Basics

```
ass C {
int mRepeat = 5;
void Process() {
  for (;mRepeat > 0; mRepeat--)
    Console.Write("X");
}

.
c1 = new C();
c2 = new C();
read t1 = new Thread(c1.Process);
read t2 = new Thread(c2.Process);
.Start();
.Start();
```

9-11: C# Thread Basics

```
ass C {
int mRepeat = 5;
void Process() {
  for (;mRepeat > 0; mRepeat--)
    Console.Write("X");
}

.
c1 = new C();
read t1 = new Thread(c1.Process);
read t2 = new Thread(c1.Process);
.Start();
.Start();
```

9-12: C# Thread Basics

- In the second example, the single variable `mRepeat` is shared among both threads
- Sharing data among threads is a powerful way for threads to communicate, can lead to difficulties ...

9-13: Shared Memory

```
ass Program {
public static int counter = 0;
static void increment() {
  for (int i = 0; i < 1000; i++)
    counter = counter + 1;
  Console.WriteLine("Counter = " + counter.ToString());
}
public static void main(string[] args) {
  Thread t1 = new Thread(increment);
  Thread t2 = new Thread(increment);
  Thread t3 = new Thread(increment);
  Thread t4 = new Thread(increment);
  t1.Start();
  t2.Start();
  t3.Start();
  t4.Start();
}
```

9-14: Shared Memory

- Since the commands in the previous example could be interleaved in any way, we have no way of knowing what will be printed out for each thread
- Do we know anything about what the output will be?
- What about the *last* counter value that is printed out?

9-15: Shared Memory

- Alas, we don't know very much at all about what *any* of the print statements will be, *including the last one!*
- Look at the statement

```
counter = counter + 1;
```

  - First, the right-hand side of the statement is evaluated
  - Next, the value is stored in counter

9-16: Shared Memory

Counter Value	Thread1	Thread2
0	Evaluate counter+1 (1)	
1	Set Counter = 1	
1		Evaluate counter+1 (2)
2		Set Counter = 2
2	Evaluate counter+1 (3)	
3	Set Counter = 3	
3		Evaluate counter+1 (4)
4		Set Counter = 4

9-17: Shared Memory

Counter Value	Thread1	Thread2
0	Evaluate counter+1 (1)	
0		Evaluate counter+1 (1)
1	Set Counter = 1	
1		Set Counter = 1
1	Evaluate counter+1 (2)	
1		Evaluate counter+1 (2)
2		Set Counter = 2
2	Set Counter = 2	

9-18: **Shared Memory**

- What if we change:
  - counter = counter+1
- to:
  - counter++

9-19: Shared Memory

- What if we change:
  - counter = counter+1
- to:
  - counter++
- Alas, still have the same problem

9-20: Thread Safety

- A program or method is *Thread Safe* if it can be called from multiple threads without unwanted interaction between the threads.
- A program or method that is not thread-safe will potentially have different behavior each time you run it
  - This kind of non-determinism is *very, very bad*
- How can we make our code Thread safe?

9-21: Thread Safety

- Most Thread-unsafe behavior comes from data shared between threads
  - Thread1 move some data from memory to register / cache
  - Thread2 changes the value of this memory location
  - Thread1 writes a value back to the same memory location
- We could eliminate the problem by eliminating any shared data between threads – but that's not practical

9-22: **Locking**

- We need to prevent two different threads from accessing the same data at the same time
- Use *Locks*

9-23: Locking

- Operating System mainains a token
- Threads can ask for the token
  - If the token is available, the thread gets the token
  - If not the thread *blocks*, and waits for the token to become available
  - When a thread gives the token back, it becomes available for other threads to use

9-24: Locking

- To use a lock:
  - Before accessing a shared variable, first acquire a lock
  - Do any calculation that you want to do
  - When you are done, give up the lock so that other threads can accses the variable

9-25: Locking

```
ass Program {
public static Object token = new Object();
public static int counter = 0;
static void increment() {
  for (int i = 0; i < 1000; i++)
    lock (token)
    {
      counter = counter + 1;
    }
  Console.WriteLine("Counter = " + counter.ToString());
}
public static void main(string[] args) {
  Thread t1 = new Thread(increment);
  Thread t2 = new Thread(increment);
  Thread t3 = new Thread(increment);
  Thread t4 = new Thread(increment);
  t1.Start();
  t2.Start();
  t3.Start();
  t4.Start();
```

Locking

- When we run this program, the intermediate values printed out for the counter will all be different – but the last one will be 4000
- Before any thread tries to access counter, it first tries to get a token
  - If the token is available, it takes the token and does its work
  - If the token is not available, it waits until the token is available
- No longer have any of the interleaving problems that we did before

9-27: Locking

- In order for Locking to work, acquiring a lock needs to be an atomic operation
  - Can’t have two threads ask for a lock simultaneously, and both get it
- Fortunately, locks *are* atomic, and you don’t need to worry too much about how they are implemented, just that they work

9-28: Locking

- Any object can be locked
- We can have as many different locks as we like
- Object(s) we are using as locks need not have any relation to the data we are modifying
  - A lock is just a token source

9-29: Locking

- Class Example: Locking & Tokens

9-30: Locking

- What if you forget a lock?

9-31: Locking

```
ass Program {
public static readonly Object token = new Object();
public static int counter = 0;

static void inc() {
  for (int i = 0;
       i < 1000;
       i++)
    lock (token)
    {
      counter = counter + 1;
    }
  Console.WriteLine(counter);
}

public static void inc2() {
  for (int i = 0;
       i < 1000;
       i++)
  {
     counter = counter + 1;
  }

public static void main(string[] args) {
  Thread t1 = new Thread(inc);
  Thread t2 = new Thread(inc);
  Thread t3 = new Thread(inc2);
  Thread t4 = new Thread(inc);
  t1.Start();
```

9-32: Locking

- In order for locking to work properly, you need to acquire a lock *every time* you want to access a variable
- Often be a good idea to acquire a lock, even if you are just examining the value of a variable (why?)

9-33: Locking

- In order for locking to work properly, you need to acquire a lock *every time* you want to access a variable
- Often be a good idea to acquire a lock, even if you are just examining the value of a variable (why?)
  - Some other thread may be currently modifying the variable, might be in an in-between state
  - If you can acquire a lock, you know that no one is currently modifying the variable $\rightarrow$ not in an inconsistent state

9-34: What to Lock

- In C#, any object can be used as a lock
  - Remember, a lock object is just a token source
  - Does not need to have any relation to the data you are accessing
- We can however, use the actual object we are modifying as a lock object

9-35: What to Lock

```
ass MyStack

 int[] data;
 int top;

 void Push(int elem)
 {
     lock(this)
     {
         data[top++] = elem;
     }
 }
 int Pop()
 {
     lock(this)
     {
        return data[--top];
     }
 }
```

9-36: What to Lock

- We can use one token to lock a large group of variables
  - An entire large data structure
- We can to finer grained locks
  - Use a number of tokens to lock different pieces of a larger data structure
- What is the advantage of each kind of locking strategy?

9-37: C# Collections

- Most C# Collections are *not* thread safe
  - (Like the non-locked version of the stack example)
- Why not?

9-38: C# Collections

- Most C# Collections are *not* thread safe
  - Incur locking cost (which is pretty small, 20ns) even for non-parallel programs
  - Even if the structures themselves were thread-safe, often still need to use locking constructs

9-39: C# Collections

- Even if `myList` was thread-safe, the following would not be:

```
f (!myList.Contains(newItem))
  myList.Add(nwItem)
```

9-40: C# Collections

- Even if `myList` was thread-safe, the following would not be:

```
f (!myList.Contains(newItem))
  myList.Add(nwItem)
```

- Solution: Wrap this access (and *all other* accesses of myList inside a lock
- If myList *was* thread-safe, duplicated effort.

9-41: Nested Locking

```
blic static readonly lock1 = new Object();

atic void run1()

 lock(lock1)
 {
    lock(lock1)
    {
       // do something
    }
 }
```

- Will only block a thread on the outermost lock
- Why would you ever want to do nested locking?

9-42: Nested Locking

```
ass Stack {
 bool Empty()
 {
     lock(this)
     {
        return mTop == 0;
     }
 }
 void Pop()
 {
    lock(this)
    {
       if (!Empty)
          return Data[--mTop]
    }
 }
 ... (rest of class definition)
```

9-43: Deadlock

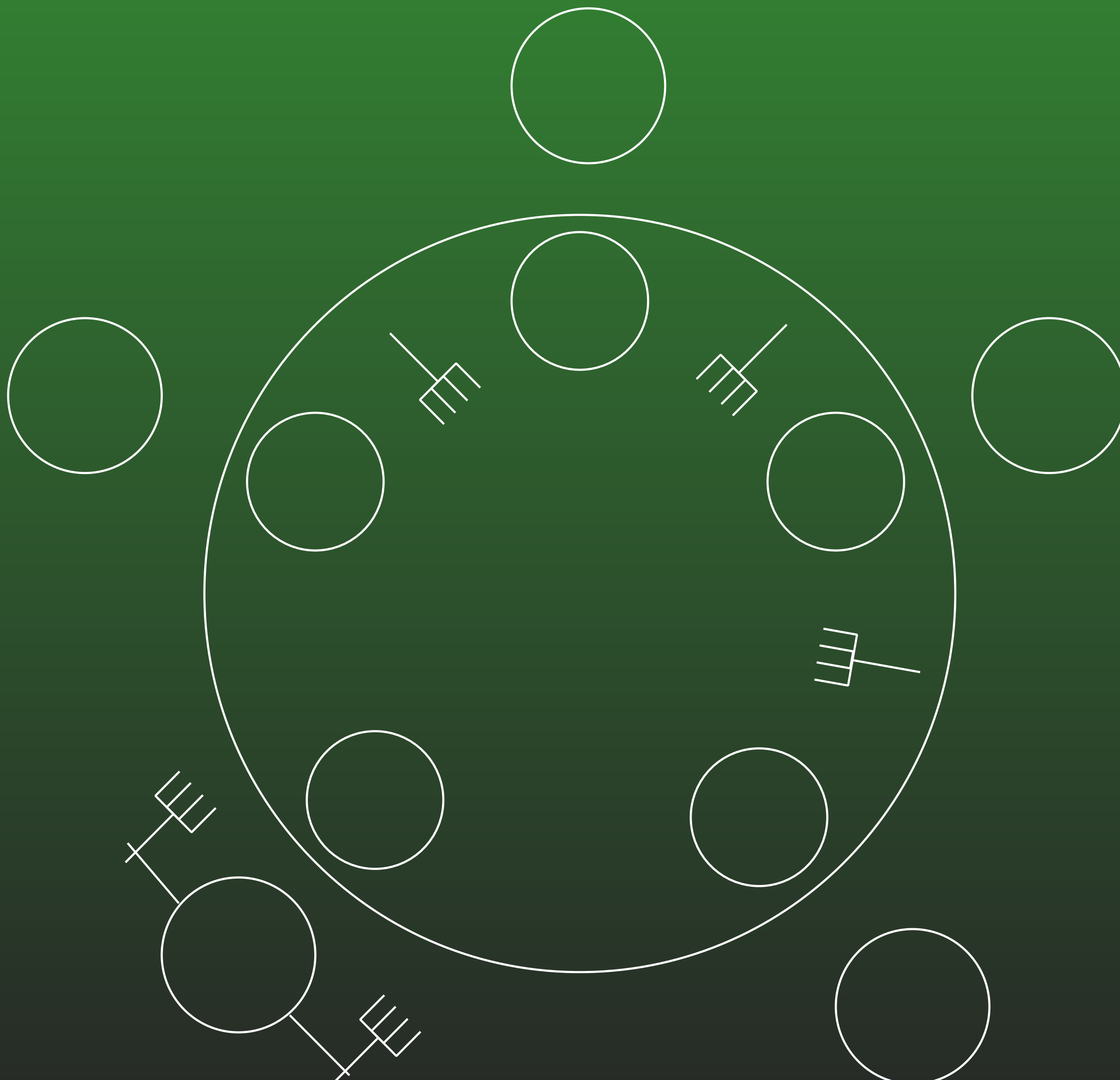
```
blic static readonly lock1 = new Object();
blic static readonly lock2 = new Object();
atic void run1()

 lock(lock1)   {
    Thread.Sleep(1000);
    lock (lock2);
 }

atic void run2()

 lock(lock2)   {
    Thread.Sleep(1000);
    lock (lock1);
 }

read t1 = new Thread(run1);
read t2 = new Thread(run2);
.Start();  t2.Start();
```

9-44: Deadlock

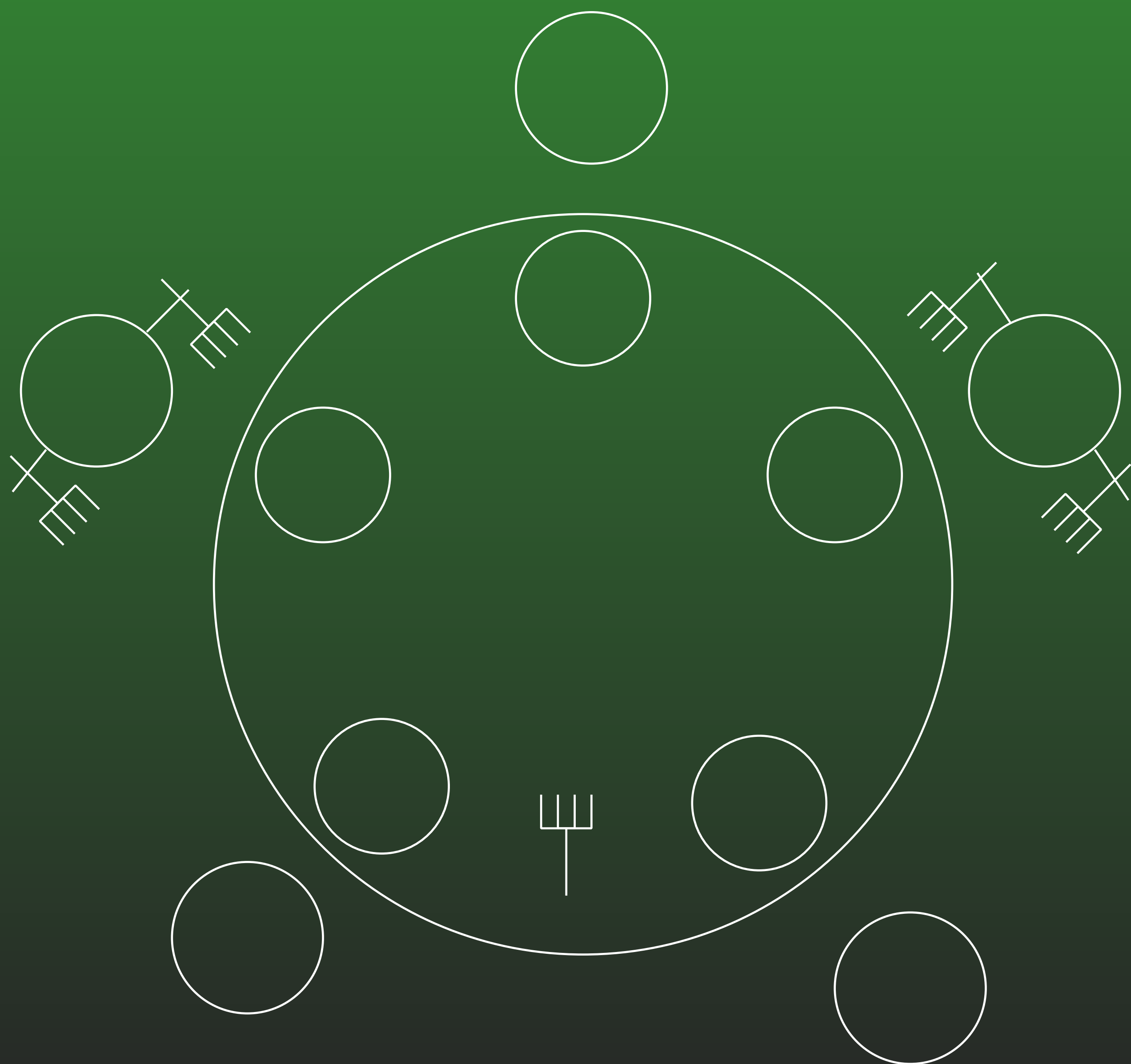
- Dinining Philosophers
  - 5 Philosophers sit around a table
  - Philosophers alternate between thinking and eating
  - In order to eat, need to pick up both forks, eat, put them down

 # Dining Philosphers

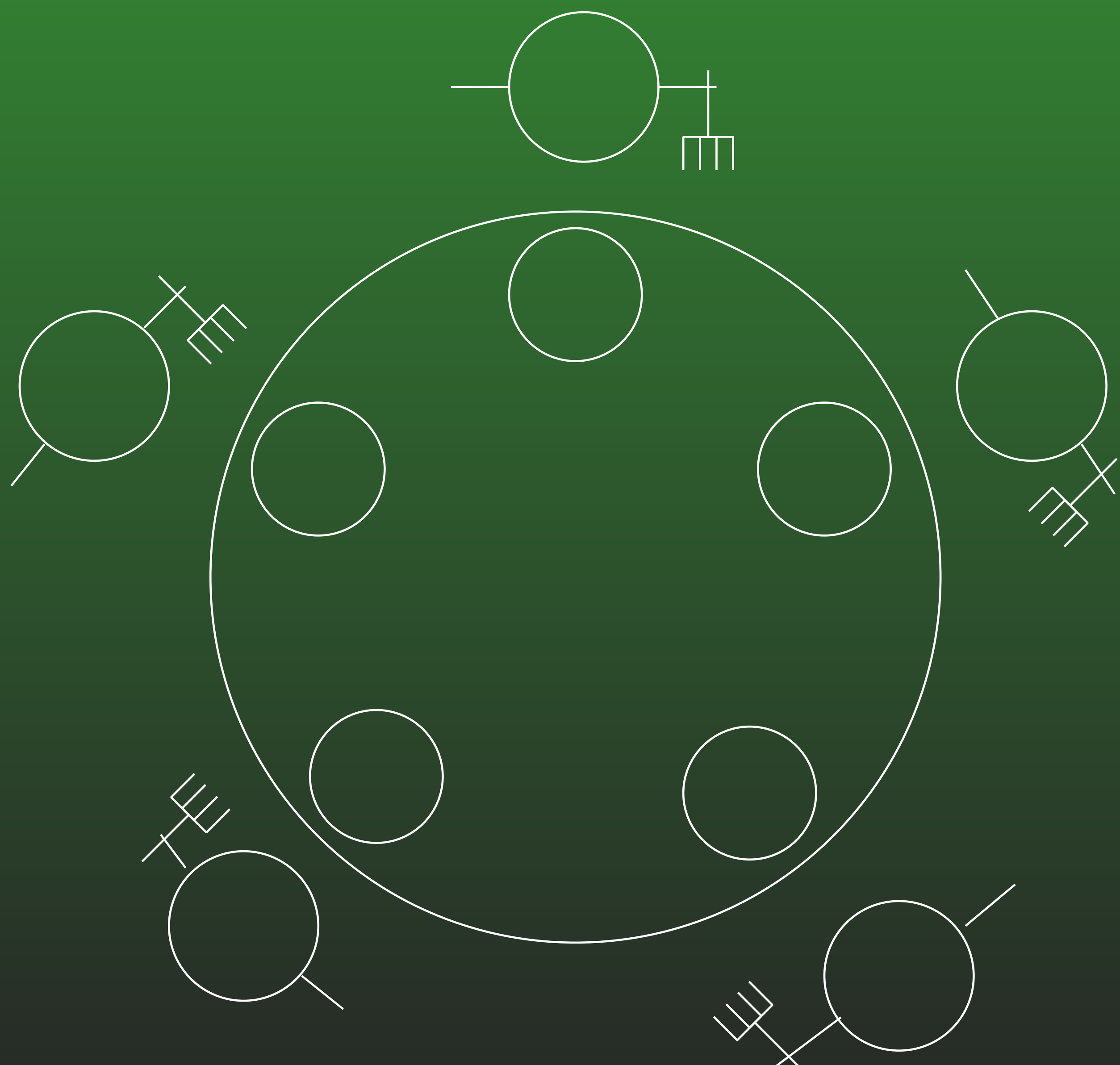
9-46: Dining Philosphers



 # Dining Philosphers



9-48: Dining Philosphers



 # Dining Philosphers

- Potential Solution:
  - Try to pick up first fork
    - If not available, block
  - Try to pick up second fork
    - If not available, put down both forks, wait 5 minutes

9-50: Dining Philosphers

- Everyone picks up left fork
- Everyone tries to pick up right fork
  - None available
  - Everone puts down left fork, waits 5 minutes
- Repeat

Dining Philosphers

- Order the forks, $1 \rightarrow 5$
- Each philosopher tries to pick up the smaller numbered fork first.
- If the first fork is successfully picked up, try the second fork
- Assuming that no philosopher dies mid meal, will this work?

9-52: Thread Synchronization

- What if you want all of your threads to syncronize
  - All frames to agree on what frame we’re currently on
  - All frames to start each frame at the same time

9-53: AutoResetEvent

- An AutoResetEvent is like a ticket turnstile
  - When closed, no one can get through
  - If you insert a ticket, the turnstile opens to let exacty one person through, then closes again
    - Called “AutoReset” because of this automatic closing (reseting) after someone has gone through

9-54: AutoResetEvent

```
atic EventWaitHandle waitHandle = new AutoResetEvent(false);

atic void Main()

  new Thread (Waiter).Start();
  Thread.Sleep (1000);
  waitHandle.Set();

atic void Waiter()

  Console.WriteLine ("Waiting...");
  waithandle.WaitOne();
  Console.WriteLine ("Notified");
```

9-55: AutoResetEvent

- When we create an AutoResetEvent, we pass in a boolean to note if we want the turnstile to start out open (true) or closed (false)
- Call Set() method to open the turnstile (putting in a ticket)
- Call WaitOne() method to wait for the turnstile to be open

9-56: AutoResetEvent

- When we call Set() on an AutoResetEvent, it stays open until a thread goes through and closes it
- Calling Set() on an open AutoResetEvent is a no-op

9-57: AutoResetEvent

```
atic EventWaitHandle waitHandle - new AutoResetEvent(false);
atic void Main()

 new Thread (Work).Start();
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();

atic void Work()

 waitHandle.WaitOne();
 Console.WriteLine("Step 1");
 Thread.Sleep(3);
 waitHandle.WaitOne();
 Console.WriteLine("Step 2");
 Thread.Sleep(3);
 waitHandle.WaitOne();
 Console.WriteLine("Step 3");
```

AutoResetEvent

```
atic EventWaitHandle waitHandle - new AutoResetEvent(false);
atic void Main()

 new Thread (Work).Start();
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();

atic void Work()

 waitHandle.WaitOne();
 Console.WriteLine("Step 1");
 Thread.Sleep(300);
 waitHandle.WaitOne();
 Console.WriteLine("Step 2");
 Thread.Sleep(300);
 waitHandle.WaitOne();
 Console.WriteLine("Step 3");
```

9-59: AutoResetEvent

```
atic EventWaitHandle waitHandle - new AutoResetEvent(false);
atic void Main()

 new Thread (Work).Start();
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();
 Thread.Sleep(100);
 waitHandle.Set();

atic void Work()

 waitHandle.WaitOne();
 Console.WriteLine("Step 1");
 Thread.Sleep(100);
 waitHandle.WaitOne();
 Console.WriteLine("Step 2");
 Thread.Sleep(100);
 waitHandle.WaitOne();
 Console.WriteLine("Step 3");
```

9-60: AutoResetEvent

- Main thread signals worker thread several times
- Don’t want to miss any of the signals
- What can we do?

9-61: AutoResetEvent

- Main thread signals worker thread several times
- Don’t want to miss any of the signals
- What can we do?
  - Hint: It’s OK for the main thread to wait on the worker ...

9-62: 2-Way Signalling

- Two AutoResetEvents
  - One for the worker thread, waiting for work to be ready
  - One for the main thread, waiting for worker to be done

9-63: 2-Way Signalling

```
static EventWaitHandle workerReady = new AutoResetEvent (false);
static EventWaitHandle workerGo = new AutoResetEvent (false);
static readonly object locker = new object();
static string message;

static void Main()
{
  new Thread (Work).Start();
  workerReady.WaitOne();
  lock (locker) message = "First Message";
  workerGo.Set();
  workerReady.WaitOne();
  lock (locker) message = "Second Message";
  workerGo.Set();
  workerReady.WaitOne();
  lock (locker) message = "Third Message";
  workerGo.Set();
  workerReady.WaitOne();
  lock (locker) message = null;
  workerGo.Set();
```

9-64: 2-Way Signalling

```
static void Work()
{
  while (true)
  {
    workerReady.Set();
    workerGo.WaitOne();
    lock (locker)
    {
      if (message == null)
         return;
      Console.WriteLine (message);
    }
  }
}
```

9-65: Producer/Consumer Queue

- Main thread adds tasks to a task queue
- One or more worker Threads pull tasks off the queue
- Code on website