12 Concurrency

Instructor: Ke Wei（柯韋）

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

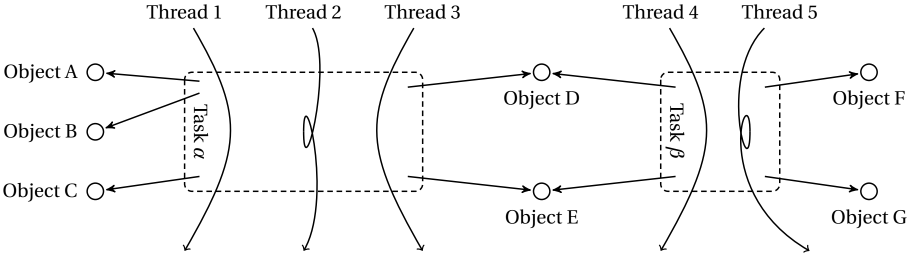
November 26(28), 2019

Outline

1. Threads
2. The *Thread* Class
3. Thread Synchronization
4. Semaphores
5. Summary of Thread States

Threads

- A thread is a *flow of execution*, of a *task* in a program, running *independently*.
- You can launch multiple threads from a program *concurrently*.
- Threads from a single program share resources, such as tasks and object instances.
- Threads can be executed *simultaneously* in multiprocessor systems



- Java provides exceptionally good support for creating and running threads and for locking resources to prevent conflicts.

Creating Tasks and Threads

- A task is a *Runnable* object that encapsulates a segment of code and the related resources (data).

```
class MyTask implements Runnable {
    private int n;
    public MyTask(int n) { this.n = n; }
    @Override public void run() { System.out.println(n); }
}
```

- A task must be executed in a thread. The *Thread* class contains the constructors for creating threads and many useful methods for controlling threads.

```
MyTask task = new MyTask(10);
Thread thread = new Thread(task);
```

- You invoke the *start* method to start the thread when it is ready to run.

```
thread.start();
```

The *Thread* Class

- Since the *Thread* class implements *Runnable*, when a task is to be executed by only one thread, you could declare a class that extends *Thread* and overrides the *run* method.
- `void` *start*() — causes `this` thread to begin execution.
- `boolean` *isAlive*() — tests if `this` thread is alive. A thread is alive when it is running in the *run* method of the task.
- `void` *join*() — waits for `this` thread to die.
- `static void` *sleep*(`long` *millis*) — causes the *current thread* to sleep for the specified number of milliseconds, approximately.
- `static void` *yield*() — gives a hint to the scheduler that the current thread is willing to yield its current use of a processor.
- If, in the running of *threadA*, a call *threadB*.*join*() is made, then *threadA* is the current thread for the entire call, and *threadB* is the `this` thread for method *join*. That is, *threadA* waits for *threadB* to die.

Example: Passing Arguments and Returning Results

ComputeProduct computes the product of $from \times (from + step) \times \cdots \times to$. The arguments and result are stored as data fields, which can be accessed before and after the thread execution.

```java
public class ComputeProduct extends Thread {
    private long from, to, step;
    private BigInteger res = BigInteger.valueOf(1);
    public ComputeProduct(long from, long to, long step) {
        this.from = from; this.to = to; this.step = step;
    }
    @Override public void run() {
        for ( ; from <= to; from += step )
            res = res.multiply(BigInteger.valueOf(from));
    }
    public BigInteger getRes() { return res; }
}
```

Example: Waiting for Threads to Complete

We start a number of threads to compute the products of the sub-sequences concurrently, and wait for the results and accumulatively multiply them together.

```java
public static void main(String[] ss) throws InterruptedException {
    final int NT = 4;
    ComputeProduct[] ts = new ComputeProduct[NT];
    for ( int i = 0; i < NT; ++i ) {
        ts[i] = new ComputeProduct(i+1, 60000, NT);
        ts[i].start();
    }
    BigInteger r = BigInteger.ONE;
    for ( int i = 0; i < NT; ++i ) {
        ts[i].join();
        r = r.multiply(ts[i].getRes());
    }
}
```

The synchronized Keyword

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- Certain sequence of actions on an object cannot be interleaved with other actions.
- It is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*.
- A `synchronized` method acquires a lock (on `this` object, or the class) before it executes.

```
public synchronized void deposit(double amount) {
    this.balance = this.balance+amount;
}
```

- A synchronized statement can be used to acquire a lock on any object, when executing a block of statements.

```
synchronized ( obj ) { obj.use(); }
```

Example: Dining Philosophers

```java
public class Philosopher extends Thread {
    private int id;
    private Object firstFork, secondFork;
    public Philosopher(int id, Object first, Object second) {
        this.id = id; this.firstFork = first; this.secondFork = second;
    }
    @Override public void run() {
        for ( ; ; ) {
            System.out.println("Philo␣#"+id+"␣is␣thinking.");
            synchronized ( firstFork ) {
                System.out.println("Philo␣#"+id+"␣is␣taking␣the␣1st␣fork.");
                synchronized ( secondFork ) {
                    System.out.println("Philo␣#"+id+"␣is␣eating.");
                } ...
```

Example: Dining Philosophers (2)

```
public static void main(String[] args) {
    final int N = 5;
    Object[] forks = new Object[N];
    for ( int i = 0; i < N; ++i )
        forks[i] = new Object();

    Philosopher[] philos = new Philosopher[N];
    for ( int i = 0; i < N; ++i ) {
        philos[i] = new Philosopher(i+1, forks[i], forks[(i+1)%N]);
        philos[i].start();
    }
}
```

The above fork allocation could lead to the case that all philosophers are holding the first forks $(0,1,2,3,4)$ and waiting for the second forks $(1,2,3,4,0)$, that is, a *dead lock*.

Avoiding Deadlocks

- Two or more threads may need to acquire the locks on several shared objects.
- This could cause a deadlock, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.

Thread 1:

```
synchronized ( a ) {
    ...
    synchronized ( b ) { ★
        ...
    }
}
```

Thread 2:

```
synchronized ( b ) {
    ...
    synchronized ( a ) { ★
        ...
    }
}
```

- Deadlock can be avoided by using a simple technique known as *resource ordering*.
- You assign an order on all the locks and ensure that each thread acquires the locks in that order.

Avoiding the Deadlock in Dining Philosophers

```
public static void main(String[] args) {
    final int N = 5;
    Object[] forks = new Object[N];
    for ( int i = 0; i < N; ++i )
        forks[i] = new Object();

    Philosopher[] philos = new Philosopher[N];
    for ( int i = 0; i < N; ++i ) {
        if ( i < N-1 )
            philos[i] = new Philosopher(i+1, forks[i], forks[i+1]);
        else
            philos[i] = new Philosopher(i+1, forks[0], forks[i]);
        philos[i].start();
    }
}
```

Synchronized Collections

- The classes in the Java Collections Framework are not *thread-safe*, that is, their contents may be corrupted if they are accessed and updated concurrently by multiple threads.
- You can protect the data in a collection by locking it, that is, a synchronized collection.
- The *Collections* class provides six static methods for wrapping a collection into a synchronized version.

```
static ⟨T⟩ Collection⟨T⟩ synchronizedCollection(Collection⟨T⟩ c)
static ⟨T⟩ List⟨T⟩ synchronizedList(List⟨T⟩ list)
static ⟨K, V⟩ Map⟨K, V⟩ synchronizedMap(Map⟨K, V⟩ m)
static ⟨T⟩ Set⟨T⟩ synchronizedSet(Set⟨T⟩ s)
static ⟨K, V⟩ SortedMap⟨K, V⟩ synchronizedSortedMap(SortedMap⟨K, V⟩ m)
static ⟨T⟩ SortedSet⟨T⟩ synchronizedSortedSet(SortedSet⟨T⟩ s)
```

- Synchronized collections are thread-safe, but the iterator is *fail-fast*. You need to acquire a lock on the synchronized collection when traversing it.

Semaphores

- Semaphores can be used to restrict the number of threads that access a shared resource.
- Before accessing the resource, a thread must acquire a permit from the semaphore.
- After finishing with the resource, the thread must return the permit back to the semaphore.
- To create a semaphore, you have to specify the number of initial permits.

```
Semaphore semaphore = new Semaphore(10);
...
semaphore.acquire();
...
semaphore.release();
```

Example: Adding a Sequence of Numbers Using a Buffer

- The main thread feeds a large number of integer elements into a buffer of a fixed and smaller size.
- An adder thread takes two elements from the buffer and puts the sum of the two back to the buffer, thus the total number of elements is decreased by 1.
- All adder threads keep doing this until there's only one element left.
- We use a pair of semaphores to control the buffer access.

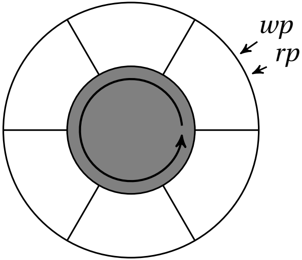
Adding Numbers — Illustrated

Input from *main*: 1 2 3 4 5 6 7 8 9 10

Adder #1: ○+○=○

Adder #2: ○+○=○

Number of Ops: 9

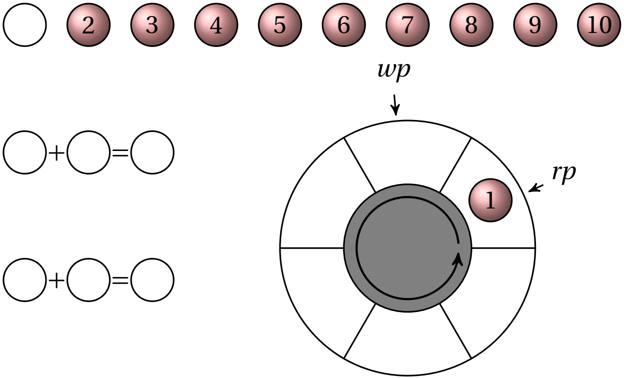


Adding Numbers — Illustrated

Input from *main*:

Adder #1:

Adder #2:



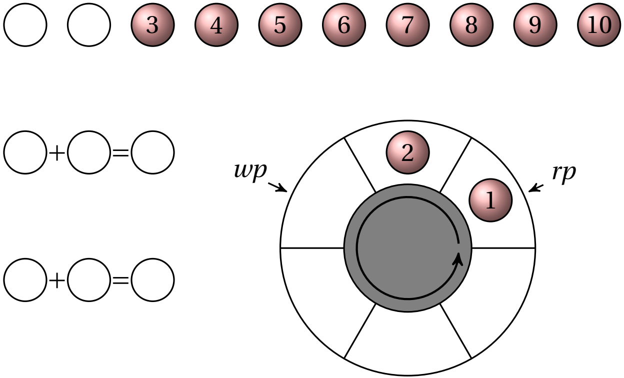
Number of Ops: 9

Adding Numbers — Illustrated

Input from *main*:

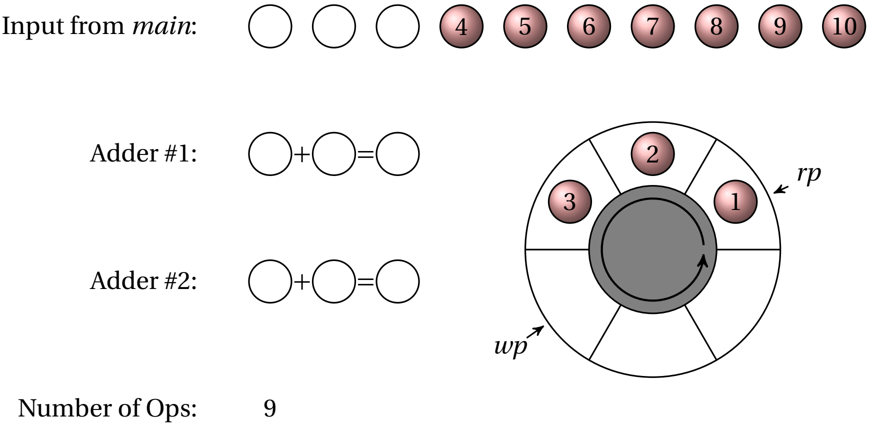
Adder #1:

Adder #2:



Number of Ops: 9

Adding Numbers — Illustrated



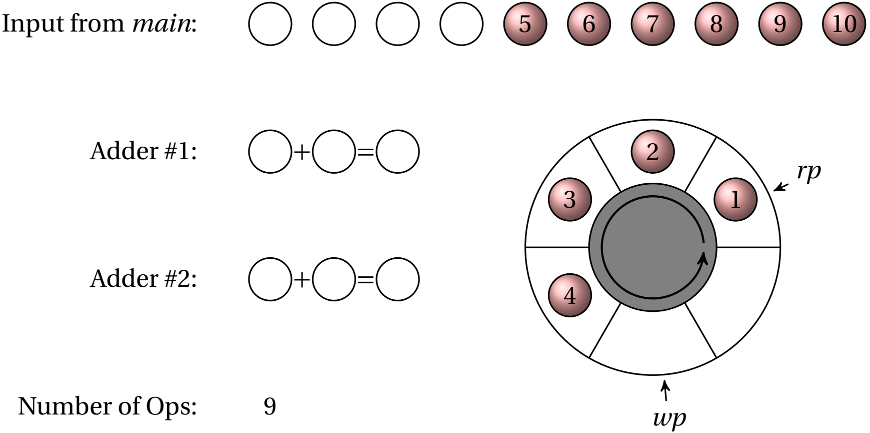
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 9

Adding Numbers — Illustrated



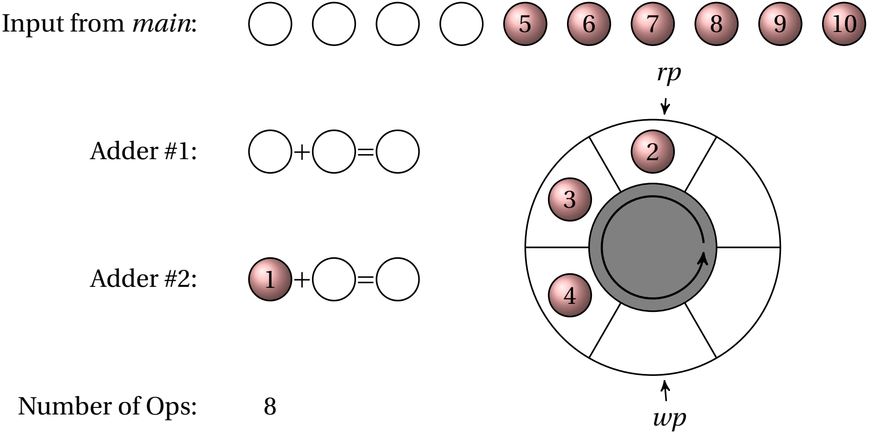
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 9

Adding Numbers — Illustrated



Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 8

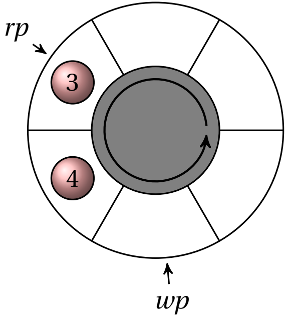
Adding Numbers — Illustrated

Input from *main*: ○ ○ ○ ○ 5 6 7 8 9 10

Adder #1: 2 + ○ = ○

Adder #2: 1 + ○ = ○

Number of Ops: 7



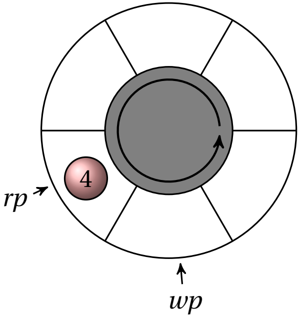
Adding Numbers — Illustrated

Input from *main*: ○ ○ ○ ○ 5 6 7 8 9 10

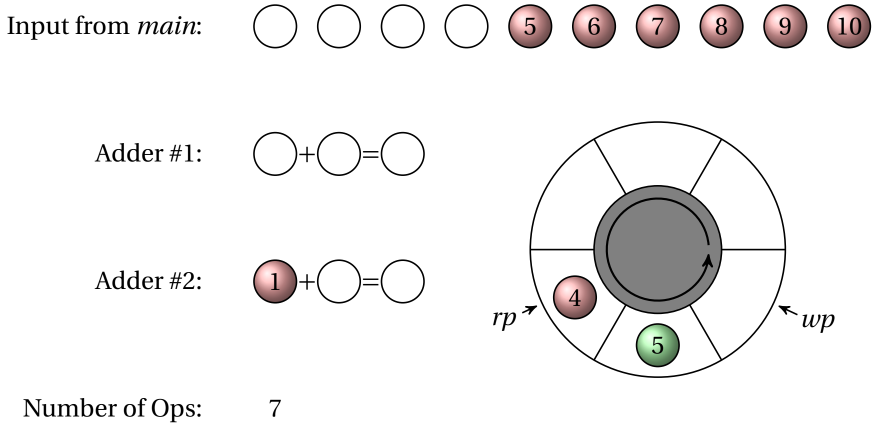
Adder #1: 2 + 3 = 5

Adder #2: 1 + ○ = ○

Number of Ops: 7



Adding Numbers — Illustrated



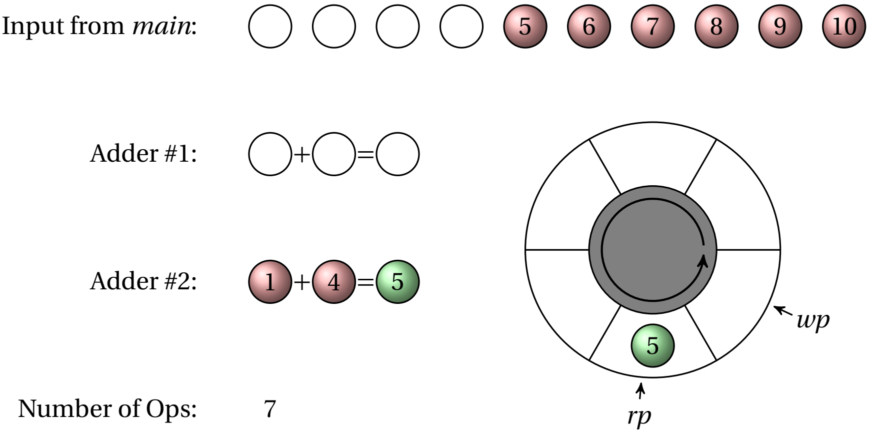
Input from *main*: 5 6 7 8 9 10

Adder #1: + =

Adder #2: 1 + =

Number of Ops: 7

Adding Numbers — Illustrated



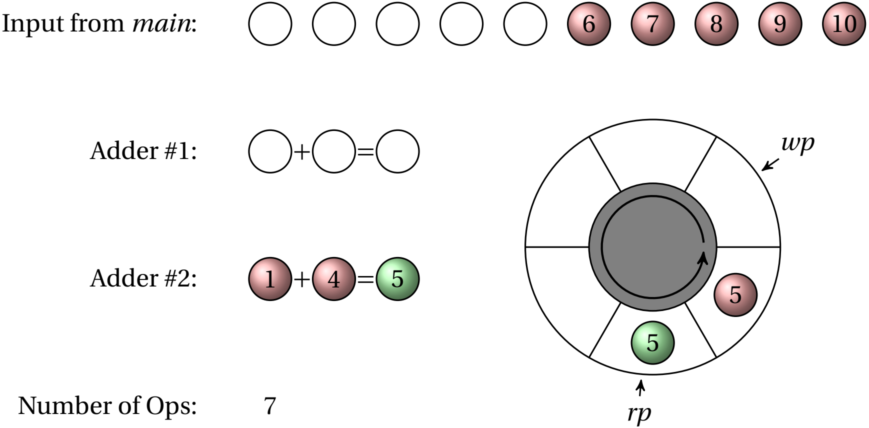
Input from *main*: 5 6 7 8 9 10

Adder #1: + =

Adder #2: 1 + 4 = 5

Number of Ops: 7

Adding Numbers — Illustrated



Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 7

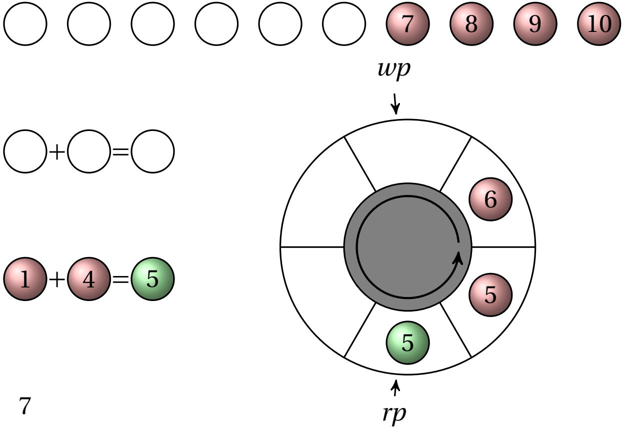
Adding Numbers — Illustrated

Input from *main*:

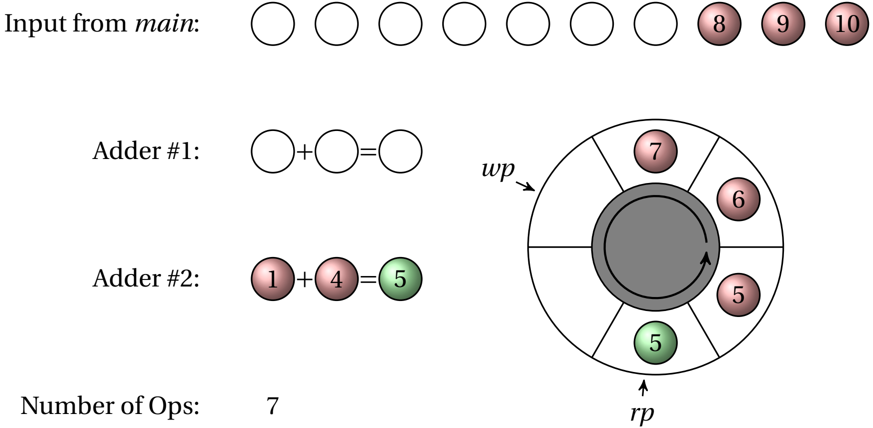
Adder #1:

Adder #2:

Number of Ops: 7



Adding Numbers — Illustrated



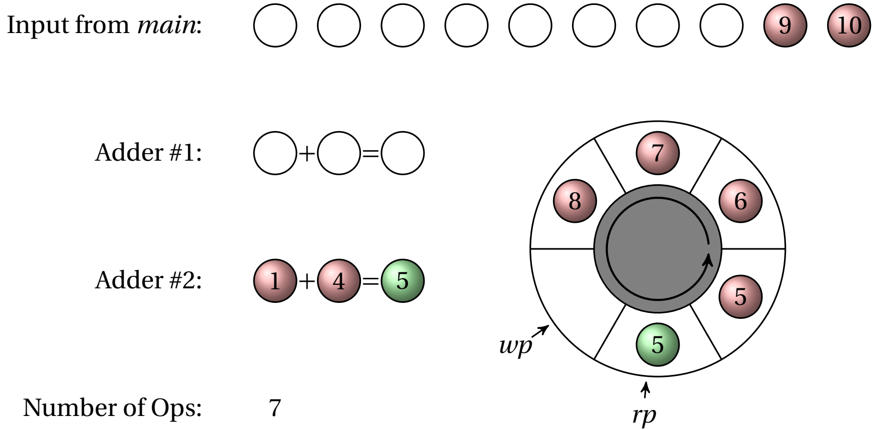
Input from *main*: 8 9 10

Adder #1: + =

Adder #2: 1 + 4 = 5

Number of Ops: 7

Adding Numbers — Illustrated



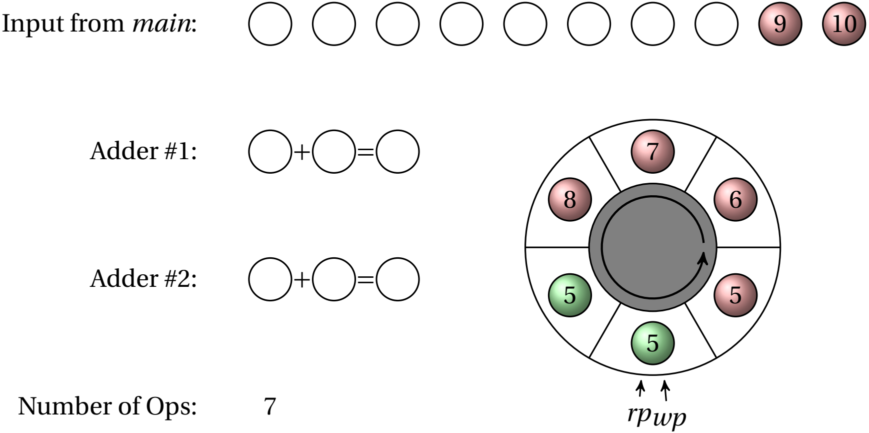
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 7

Adding Numbers — Illustrated



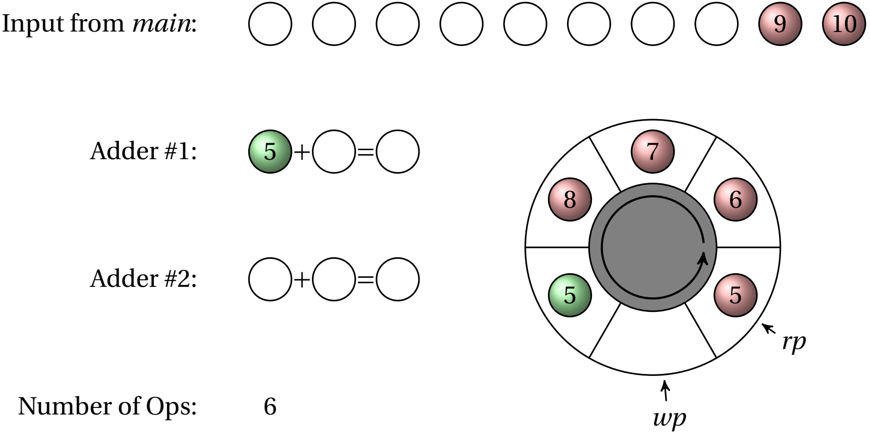
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 7

Adding Numbers — Illustrated



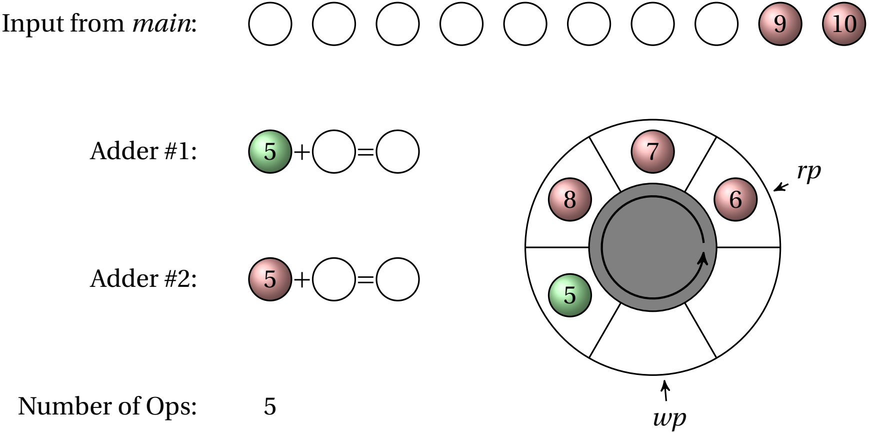
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 6

Adding Numbers — Illustrated



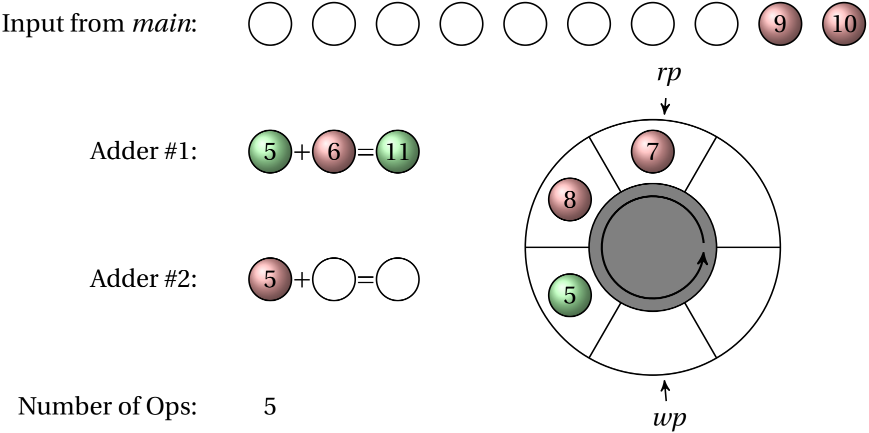
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 5

Adding Numbers — Illustrated



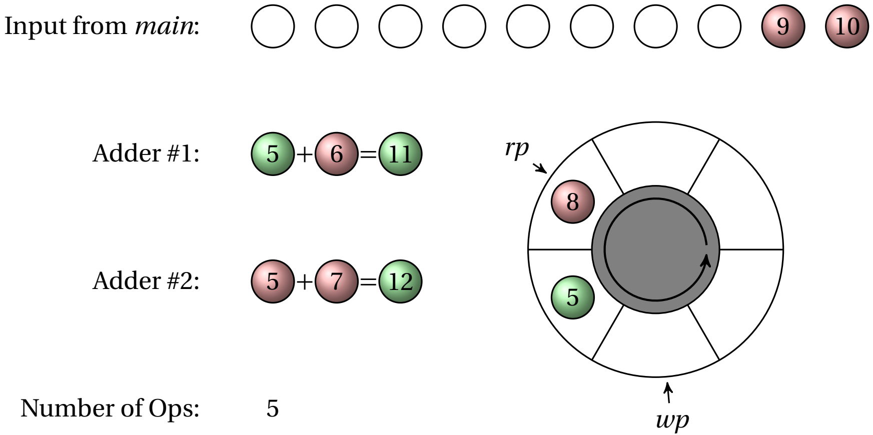
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 5

Adding Numbers — Illustrated



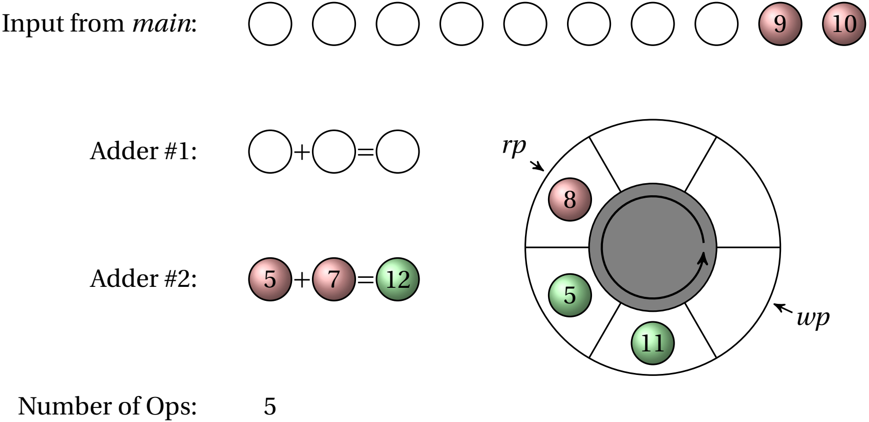
Input from *main*:

Adder #1: 5 + 6 = 11

Adder #2: 5 + 7 = 12

Number of Ops: 5

Adding Numbers — Illustrated



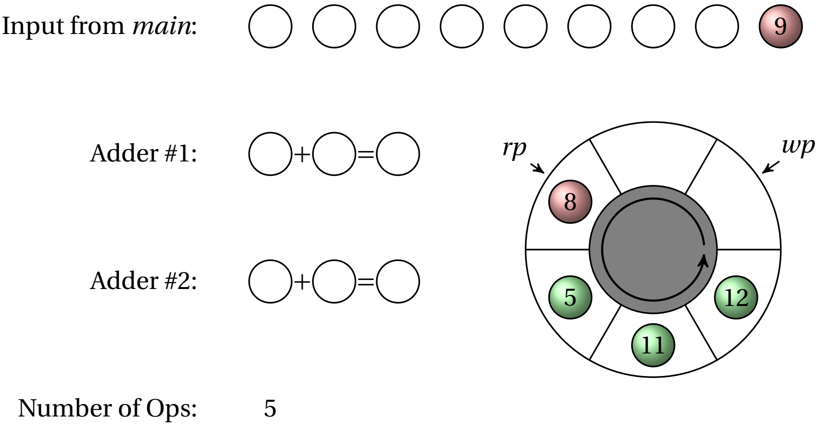
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 5

Adding Numbers — Illustrated



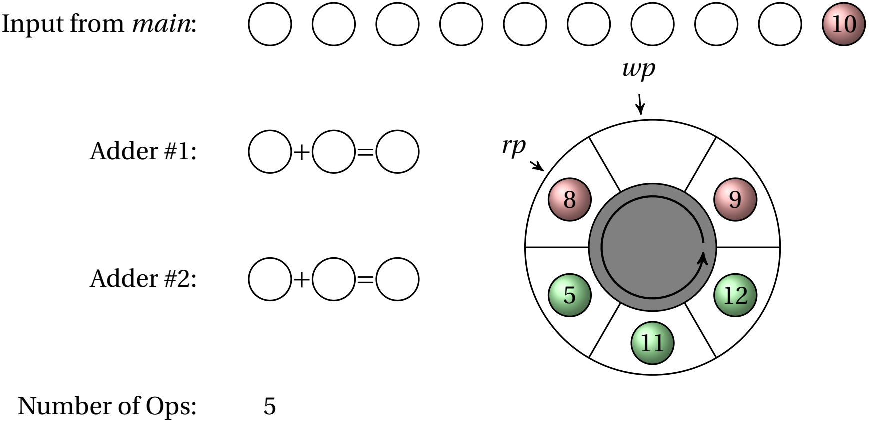
Input from *main*:

Adder #1: ○+○=○

Adder #2: ○+○=○

Number of Ops: 5

Adding Numbers — Illustrated



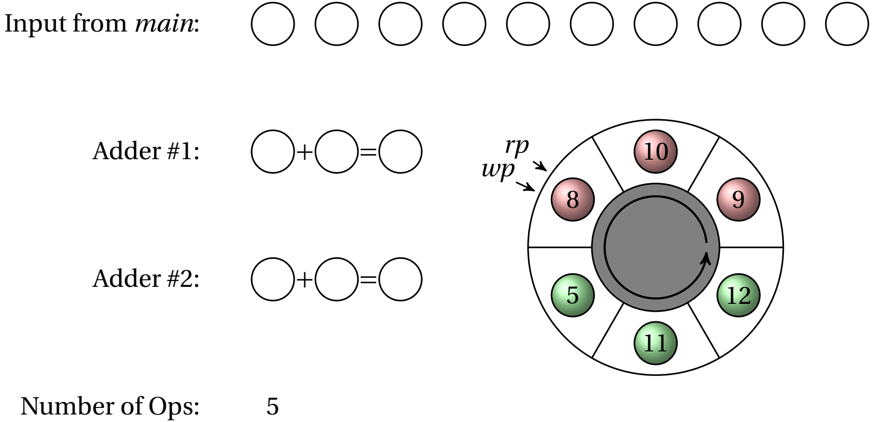
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 5

Adding Numbers — Illustrated



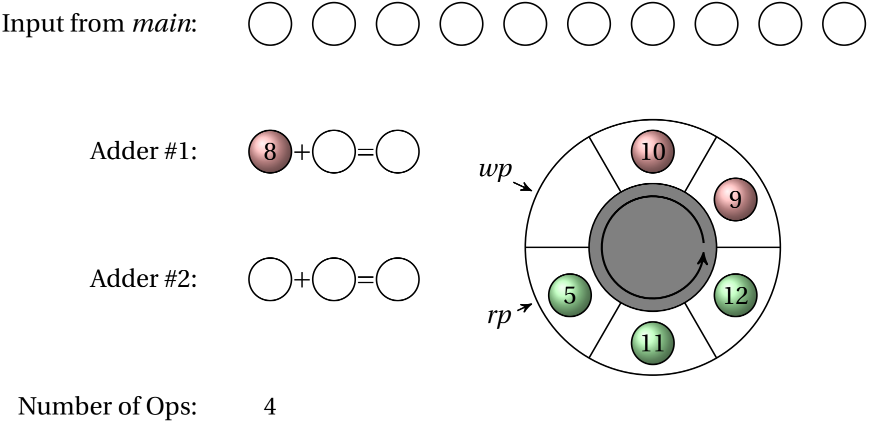
Input from *main*:

Adder #1: ○+○=○

Adder #2: ○+○=○

Number of Ops: 5

Adding Numbers — Illustrated



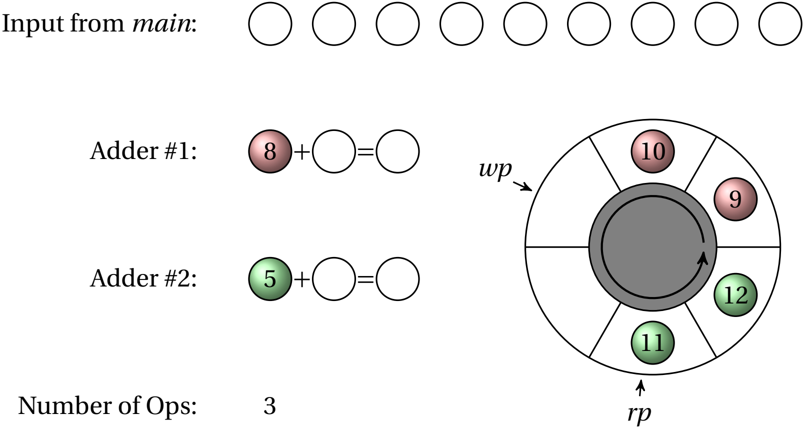
Input from *main*:

Adder #1: 8 + =

Adder #2: + =

Number of Ops: 4

Adding Numbers — Illustrated



Input from *main*:

Adder #1: 8 + =

Adder #2: 5 + =

Number of Ops: 3

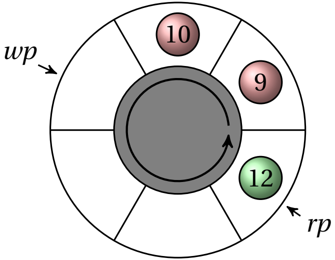
Adding Numbers — Illustrated

Input from *main*: ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

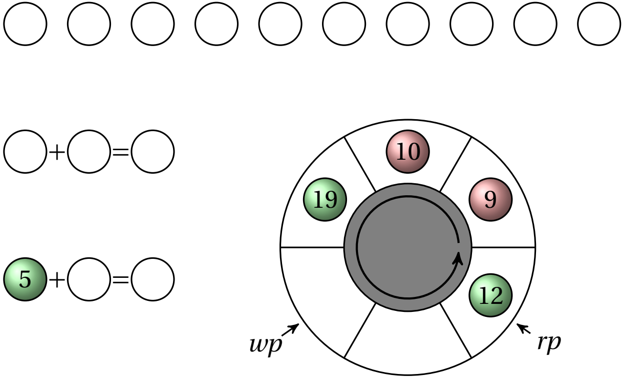
Adder #1: 8 + 11 = 19

Adder #2: 5 + ○ = ○

Number of Ops: 3



Adding Numbers — Illustrated



Input from *main*: ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

Adder #1: ○+○=○

Adder #2: 5+○=○

Number of Ops: 3

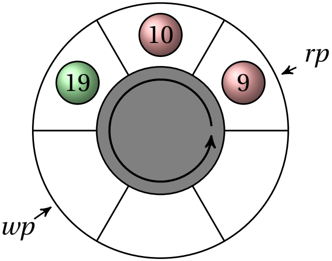
Adding Numbers — Illustrated

Input from *main*: ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

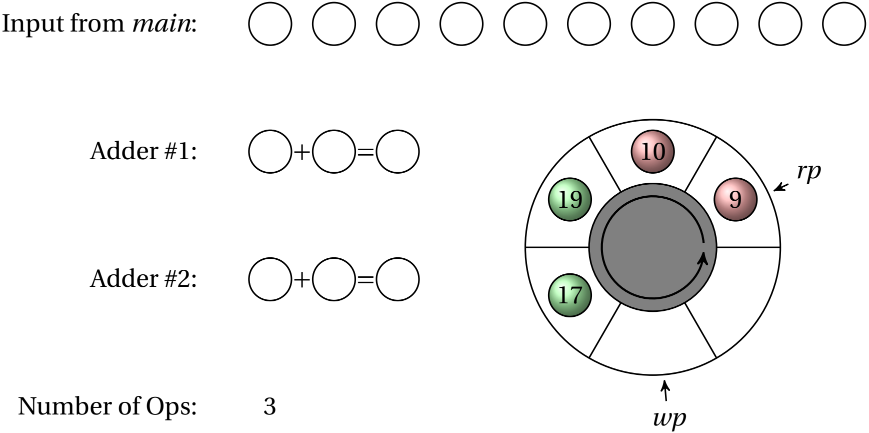
Adder #1: ○+○=○

Adder #2: 5+12=17

Number of Ops: 3



Adding Numbers — Illustrated



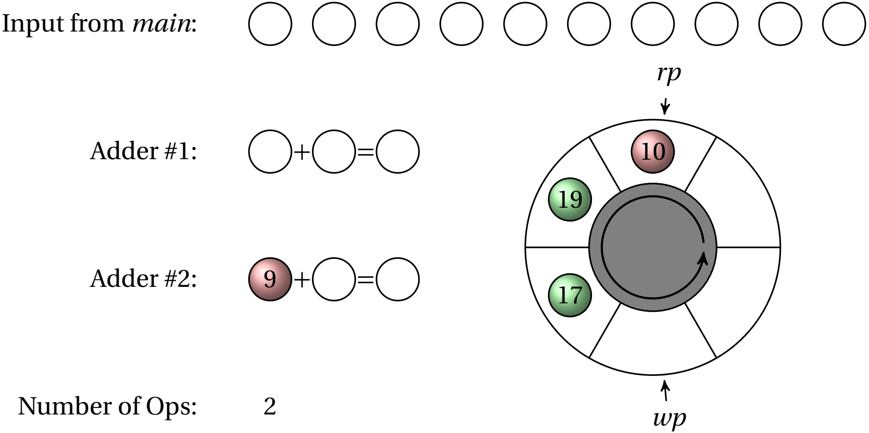
Input from *main*: ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

Adder #1: ○+○=○

Adder #2: ○+○=○

Number of Ops: 3

Adding Numbers — Illustrated



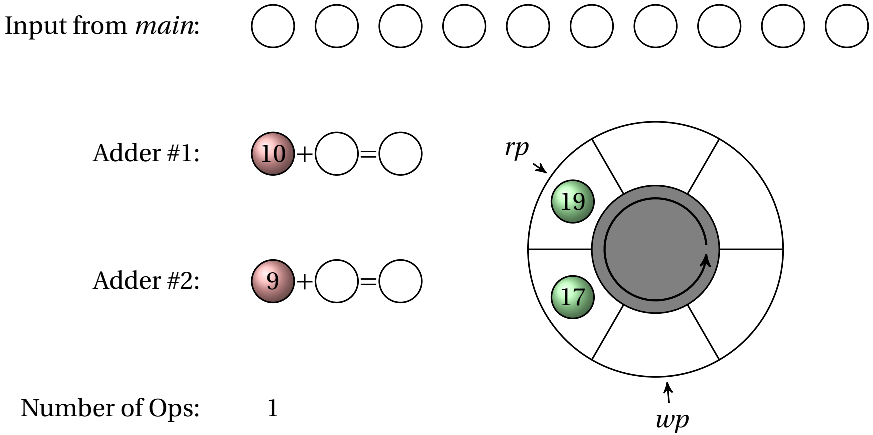
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 2

Adding Numbers — Illustrated



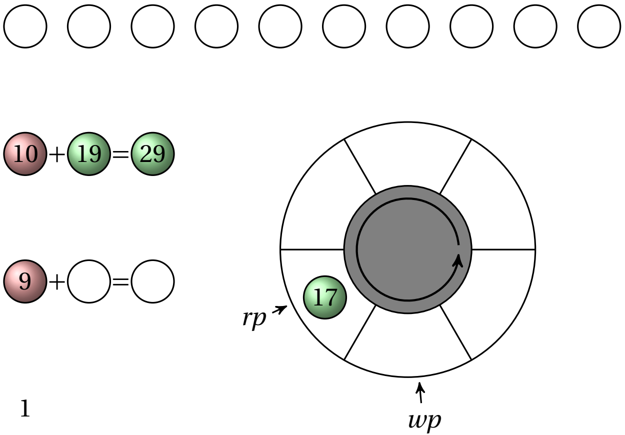
Input from *main*:

Adder #1: 10 + =

Adder #2: 9 + =

Number of Ops: 1

Adding Numbers — Illustrated



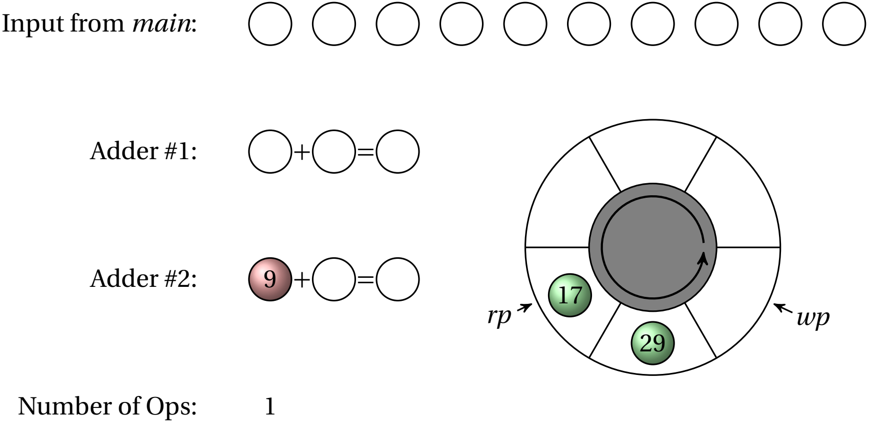
Input from *main*:

Adder #1: 10 + 19 = 29

Adder #2: 9 + =

Number of Ops: 1

Adding Numbers — Illustrated



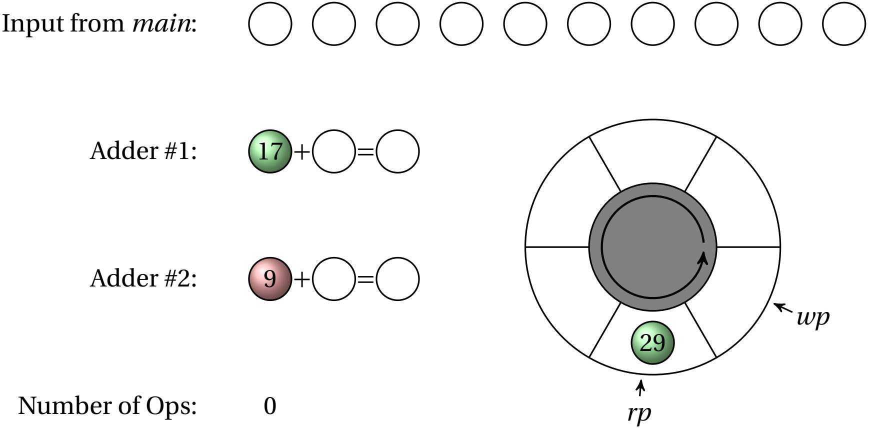
Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 1

Adding Numbers — Illustrated



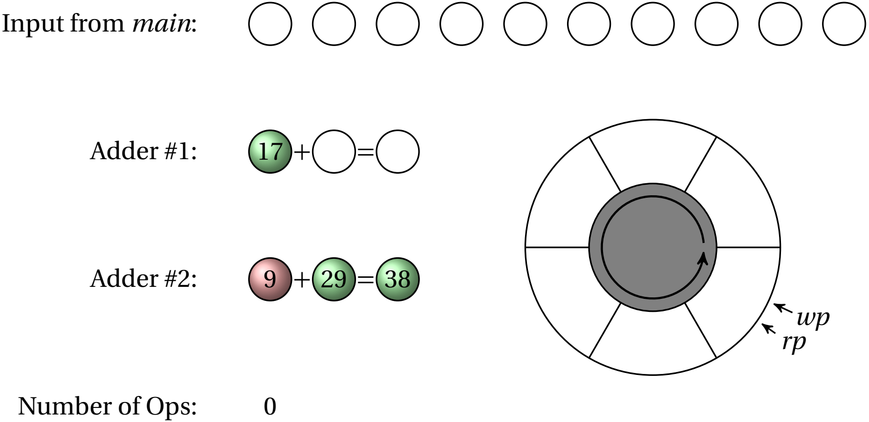
Input from *main*:

Adder #1: + =

Adder #2: + =

Number of Ops: 0

Adding Numbers — Illustrated



Input from *main*:

Adder #1: 17 + =

Adder #2: 9 + 29 = 38

Number of Ops: 0

wp

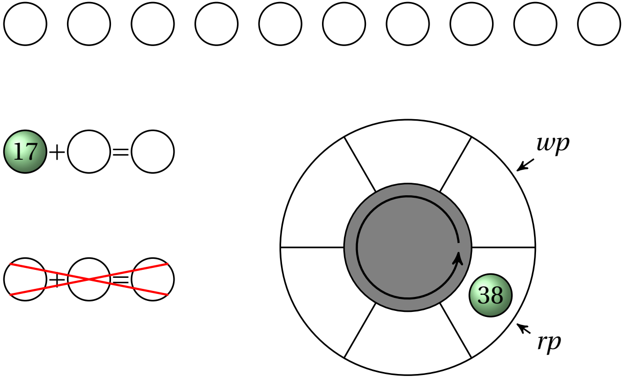
rp

Adding Numbers — Illustrated

Input from *main*:

Adder #1:

Adder #2:



Number of Ops: 0

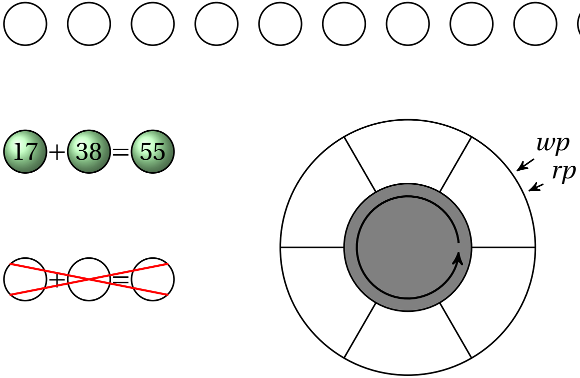
Adding Numbers — Illustrated

Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 0



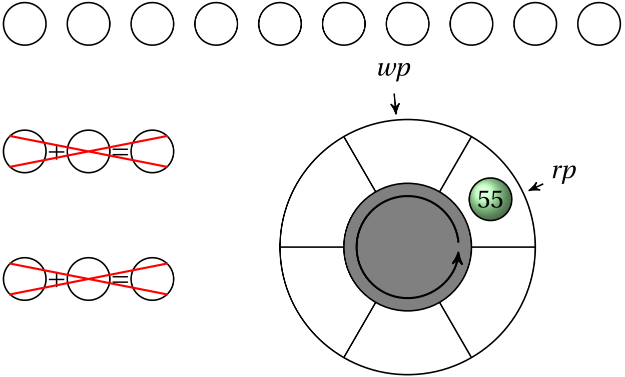
Adding Numbers — Illustrated

Input from *main*:

Adder #1:

Adder #2:

Number of Ops: 0



The *Buffer* Class

```
public class Buffer {
    private Semaphore free, elem;
    private long[] a;
    private int rp, wp;
    private int nOp;
    private Object nOpLock = new Object();
    public Buffer(int nOp, int size) {
        this.nOp = nOp;
        a = new long[size];
        rp = wp = 0;
        free = new Semaphore(size);
        elem = new Semaphore(0);
    }
    public Semaphore getFree() { return free; }
    public Semaphore getElem() { return elem; }                                  // ...
```

The *Buffer* Class (2)

```
    public boolean decNumOp() {
        synchronized ( nOpLock ) { return --nOp >= 0; }
    }
    public long deq() {
        synchronized ( a ) {
            long x = a[rp]; rp = (rp+1)%a.length; return x;
        }
    }
    public void enq(long x) {
        synchronized ( a ) { a[wp] = x; wp = (wp+1)%a.length; }
    }
}
```

The *Adder* Task

```java
public class Adder implements Runnable {
    private Buffer b;
    public Adder(Buffer b) { this.b = b; }
    @Override public void run() {
        try {
            while ( b.decNumOp() ) {
                b.getElem().acquire();
                long x = b.deq();
                b.getFree().release();
                b.getElem().acquire();
                long y = b.deq();
                b.enq(x+y);
                b.getElem().release();
            }
        } catch ( InterruptedException ex ) { } ...
```

Creating and Starting Adder Threads

```java
public static void main(String[] ss) throws InterruptedException {
    final int NT = 4;
    final int NE = 10000;

    Thread[] ts = new Thread[NT];
    Buffer b = new Buffer(NE-1, 32);
    Adder adder = new Adder(b);

    for ( int i = 0; i < NT; ++i ) {
        ts[i] = new Thread(adder);
        ts[i].start();
    }                                                        // ...
```

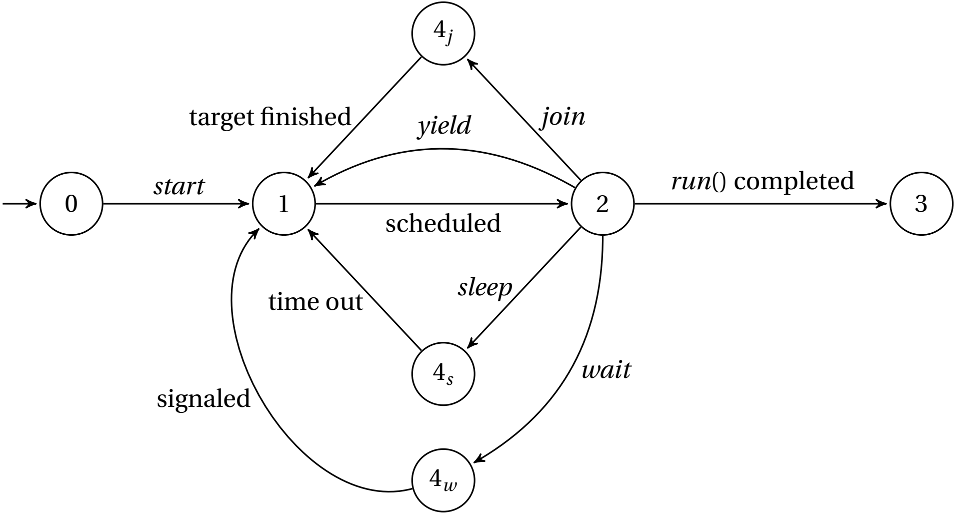
Feeding Elements and Waiting for the Result

```
    for ( int n = 1; n <= NE; ++n ) {
        b.getFree().acquire();
        b.enq(n);
        b.getElem().release();
    }

    for ( int i = 0; i < NT; ++i )
        ts[i].join();

    b.getElem().acquire();
    System.out.println(b.deq());
    b.getFree().release();
}
```

Summary of Thread States



0 — new

1 — ready

2 — running

3 — finished

4 — blocked