Threads in Java

Threads

- Difference between a process and a thread:
  - 2 processes have separate memory space.
  - 2 threads **share** the same memory space. **This means common access to the same variables!**

Two major problems with multithreading

- To know when a thread may or may not access certain data in memory.
  - Data integrity problems
  - Possible incorrect execution
- To know when the necessary data is available for access.
  - Performance issues.
  - Optimal use of resources

Make threads in Java-Example

```
public class Producer implements Runnable {
    public void run(){...}
    }
...
}
```

Make Threads in Java-Example

```
public class ThreadManager
{
    public void foo(){
     SimpleBuffer sharedLocation=new SimpleBuffer();
     //create the objects
    Consumer c=new Consumer(sharedLocation);
    Producer p=new Producer(sharedLocation);
    //Assign the objects to threads
    Thread t1=new Thread(c);
    Thread t2=new Thread(p);
    ...
    ExecutorService es=Executors.newCachedThreadPool();
    es.execute(t1);
    ...
    }
}
```

Consumer-Producer Problem v.1.

- Consumer and Producer share a common variable.
- Producer increases the value of the variable X times.
- Consumer decreases the value of the variable X times.
- Will they agree on the variable's value when they are finished?

Simple Buffer

```
public class SimpleBuffer{

int value=0;
public void add(int x){
      value=value+x;
      return value;
}
public void subtract(int x){
      value=value-x;
      return value;
}
public int readValue(){
return value;
}
public String sayHello(){
return “Hello world!”;
}
```

Producer

```
public class Producer implements Runnable {
	SimpleBuffer q;

	public Producer(SimpleBuffer q) {
	 this.q = q;
	}

	@Override
	public void run() {
		produce();
	}

	public void produce() throws InterruptedException {
	 for (int i = 0; i < Constants.ROUNDS; i++) {// ROUNDS is 10^7
		System.out.print("");
		q.add(1);
	 }
	 System.out.println("P" + q.readValue());
	}
}
```

Consumer

```
public class Consumer implements Runnable {
	SimpleBuffer q;

	public Consumer(SimpleBuffer q) {
	 this.q = q;
	}

	@Override
	public void run() {
		consume();
	}
	public void consume() throws InterruptedException {
	 for (int i = 0; i < Constants.ROUNDS; i++) {
		System.out.print("");
		q.subtract(1);
	 }
	 System.out.println("C" + q.readValue());
	}
}
```

Conclusions from execution

Both threads access the same memory space simultaneously.

Producer
temp1 ← read buffer;
temp1 ← temp+x;
buffer ← store temp1;

Consumer
temp2 ← read buffer;
temp2 ← temp2-y;
buffer← store temp2;

Memory space
b=1, t1=1
b=1, t1=1+x, t2=1
b=1+x, t2=1-y
b=1-y

Wrong Execution!!

b=buffer
t1=temp1
t2=temp2

Solution

- **Synchronized**:As long as a thread owns the lock, no other thread can acquire the same lock. Other threads will be blocked waiting for the lock to be released. The lock is released when the method returns

Synchronized

- It is the shared memory space that has to be under synchronization

```
public class SimpleBuffer{

public synchronized void add(int value)  {
       buffer += value; //critical point
}

public synchronized subtract(int value){
      buffer-=value; //critical point
}

public String sayHello(){ // we do not need to synchronize this one
…
}
```

Synchronized blocks

- Synchronize only the critical blocks of code. More complicated but allows better performance

```
public  void add(int value){
       synchronized(this){
              buffer += value;
       }
       System.out.println("Hello!!");
      // other thread do not have to wait for the message to be printed to access
      //the buffer
}
```

Wait/Notify

- Synchronized ensures that two threads will not access the same address in memory simultaneously (Correct execution).
- Synchronized cannot provide information about when the data is available.
- Theoretically a program can function without wait/notify but we will not have optimal use of resources and therefore bad performance.

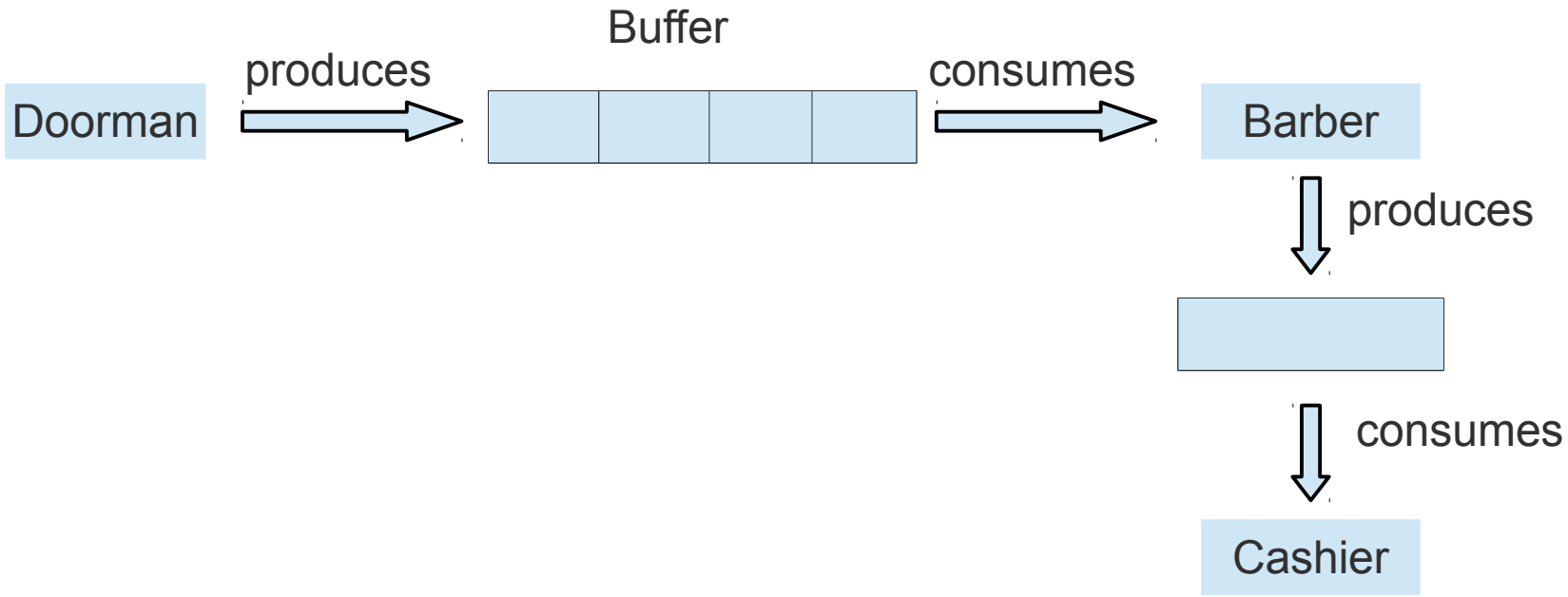
Consumer-Producer problem v.2

- Consumer and Producer share a common variable.
- Producer produces a number,updates the variable, sets a flag that there variable has been updated, and updates **its local sum**
- Consumer reads the value, sets a flag that the variable has been read and updates **its local sum.**
- Will consumer and producer agree on the total sum?

Synchronized Buffer

```
public class SynchBuffer
{
	int buffer=Integer.MIN_VALUE;
	boolean bufferFull=false;
public synchronized void setNumber(int x) throws InterruptedException{
	while(!bufferFull)
	wait();
	buffer=x;
	bufferFull=true;
	notifyAll();
}
public synchronized int getNumber() throws InterruptedException{
	int x=0;
	while(bufferFull)
	wait();
	x=buffer;
	bufferFull=false;
	notifyAll();
	return x;
}
}
```

Barber shop


Buffer
Doorman
produces
consumes
Barber
produces
consumes
Cashier