

java

 Copy code


```
class SharedResource {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) throws InterruptedException {
        while (available) { // wait if data is not consumed yet
            wait();
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify(); // wake up consumer
    }

    public synchronized int consume() throws InterruptedException {
        while (!available) { // wait if no data
            wait();
        }
        available = false;
        System.out.println("Consumed: " + data);
        notify(); // wake up producer
        return data;
    }
}
```

Producer thread:

java

 Copy code


```
class Producer extends Thread {
    private SharedResource resource;

    public Producer(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
                Thread.sleep(500); // simulate work
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Consumer thread:

java

 Copy code


```
class Consumer extends Thread {
    private SharedResource resource;

    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
                Thread.sleep(1000); // simulate work
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Running the example:


java

 Copy code

```
public class WaitNotifyExample {  
    public static void main(String[] args) {  
        SharedResource resource = new SharedResource();  
        Producer producer = new Producer(resource);  
        Consumer consumer = new Consumer(resource);  
  
        producer.start();  
        consumer.start();  
    }  
}
```

Sample Output

makefile

 Copy code

```
Produced: 1  
Consumed: 1  
Produced: 2  
Consumed: 2  
...
```

1. Lock and Object

- A **lock** belongs to an **object**, not a thread.
- A **thread** acquires/release a lock when entering/exiting a `synchronized` block on that object.
- `wait()` simply makes the **current thread pause** and **release the object's lock** it is holding.

2. Sleep

- `sleep()` is just a timer → "pause me for X ms" → belongs to `Thread`.
- `wait()` is about coordination on a shared **object monitor** → belongs to `Object`.

3. Context Switch

What is a Context Switch?

A **context switch** is when the CPU stops running one thread (or process) and starts running another.

Since a CPU core can only execute **one thread at a time**, the operating system (OS) scheduler rapidly switches between threads to give the illusion of parallelism (unless you have multiple CPU cores).

4. CPU-intensive and IO-intensive

Will running multiple threads on a single-core CPU be efficient?

Whether a single-core CPU can run multiple threads simultaneously efficiently depends on the type of thread and the nature of the task. Generally speaking, there are two types of threads:

1. **CPU-intensive** : CPU-intensive threads mainly perform calculations and logic processing, and require a large amount of CPU resources.
2. **IO-intensive** : IO-intensive threads mainly perform input and output operations, such as reading and writing files, network communication, etc., and need to wait for the response of IO devices without taking up too many CPU resources.

On a single-core CPU, only one thread can run at a time; other threads must wait for CPU time slices. If the thread is CPU-bound, running multiple threads simultaneously will result in frequent thread switching, increasing system overhead and reducing efficiency. If the thread is I/O-bound, running multiple threads simultaneously can utilize the CPU's idle time while waiting for I/O, improving efficiency.

Therefore, for a single-core CPU, if the task is CPU-intensive, then running many threads will affect efficiency; if the task is I/O-intensive, then running many threads will improve efficiency. Of course, the "many" here should be moderate and should not exceed the upper limit of the system's tolerance.

5. Prevent Deadlock

1. Lock ordering

Always acquire locks in a fixed **global order**.

```
java Copy code  
  
Object lockA = new Object();  
Object lockB = new Object();  
  
void safeMethod() {  
    Object first = lockA.hashCode() < lockB.hashCode() ? lockA : lockB;  
    Object second = (first == lockA) ? lockB : lockA;  
  
    synchronized (first) {  
        synchronized (second) {  
            // safe work  
        }  
    }  
}
```

2. Use `tryLock()` with timeout

Instead of blocking forever, attempt to acquire a lock with a timeout.

```
java Copy code  
  
Lock lockA = new ReentrantLock();  
Lock lockB = new ReentrantLock();  
  
if (lockA.tryLock(100, TimeUnit.MILLISECONDS)) {  
    try {  
        if (lockB.tryLock(100, TimeUnit.MILLISECONDS)) {  
            try {  
                // safe work  
            } finally {  
                lockB.unlock();  
            }  
        }  
    } finally {  
        lockA.unlock();  
    }  
}
```

👉 If the lock isn't acquired, you can retry or back off.