Programming language support for concurrency

Madhavan Mukund, S P Suresh

Programming Language Concepts Lecture 12, 20 February 2025

Race conditions

- Concurrent update of a shared variable can lead to data inconsistency
 - Race condition
- Control behaviour of threads to regulate concurrent updates
 - Critical sections sections of code where shared variables are updated
 - Mutual exclusion at most one thread at a time can be in a critical section
- We can construct protocols that guarantee mutual exclusion to critical sections
 - Watch out for starvation and deadlock
- These protocols cleverly use regular variables
 - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization

Madhavan Mukund/S P Suresh

Language support for concurrency

• The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive
- Language primitives like compare-and-swap (CAS) execute this atomically

- The fundamental issue preventing consistent concurrent updates of shared variables is test-and-set
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive
- Language primitives like compare-and-swap (CAS) execute this atomically
- We shall look at other structures semaphores and monitors

• Programming language support for mutual exclusion

- Programming language support for mutual exclusion
- Dijkstra's semaphores
 - Integer variable with atomic test-and-set operation

- Programming language support for mutual exclusion
- Dijkstra's semaphores
 - Integer variable with atomic test-and-set operation
- A semaphore S supports two atomic operations
 - P(s) from Dutch passeren, to pass
 - V(s) from Dutch vrygeven, to release

- Programming language support for mutual exclusion
- Dijkstra's semaphores
 - Integer variable with atomic test-and-set operation
- A semaphore S supports two atomic operations
 - P(s) from Dutch passeren, to pass
 - V(s) from Dutch vrygeven, to release

• P(S) atomically executes the following

if	(s > ())				
	decre	ement	S;			
els	e					
	wait	on S	to	become	positive;	

- Programming language support for mutual exclusion
- Dijkstra's semaphores
 - Integer variable with atomic test-and-set operation
- A semaphore S supports two atomic operations
 - P(s) from Dutch passeren, to pass
 - V(s) from Dutch vrygeven, to release

• P(S) atomically executes the following

if	(S >	0)				
	decr	ement	: S;			
els	е					
	wait	on S	5 to	become	<pre>positive;</pre>	

• V(S) atomically executes the following

Using semaphores

• Mutual exclusion using semaphores

Thread 1	Thread 2
 P(S); // Enter critical section	 P(S); // Enter critical section
<pre> // Leave critical section V(S);</pre>	<pre> // Leave critical section V(S);</pre>

Using semaphores

• Mutual exclusion using semaphores

Thread 1	Thread 2
 P(S); // Enter critical section	 P(S); // Enter critical section
<pre> // Leave critical section V(S);</pre>	<pre> // Leave critical section V(S);</pre>

- Semaphores guarantee
 - Mutual exclusion
 - Freedom from starvation
 - Freedom from deadlock

• Too low level

- Too low level
- No clear relationship between a semaphore and the critical region that it protects

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each P(S) has a matching V(S)

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each P(S) has a matching V(S)
- Can even execute V(S) without having done P(S)

Summary

- Test-and-set is at the heart of most race conditions
- Need a high level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors

• Attach synchronization control to the data that is being protected

- Attach synchronization control to the data that is being protected
- Monitors Per Brinch Hansen and CAR Hoare

- Attach synchronization control to the data that is being protected
- Monitors Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
 - Data definition to which access is restricted across threads
 - Collections of functions operating on this data — all are implicitly mutually exclusive

```
monitor bank account {
    double accounts[100]:
    boolean transfer (double amount,
                     int source, int target) {
        if (accounts[source] < amount) {</pre>
            return false;
        accounts[source] -= amount;
        accounts[target] += amount:
        return true:
    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for (int i = 0; i < 100; i++) {</pre>
            balance += accounts[i];
        return balance:
```

- Attach synchronization control to the data that is being protected
- Monitors Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
 - Data definition to which access is restricted across threads
 - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion if one function is active, any other function will have to wait for it to finish

```
monitor bank account {
    double accounts[100]:
    boolean transfer (double amount,
                     int source, int target) {
        if (accounts[source] < amount) {</pre>
            return false;
        accounts[source] -= amount;
        accounts[target] += amount:
        return true:
    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for (int i = 0; i < 100; i++) {</pre>
            balance += accounts[i];
        return balance:
```

Monitors: external queue

 Monitor ensures transfer and audit are mutually exclusive

```
monitor bank account {
    double accounts[100];
    boolean transfer (double amount,
                    int source, int target) {
        if (accounts[source] < amount) {</pre>
            return false;
        accounts[source] -= amount;
        accounts[target] += amount;
        return true;
    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for (int i = 0; i < 100; i++) {</pre>
            balance += accounts[i]:
        return balance;
```

Monitors: external queue

- Monitor ensures transfer and audit are mutually exclusive
- If Thread 1 is executing transfer and Thread 2 invokes audit, it must wait

```
monitor bank account {
    double accounts[100]:
    boolean transfer (double amount,
                    int source, int target) {
        if (accounts[source] < amount) {</pre>
            return false;
        accounts[source] -= amount;
        accounts[target] += amount;
        return true;
    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for (int i = 0; i < 100; i++) {</pre>
            balance += accounts[i];
        return balance:
```

Monitors: external queue

- Monitor ensures transfer and audit are mutually exclusive
- If Thread 1 is executing transfer and Thread 2 invokes audit, it must wait
- Implicit queue associated with each monitor
 - Contains all processes waiting for access
 - In practice, this may be just a set, not a queue

```
monitor bank account {
    double accounts[100]:
    boolean transfer (double amount,
                     int source, int target) {
        if (accounts[source] < amount) {</pre>
            return false;
        accounts[source] -= amount;
        accounts[target] += amount;
        return true:
    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for (int i = 0; i < 100; i++) {</pre>
            balance += accounts[i];
        return balance:
```

• Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

• Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

This should always succeed if accounts[i] > 500

• Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if accounts[i] > 500
- If these calls are reordered and accounts[j] < 400 initially, this will fail

• Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if accounts[i] > 500
- If these calls are reordered and accounts[j] < 400 initially, this will fail
- A possible fix let an account wait for pending inflows

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

```
Monitors — wait()
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

• All other processes are blocked out while this process waits!

```
Monitors — wait()
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor

```
Monitors — wait()
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
```
Monitors — wait()
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait

```
Monitors — wait()
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount) {
        // wait for another transaction to transfer money into accounts[source]
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}</pre>
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait
- Dual operation to **notify** and wake up suspended processes

Madhavan Mukund/S P Suresh

Language support for concurrency

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount){ wait(); }
    accounts[source] -= amount;
    accounts[target] += amount;
    notify();
    return true;
}</pre>
```

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount){ wait(); }
    accounts[source] -= amount;
    accounts[target] += amount;
    notify();
    return true;
}</pre>
```

• What happens when a process executes notify()?

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount){ wait(); }
    accounts[source] -= amount;
    accounts[target] += amount;
    notify();
    return true;
}</pre>
```

- What happens when a process executes notify()?
- Signal and exit notifying process immediately exits the monitor
 - notify() must be the last instruction

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount){ wait(); }
    accounts[source] -= amount;
    accounts[target] += amount;
    notify();
    return true;
}</pre>
```

- What happens when a process executes notify()?
- Signal and exit notifying process immediately exits the monitor
 - notify() must be the last instruction
- Signal and wait notifying process swaps roles and goes into the internal queue of the monitor

```
boolean transfer (double amount, int source, int target) {
    if (accounts[source] < amount){ wait(); }
    accounts[source] -= amount;
    accounts[target] += amount;
    notify();
    return true;
}</pre>
```

- What happens when a process executes notify()?
- Signal and exit notifying process immediately exits the monitor
 - notify() must be the last instruction
- Signal and wait notifying process swaps roles and goes into the internal queue of the monitor
- Signal and continue notifying process keeps control till it completes and then one of the notified processes steps in

Madhavan Mukund/S P Suresh

Language support for concurrency

Monitors — wait() and notify()

- Should check the wait() condition again on wake up
 - Change of state may not be sufficient to continue e.g., not enough inflow into the account to allow transfer

Monitors — wait() and notify()

- Should check the wait() condition again on wake up
 - Change of state may not be sufficient to continue e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
 - At wake-up, the state was fine, but it has changed again due to some other concurrent action

Monitors — wait() and notify()

- Should check the wait() condition again on wake up
 - Change of state may not be sufficient to continue e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
 - At wake-up, the state was fine, but it has changed again due to some other concurrent action
- wait() should be in a while, not in an if

```
boolean transfer (double amount, int source, int target) {
   while (accounts[source] < amount){ wait(); }
   accounts[source] -= amount;
   accounts[target] += amount;
   notify();
   return true;
}</pre>
```

Condition variables

 After transfer, notify() is only useful for threads waiting for target account of transfer to change state

Condition variables

- After transfer, notify() is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue

Condition variables

- After transfer, notify() is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue
- Monitor can have **condition variables** to describe internal queues

```
monitor bank account {
    double accounts[100];
    queue q[100]; // one internal queue
                    // for each account
    boolean transfer (double amount,
                    int source, int target) {
        while (accounts[source] < amount) {</pre>
            q[source].wait(); // wait in the queue
                                // associated with
                                      source
        accounts[source] -= amount;
        accounts[target] += amount:
        q[target].notify(); // notify the queue
                            // associated with target
        return true:
    // compute the balance across all accounts
    double audit(){ ... }
```

Summary

- Concurrent programming with atomic test-and-set primitives is error prone
- Monitors are like abstract datatypes for concurrent programming
 - Encapsulate data and methods to manipulate data
 - Methods are implicitly atomic, regulate concurrent access
 - Each object has an implicit external queue of processes waiting to execute a method
- wait() and notify() allow more flexible operation
- Can have multiple internal queues controlled by condition variables

 Monitors incorporated within existing class definitions

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i];
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0:
        for (int i = 0: i < 100: i++)
            balance += accounts[i]:
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically
- Each object has a lock
 - To execute a synchronized method, thread must acquire lock
 - Thread gives up lock when the method exits
 - Only one thread can have the lock at any time

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i];
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically
- Each object has a lock
 - To execute a synchronized method, thread must acquire lock
 - Thread gives up lock when the method exits
 - Only one thread can have the lock at any time
- Wait for lock in external queue

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i];
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

• wait() and notify() to suspend and
resume

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i];
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

- wait() and notify() to suspend and
 resume
- Wait single internal queue

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i]:
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

- wait() and notify() to suspend and
 resume
- Wait single internal queue
- Notify
 - notify() signals one (arbitrary) waiting process
 - notifyAll() signals all waiting processes
 - Java uses signal and continue

```
public class bank account {
    double accounts[100]:
    public synchronized boolean transfer(
      double amount, int source, int target) {
        while (accounts[source] < amount){wait():}</pre>
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true:
    public synchronized double audit() {
        double balance = 0.0;
        for (int i = 0: i < 100: i++)
            balance += accounts[i]:
        return balance:
    public double current balance(int i) {
        return accounts[i]; // not synchronized!
```

• Use object locks to synchronize arbitrary blocks of code

```
public class XYZ {
    Object o = new Object();
    public int f() {
        ...
        synchronized(o){ ... }
    }
    public double g() {
        ...
        synchronized(o){ ... }
    }
}
```

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o

```
public class XYZ {
    Object o = new Object();
    public int f() {
        ...
        synchronized(o){ ... }
    }
    public double g() {
        ...
        synchronized(o){ ... }
    }
}
```

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o
- Each object has its own internal queue

```
Object o = new Object();
public int f() {
     ...
    synchronized(o) {
         ...
        o.wait(); // Wait in gueue attached to "o"
public double g() {
    synchronized(o) {
         . . .
        o.notifyAll(); // Wake up gueue attached to "o"
         ...
```

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized

<pre>public double h() { synchronized(this){</pre>	
}	

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized
- "Anonymous" wait(), notify(), notifyAll() abbreviate this.wait(), this.notify(), this.notifyAll()

publ	<pre>lic double h() { synchronized(this){</pre>
}	·

• Actually, wait() can be "interrupted" by an InterruptedException

- Actually, wait() can be "interrupted" by an InterruptedException
- Should write

```
try{
    wait();
}
catch (InterruptedException e) {
    ...
};
```

- Actually, wait() can be "interrupted" by an InterruptedException
- Should write

```
try{
    wait();
}
catch (InterruptedException e) {
    ...
};
```

- Error to use wait(), notify(), notifyAll() outside synchronized method
 - IllegalMonitorStateException

- Actually, wait() can be "interrupted" by an InterruptedException
- Should write

```
try{
    wait();
}
catch (InterruptedException e) {
    ...
};
```

- Error to use wait(), notify(), notifyAll() outside synchronized method
 - IllegalMonitorStateException
- Likewise, use o.wait(), o.notify(), o.notifyAll() only in block synchronized on o

Separate ReentrantLock class

```
public class Bank {
    private Lock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, int
         amount) {
        bankLock.lock();
        trv {
            accounts[from] -= amount;
            accounts[to] += amount;
        } finally {
            bankLock.unlock();
```

- Separate ReentrantLock class
- Similar to a semaphore
 - lock() is like P(S)
 - unlock() is like V(S)

```
public class Bank {
    private Lock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, int
         amount) {
        bankLock.lock();
        trv {
            accounts[from] -= amount;
            accounts[to] += amount;
        } finally {
            bankLock.unlock();
```

- Separate ReentrantLock class
- Similar to a semaphore
 - lock() is like P(S)
 - unlock() is like V(S)
- Always unlock() in finally avoid abort while holding lock

```
public class Bank {
    private Lock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, int
         amount) {
        bankLock.lock();
        trv {
            accounts[from] -= amount;
            accounts[to] += amount;
        } finally {
            bankLock.unlock();
```

- Separate ReentrantLock class
- Similar to a semaphore
 - lock() is like P(S)
 - unlock() is like V(S)
- Always unlock() in finally avoid abort while holding lock
- Why reentrant?
 - Thread holding lock can reacquire it
 - transfer() may call getBalance() that also locks bankLock
 - Hold count increases with lock(), decreases with unlock()
 - Lock is available if hold count is 0

```
public class Bank {
    private Lock bankLock = new ReentrantLock():
    public void transfer(int from, int to, int
         amount) {
        bankLock.lock();
        trv {
            accounts[from] -= amount;
            accounts[to] += amount;
        } finally {
            bankLock.unlock();
```

Summary

- Every object in Java implicitly has a lock
- Methods tagged synchronized are executed atomically
 - Implicitly acquire and release the object's lock
- Associated condition variable, single internal queue
 - wait(),notify(),notifyAll()
- Can synchronize an arbitrary block of code using an object
 - sycnchronized(o) { ... }
 - o.wait(),o.notify(),o.notifyAll()
- Reentrant locks work like semaphores

Creating threads in Java

- Have a class extend Thread
- Define a function **run()** where execution can begin in parallel
- Invoking p[i].start() initiates p[i].run() in a separate thread
 - Directly calling p[i].run() does not execute in separate thread!
- sleep(t) suspends thread for t
 milliseconds
 - Static function use Thread.sleep() if current class does not extend Thread
 - Throws InterruptedException

```
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
    public void run() {
        for (int j = 0; j < 100; j++) {
            System.out.println("My id is "+id);
            trv {
                sleep(1000); // Sleep for 1000 ms
            } catch(InterruptedException e) {}
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i):
            p[i].start(); // Start p[i].run()
                            // in concurrent thread
```
Creating threads in Java

- Have a class extend Thread
- Define a function **run()** where execution can begin in parallel
- Invoking p[i].start() initiates p[i].run() in a separate thread
 - Directly calling p[i].run() does not execute in separate thread!
- sleep(t) suspends thread for t milliseconds
 - Static function use Thread.sleep() if current class does not extend Thread
 - Throws InterruptedException

Typical output

Му	id	is	0	
Му	id	is	3	
Му	id	is	2	
Му	id	is	1	
Му	id	is	4	
Му	id	is	0	
Му	id	is	2	
Му	id	is	3	
Му	id	is	4	
Му	id	is	1	
Му	id	is	0	
Му	id	is	3	
Му	id	is	1	
Му	id	is	2	
Му	id	is	4	
Му	id	is	0	

Java threads ...

- Cannot always extend Thread
 - Single inheritance
- Instead, implement Runnable
- To use Runnable class, explicitly create a Thread and start() it

```
public class Parallel implements Runnable {
    // only the line above has changed
    private int id;
    public Parallel(int i){ ... } // Constructor
    public void run(){ ... }
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        Thread t[] = new Thread[5]:
        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i);
            t[i] = new Thread(p[i]):
            // Make a thread t[i] from p[i]
            t[i].start(); // Start off p[i].run()
                            // Note: t[i].start().
                            // not p[i].start()
```

A thread can be in six states

• New: Created but not start()ed.

- New: Created but not start()ed.
- Runnable: start()ed and ready to be scheduled.
 - Need not be actually "running"
 - No guarantee made about how scheduling is done
 - Most Java implementations use time-slicing

- New: Created but not start()ed.
- Runnable: start()ed and ready to be scheduled.
 - Need not be actually "running"
 - No guarantee made about how scheduling is done
 - Most Java implementations use time-slicing
- Not available to run
 - Blocked waiting for a lock, unblocked when lock is granted
 - Waiting suspended by wait(), unblocked by notify() or notfifyAll()
 - Timed wait within sleep(..), released when sleep timer expires

- New: Created but not start()ed.
- Runnable: start()ed and ready to be scheduled.
 - Need not be actually "running"
 - No guarantee made about how scheduling is done
 - Most Java implementations use time-slicing
- Not available to run
 - Blocked waiting for a lock, unblocked when lock is granted
 - Waiting suspended by wait(), unblocked by notify() or notfifyAll()
 - Timed wait within sleep(..), released when sleep timer expires
- Dead: thread terminates

A thread can be in six states — thread status via t.getState()

- New: Created but not start()ed.
- Runnable: start()ed and ready to be scheduled.
 - Need not be actually "running"
 - No guarantee made about how scheduling is done
 - Most Java implementations use time-slicing
- Not available to run
 - Blocked waiting for a lock, unblocked when lock is granted
 - Waiting suspended by wait(), unblocked by notify() or notfifyAll()
 - Timed wait within sleep(..), released when sleep timer expires
- Dead: thread terminates

Interrupts

- One thread can interrupt another using interrupt()
 - p[i].interrupt(); interrupts thread
 p[i]

Interrupts

- One thread can interrupt another using interrupt()
 - p[i].interrupt(); interrupts thread
 p[i]
- Raises InterruptedException within wait(), sleep()

Interrupts

- One thread can interrupt another using interrupt()
 - p[i].interrupt(); interrupts thread
 p[i]
- Raises InterruptedException within wait(), sleep()
- No exception raised if thread is running!
 - interrupt() sets a status flag
 - interrupted() checks interrupt status and clears the flag
- Detecting an interrupt while running or waiting

```
public void run() {
    trv {
        i = 0:
        while (!interrupted() & j < 100) {</pre>
            System.out.println("My id is "+
                 id):
            sleep(1000); // Sleep for
                 1000 ms
            i++:
    } catch(InterruptedException e){}
```

- Check a thread's interrupt status
 - Use t.isInterrupted() to check status of t's interrupt flag
 - Does not clear flag

- Check a thread's interrupt status
 - Use t.isInterrupted() to check status of t's interrupt flag
 - Does not clear flag
- Can give up running status
 - yield() gives up active state to another thread
 - Static method in Thread

- Check a thread's interrupt status
 - Use t.isInterrupted() to check status of t's interrupt flag
 - Does **not** clear flag
- Can give up running status
 - yield() gives up active state to another thread
 - Static method in Thread
 - Normally, scheduling of threads is handled by OS preemptive
 - Some mobile platforms use cooperative scheduling thread loses control only if it yields

- Check a thread's interrupt status
 - Use t.isInterrupted() to check status of t's interrupt flag
 - Does not clear flag
- Can give up running status
 - yield() gives up active state to another thread
 - Static method in Thread
 - Normally, scheduling of threads is handled by OS preemptive
 - * Some mobile platforms use cooperative scheduling thread loses control only if it yields
- Waiting for other threads
 - t.join() waits for t to terminate

Summary

- To run in parallel, need to extend Thread or implement Runnable
 - When implmenting Runnable, first create a Thread from Runnable object
- t.start() invokes method run() in parallel
- Threads can become inactive for different reasons
 - Block waiting for a lock
 - Wait in internal queue for a condition to be notified
 - Wait for a sleep timer to elapse
- Threads can be interrupted
 - Be careful to check both interrupted status and handle InterruptException
- Can yield control, or wait for another thread to terminate

Madhavan Mukund/S P Suresh