Threads

Hyang-Won Lee
Department of Computer Science & Engineering
Konkuk University
leehw@konkuk.ac.kr
https://sites.google.com/view/leehwko/

Web Servers

- Program runs much faster than network can supply input
  - network speed: 10Mbps
  - CPU clock speed: 3GHz = $3*10^9$ cycles/s
    - e.g., 3 cycles needed to process a bit, i.e., 3cycles/bit ⇒ 1Gbps
- Handle hundreds of requests simultaneously
  - there can be a mixture of slow and fast clients
- Need something that enables efficient utilization of CPU resource (and other resources as well)

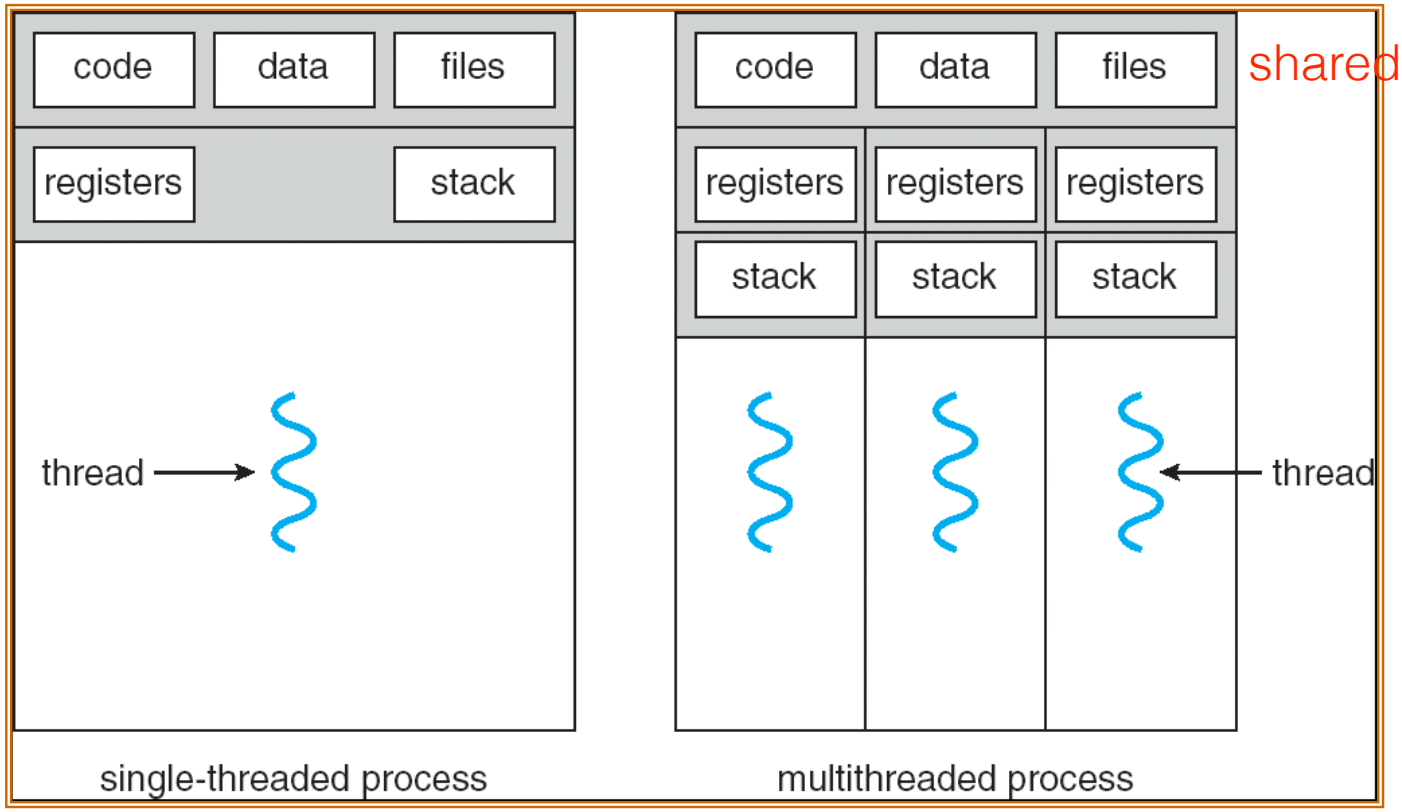
Handling Multiple Requests

- There are several approaches to handling multiple requests
- Approach I
  - fork a new process for every new request
    - a process takes separate memory space
  - poor scalability
- Approach II (process pool)
  - fixed # of processes are spawned and reused
- Approach III (multi-threaded program)
  - use lightweight threads

thread pool is also an option

Threads

- Thread
  - basic unit of CPU utilization
  - separate, independent path of execution



can be different depending on OS

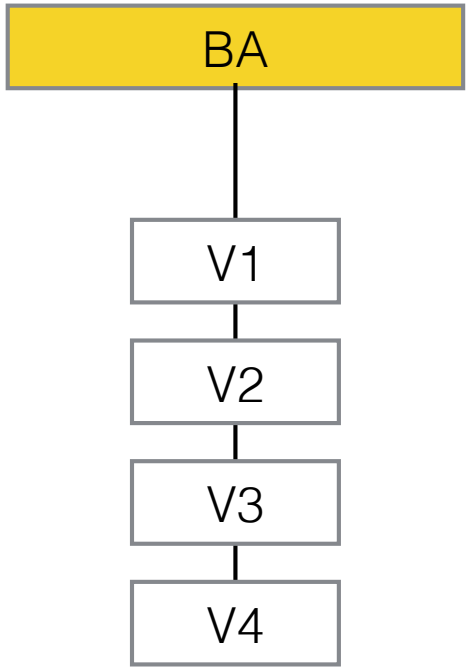
Multi-threading

- In game programming
  - characters move at the same time (each character ~ thread)

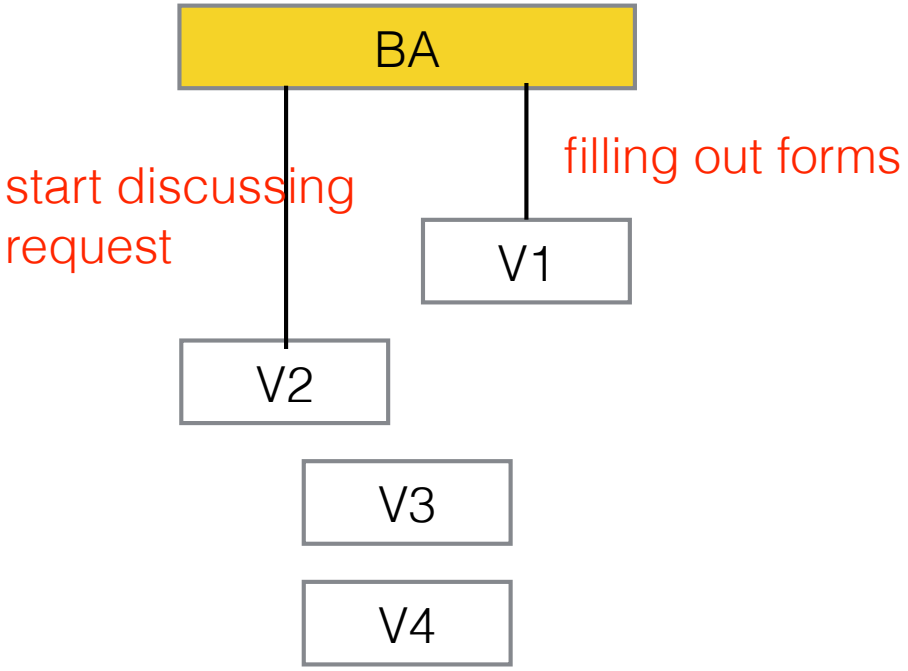
- In network programming
  - socket waiting requests
  - socket handling data transmission and reception

Human Analogy

- Bank accountant (BA), Visitor (V)



single-threaded



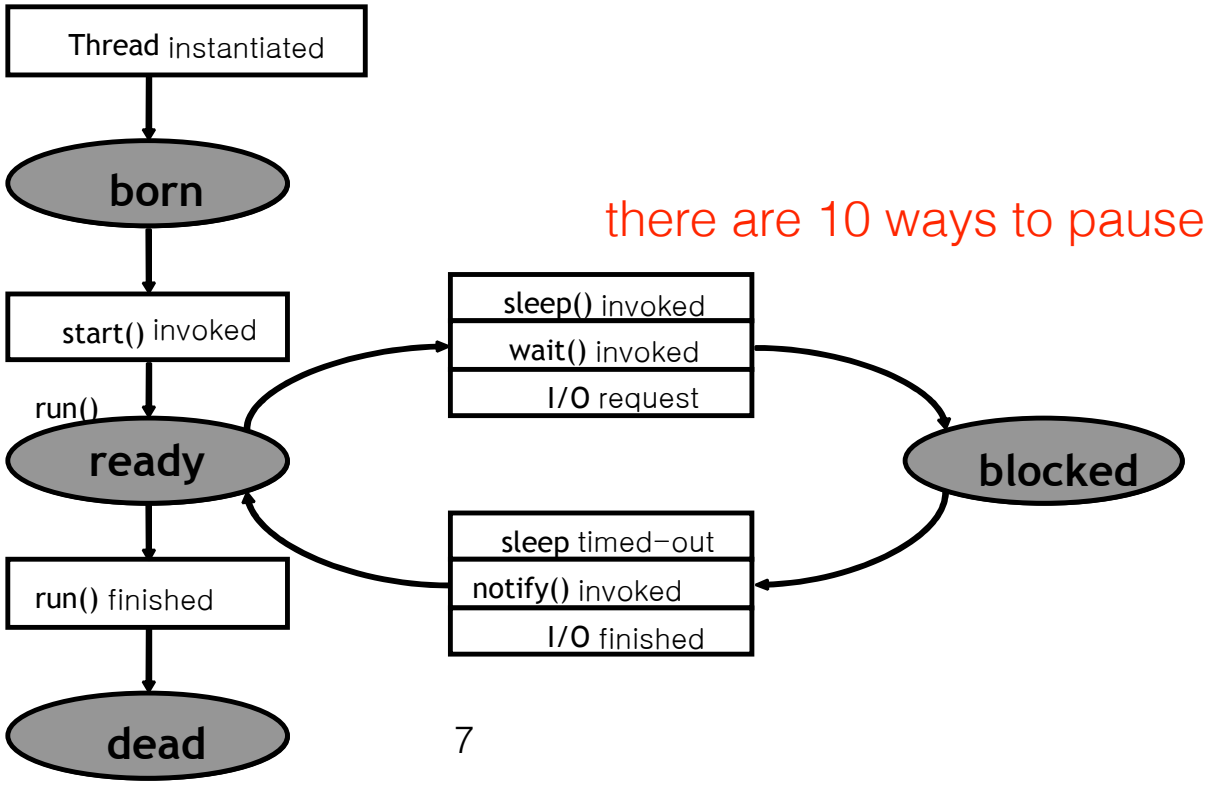
multi-threaded

order of
job completion: V1, V2, V3, V4 (FIFO)

order of
job completion: not necessarily FIFO

Running Threads in Java

- Two ways
  - subclassing Thread class
  - implementing Runnable interface
- Life cycle of a thread



there are 10 ways to pause thread

Thread Class

- Constructors

```
Thread()
Thread(Runnable target)
Thread(Runnable target, string name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, String name)
```

- Methods will be discussed later

Subclassing Thread Class

Subclassing Thread

- Construct an instance of Thread class and invoke start()

```
Thread t = new Thread();
t.start();
```

- Giving a thread something to do
  - override run() method of Thread class

```
public void run()
```

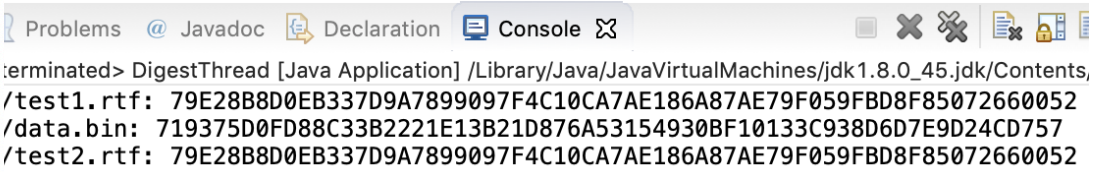
    - put all the work in this method
    - thread starts here and stops here
    - run() completes ⇒ thread dies
    - run() method to thread ≡ main() method to non-threaded program

Example

- Program that calculates Secure Hash Algorithm (SHA) digest (hash value) for many files
  - I/O-bound program (i.e., speed limited by the amount of time to read files from disk)
  - single-threaded program will spend a lot of time blocked, waiting for hard drive to return data
- Multi-threaded program
  - let other files be processed, while waiting for hard drive to return data
- DigestInputStream
  - calculate cryptographic hash function as it reads from files
  - digest() method does this

Example

```java
import java.io.*;
import java.security.*;

public class DigestThread extends Thread {

    private String filename;

    public DigestThread(String filename) {
        this.filename = filename;
    }

    @Override
    public void run() {
        try {
            FileInputStream in = new FileInputStream(filename);
            MessageDigest sha = MessageDigest.getInstance("SHA-256");
            DigestInputStream din = new DigestInputStream(in, sha);
            while (din.read() != -1) ;
            din.close();
            byte[] digest = sha.digest();

            //*good
            StringBuilder result = new StringBuilder(filename);
            result.append(": ");
            result.append(toHexString(digest));
            System.out.println(result);
            //*/
        } catch (IOException ex) {
            System.err.println(ex);
        } catch (NoSuchAlgorithmException ex) {
            System.err.println(ex);
        }
    }
    public static String toHexString(byte[] bytes) {
        StringBuilder hexString = new StringBuilder();

        for (int i = 0; i < bytes.length; i++) {
            String hex = Integer.toHexString(0xFF & bytes[i]);
            if (hex.length() == 1) {
                hexString.append('0');
            }
            hexString.append(hex);
        }

        return hexString.toString();
    }

    public static void main(String[] args) {
        for (String filename : args) {
            Thread t = new DigestThread(filename);
            t.start();
        }
    }
}
```



Implementing Runnable Interface

Implementing Runnable Interface

- Implement the Runnable interface and pass the Runnable object to the Thread constructor

```
public class MyRunnable implements Runnable {
…
    public static void main(String[] args) {
        MyRunnable myRunnableObject = new MyRunnable(f);
        Thread t = new Thread(myRunnableObject);
        t.start();
    }
}
```

```java
import java.io.*;
import java.security.*;

public class DigestRunnable implements Runnable {

  private String filename;

  public DigestRunnable(String filename) {
   this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ;
      din.close();
      byte[] digest = sha.digest();

      StringBuilder result = new StringBuilder(filename);
      result.append(": ");
//    result.append(DatatypeConverter.printHexBinary(digest));
      result.append(toHexString(digest));
      System.out.println(result);
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }

    public static String toHexString(byte[] bytes) {

  public static void main(String[] args) {
    for (String filename : args) {
      DigestRunnable dr = new DigestRunnable(filename);
      Thread t = new Thread(dr);
      t.start();
    }
  }
}
```

Subclassing vs. Runnable Interface

- Subclassing (white box)
  - useful when it is necessary to use methods in Thread within the constructor of subclass
  - task that thread undertakes is not really a kind of thread (better to separate thread and task, rather than place task in a subclass of Thread)

- In favor of Runnable interface (black box)
  - superclass better protected
  - separate thread and task

Returning Information from a Thread

Adding Getter

- Subclassing
  - add field(s) in the subclass, and a getter method to get access to the field
  - in main(), use the getter method to access fields

- Runnable interface
  - add field(s) in the Runnable interface, and a getter method to get access to the field
  - in main(), use the getter method to access fields

Example

```
import java.io.*;
import java.security.*;

public class ReturnDigest extends Thread {

  private String filename;
  private byte[] digest;

  public ReturnDigest(String filename) {
    this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // read entire file
      din.close();
      digest = sha.digest();
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }

  public byte[] getDigest() {
    return digest;
  }
}
```

Example

```
public static void main(String[] args) {
    for (String filename : args) {
        // Calculate the digest
        ReturnDigest dr = new ReturnDigest(filename);
        dr.start();

        // Now print the result
        StringBuilder result = new StringBuilder(filename);
        result.append(": ");
        byte[] digest = dr.getDigest();
        result.append(toHexString(digest));
        System.out.println(result);
    }
} //*/
```

Problems @ Javadoc Declaration Console

```
<terminated> ReturnDigestUserInterface [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bi
Exception in thread "main" java.lang.NullPointerException
        at javax.xml.bind.DatatypeConverterImpl.printHexBinary(DatatypeConverterImpl.java:475)
        at javax.xml.bind.DatatypeConverter.printHexBinary(DatatypeConverter.java:626)
        at ReturnDigestUserInterface.main(ReturnDigestUserInterface.java:15)
```

Discussion

- What is wrong with the previous example?

First Try

- Start all the threads and get digest

```
public static void main(String[] args) {

  ReturnDigest[] digests = new ReturnDigest[args.length];

  for (int i = 0; i < args.length; i++) {
    // Calculate the digest
    digests[i] = new ReturnDigest(args[i]);
    digests[i].start();
  }

  for (int i = 0; i < args.length; i++) {
    // Now print the result
    StringBuffer result = new StringBuffer(args[i]);
    result.append(": ");
    byte[] digest = digests[i].getDigest();
    result.append(DatatypeConverter.printHexBinary(digest));

    System.out.println(result);
  }
}
```

Race Condition

- Can still have the same problem
  - ReturnDigest threads spawned in the first loop may not finish before getDigest() method is called
- Many factors can affect the result
  - # threads the program spawns
  - speed of CPU
  - # CPUs used
  - algorithm that JVM uses to allot time to different threads
- Race condition
  - getting correct results depends on relative speeds of different threads, which cannot be controlled

Polling

- Proceed if returned value is valid
- Exercise
  - modify the second for loop so that result is printed only if digest is not null

Polling (contd.)

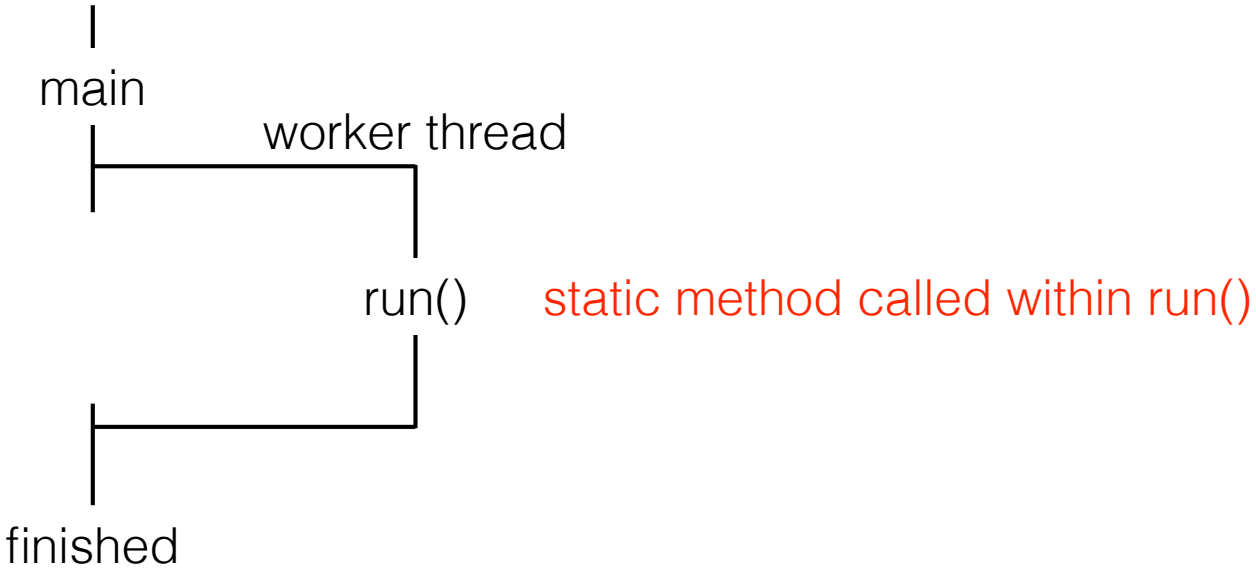
- Polling "may" solve the problem
  - on some virtual machines, main thread (while loop) takes all the time available checking for job completion and leaves no time for actual worker threads
  - job may never complete (depending on platform)

- Caveats (even if it works)
  - it does a lot of work (while loop)

Callbacks

- Key idea
  - let thread tell main program when it's finished
  - thread calls its creator back when it's done ⇒ callback

- Main program can go to sleep



Example

```
import java.io.*;
import java.security.*;

public class CallbackDigest implements Runnable {

  private String filename;

  public CallbackDigest(String filename) {
   this.filename = filename;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ; // read entire file
      din.close();
      byte[] digest = sha.digest();
      CallbackDigestUserInterface.receiveDigest(digest, filename);
    } catch (IOException ex) {
      System.err.println(ex);
    } catch (NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
}
```

static method called

Example (contd.)

```
public class CallbackDigestUserInterface {

    public static String toHexString(byte[] bytes) {

    public static void receiveDigest(byte[] digest, String name) {
        StringBuilder result = new StringBuilder(name);
        result.append(": ");
        result.append(toHexString(digest));
        System.out.println(result);
    }


    public static void main(String[] args) throws InterruptedException {
        for (String filename : args) {
            // Calculate the digest
            CallbackDigest cb = new CallbackDigest(filename);
            Thread t = new Thread(cb);
            t.start();
        }
    }
}
```

main() does nothing to print

Example

- Reference to CallbackDigestUserInterface object is given an argument in the constructor

```
import java.io.*;
import java.security.*;

public class InstanceCallbackDigest implements Runnable {

  private String filename;
  private InstanceCallbackDigestUserInterface callback;
  public InstanceCallbackDigest(String filename,
   InstanceCallbackDigestUserInterface callback) {
    this.filename = filename;
    this.callback = callback;
  }

  @Override
  public void run() {
    try {
      FileInputStream in = new FileInputStream(filename);
      MessageDigest sha = MessageDigest.getInstance("SHA-256");
      DigestInputStream din = new DigestInputStream(in, sha);
      while (din.read() != -1) ;  // read entire file
      din.close();
      byte[] digest = sha.digest();
      callback.receiveDigest(digest);
    } catch (IOException | NoSuchAlgorithmException ex) {
      System.err.println(ex);
    }
  }
}
```

callback instance stored in a field

instance method called

Example

- Store digest in a field

```
import javax.xml.bind.*; // for DatatypeConverter; requires Java 6 or JAXB

public class InstanceCallbackDigestUserInterface {

  private String filename;
  private byte[] digest;

  public InstanceCallbackDigestUserInterface(String filename) {
    this.filename = filename;
  }

  public void calculateDigest() {
    InstanceCallbackDigest cb = new InstanceCallbackDigest(filename, this);
    Thread t = new Thread(cb);
    t.start();
  }

  void receiveDigest(byte[] digest) {
    this.digest = digest;
    System.out.println(this);
  }
```

```
    @Override
    public String toString() {
        String result = filename + ": ";
        if (digest != null) {
            result += toHexString(digest);
        } else {
            result += "digest not available";
        }
        return result;
    }

    public static String toHexString(byte[] bytes) {

    public static void main(String[] args) {
        for (String filename : args) {
            // Calculate the digest
            InstanceCallbackDigestUserInterface d
            = new InstanceCallbackDigestUserInterface(filename);
            d.calculateDigest();
        }
    }
}
```

advantage

-each instance maps to exactly one file

-can recalculate if needed

Callback vs. Polling

- Advantages of callback (compared with polling)
  - it doesn't waste so many CPU cycles
  - more flexible
    - if more than one object is interested in the result of thread's calculation, the thread can keep a list of objects to call back

Futures, Callables, Executors

- ExecutorService
  - create threads as needed
- Callable<V>
  - implementors must define the method "V call()" which returns data type V
  - recall that Runnable does not return any value
- How to program
  - submit Callable jobs to ExecutorService, and get back a Future instance
  - later point, ask the Future instance for the result of job

can achieve huge performance gain, in resource-rich environment

Example

```
import java.util.concurrent.Callable;

class FindMaxTask implements Callable<Integer> {

  private int[] data;
  private int start;
  private int end;

  FindMaxTask(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  public Integer call() {
    int max = Integer.MIN_VALUE;
    for (int i = start; i < end; i++) {
      if (data[i] > max) max = data[i];
    }
    return max;
  }
}
```

Example (contd.)

```
import java.util.concurrent.*;

public class MultithreadedMaxFinder {

  public static int max(int[] data) throws InterruptedException, ExecutionException {

    if (data.length == 1) {
      return data[0];
    } else if (data.length == 0) {
      throw new IllegalArgumentException();
    }

    // split the job into 2 pieces
    FindMaxTask task1 = new FindMaxTask(data, 0, data.length/2);
    FindMaxTask task2 = new FindMaxTask(data, data.length/2, data.length);

    // spawn 2 threads
    ExecutorService service = Executors.newFixedThreadPool(2);

    Future<Integer> future1 = service.submit(task1);
    Future<Integer> future2 = service.submit(task2);

    return Math.max(future1.get(), future2.get());
  }
}
```

can possibly run almost twice as fast

future1.get() blocks until first FindMaxTask finished => future2.get() called

Advantages

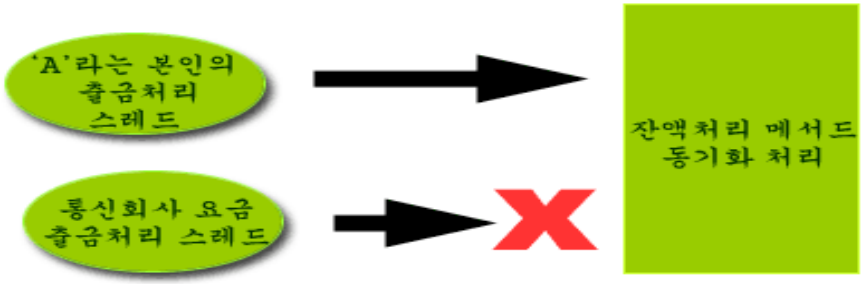
- Convenient if want to work on different pieces of a problem
- Executors and executor services let programmers assign jobs to different threads with different strategies
  - in the previous example, two threads are spawned
- Programmers don't need to worry about asynchronicity

Synchronization

Shared Resources

- Threads make programs more efficient by sharing memory, file handles, sockets and other resources
- Downside
  - if two threads want the same resource at the same time, one of them will have to wait for the other to finish

만약 A 라는 사람이 현금인출기를 통해서 돈을 출금하려고 한다고 하자. 현재 총금액은 1000원이고 출금액은 100원이라고 한다면 잔액은 900원이 될 것이다. 그럼 은행 프로그램에서 금액 1000원에서 잔액을 900으로 처리하는 순간, B라는 통신회사에서 전화요금 200원을 출금해 간다면.....
아직 잔액 900처리가 끝나지 않은 상태이므로 여전히 총금액은 1000원이다. 따라서 통신회사에서 200원을 출금하고 잔액은 800원이 될것이고 본인이 100원을 출금했으니, 900원이 잔액이 된다. 이는 결과는 800원 혹은 900원이 될것이다.
이렇게 된다면 정말 큰일이다. 이는 잔액을 처리하는 메서드에 동시에 여러 객체가 접근하기 때문이다. 따라서 이 메서드에 동기화처리를 해주면 이런 문제는 말끔히 처리가 된다.



< 동기화를 사용해서 데이터를 보존 >

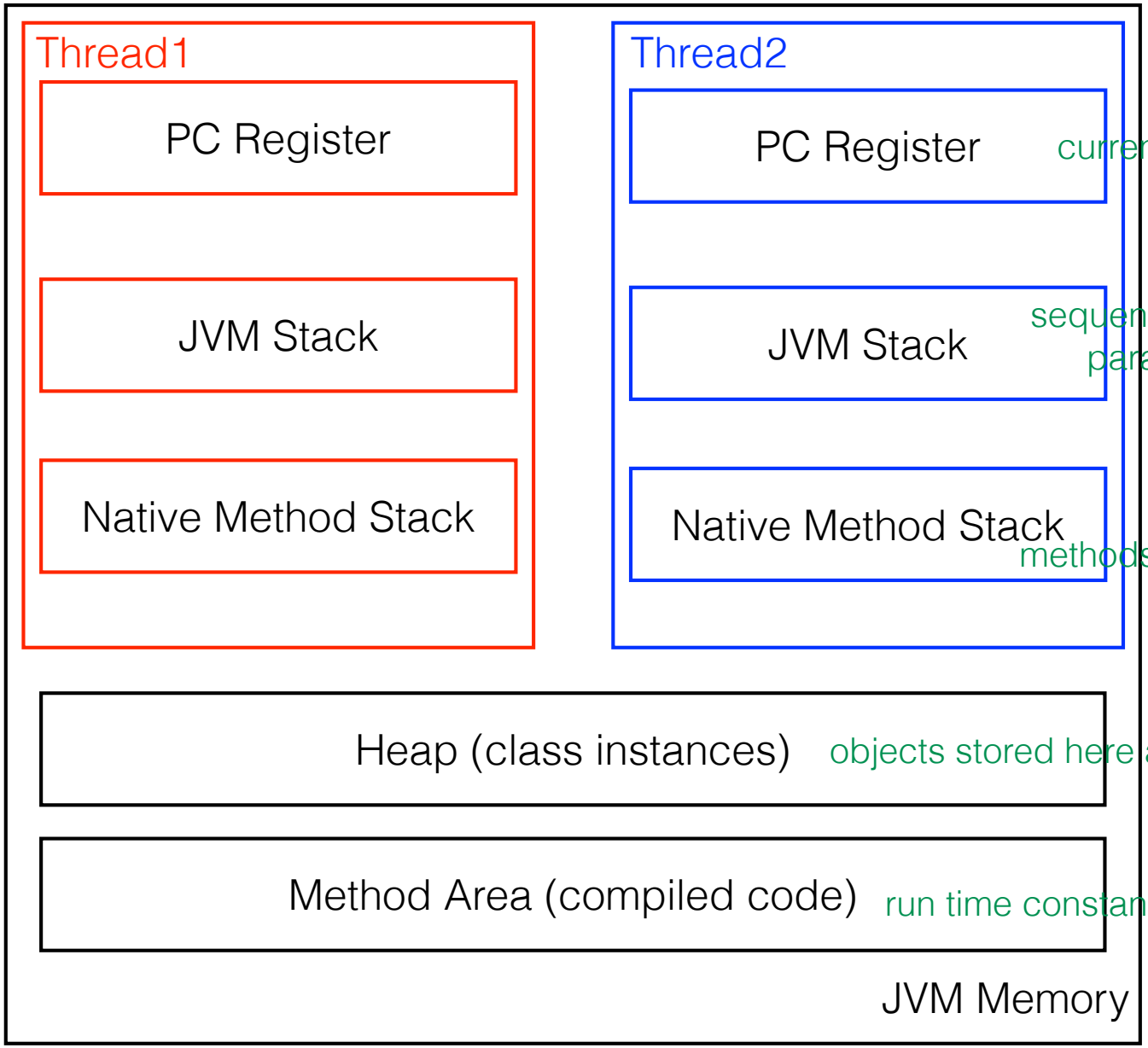
Example

```
@Override
public void run() {
  try {
    FileInputStream in = new FileInputStream(filename);
    MessageDigest sha = MessageDigest.getInstance("SHA-256");
    DigestInputStream din = new DigestInputStream(in, sha);
    while (din.read() != -1) ; // read entire file
    din.close();
    byte[] digest = sha.digest();
    System.out.print(input + ": ");
    System.out.print(toHexBinary(digest));
    System.out.println();
  } catch (IOException ex) {
    System.err.println(ex);
  } catch (NoSuchAlgorithmException ex) {
    System.err.println(ex);
  }
}
```

```
Triangle.java: B4C7AF1BAE952655A96517476BF9DAC97C4AF02411E40DD386FECB58D94CC769
InterfaceLister.java: Squares.java: UlpPrinter.java:
C8009AB1578BF7E730BD2C3EADA54B772576E265011DF22D171D60A1881AFF51
267D0EFE73896CD550DC202935D20E87CA71536CB176AF78F915935A6E81B034
DA2E27EA139785535122A2420D3DB472A807841D05F6C268A43695B9FDFE1B11
```

what happened?

JVM Memory Architecture



current instructions and addresses

sequence of methods, local vars., parameters, local obj. refs.

methods written in other languages

objects stored here and referenced by JVM Stack

run time constant pool, constructors

What Happened?

- System.out is shared by threads
  - right to access System.out can be changed
  - file names and digests can mix up

- Solution
  - exclusive access to System.out needed
  - "synchronized" block does this

```
synchronized (System.out) {
  System.out.print(input + ": ");
  System.out.print(toHexBinary(digest));
  System.out.println();
}
```

once one thread starts printing out the values,
all other threads will have to wait until the current thread finishes printing

Another Case

- System.out: static class variable
- Instance variables can also be a problem
- Example
  - logfile in web server
  - web server uses multiple threads to handle incoming connections
  - each of those threads needs access to the same logfile and consequently same LogFile (defined in the next slide) object

Example

```
import java.io.*;
import java.util.*;

public class LogFile {

  private Writer out;

  public LogFile(File f) throws IOException {
    FileWriter fw = new FileWriter(f);
    this.out = new BufferedWriter(fw);
  }

  public void writeEntry(String message) throws IOException {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

  public void close() throws IOException {
    out.flush();
    out.close();
  }
}
```

log info can mix up due to multiple threads interrupting each other

Solution 1

- Synchronize Writer object out

```
public void writeEntry(String message) throws IOException {
  synchronized (out) {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
```

Only a single thread can execute the code inside the synchronized block

Solution 11

- Synchronize LogFile object itself

```
public void writeEntry(String message) throws IOException {
  synchronized (this) {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
```

Only a single thread can execute the code inside the synchronized block

Solution III

- Synchronize methods

```
public synchronized void writeEntry(String message) throws IOException {
  Date d = new Date();
  out.write(d.toString());
  out.write('\t');
  out.write(message);
  out.write("\r\n");
}
```

-

Only a single thread can execute the code inside the synchronized block

Notes

- Synchronizing all the methods can be harmful
  - performance can be deteriorated
  - dramatically increase the chance of deadlock
    - deadlock do be discussed later
  - synchronizing on instance of method's class may fail
    - e.g., if some other class had a reference to out completely unrelated to LogFile (out used somewhere else other than synchronized block)

Deadlock

Deadlock

- Synchronization can lead to another problem: deadlock
  - two threads need exclusive access to the same set of resources and each thread holds the lock on a different subset of those resources
  - neither thread is willing to give up the resources it has => both threads come to an indefinite halt
- Deadlock
  - viewed as normal behavior to OS
  - hard to detect since it can originate from sporadic timing issues
- Synchronization should be a last resort for ensuring thread safety

Alternatives to Synchronization

- Use local variables if possible
- Method arguments of primitive types are safe from modification in separate threads
  - java passes arguments by value rather than by reference
  - e.g., Math.sqrt() perform some calculation and return value without ever interacting with the fields of any class, and hence, thread-safe
    - Math.sqrt(a) take the value of a and performs calculations (call by value)
  - Method arguments of object types can be tricky
- Use immutability
  - String, Integer, Double,…
  - make object immutable by declaring it private and final

Thread Scheduling

Need for Scheduling

- I/O-bound threads wait for input a lot of time
  - CPU much faster than I/O speed
- CPU-bound threads
  - e.g., jobs that have nothing to do with I/O
  - can take all the available CPU resources and starve all other threads
- Need to pose some kind of priorities to avoid starvation problem

Priorities

- Each thread has a priority
  - integer from 0(lowest priority) to 10(highest)
  - default: 5

```
public static final int MIN_PRIORITY  = 1;
public static final int NORM_PRIORITY = 5;
public static final int MAX_PRIORITY  = 10;

public final void setPriority(int newPriority)
```

- Some norms to setting priority
  - threads that interact with user: high
  - threads that calculate in the background: low
  - threads that complete quickly: high
  - threads that take long to complete: low

Example

```
public class ThreadPriorityTest extends Thread {

    public void run() {
        for (int i=1; i<=10; i++)
            System.out.println(getName()+": "+i);
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ThreadPriorityTest t1 = new ThreadPriorityTest();
        ThreadPriorityTest t2 = new ThreadPriorityTest();

        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MIN_PRIORITY);

        t1.start();
        t2.start();

    }

}
```

change priorities and see what happens

Thread Scheduling Methods

- Preemptive
  - determine when a thread has had its fair share
  - pause the thread
  - hand off control of CPU to a different thread

- Cooperative
  - wait for running thread to pause itself before handing off control of CPU
  - more susceptible to starvation

Java VMs

- All java VMs use preemptive thread scheduling between priorities
  - running thread: low priority
  - another thread ready to run: high priority
  - will soon preempt low-priority thread and allow high-priority thread to run

- Sometimes, it is necessary to pause running thread purposely

Pausing a Thread

10 Ways to Pause

- Block on I/O
- Block on synchronized object
- Yield
- Sleep
- Join another thread
- Wait on an object
- Finish
- Preempt
- Suspend (deprecated)
- Stop (deprecated)

Blocking

- Two cases
  - (blocking on I/O) a thread has to stop and wait for a resource it doesn't possess
    - e.g., network program: a thread voluntarily gives up control of CPU while waiting for data to arrive from or be sent out to network
  - (blocking on lock) a thread pause if it enters a synchronized method or block for which it doesn't have the lock
    - if the lock is never released, thread is permanently stopped
    - lock: some kind of token.
    - lock is attached to every instantiated object stored in heap area.
    - lock is activated with synchronized keyword
    - can incur deadlock (T1 waits for T2's locks to be released, T2 waits for T1's locks to be released)

Yielding

- Thread gives up control of CPU by invoking Thread.yield()

```
public void run() {
  while (true) {
    // Do the thread's work...
    Thread.yield();
  }
}
```

  - this signals to VM that it can run another thread, giving other threads of the same priority the opportunity to run
- yield() does not release any locks the thread holds
  - other threads can't run if they need synchronized resources held by the yielding thread

Sleeping

- Sleeping thread pauses whether there are other threads ready to run or not
  - yielding does not necessarily result in pause, e.g., if no other threads are ready to run
- A thread going to sleeping does not release the locks it holds
  - other threads that need the same locks will be blocked even if CPU is available
- Two methods

```
public static void sleep(long milliseconds) throws InterruptedException
public static void sleep(long milliseconds, int nanoseconds)
    throws InterruptedException
```

for more fine control of sleeping time
(not guaranteed to work as desired)

Sleeping (contd.)

- Example

```
public void run() {
  while (true) {
    if (!getPage("http://www.ibiblio.org/")) {
      mailError("webmaster@ibiblio.org");
    }
    try {
      Thread.sleep(300000); // 300,000 milliseconds == 5 minutes
    } catch (InterruptedException ex) {
      break;
    }
  }
}
```

  - run() method attempts to load a page every five minutes
- Sleeping doesn't guarantee to sleep as long as it wants to
  - it can pass wake-up time if VM is busy doing other jobs
  - or other threads can wake up (with interruption)

Sleeping (contd.)

- Interruption

  ```
  public void interrupt()
  ```

  - Other threads can invoke sleeping thread's interrupt() method
  - Sleeping thread will experience InterruptedException

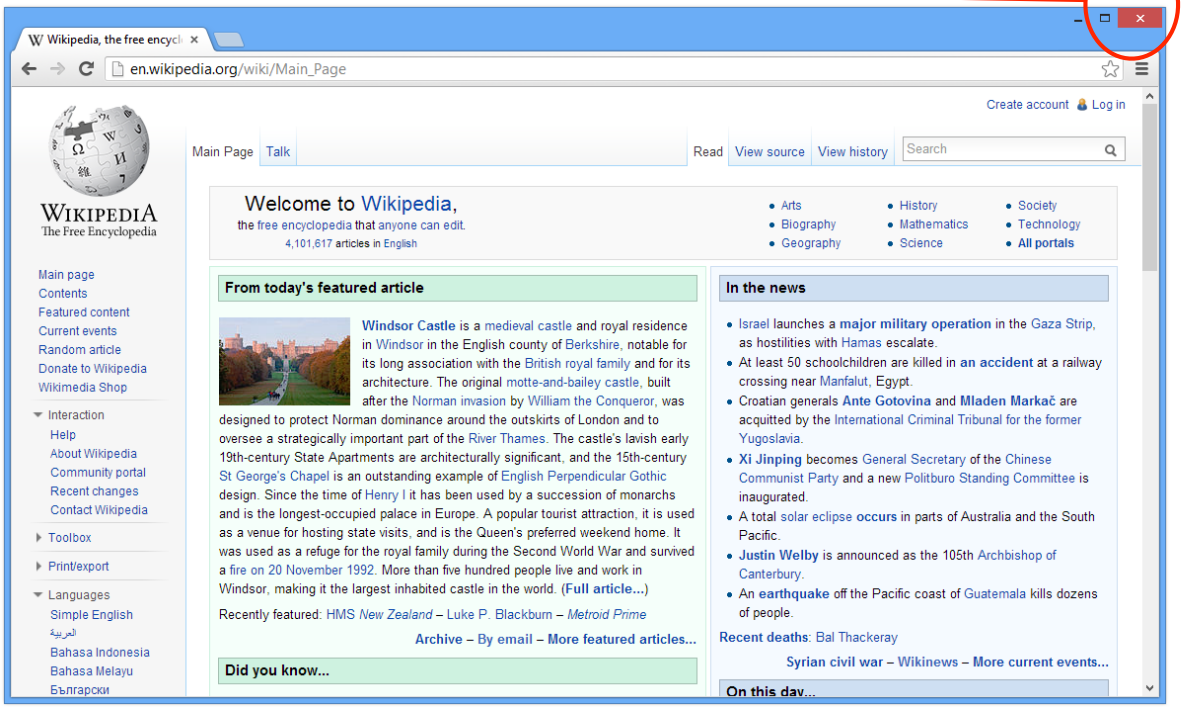
- Example

```
public class ThreadInterruptExample extends Thread {

    public void run() {
        try {
            Thread.sleep(2000);
            System.out.println("run(): after sleeping 2000ms");
        }catch(InterruptedException e){
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        ThreadInterruptExample t1=new ThreadInterruptExample();
        t1.start();
        try {
            Thread.sleep(1000);
            t1.interrupt();
        }catch(Exception e){System.out.println("Exception handled "+e);}
    }

}
```

Example

```
public void run() {
  while (true) {
    if (!getPage("http://www.ibiblio.org/")) {
      mailError("webmaster@ibiblio.org");
    }
    try {
      Thread.sleep(300000); // 300,000 milliseconds == 5 minutes
    } catch (InterruptedException ex) {
      break;
    }
  }
}
```

Infinite loop should broken when user selects Exit



Sleeping (contd.)

- Other threads can't work with a sleeping thread, but can work with the corresponding Thread object
  - e.g., other threads can invoke the sleeping Thread's object's interrupt() method
  - This point forward, the thread is awake and executes as normal

- Exercise
  - add Thread.yield() or Thread.sleep(ms) to synchronized block in a single thread in DigestThread.java
  - add Thread.yield() or Thread.sleep(ms) to ReturnDigestUserInterface (polling)

Joining

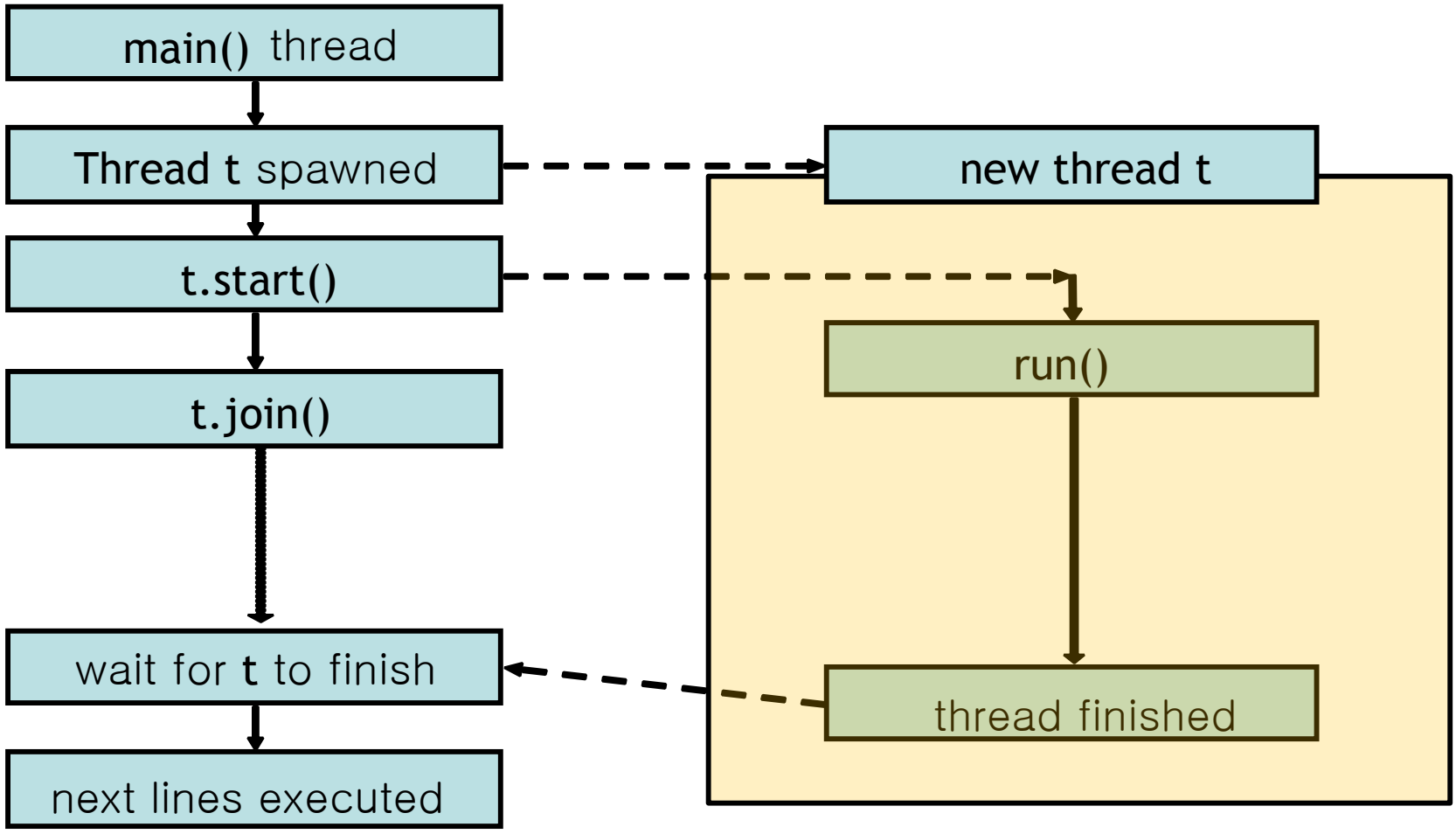
- Sometimes, one thread must wait for the result calculated by another thread
  - e.g., web browser loading an html page in one thread might spawn a separate thread to retrieve an image embedded in the page
  - if IMG elements don't have HEIGHT and WIDTH, the main thread might have to wait for the image to load before it can finish by displaying the page
- Three methods

```
public final void join() throws InterruptedException
public final void join(long milliseconds) throws InterruptedException
public final void join(long milliseconds, int nanoseconds)
    throws InterruptedException
```

wait for joined thread indefinitely or for given time

Joining (contd.)

- join() method



Joining (contd.)

- Example

```
double[] array = new double[10000];
for (int i = 0; i < array.length; i++) {
  array[i] = Math.random();
}
SortThread t = new SortThread(array);
t.start();
try {
  t.join();
  System.out.println("Minimum: " + array[0]);
  System.out.println("Median: " + array[array.length/2]);
  System.out.println("Maximum: " + array[array.length-1]);
} catch (InterruptedException ex) {
}
```

waits for SortThread to finish

- Notes
  - no reference to current thread (exists implicitly)
  - joined thread not aware of joining thread

Joining (contd.)

- Example

```
public class ThreadJoinTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Thread t = new Thread() {
            public void run() {
                try {
                    Thread.sleep(2000);
                    System.out.println("Mythread ended");
                    Thread.sleep(3000);
                } catch (Exception e) {

                }
            }
        };

        t.start();
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("main() ended");

    }

}
```

Waiting

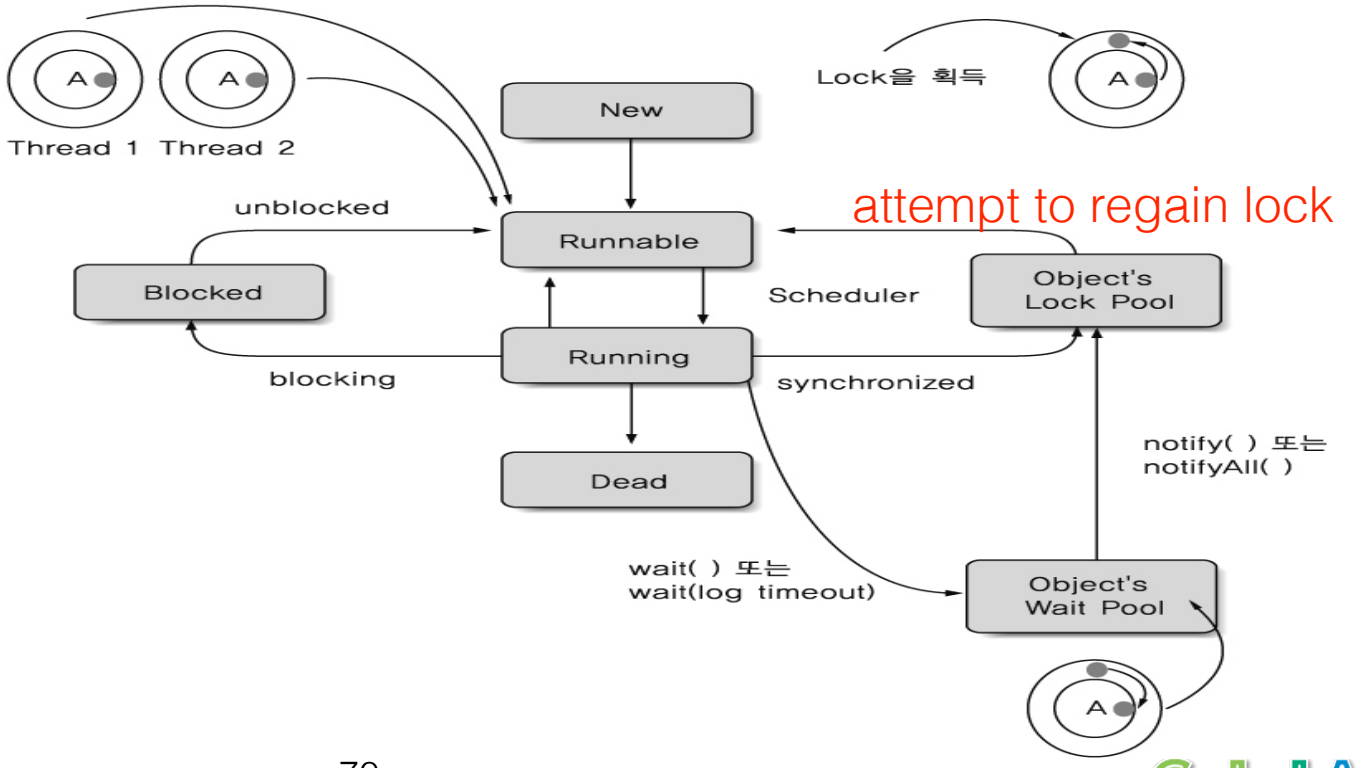
- A thread can wait until an object or resource reaches a certain state
- Methods (in java.lang.Object, can be invoked on any object of any class)

```
public final void wait() throws InterruptedException
public final void wait(long milliseconds) throws InterruptedException
public final void wait(long milliseconds, int nanoseconds)
    throws InterruptedException
```

- wait() vs. join()
  - join() pauses until joined thread finishes
  - wait() pauses until an object or resource reaches a certain state
    - wait() invoked on an object (not thread)

Notifying

- A thread that invoked wait() on an object releases the lock on the object (not the locks on the other objects)
- A thread that invoked wait() goes to sleep until one of the following events happens
  - timeout
  - interruption
  - notification



Notifying

- Methods (must be invoked on an object)

```
public final void notify()
public final void notifyAll()
```

  - before notifying, a thread must obtain the lock on the object, using synchronized method or block

- notify()
  - selects one thread at random from the list of threads waiting on the object
  - wakes the selected thread

- notifyAll()
  - wakes up every thread waiting on the given object

Waiting and Notifying

- Example

```
ManifestFile m = new ManifestFile();
JarThread    t = new JarThread(m, in);
synchronized (m) {
  t.start();
  try {
    m.wait();
    // work with the manifest file...
  } catch (InterruptedException ex) {
    // handle exception...
  }
}
```

```
ManifestFile theManifest;
InputStream in;

public JarThread(Manifest m, InputStream in) {
  theManifest = m;
  this.in= in;
}

@Override
public void run() {
  synchronized (theManifest) {
    // read the manifest from the stream in...
    theManifest.notify();
  }
  // read the rest of the stream...
}
```

homework to write producer/consumer/buffer

Thread Pools and Executors

Motivation

- Adding a thread is not free
  - starting and cleaning up takes a noticeable amount of work from VM (switching overhead between threads)
  - if threads are CPU-bound, total task could finish quickly without switching overhead
  - once you spawned enough threads to fully utilize CPU, spawning more threads just wastes memory on thread management
- Executors class (in java.util.concurrent)
  - set up thread pool
  - you just need to submit each task as a Runnable object to the pool
  - you get back a Future object you can use to check on the progress of task

Example

- Read all the files in the current directory, compress them and create new .gz files
- I/O-heavy operations
  - read and write files
- CPU-intensive operations
  - compressing
- Spawn and reuse a fixed number of threads
  - main thread list files and submit task
  - threads in the pool read, compress, and write files
  - (recommended) main thread will outpace ⇒ fill the pool first and start threads: the following example doesn't do this

```java
import java.io.*;
import java.util.zip.*;

public class GZipRunnable implements Runnable {

  private final File input;

  public GZipRunnable(File input) {
    this.input = input;
  }

  @Override
  public void run() {
    // don't compress an already compressed file
    if (!input.getName().endsWith(".gz")) {
      File output = new File(input.getParent(), input.getName() + ".gz");
      if (!output.exists()) { // Don't overwrite an existing file
        try ( // with resources; requires Java 7
          InputStream in = new BufferedInputStream(new FileInputStream(input));
          OutputStream out = new BufferedOutputStream(
            new GZIPOutputStream(
              new FileOutputStream(output)));
           ) {
            int b;
            while ((b = in.read()) != -1) out.write(b);
            out.flush();
          } catch (IOException ex) {
            System.err.println(ex);
          }
        }
      }
    }
  }
```

```java
import java.io.*;
import java.util.concurrent.*;

public class GZipAllFiles {

  public final static int THREAD_COUNT = 4;

  public static void main(String[] args) {

    ExecutorService pool = Executors.newFixedThreadPool(THREAD_COUNT);

    for (String filename : args) {
      File f = new File(filename);
      if (f.exists()) {
        if (f.isDirectory()) {
          File[] files = f.listFiles();
          for (int i = 0; i < files.length; i++) {
            if (!files[i].isDirectory()) { // don't recurse directories
              Runnable task = new GZipRunnable(files[i]);
              pool.submit(task);

            }
          }
        } else {
          Runnable task = new GZipRunnable(f);
          pool.submit(task);
        }
      }
    }

    pool.shutdown();
  }
}
```

- notifies the pool that no additional task will be submitted
- shut down once it has finished all the pending work

Exercise

- Implement polling to solve race condition

- Hint
  - Start all the threads first
  - For each thread, check the nullity of digest, and
    - If null, keep checking the nullity
    - If not null, build the string "key(=file name): value(=hash)", and print on the screen

Exercise

- Use join() method to solve race condition problem in
  - RaceConditionTest.java

- Hint
  - start all the threads first
  - and then join each of the threads

- Observe the order of keys
  - in run()
  - in main()
  - Analyze what has happened based on the results displayed on console

Exercise

- Write a multi-threaded program that
  - Takes an integer array
    - (Example) int[ ] num = {5, 2, 9, 10, 0}
  - Prints the integers in an ascending order
    - 출력순서: 0, 2, 5, 9, 10
  - Does not use relational operators to compare integers