

Java Concurrency

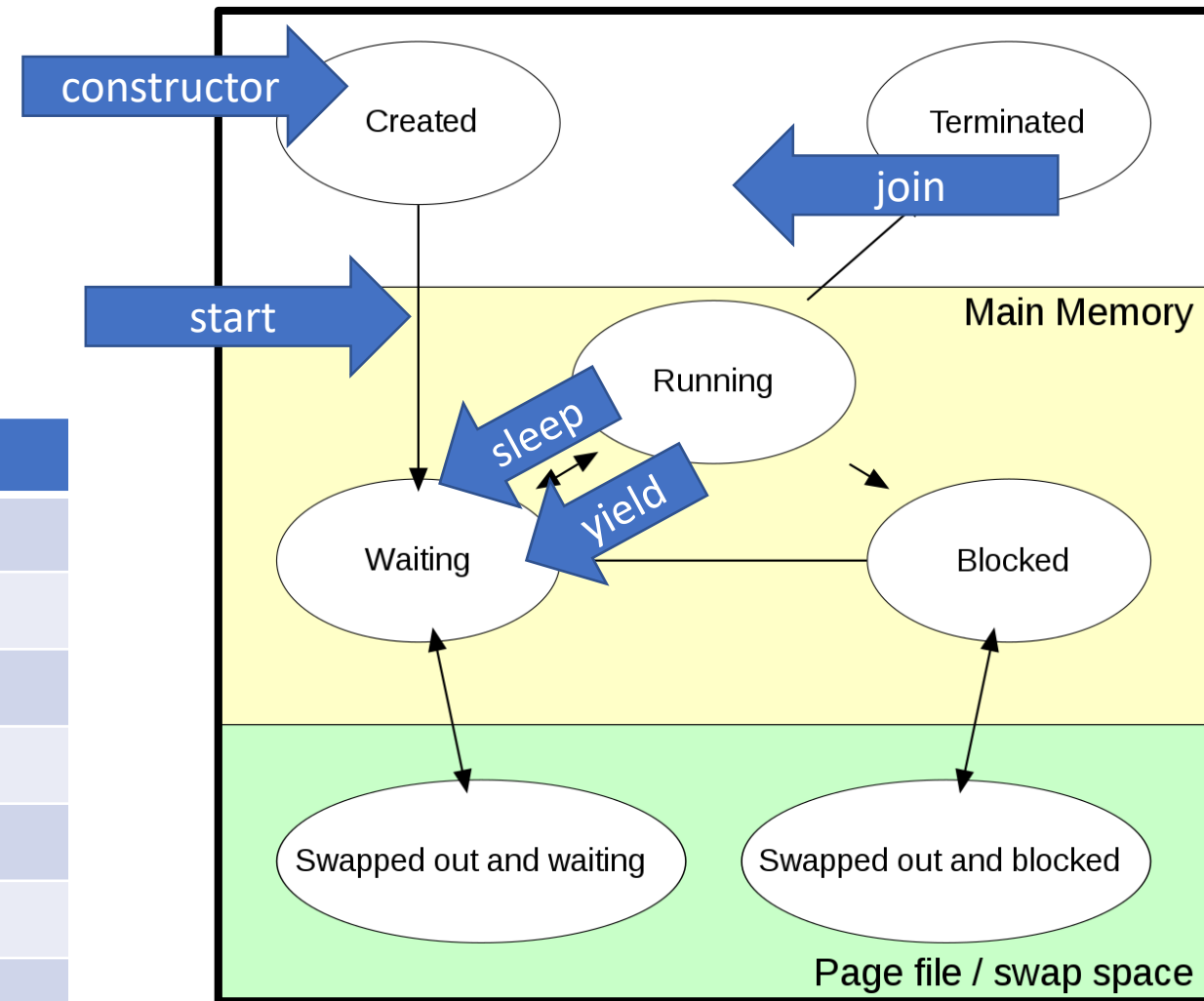
Nir Piterman

TDA 384 / DIT 391

Creating Threads

- What does a thread need to do?

Method	
start()	Start a thread by calling run() method
run()	Entry point for a thread
join()	Wait for a thread to end
isAlive()	Checks if thread is still running or not
setName()	
getName()	
getPriority()	



https://en.wikipedia.org/wiki/Process_state

Extend Thread

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        mt.start();
    }
}
```

Extend?

Hierarchy: Animals

- Animal
 - Mammal
 - Canine
 - Dog
 - Wolf
 - Feline
 - Cat
 - Fish
 - Tuna
 - Shark
 - Reptile
 - Crocodile
 - Iguana

Object - Bank Account

- Accounts have certain data and operations
 - Regardless of whether checking, savings, etc.
- Data
 - account nu
 - balance
 - owner
- Operations
 - open
 - close
 - get balance
 - deposit
 - withdraw

Kinds of Bank Accounts

- Account
 - **Checking**
 - Monthly fees
 - Minimum balance.
 - **Savings**
 - Interest rate
- Each type shares some data and operations of "account", and has some data and operations of its own.

Implement Runnable

- Java does not support multiple inheritance.
- If you need your class to inherit.

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
    }
}
```

Data Races

Data races

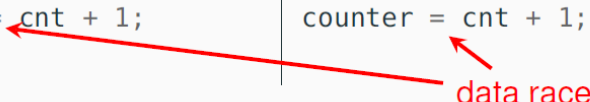
Race conditions are typically caused by a **lack of synchronization** between threads that **access shared** memory.

A **data race** occurs when two concurrent threads

- access a shared memory location,
- at least one access is a **write**,
- the threads use no explicit **synchronization mechanism** to protect the shared data.

```
int counter = 0;
```

	thread t	thread u	
	<pre>int cnt;</pre>	<pre>int cnt;</pre>	
1	<pre>cnt = counter;</pre>	<pre>cnt = counter;</pre>	1
2	<pre>counter = cnt + 1;</pre>	<pre>counter = cnt + 1;</pre>	2



6/46

Concurrency humor

Knock knock.

– “Race condition.”

– “Who’s there?”

5/46

Locks

Lock implementations in Java

The most common implementation of the `Lock` interface in Java is `class ReentrantLock`.

Mutual exclusion:

- `ReentrantLock` guarantees **mutual exclusion**

Starvation:

- `ReentrantLock` does **not** guarantee freedom from starvation by default
- however, calling the constructor with `new ReentrantLock(true)` “favors granting access to the **longest-waiting** thread”
- this still does not guarantee that thread **scheduling** is fair

Deadlocks:

- one thread will succeed in acquiring the lock
- however, deadlocks may occur in systems that use multiple locks (remember the dining philosophers)

```
interface Lock {  
    void lock();    // acquire lock  
    void unlock(); // release lock  
}
```

Implicit Locking

Built-in locks in Java

Every object in Java has an **implicit** lock, which can be accessed using the keyword **synchronized**.

Whole method locking
(**synchronized methods**):

```
synchronized T m() {  
    // the critical section  
    // is the whole method  
    // body  
}
```

Every call to `m` **implicitly**:

1. acquires the lock
2. executes `m`
3. releases the lock

Block locking
(**synchronized block**):

```
synchronized(this) {  
    // the critical section  
    // is the block's content  
}
```

Every execution of the block **implicitly**:

1. acquires the lock
2. executes the block
3. releases the lock

Semaphores

```
interface Semaphore {  
    int count();    // current value of counter  
    void up();      // increment counter  
    void down();    // decrement counter  
}
```

Mutual exclusion for **two** processes with semaphores

With semaphores the entry/exit protocols are trivial:

- initialize semaphore to 1
- **entry protocol**: call `sem.down()`
- **exit protocol**: call `sem.up()`

```
Semaphore sem = new Semaphore(1);
```

	thread t	thread u	
	<pre>int cnt;</pre>	<pre>int cnt;</pre>	
1	<pre>sem.down();</pre>	<pre>sem.down();</pre>	5
2	<pre>cnt = counter;</pre>	<pre>cnt = counter;</pre>	6
3	<pre>counter = cnt + 1;</pre>	<pre>counter = cnt + 1;</pre>	7
4	<pre>sem.up();</pre>	<pre>sem.up();</pre>	8

The implementation of the Semaphore interface guarantees mutual exclusion, deadlock freedom, and starvation freedom.

Producer-consumer (PC) finite buffer, with semaphores

A finite buffer B holds up to N items produced by *producer*, p , and consumed by *consumer*, c . The conditions: c must wait if B is empty, and p must wait if B is full. Semaphores E and F are used to ensure this.

$\text{queue [capacity } N] \text{ of int } B := \emptyset$ $\text{sem } E := \langle 0, \emptyset \rangle, \quad \text{sem } F := \langle N, \emptyset \rangle$	
process p	process c
<pre>int d; while true { p1: wait(F); //pre-protocol p2: append(d, B); p3: signal(E); //post-protocol };</pre>	<pre>int d; while true { c1: wait(E); //pre-protocol c2: d := take(B); c3: signal(F); //post-protocol };</pre>

NB: p does $\text{wait}(F)$ and $\text{signal}(E)$, while q does $\text{wait}(E)$ and $\text{signal}(F)$.

The *PC safety* requirement is that c never removes an item from an empty buffer, and that p never puts an item into a full buffer.

Polling vs waiting

Common reasons for rejection

Using *polling/busy waiting* for synchronization is a common mistake that leads to submissions being *rejected*.

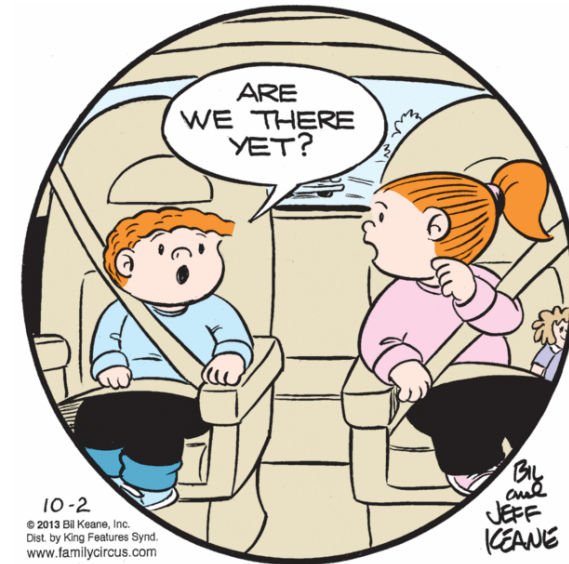
Here are some examples of polling/busy waiting in pseudo code. Loops that behave similarly to the situations below (where the dots do not include any *blocking wait*) are considered as polling.

```
while (e) { // POLLING!  
  ...  
  sleep(t);  
  ...  
}
```

Using a blocking operation within a loop is not considered as polling

```
while (e) { // NO POLLING!  
  ...  
  wait(o);  
  ...  
}
```

provided that it is **not** the case that that the waiting process is woken up from its wait at regular intervals. Thus, the following example is also an instance of polling:



“It depends on where you think we’re going.”

Q2 (15p). A small building firm can only build one house at a time, and cannot start on a new one till the present one is completed. The firm has N specialist workers such as a mason, a carpenter, an electrician, a plumber, etc. They are told to start on a house by the team manager, who then waits till each worker reports that they are done on this house, before starting the team on the next house. The firm never stops building houses.

(Part a). Write the declarations of the semaphores you will use in your solution. For each semaphore, indicate its name and the number of permits with which it is initialised. What should the scope of these semaphores be? (2p)

(Part b). Write the implementation of the method `run()` of the class `Worker` according to the description above. Remember that the only shared variables among threads are `NumSpecialists` and perhaps the semaphores you defined in Part a. (5p)

(Part c). Write the implementation of the method `run()` of the class `TeamManager` according to the description above. Remember that the only shared variables among threads are `NumSpecialists` and perhaps the semaphores you defined in Part a. (5p)

(Part d). How would your solution change if you only use binary semaphores? (3p)