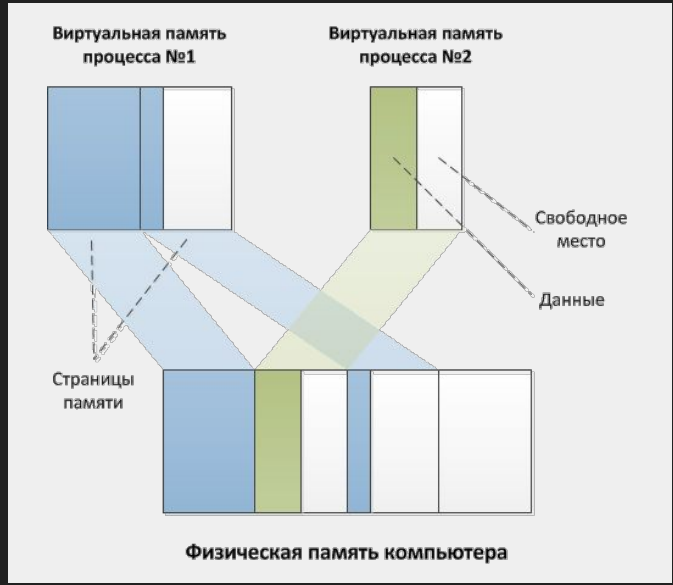
Java - Lesson 2

Multithreading

Memory, process, thread

- OS manages memory as pages
- Process is an instance of a program, being executed
- Process has a memory assosiated with it and one or multiple execution threads
- OS runs multiple independent processes to avoid system failures
- Process has MAIN thread of execution



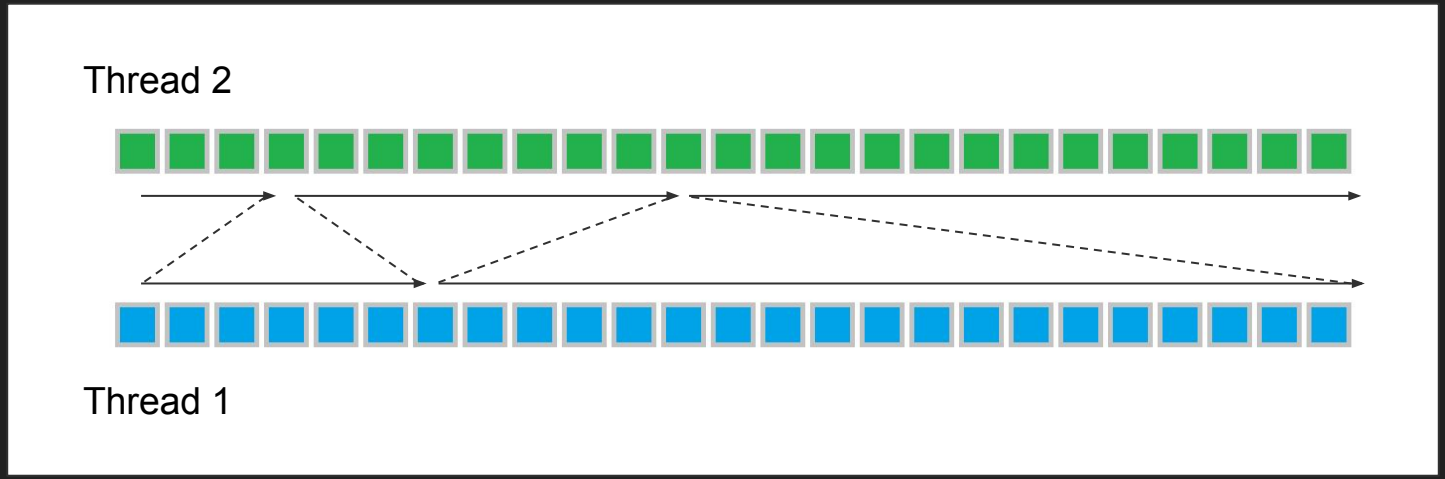
- Thread is a sequence of commands
- Threads exist only as a part of a process
- Multiple threads share state, memory and other resources of a parent process
- Context switching of threads is faster than context switching of processes

Process

- Memory (typically some region of virtual memory)
  - executable code
  - process-specific data (input and output)
  - call stack
  - heap to hold data generated during run time
- OS descriptors such as file descriptors or handles, data sources and sinks.
- Security attributes, such as the process owner and the set of permissions
- Processor state (context)
  - content of registers and physical memory addressing
  - state is typically stored in computer registers when the process is executing, and in memory otherwise
- OS holds most of this information about active processes in process control blocks
- Most OS have IPC mechanisms

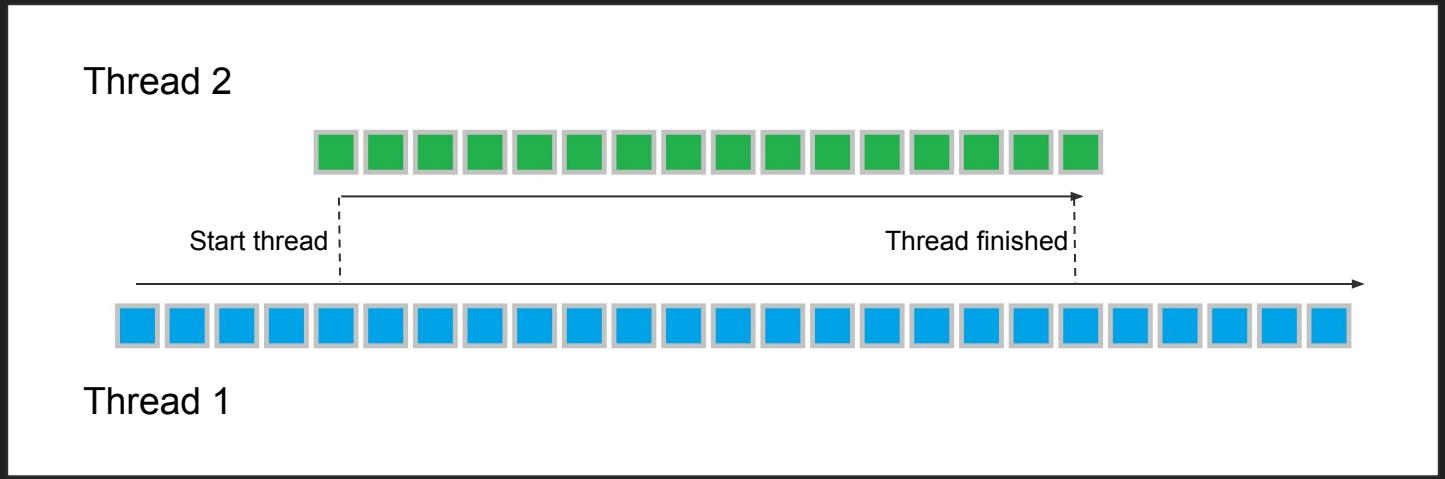
Threading

- Multiple threads can run on a single CPU core
- Process switches between threads, so they are executed consequently
- No need for synchronization
- No performance bonus, instead overhead when switching between threads



Threading

- Multiple threads can run on several CPU cores simultaneously
- Due to asynchronous behavior, might require synchronization (e.g. use of shared data)
- Potential deadlocks, race conditions, etc
- Performance bonus, no overhead when switching between threads



Java threads

- Basic threading includes ***Thread*** class and ***Runnable*** interface

```
	public class SimpleWorkingThread extends Thread {

		private final int mThreadId;

		public SimpleWorkingThread(final int threadID) {
			super();
			mThreadId = threadID;
		}

		@Override
		public void run() {
			super.run();
			System.out.println("thread " + mThreadId + " run was called");
			try { Thread.sleep(1000); } catch (InterruptedException e) {	e.printStackTrace(); }
		}

	}
...
		// create thread 1
		SimpleWorkingThread workerThread1 = new SimpleWorkingThread(1);

		// create thread 2
		SimpleWorkingThread workerThread2 = new SimpleWorkingThread(2);

		// execute threads simultaneously
		workerThread1.start();
		workerThread2.start();
```

Java threads

- Basic threading includes ***Thread*** class and ***Runnable*** interface

```
	private static void work(final int threadID) {
		System.out.println("thread " + threadID + " run was called");
		try { Thread.sleep(1000); } catch ( InterruptedException e) { e.printStackTrace(); }
	}
...
		Thread workerThread1 = new Thread(new Runnable() {
			@Override
			public void run() {
				for(int i = 0; i < 10; ++i) {
					work(1);
				}
			}
		});

		Thread workerThread2 = new Thread(new Runnable() {
			@Override
			public void run() {
				for(int i = 0; i < 10; ++i) {
					work(2);
				}
			}
		});


		workerThread1.start();
		workerThread2.start();
```

Java threads

- Main thread can start other threads
- If main thread finishes execution, other threads will continue until they finish
- Daemon thread will be stopped irregardless of job still executed, if main thread is terminated

```
// create thread 2
Thread workerThread2 = new  Thread(new Runnable() {
        @Override
        public void  run() {
                for(int i = 0; i < 10; ++i) {
                        System.out.println("thread 2 run was called " + i + " times");
                        try {
                                // simulate thread being busy for long time
                                Thread.sleep(10);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
});

workerThread1.setDaemon(true);
// execute threads simultaneously
workerThread1.start();
workerThread2.start();
```

Java threads

- Thread can be stopped, resumed and terminated

```
thread.suspend();

thread.resume();

thread.stop();
```

- However these methods are deprecated as not safe. A thread can be terminated while modifying a state of an object, in this case it can lead to undefined behavior
- Threads should manage their life cycle by themselves

```
private volatile boolean mIsRunning = true;

public void finish() {
       mIsRunning = false;
}

public void run() {
       super.run();
       while (mIsRunning) {
          // do some magic
       }
}
```

Java threads: Interruption

- Interruption mechanism might be better, since it throws ***InterruptedException***, so it will stop if thread is suspended by sleep(), join(), yield(), etc

```
        public class SimpleWorkingThread extends Thread {
                @Override
                public void run() {
                        super.run();

                        int counter = 0;
                        while (!Thread.interrupted()) {
                                System.out.println("thread run was called " + (++counter) + " times");
                                Utils.waitFor(100, false);
                        }
                }
        }
...
        private SimpleWorkerThread mWorkerThread = new SimpleWorkingThread();

        public void execute() {
                mWorkerThread = new SimpleWorkingThread();

                // execute thread
                mWorkerThread.start();
        }

        public void stop() {
                if (mWorkerThread != null) {
                        mWorkerThread.interrupt();
                        mWorkerThread = null;
                }
        }
```

Simple thread synchronization mechanisms

```
Thread.yield();   // makes current thread stop execution and give CPU time to other threads
Thread.sleep(long millis);       // makes current thread sleep for given amount of milliseconds and give CPU time

// simple scenario
while(!msgQueue.isEmpty())              // while Message Queue is empty sleep, or yield
{
    if (useSleep)  { Thread.sleep(100); }
             else { Thread.yield(); }
}
```

```
someThread.join();               // makes current thread wait for someThread to terminate
someThread.join(long millis);    // same but will stop waiting and start execution after given amount of millis

    final Thread preprocessThread = new  Thread(new Runnable() {
            @Override
            public void  run() {
                    System.out.println("preprocessing... ");
                    Thread.sleep(1000);
                    System.out.println("Preprocessing is done!");
            }
    });

    Thread processThread = new  Thread(new Runnable() {
            @Override
            public void  run() {
                    preprocessThread.join();
                    System.out.println("Start processing");
            }
    });

    // execute threads simultaneously
    preprocessThread.start();
    processThread.start();
```